Encoding and Running of Programs

Arnav Kumar Behera, Vedanta Mohapatra

October 2022

1 Supported Instruction Set

Here, we discuss the instructions supported by the pipeline and their respective OP-Codes and their encoding scheme.

Encoding Scheme			
Instruction Syntax	RTN	OP-Code	Encoding Scheme
MOV R_i, R_j	$R_i \leftarrow [R_j]$	000000	$[R_j]_{31-27}[R_i]_{26-22}[*]_{21-6}[OP]_{5-0}$
MVI R_i, X	$R_i \leftarrow X$	000001	$[*]_{31-27}[R_i]_{26-22}[X]_{21-6}[OP]_{5-0}$
LOAD $R_i, X(R_j)$	$R_i \leftarrow [X + [R_j]]$	000010	$[R_j]_{31-27}[R_i]_{26-22}[X]_{21-6}[OP]_{5-0}$
STORE $R_i, X(R_j)$	$R_i \rightarrow [X + [R_j]]$	000011	$[R_j]_{31-27}[R_i]_{26-22}[X]_{21-6}[OP]_{5-0}$
ADD R_i, R_j, R_k	$R_i \leftarrow [R_j] + [R_k]$	000100	$[R_j]_{31-27}[R_k]_{26-22}[R_i]_{21-17}[*]_{16-6}[OP]_{5-0}$
ADI R_i, R_j, X	$R_i \leftarrow [R_j] + X$	000101	$[R_j]_{31-27}[R_i]_{26-22}[X]_{21-6}[OP]_{5-0}$
SUB R_i, R_j, R_k	$R_i \leftarrow [R_j] - [R_k]$	000110	$[R_j]_{31-27}[R_k]_{26-22}[R_i]_{21-17}[*]_{16-6}[OP]_{5-0}$
SUI R_i, R_j, X	$R_i \leftarrow [R_j] - X$	000111	$[R_j]_{31-27}[R_i]_{26-22}[X]_{21-6}[OP]_{5-0}$
AND R_i, R_j, R_k	$R_i \leftarrow [R_j] \& [R_k]$	001000	$[R_j]_{31-27}[R_k]_{26-22}[R_i]_{21-17}[*]_{16-6}[OP]_{5-0}$
ANI R_i, R_j, X	$R_i \leftarrow [R_j] \& X$	001001	$[R_j]_{31-27}[R_i]_{26-22}[X]_{21-6}[OP]_{5-0}$
OR R_i, R_j, R_k	$R_i \leftarrow [R_j] \vert\vert [R_k]$	001010	$[R_j]_{31-27}[R_k]_{26-22}[R_i]_{21-17}[*]_{16-6}[OP]_{5-0}$
ORI R_i, R_j, X	$R_i \leftarrow [R_j] \vert\vert X$	001010	$[R_j]_{31-27}[R_i]_{26-22}[X]_{21-6}[OP]_{5-0}$
HLT		001100	$[*]_{31-6}[OP]_{5-0}$

The general Encoding Scheme followed is $[Src1][Src2(opt)][Dst(opt)][Imm][Op]$ where
$Scr1$= Address of the First Source register
$Scr2$= Address of the Second Source register(optional)
Dst= Address of the Destination register
Imm= 16-Bit Immediate Value
Op= Opcode of the Instruction

2 Encoding Scheme

The position of the respective fields in the instruction is fixed, which allows the values(and addresses) to be loaded even before the instruction is decoded This allows us to complete the decoding and fetching of values from the register file in one clock cycle.

Let us now see how instructions can be encoded using our encoding scheme:
Consider the Instruction ADD R_1, R_2, R_3 , the opcode corresponding to ADD is 000100 (can be seen from the table), the first Source register's Address is 00010 (Since it is R_2), the second Source register's Address is 00011 (Since it is R_3), and the destination register's address is 00001 (Since it is R_1). The rest of the bits in the instruction have no significance so that they can be zero-padded.
Following the encoding scheme given in the table:

$$[R_j]_{31-27}[R_k]_{26-22}[R_i]_{21-17}[*]_{16-6}[OP]_{5-0}$$

we arrive at our instructions:

$$\overbrace{00010}^{\text{Src1}}\ \overbrace{00011}^{\text{Src2}}\ \overbrace{00001}^{\text{Dst}}\ \overbrace{00000000000}^{\text{0padding}}\ \overbrace{000100}^{\text{Op}}$$

