

Creators: Christian Apostol, Marcus San Antonio

Pledge: *I pledge my honor that I have abided by the Stevens Honor System.*

CPU Name: *Budget CPU*

Budget CPU User Guide

Job Assignments

Christian Apostol: Hardware Design. Used Logisim Evolution to create a circuit (.circ) for the Budget CPU design.

- Designed machine code for each 8 bit instruction (explained down below) to work for three operations Add, Subtract, and Load.
- Incorporated Automatic Memory reader (Lab 9) to serve as the instruction memory of the CPU by splitting the output of each hexadecimal instruction into 8 distinct binary bits
- Create a register file (4 registers) that receives input from the instruction decoding phase to select certain registers for arithmetic or data load operations
- Created a simple ALU that decides between addition and subtraction
- Created a data memory (separate timer from instruction memory and register file) with 16 addressable bytes
- Used multiplexor in the Write Back (WB) phase to decide if ALU calculation or Data Memory is returned to the register file
- Created sample binary/hex test instructions
- Labeled and added captions on the CPU diagram

Marcus San Antonio: Compiler. Used python to write a *compile.py* program that outputs two image files (*instruction_mem.txt* & *data_mem.txt*)


- Created function that read assembly from *assembly.txt* and divided each section into two arrays. (1) Instruction array, (2) Data Memory assignments
- Created function that converts instructions into hexadecimal instructions based on the mnemonics from the hardware
- Created function that writes data values to corresponding spots in a text file (*data_mem.txt*)
- Created and tested demo program in *assembly.txt*
- Commented code for clarification

Instructions - How to use the Budget CPU

The zip file should contain 5 files: *budget_cpu.circ*, *compile.py*, *assembly.txt*, *instruction_mem.txt*, *data_mem.txt*. If the last 2 files are not included, that is fine since the program will generate them in that case. However, make sure the three other files are included in the same directory.

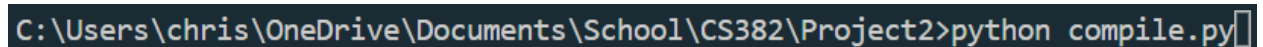
Step 1: Write Assembly Code in *assembly.txt*. The guide to each instruction is shown below. Make sure to follow it closely, and place each instruction on its own line in *assembly.txt*. Empty lines are allowed. To specify data, have one line that reads “DATA”. To assign memory, after the “DATA” line write M (for memory) and then a decimal value of 0 to 15 inclusive representing the address you want to write to. Follow this with a semicolon and space and add the value that you want to write to the register. Comments are not supported

A sample program is provided already in *assembly.txt*:

A screenshot of a text editor showing the contents of a file named 'assembly.txt'. The file contains assembly code for a CPU. It starts with five instructions: LDR R0 M0, LDR R1 M1, ADD R2 R1 R0, and SUB R3 R0 R1. Line 5 is empty. Line 6 is 'DATA'. Line 7 is 'M0: 4' and line 8 is 'M1: 3'. Line 9 is empty and has a green cursor. The instructions use registers R0, R1, R2, R3 and memory locations M0, M1.

```
≡ assembly.txt
1    LDR R0 M0
2    LDR R1 M1
3    ADD R2 R1 R0
4    SUB R3 R0 R1
5
6    DATA
7    M0: 4
8    M1: 3
9
```

