MEF

EE 203

DIGITAL SYSTEM DESIGN

MULTI- CYCLE CPU

Presented By

BAHA KARADAGLI 042101106

SEMIH YILMAZ 042101116

EMIR ONALAN 042201177

TABLE OF CONTENTS

OBJECTIVES

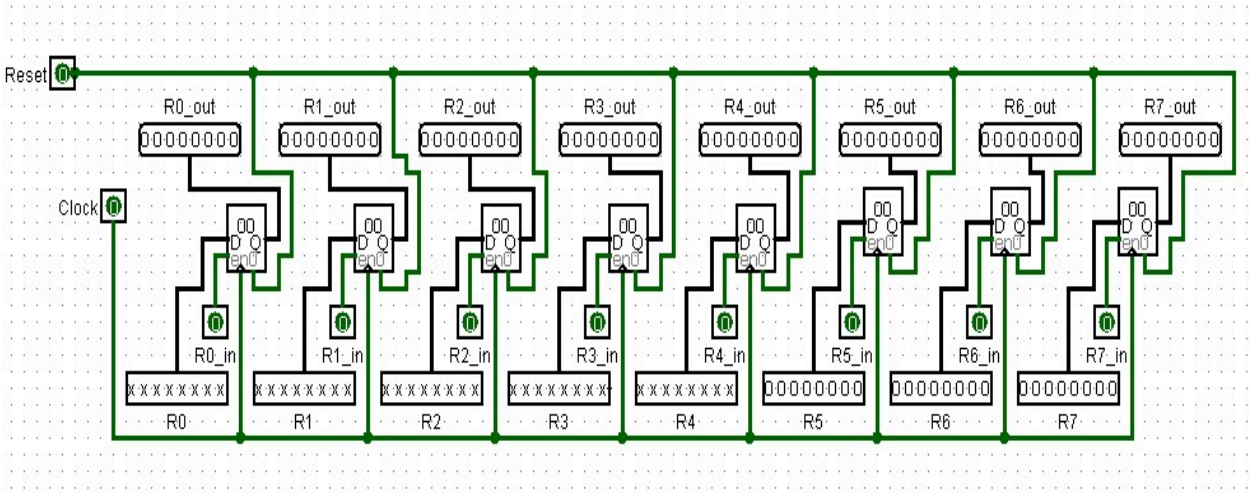
This project encourages participants to explore the complexities of digital system design and acts as a crucial learning opportunity. Participants' practical abilities are elevated by the focus on designing a multi-cycle CPU with an 8-bit data processing capacity, backed by a well-organized datapath and control unit. The development of eight flexible registers improves the ability to manipulate data, and template analysis fosters a clear comprehension of architectural details. The use of control words, microoperations, and an advanced ALU guarantees a nuanced execution strategy for instructions. Participants get a comprehensive grasp of CPU design, ALU operations, and the smooth integration of theoretical principles into practical situations through collaborative learning and hands-on applications. The features of reporting and documenting highlight how crucial it is to communicate effectively and provide results in an organized way. All in all, this project provides participants with invaluable skills for future applications and offers an extensive exploration of the complex world of digital system design.

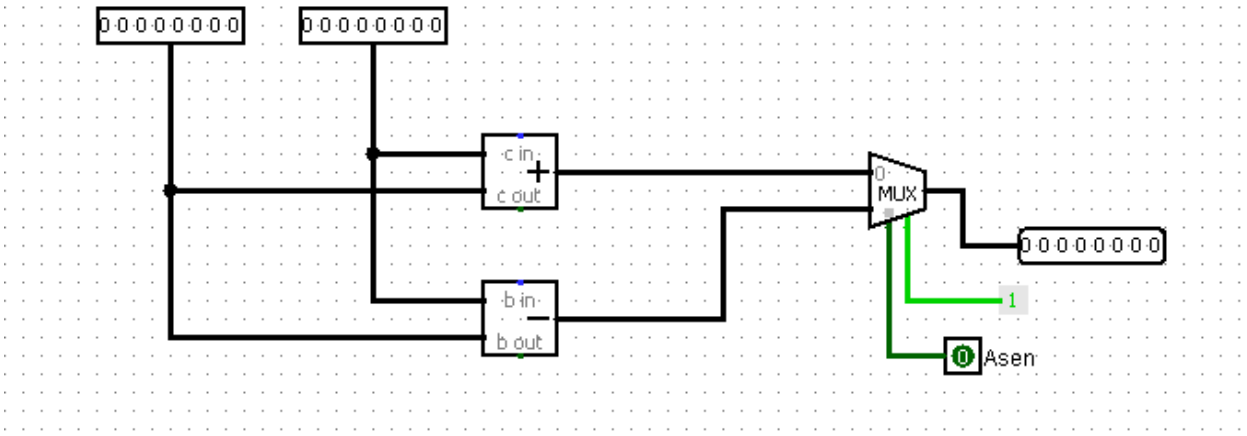
- Design and Implementation of Multi-Cycle CPU
- General Purpose Registers
- Template Analysis
- Arithmetic Logic Unit (ALU) Design
- Microoperations and Control Words
- Learning Testing and Demonstration Techniques
- Integration of ALU Operations
- Documentation and Reporting