The 32-Bit Decimal Code for the instruction is therefore:

$$00010000110000100000000000000100$$

Grouping 4-Bits at a time we get the equivalent hexadecimal code:

$$\underbrace{0001}_{1}\ \underbrace{0000}_{0}\ \underbrace{1100}_{C}\ \underbrace{0010}_{2}\ \underbrace{0000}_{0}\ \underbrace{0000}_{0}\ \underbrace{0000}_{0}\ \underbrace{0100}_{4}$$

Therefore the Equivalent Hexadecimal Instruction for ADD R_1, R_2, R_3 is 10C20004. A few other Encoding examples are given below:
ADI $R_1, R_2, 0BA0$: 1042E805
LOAD $R_1, 000A(R_2)$: 10400282
STORE $R1, 000B(R_2)$: 104002C3

3 Sample Encoding

Now, we provide instructions for Writing a Program, encoding it and then running it on the pipeline by means of an example.

Let's say we want to write a program to add two numbers present in the memory and write the answer back to the memory. Assuming that the first 32 addresses (0 through 31) in Memory are used for instructions, we can use the locations 32 onward for storing our data. Let's say the two operands are stored at locations 35 (23H) and 38 (26H). And we want to store their sum at location 40 (28H); we are taking any arbitrary register (initially loaded with 0) to be our base register, say, R_5.

The program that does this can be written as:

```
MVI R5,0H
LOAD R1,23H(R5)
LOAD R2,26H(R5)
ADD R3,R1,R2
STORE R3,28H(R5)
HLT
```

We can encode these instruction as discussed above:

- MVI R5,0H : We note in advance that the don't care bits are 0- padded

$$\overbrace{00000}^{\text{Src1(0padding)}}\ \overbrace{00101}^{\text{Dst}}\ \overbrace{0000000000000000}^{\text{Imm}}\ \overbrace{000001}^{\text{Op}}$$

  Grouping 4-Bits at a time we get the equivalent hexadecimal code:

$$\underbrace{0000}_{0}\ \underbrace{0001}_{1}\ \underbrace{0000}_{0}\ \underbrace{0000}_{0}\ \underbrace{0000}_{0}\ \underbrace{0000}_{0}\ \underbrace{0000}_{0}\ \underbrace{0001}_{1}$$

  The Hexadecimal encoding for the given instruction is 01000001

- LOAD R1,23H(R5) : We note in advance that the don't care bits are 0- padded

$$\overbrace{00101}^{\text{Src1}}\ \overbrace{00001}^{\text{Dst}}\ \overbrace{0000000000100011}^{\text{Imm}}\ \overbrace{000010}^{\text{Op}}$$

  Grouping 4-Bits at a time we get the equivalent hexadecimal code:

$$\underbrace{0010}_{2}\ \underbrace{1000}_{8}\ \underbrace{0100}_{4}\ \underbrace{0000}_{0}\ \underbrace{0000}_{0}\ \underbrace{1000}_{8}\ \underbrace{1100}_{C}\ \underbrace{0010}_{2}$$

  The Hexadecimal encoding for the given instruction is 284008C2

- LOAD R2,26H(R5) : We note in advance that the don't care bits are 0- padded

$$\overbrace{00101}^{\text{Src1}}\ \overbrace{00010}^{\text{Dst}}\ \overbrace{0000000000100110}^{\text{Imm}}\ \overbrace{000010}^{\text{Op}}$$

  Grouping 4-Bits at a time we get the equivalent hexadecimal code:

$$\underbrace{0010}_{2}\ \underbrace{1000}_{8}\ \underbrace{1000}_{8}\ \underbrace{0000}_{0}\ \underbrace{0000}_{0}\ \underbrace{1001}_{9}\ \underbrace{1000}_{8}\ \underbrace{0010}_{2}$$

  The Hexadecimal encoding for the given instruction is 28800982

- ADD R3,R1,R2 : We note in advance that the don't care bits are 0- padded

$$\overbrace{00001}^{\text{Src1}}\ \overbrace{00010}^{\text{Src2}}\ \overbrace{00011}^{\text{Dst}}\ \overbrace{00000000000}^{\text{0padding}}\ \overbrace{000100}^{\text{Op}}$$

  Grouping 4-Bits at a time we get the equivalent hexadecimal code:

$$\underbrace{0000}_{0}\ \underbrace{1000}_{8}\ \underbrace{1000}_{8}\ \underbrace{0110}_{6}\ \underbrace{0000}_{0}\ \underbrace{0000}_{0}\ \underbrace{0000}_{0}\ \underbrace{0100}_{4}$$