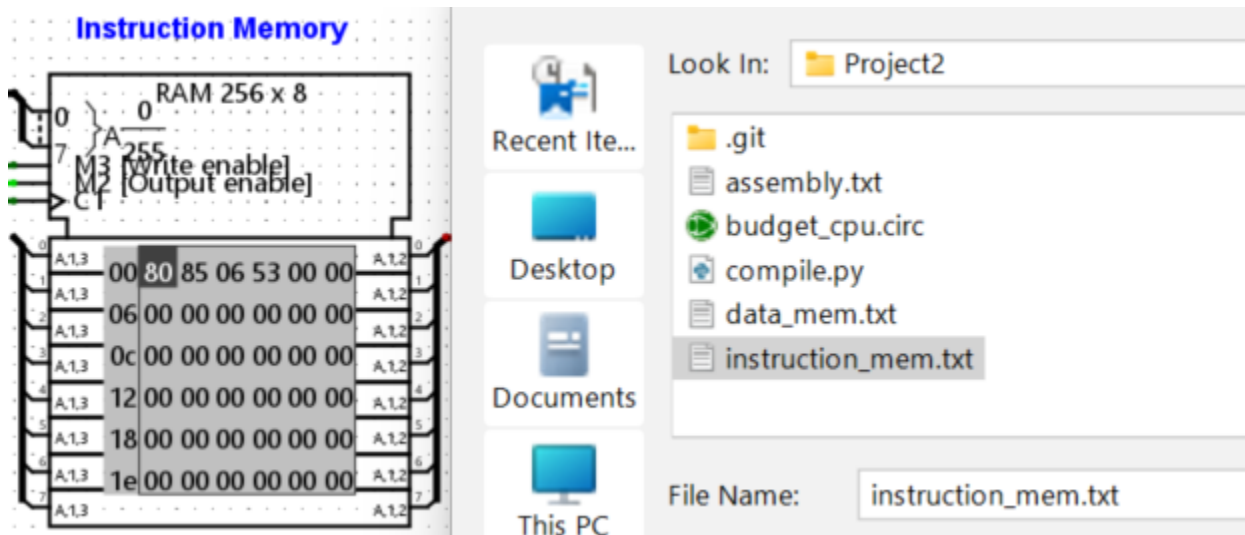
Step 2: Compile assembly into image files. For this step, make sure you have the most recent version of Python 3 downloaded on your computer. Open a terminal and change directories into the current directory with all of the project files. Run “python compile.py” or “py compile.py” : this may vary depending on how you downloaded python.

A screenshot of a terminal window showing the command 'python compile.py' being executed in the directory 'C:\Users\chris\OneDrive\Documents\School\CS382\Project2'.

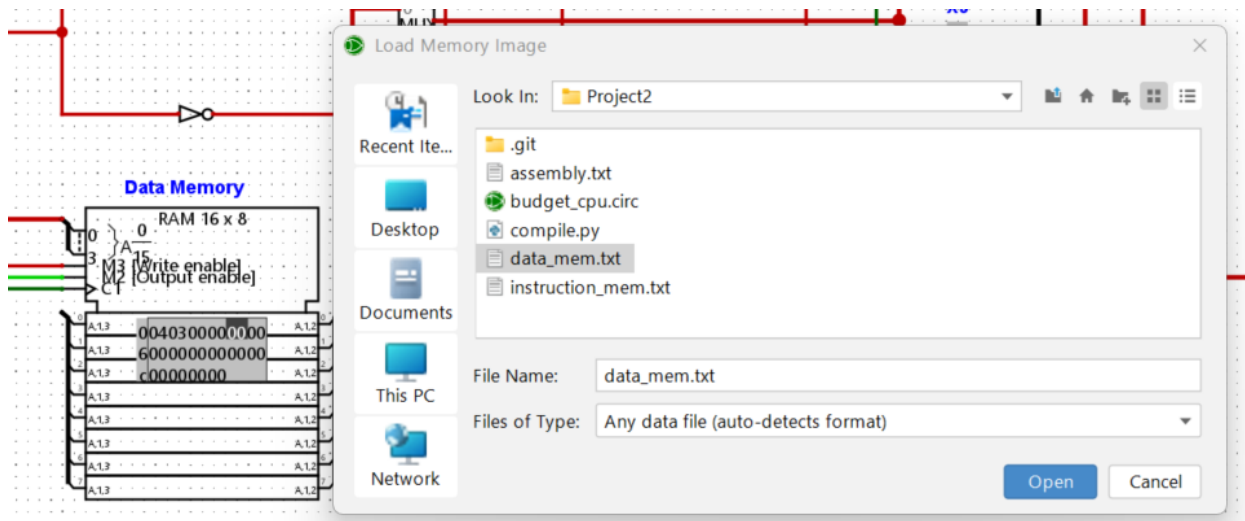
```
C:\Users\chris\OneDrive\Documents\School\CS382\Project2>python compile.py
```

Step 3: Load image files into the CPU. Open *budget_cpu.circ* and make sure that the simulation is reset before loading. Also, set the auto-tick frequency to 0.25 hz (can find this setting under the Simulate tab in Logisim Evolution).