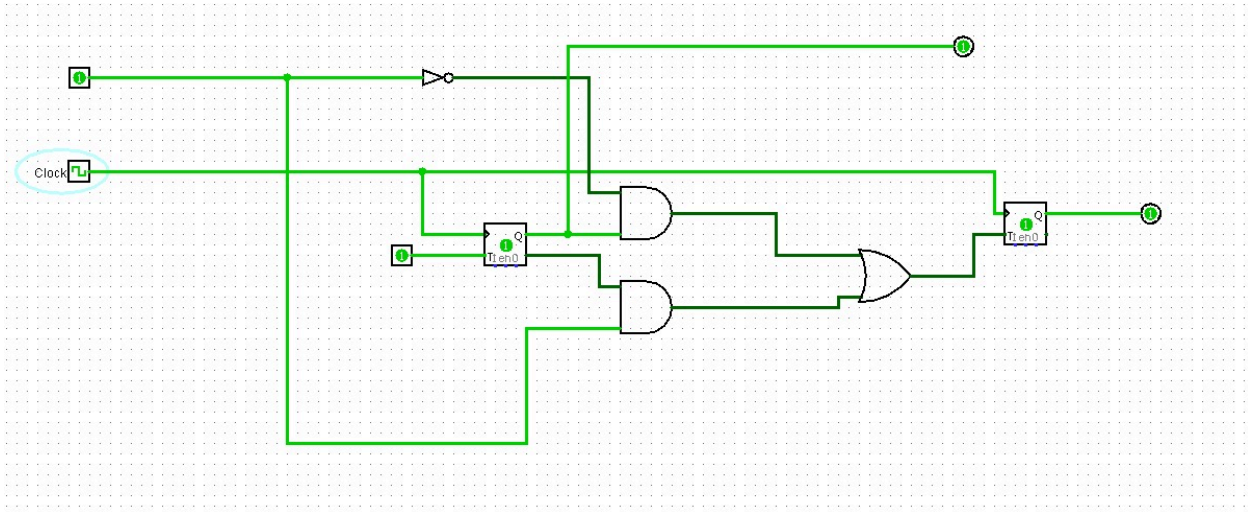
Description and The Design Work

2.1

The datapath of a multi-cycle computer, which includes registers, an ALU, multiplexers, and buses, is an essential conduit for data processing. It carries out I/O, logic, memory access, and arithmetic instruction execution. Being able to perform a wide range of operations, including addition, subtraction, multiplication, division, and logical bitwise operations, makes the ALU essential. The datapath works in tandem with the ALU to carry out essential operations, move data between registers, and execute instructions across a number of clock cycles. This procedure is coordinated by the control unit, which sends out control signals in response to each command. Registers R0 through R7, an extra A register, an IR for 8-bit instructions, an ALU with multiple operations, an 8-bit multiplexer, parallel data transfer buses, a 2-bit counter, clock edges, and labeled control signals are all included in the detailed design, which guarantees the multi-cycle computer architecture operates neatly and effectively.







We used the 2-bit counter by taking it from my login, but additionally we drew it in a hand-made 2-bit counter. For this counter, 2 T flip flops, 2 and gate, 1 or gate, 1 clock and 1 input , not gate and 2 output were used.

2.2

Logisim was used to implement the final circuit. Interestingly, because they were single loops, locations in the truth table labeled "T0" and "T1" with values of "11" and "10" were tagged as "X".The arithmetic logic unit (ALU) and registers, among other parts of the CPU, are controlled

by the control logic unit, which also determines how to run the CPU by sending control signals to execute instructions. The instruction register, which holds the current instruction being performed, is where input is received. This input is used by the control logic unit, together with signals like clock and reset, to make decisions like choosing data for registers and the ALU according to the instruction's opcode. The control logic unit generates control signals for the CPU's functioning using multiplexers, flip-flops, and logical gates. The implemented control logic contains a "DONE" output for counter clearing and selects registers using a MUX and a decoder, among other things.