  The Hexadecimal encoding for the given instruction is 08860004

- STORE R3,28H(R5) : We note in advance that the don't care bits are 0- padded

$$\overbrace{00101}^{\text{Src1}}\ \overbrace{00011}^{\text{Src2}}\ \overbrace{0000000000101000}^{\text{0padding}}\ \overbrace{000011}^{\text{Op}}$$

Grouping 4-Bits at a time we get the equivalent hexadecimal code:

$$\underbrace{0010}_{2}\ \underbrace{1000}_{8}\ \underbrace{1100}_{C}\ \underbrace{0000}_{0}\ \underbrace{0000}_{0}\ \underbrace{1010}_{A}\ \underbrace{0000}_{0}\ \underbrace{0011}_{3}$$

The Hexadecimal encoding for the given instruction is 28C00A03

- The Hexadecimal encoding for HLT is 0000000C.

4 Sample Execution

Since we are done encoding the program, we move on to executing it on the pipeline. Follow the steps discussed below:

- Open Logisim-Evolution and open the circuit provided to you.
- Once the circuit is open, go to Simulate section as shown:

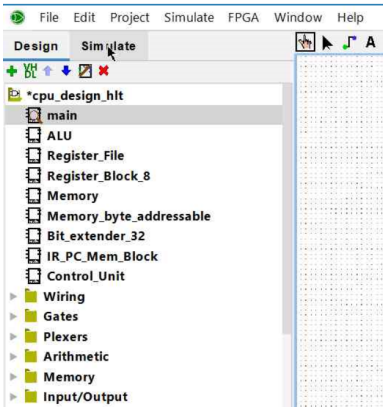


Figure 1: Step 1

- Then open the IR_PC_MEM Block under the subcircuits of the main by double-clicking on it.

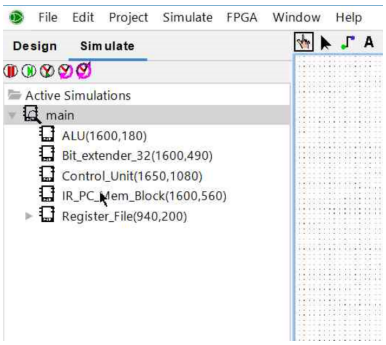


Figure 2: Step 2

- Zoom into the Memory Block (RAM) since we need to write our instructions to memory from 0 location onwards.

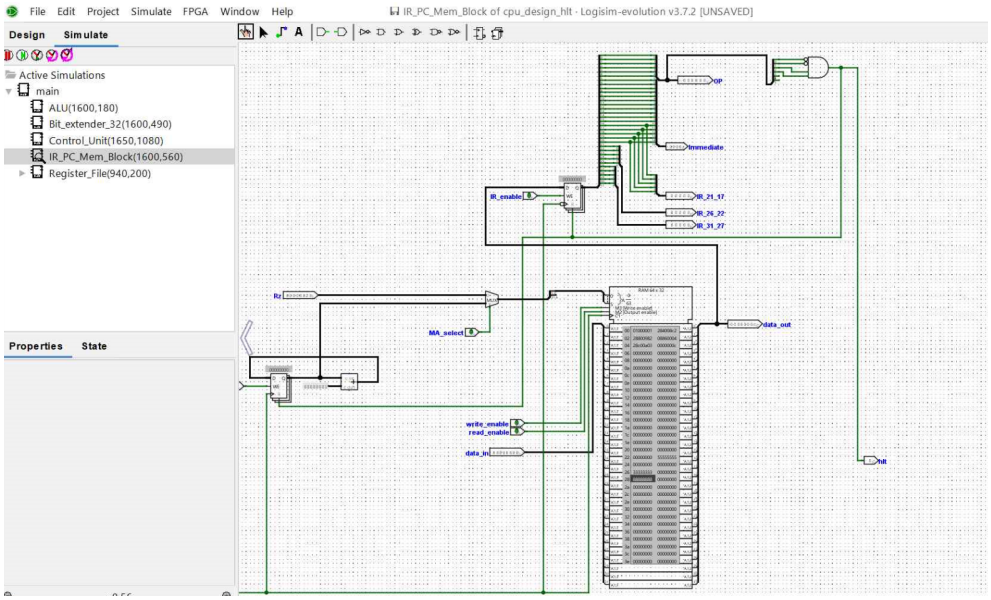


Figure 3: Step 3

- Now, we can enter the encoded instructions from memory location 0 onwards, as shown below. We click a field and then input the instruction by keyboard (order of insertion is left→right→down). With this, we have written our program into memory.