Load *instruction_mem.txt* into the RAM labeled “Instruction Memory”



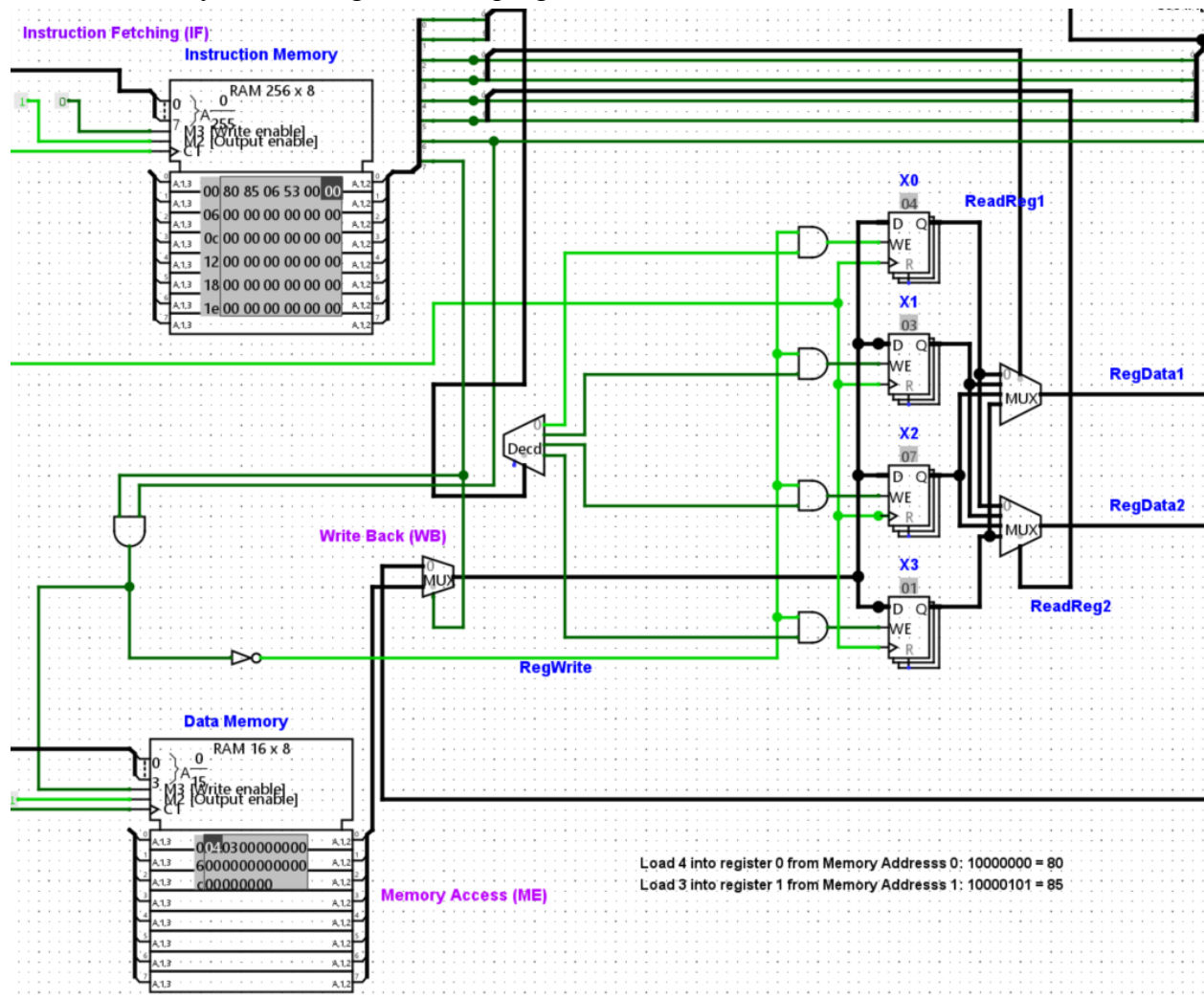
Load *data_mem.txt* into the RAM labeled “Data Memory”



If you are using the sample code in *assembly.txt*, then the RAMs should look exactly as depicted above.

Step 4: Run the program by going to Simulate → Toggle Auto-Tick Enabled (or just Ctrl + k)