TRUTH TABLE

OP1	OP0	T1	T0	DONE	RX_IN	RX_OUT	RY_OUT	A_IN	B_IN	B_OUT	ASEN	DIN_OUT	IR_IN
0	0	0	0	0	0	0	0	0	0	0	0	0	1
0	0	0	1	1	1	0	1	0	0	0	0	0	0
0	0	1	0	x	x	x	x	x	x	x	x	x	x
0	0	1	1	x	x	x	x	x	x	x	x	x	x
0	1	0	0	0	0	0	0	0	0	0	0	0	1
0	1	0	1	1	1	0	0	0	0	0	0	1	0
0	1	1	0	x	x	x	x	x	x	x	x	x	x
0	1	1	1	x	x	x	x	x	x	x	x	x	x
1	0	0	0	0	0	0	0	0	0	0	0	0	1
1	0	0	1	0	0	1	0	1	0	0	0	0	0
1	0	1	0	0	0	0	1	0	1	0	0	0	0
1	0	1	1	1	1	0	0	0	0	1	0	0	0
1	1	0	0	0	0	0	0	0	0	0	0	0	1
1	1	0	1	0	0	1	0	1	0	0	0	0	0
1	1	1	0	0	0	0	1	0	1	0	1	0	0
1	1	1	1	1	1	0	0	0	0	1	0	0	0