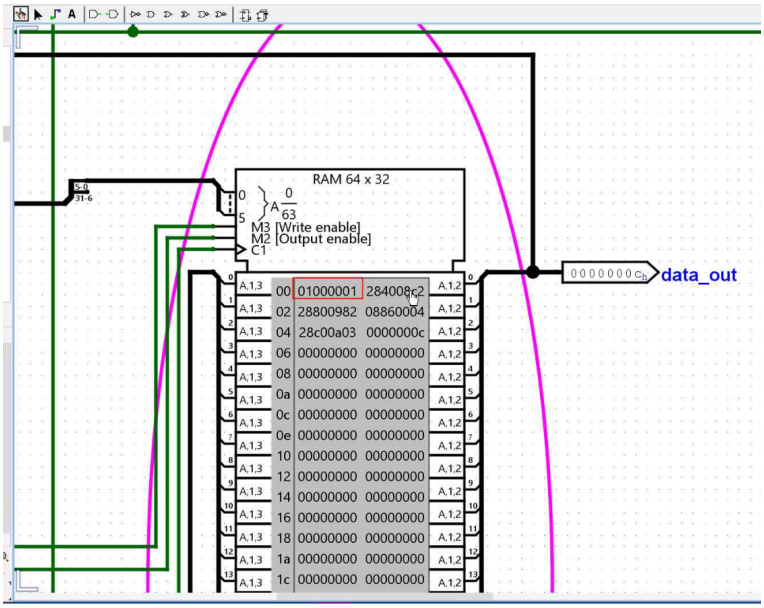


Figure 4: Step 4

- We also note that we need to enter appropriate values at memory locations 23H and 26H for the program to access. We load these locations with 55555555H and 33333333H respectively. We also set the value at 28H to 0 so that we can later check the working of the program. This has been done similarly as shown below:

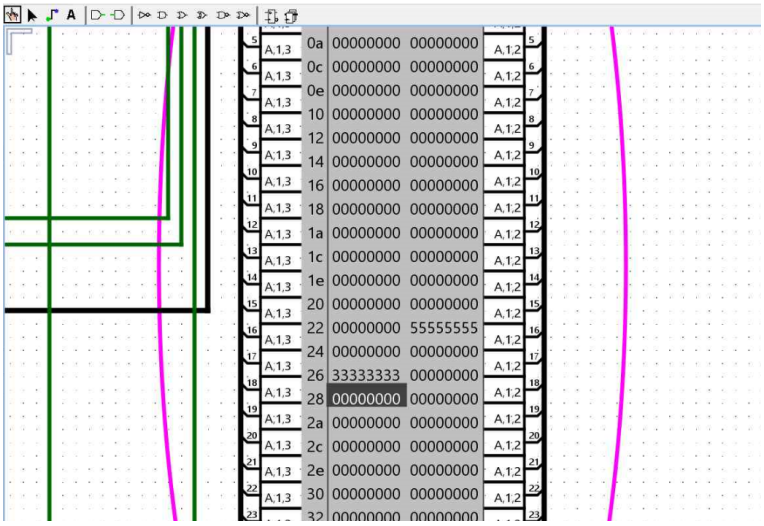


Figure 5: Step 5

- Since we have loaded both our instructions and data into memory, we go back to the main circuit. let us execute the program. To do this, we start the clock using Ctrl+K (on Windows) or Cmd+K (on Mac). Since our program has a handful of instructions and each of the instructions executes in 5 clock cycles, it could take quite a while. To speed up the execution, we increase the clock frequency, as shown below.