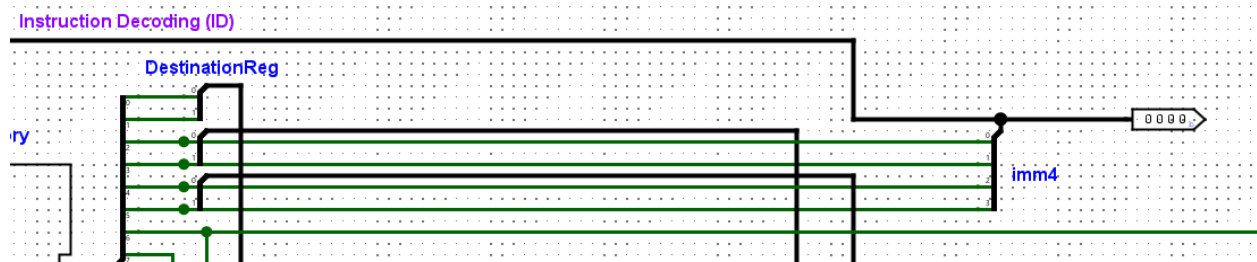
Even on 0.25 Hz, it still takes some time for the program to process, but the final result should look like this if you are using the demo program:



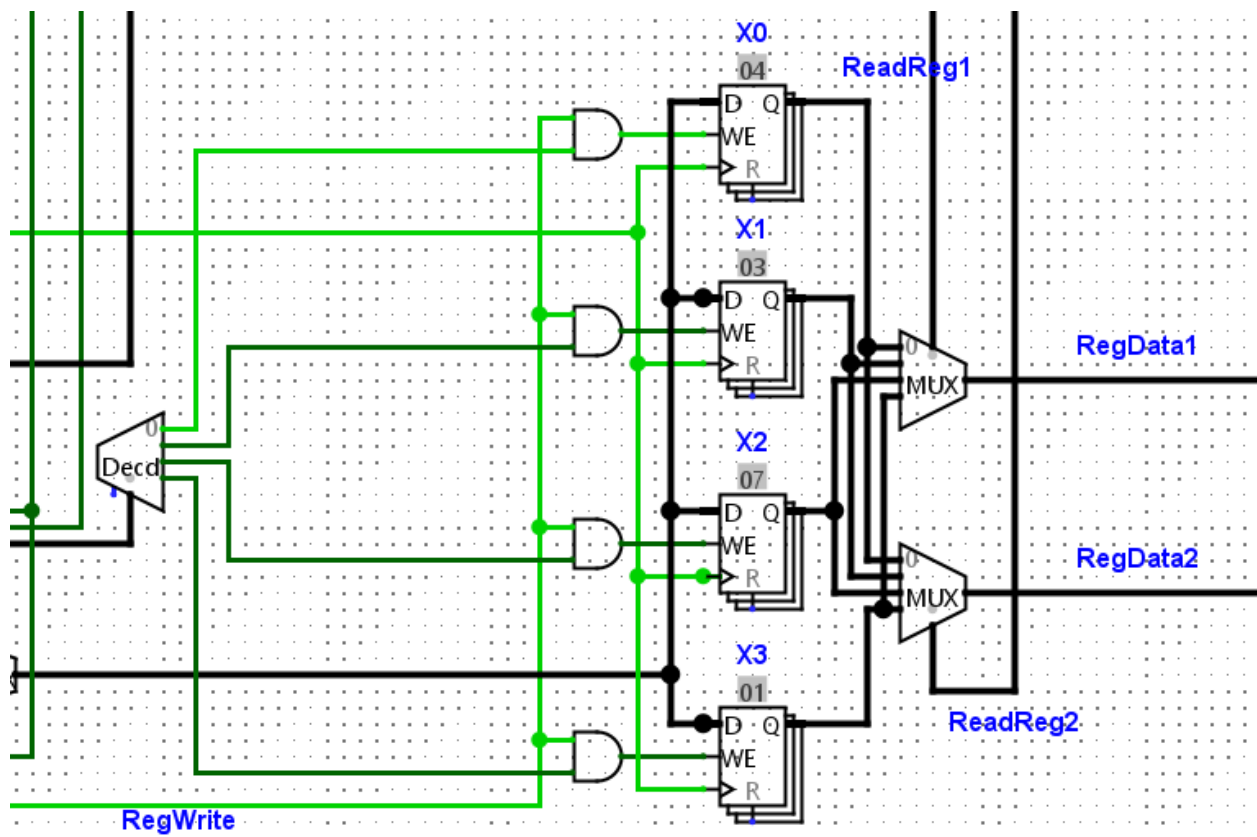
See register values.

CPU Architecture

Instruction Decoding: Splits the 8 bit instruction into different parts based on the mnemonic design (explained below). An 8 bit splitter is used to get each individual bit. Additional 2 bit splitters are used to get specific addresses for registers and a 4 bit splitter is used for the immediate.