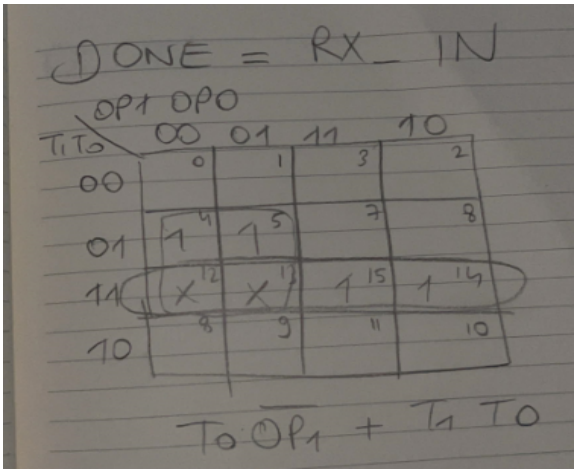


Fig.1 Karnaugh Map for DONE and RX_IN

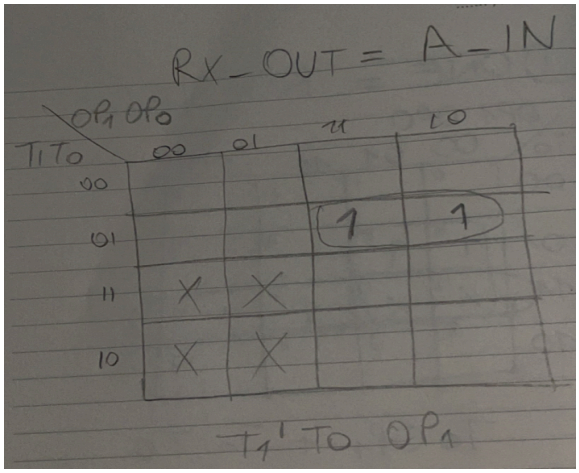


Fig.2 Karnaugh Map for RX_OUT and A_IN

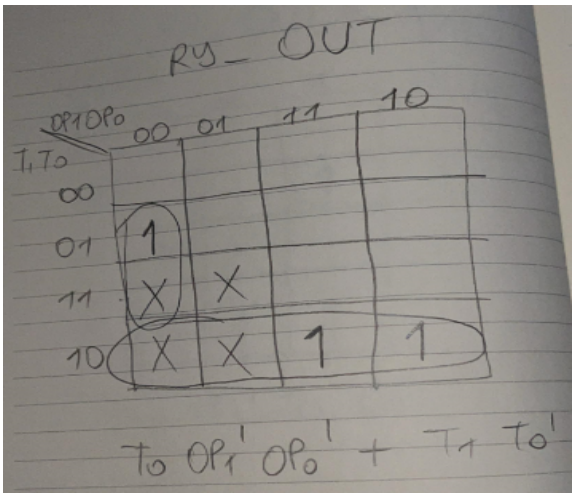


Fig.3 Karnaugh Map for RY_OUT

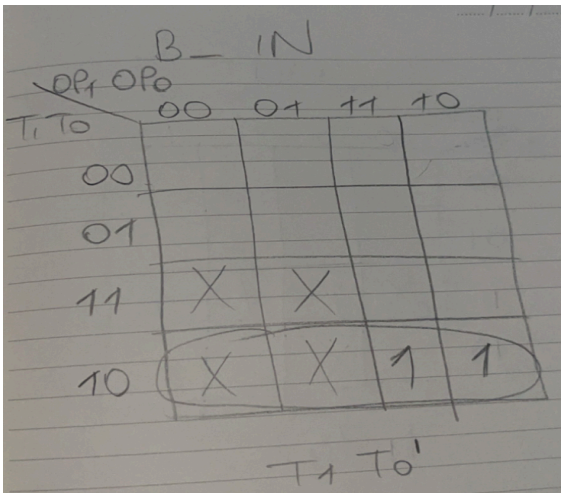


Fig.4 Karnaugh Map for B_IN

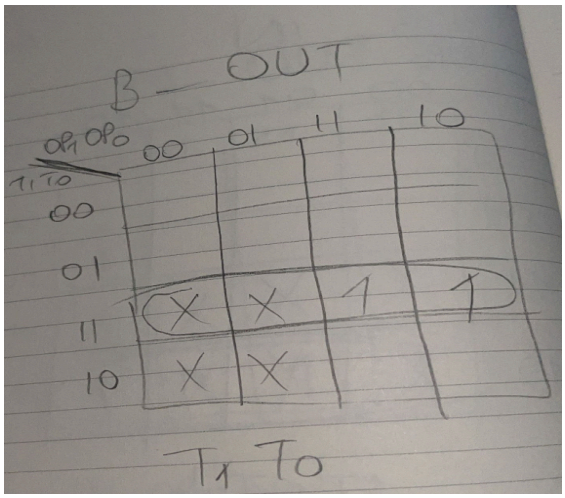


Fig.5 Karnaugh Map for B_OUT

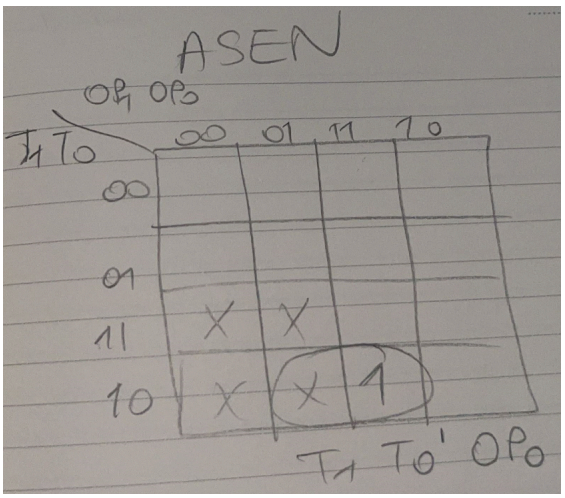


Fig.6 Karnaugh Map for ASEN

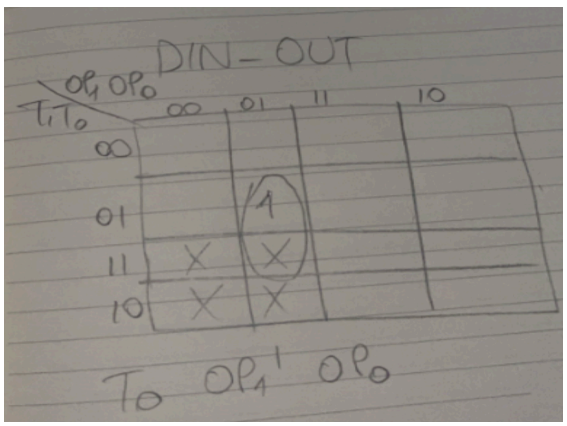


Fig.7 Karnaugh Map for DIN_OUT

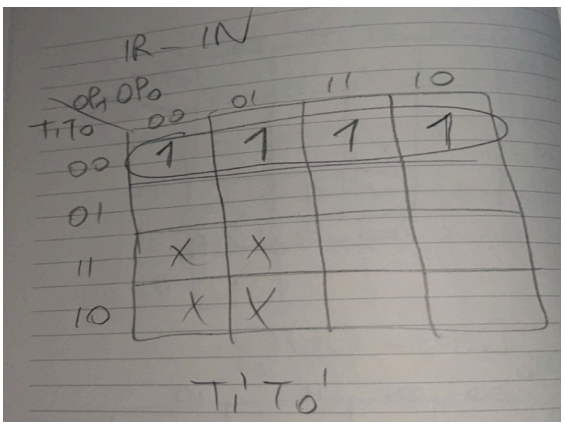
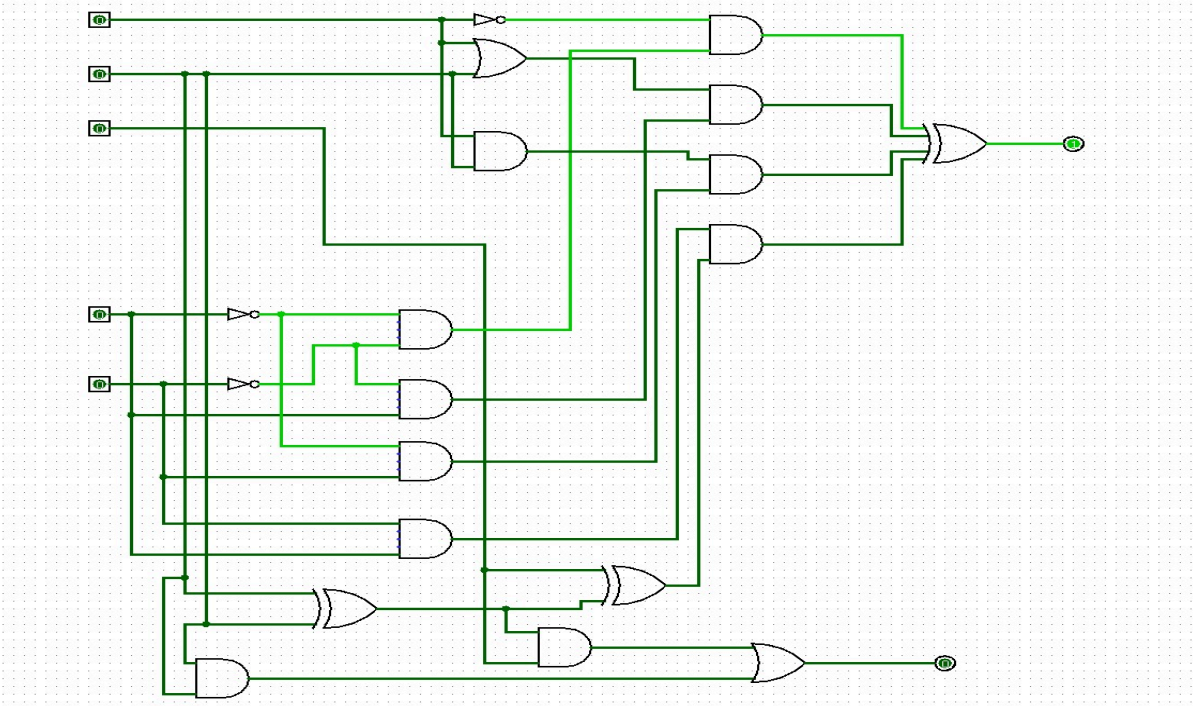


Fig.8 Karnaugh Map for IR_IN

2.3

In arithmetic operations, the carry-out and overflow outputs are important indications that support the control logic unit's decision-making process. When an arithmetic result exceeds the bit representation capacity, an overflow output signals, perhaps indicating an error or requiring additional processing. On the other hand, instances when an operation requires more bits for representation than are available are indicated by the carry-out output.The control logic unit can use these outputs as inputs to make well-informed judgments based on arithmetic results. The control unit could decide to ignore a result if there is an overflow condition, for example, or use the carry-out output to pinpoint situations that need more processing. After summation processes, the system essentially supports up to 9-bit binary results. It acts as an overflow detector when a carry-out is formed, indicating a 9-bit result. Thus, in order to preserve the integrity of the system's computing operations, the control logic unit can effectively handle overflow problems in certain circumstances.



Demonstration

3.1

First of all, we knew from the report that the values we would write to RAM were an 8-bit Instructor, the first 2 of these values were Operand 0 and Operand 1, and the other 6 values were divided into two by the splitter as xxx and yyy and specified the register paths we would process.

We set up another system to enter these 8-bit numbers into the RAM.