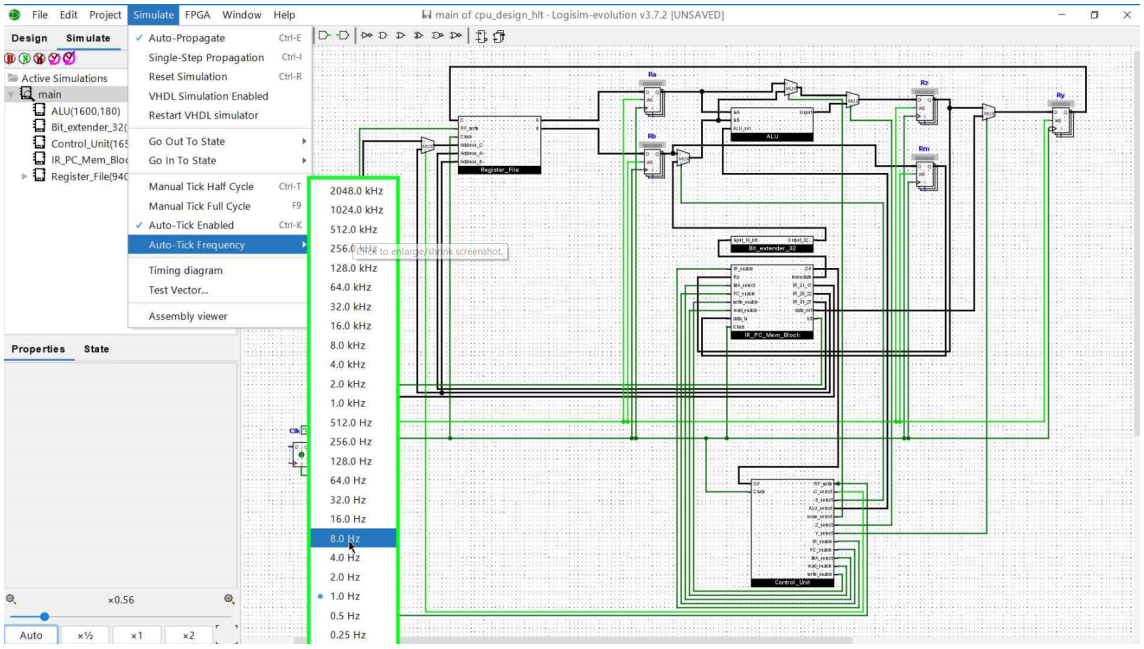
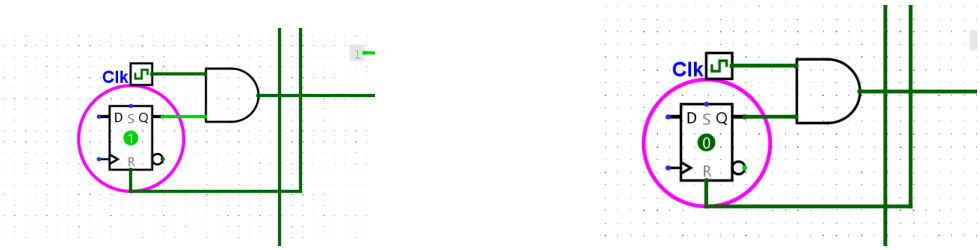


Figure 6: Step 6

- We are done with our prep. To execute the program, we just need to click on the register beside the clock in the main circuit and wait till the completion of execution. Completion is shown by the register value turning back to a dark colour from the bright green that we set it to.



(a) Starting Execution (b) Completing Execution

Figure 7: Step 7

Once we are done with execution, we can go on to check the values in memory and register files to see that the program has indeed executed properly. The contents of memory after execution are shown below:

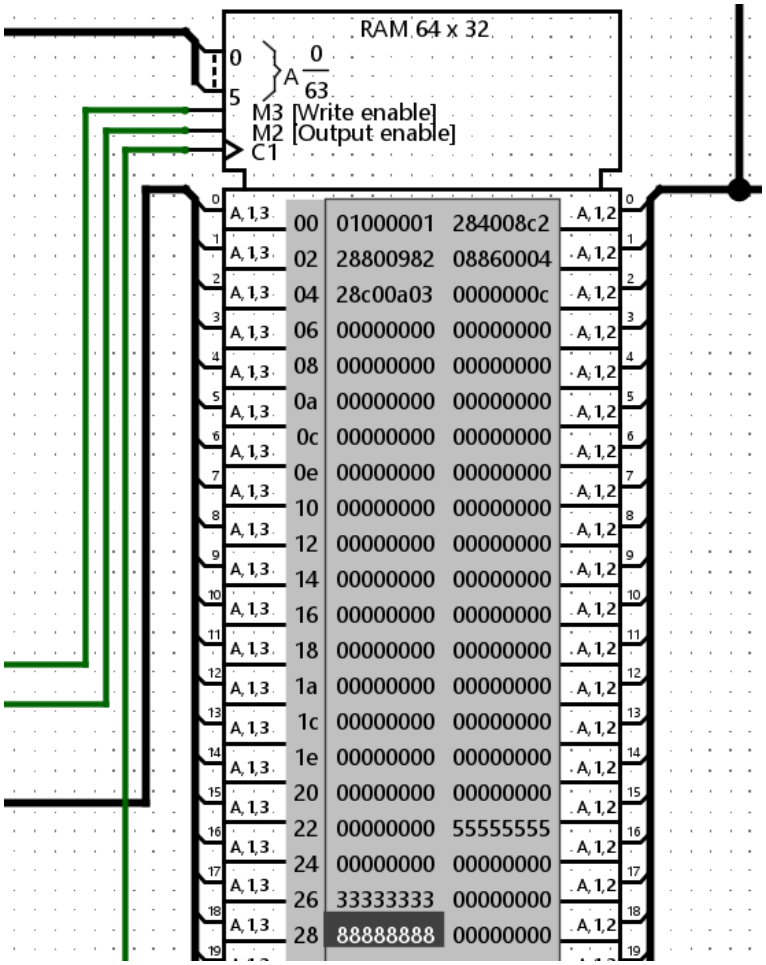


Figure 8: Verification of Memory

We can see that the location 28H is storing 88888888H, which is the correct result of the addition.

Let's also look at the contents of the register file:

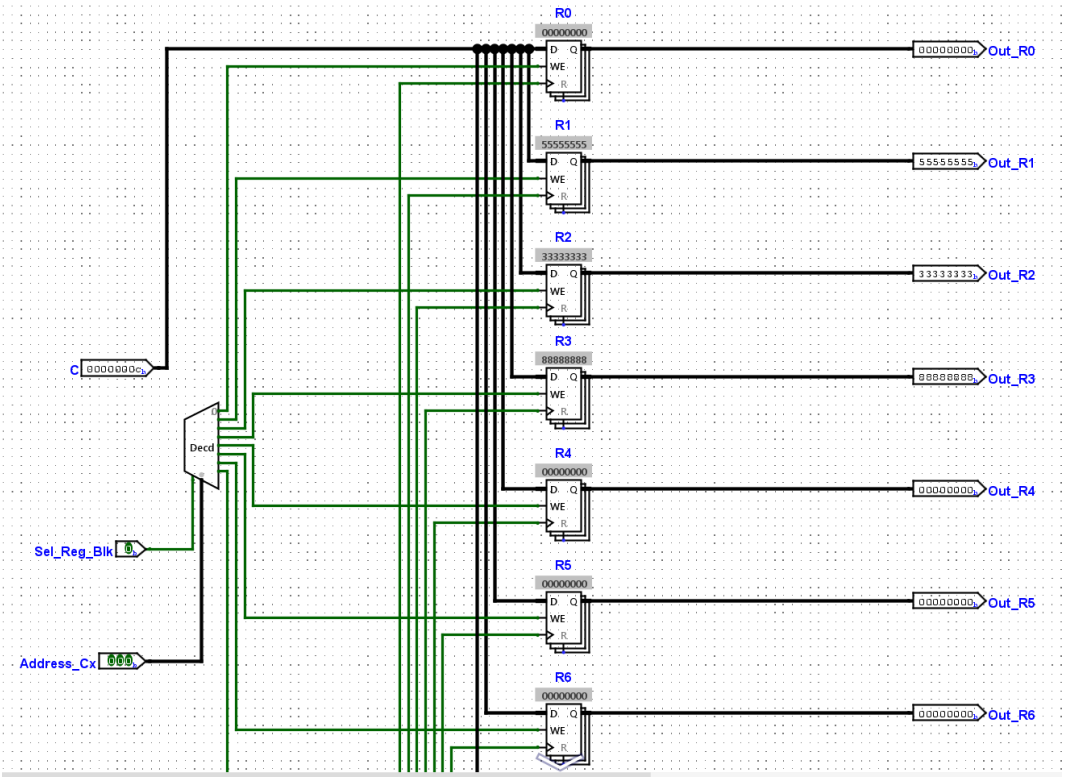


Figure 9: Verification of Memory

We can see the contents of R1 is 55555555H (loaded from 23H), the contents of R2 is 33333333H (loaded from 26H), and the contents of R3 is 88888888H (sum of [R2] and [R3]). Therefore we conclude that the program has executed successfully.