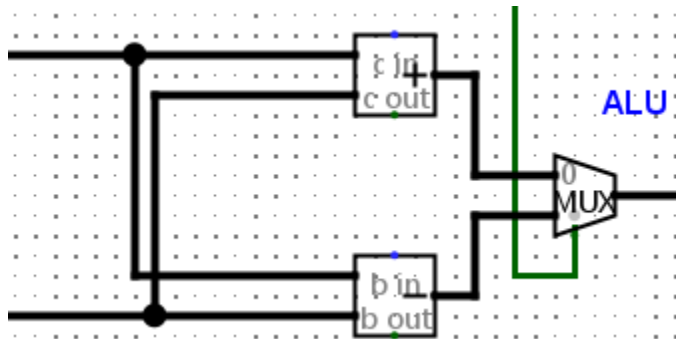
Register File: There are 4 general purposes in the CPU labeled X0 - X3. The Decoder in the middle of the circuit chooses which register is being written to based on an input of 2 bits, so 00 = X0, 01 = X1, 10 = X2, and 11 = X3. In the assembly code, it is referred to as R0 - R4, however the “R” does not really matter, as long as there is one character there present.



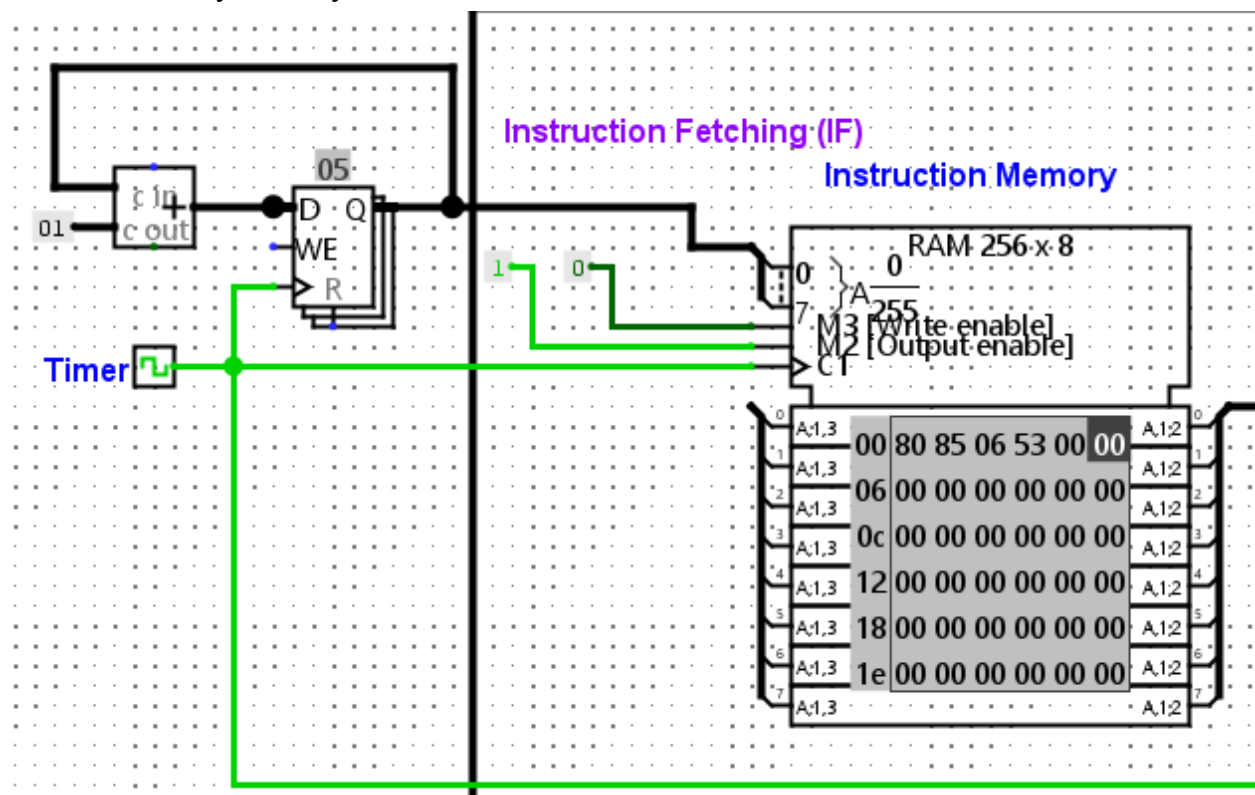
Functions: The CPU is capable of 3 operations