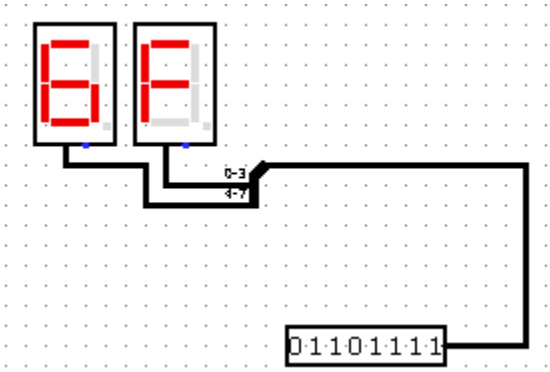


Figure 1.1 HEX Value finding system

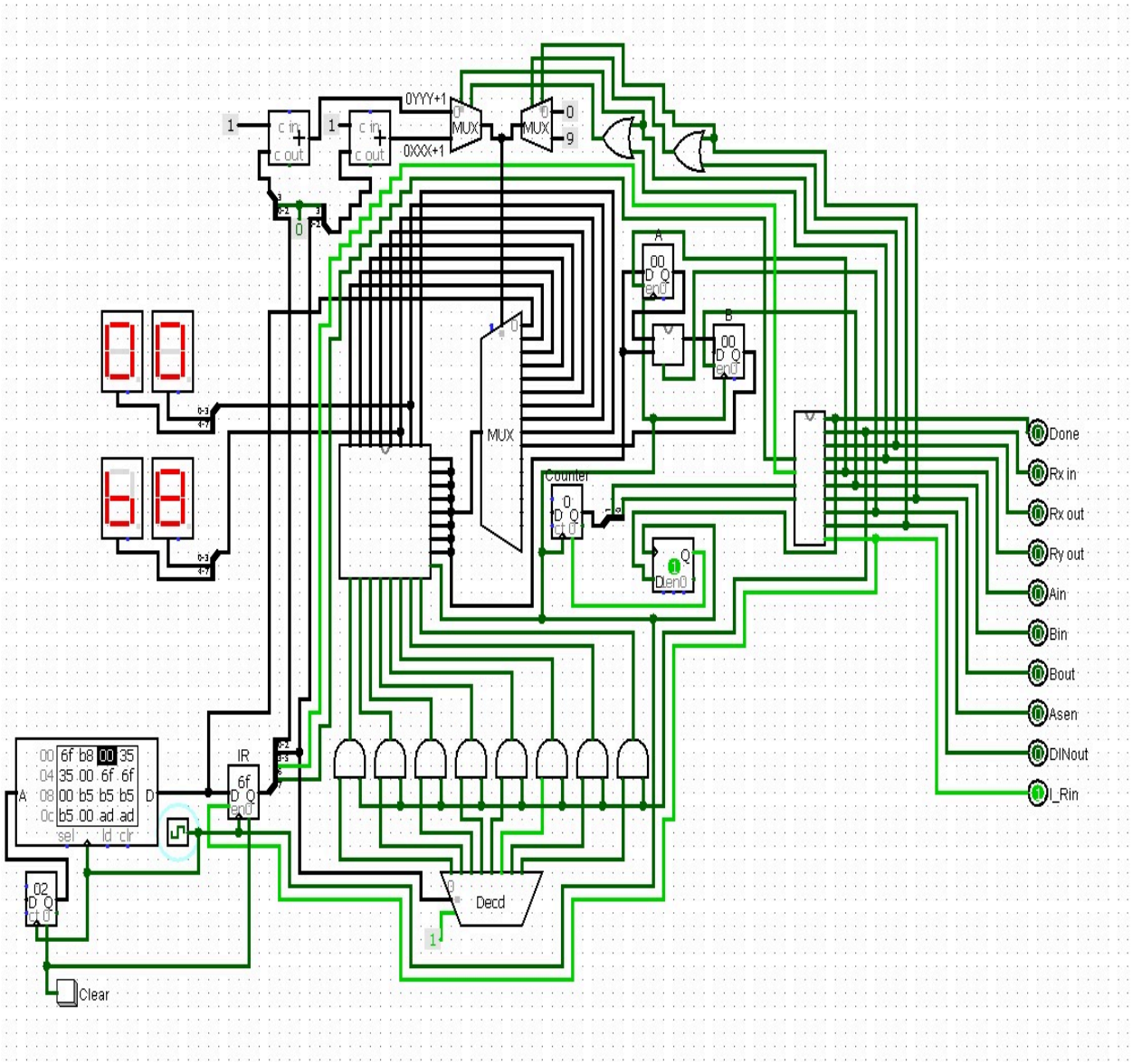
We found the HEX values to complete the demonstration part and perform the operations correctly. We put the value 00 between each operation we performed to reset the IR.

```
00  6f b8 00 35  00 00 6f 6f  00 b5 b5 b5  b5 00 ad ad
10  ad ad 00 ee  ee ee ee 00  00 00 00 00  00 00 00 00
20  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00
```

Figure 1.2 HEX Values that we should use to complete Demonstration

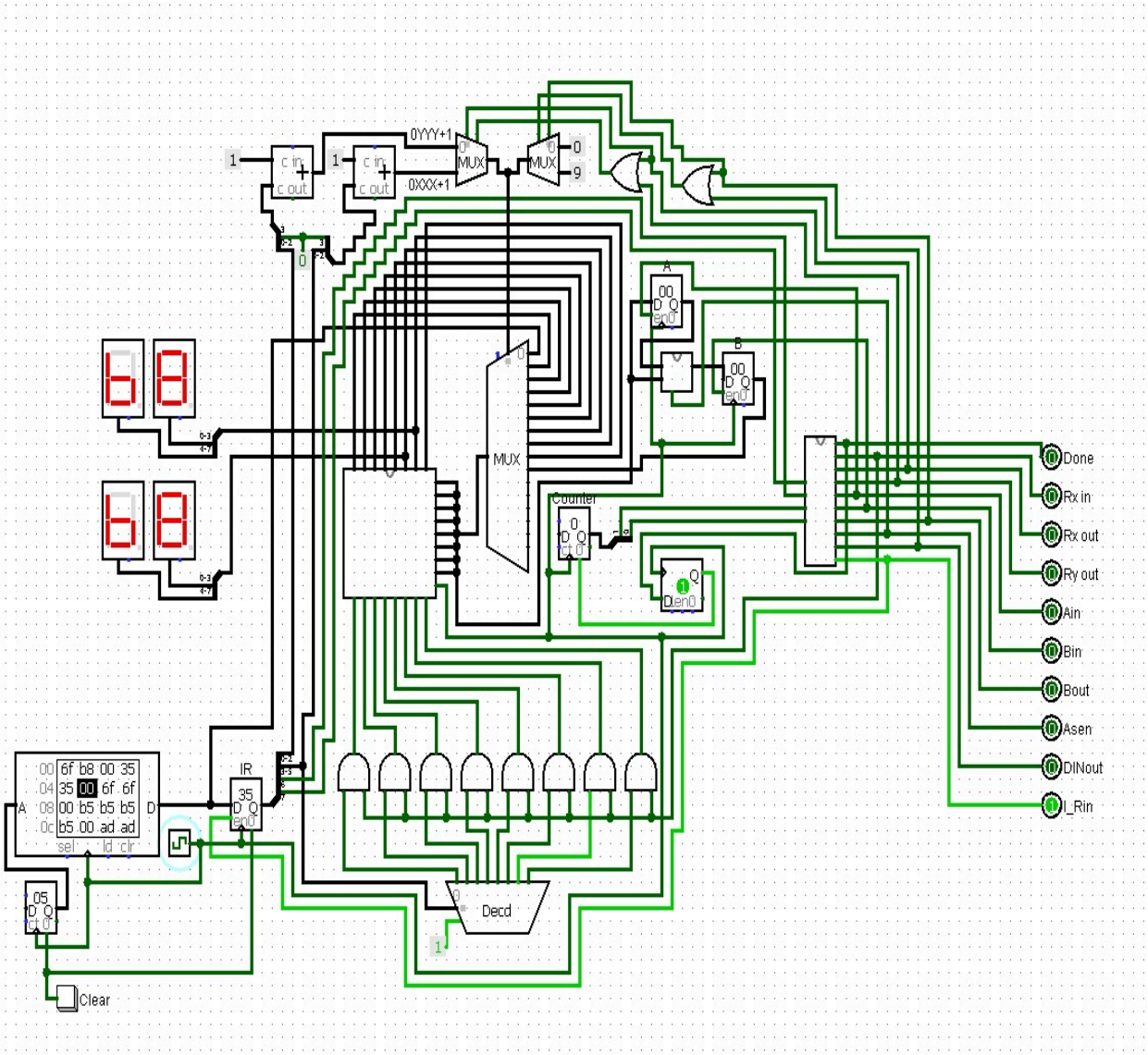
First Operation: mvi R5, 0xB8 | R5 ← 0xB8

- In the First Operation, we need to fill **R5** with the value **0xB8**.
- **mvi** command is completed twice and this command allows us to fill the register with the values we want. When we complete our First Operation, we write **6F** (that is, our IR) in the first space of the RAM. After that we write **B8** (that is, the value we want to fill) in the second space.