1. Adding values from two registers and placing the value in a destination register
2. Subtracting values from two registers and placing the value in a destination register
3. Loading values from 4 bit memory address into a register

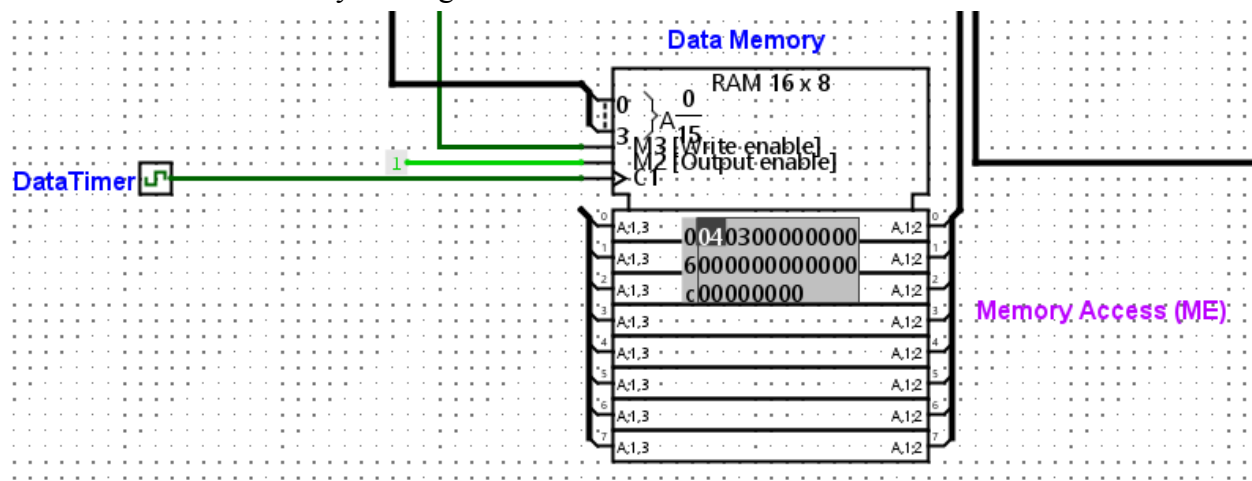
ALU: Takes two inputs: (1) The output of adding ReadReg1 and ReadReg2, (2) The output of subtracting ReadReg2 from ReadReg1. The machine code of the instruction decides which one to choose. This is done by connecting the ALU output to a multiplexor which is also connected to the output from the Data Memory. If load is the instruction, then it will choose the latter, else it will choose the former.



Instruction Memory: The design comes from a previous lab. An adder (+ 1) and a register is used to increment the memory address so that the program will automatically iterate over the instructions every clock cycle.



Data Memory: Data memory only has 16 addresses since we are using an 8 bit instruction, so we only have room to allocate a 4 bit immediate for the address. The timer is separate from the one used in instruction memory and register file.



Mnemonics/Instruction Design

Instruction Breakdown:

Instruction	Operation	Rr2	Rr1	Rd
ADD Rd,Rr1,Rr2	00	[4:5]	[2:3]	[0:1]
SUB Rd,Rr1,Rr2	01	[4:5]	[2:3]	[0:1]

There are 3 operations (ADD, SUB, LDR) hence the two bits

There are 4 registers, hence why Rd, Rr1, and Rr2 can be represented with only 2 bits.

Instruction	Operation	Mem	Rd
LDR Rd,Mem	10	[2:5]	[0:1]

For load, registers are not used, so [2:5] is used for the memory address. There are 16 addresses hence the 4 bits used to cover each one.

I[0] is rightmost bit*

XX XX XX XX
76 54 32 10

8-bit Instruction	Description	Example
I[6:7]: Operation	Denotes the operation that is being performed between two registers.	00: Addition 01: Subtraction 10: LDR 11: STR 53: 01010011 => R3 = R0 - R1
I[4:5]: ReadReg2	Denotes the second register that has an operation performed on it	53: 01010011 => R3 = R0 - R1
I[2:3]: ReadReg1	First register in the operation	53: 01010011 => R3 = R0 - R1
I[0:1]: Designation Register	Register that stores the result of the operation between ReadReg1 and ReadReg2	53: 01010011 => R3 = R0 - R1

More Examples:

LDR R0 M0 = 0x80 = 0b10000000 = “Load data stored at memory address 00 into register 0”
SUB R3 R0 R1 = 0x53 = 0b01010011 = “R3 = R0 - R1”