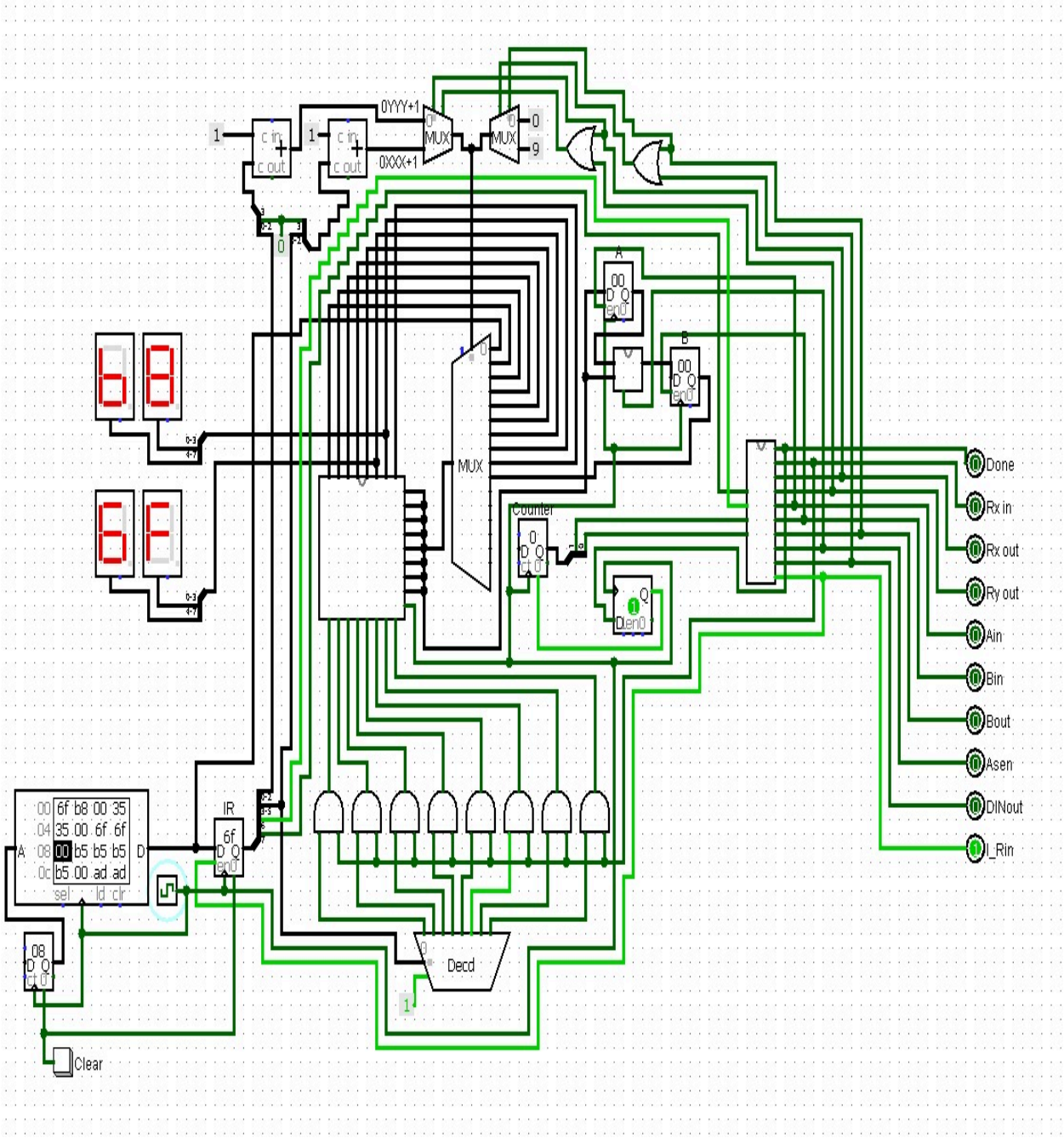
Second Operation: mv R6, R5 | R6 ← [R5]

- In the Second Operation, we need to copy **R5** to **R6**.
- **mv** command is completed twice and this command allows us to copy our xxx to yyy. When we complete our Second Operation, we write **35** (that is, our IR) in the 4th space of the RAM. After that we can type **any value** for 5th space of the RAM



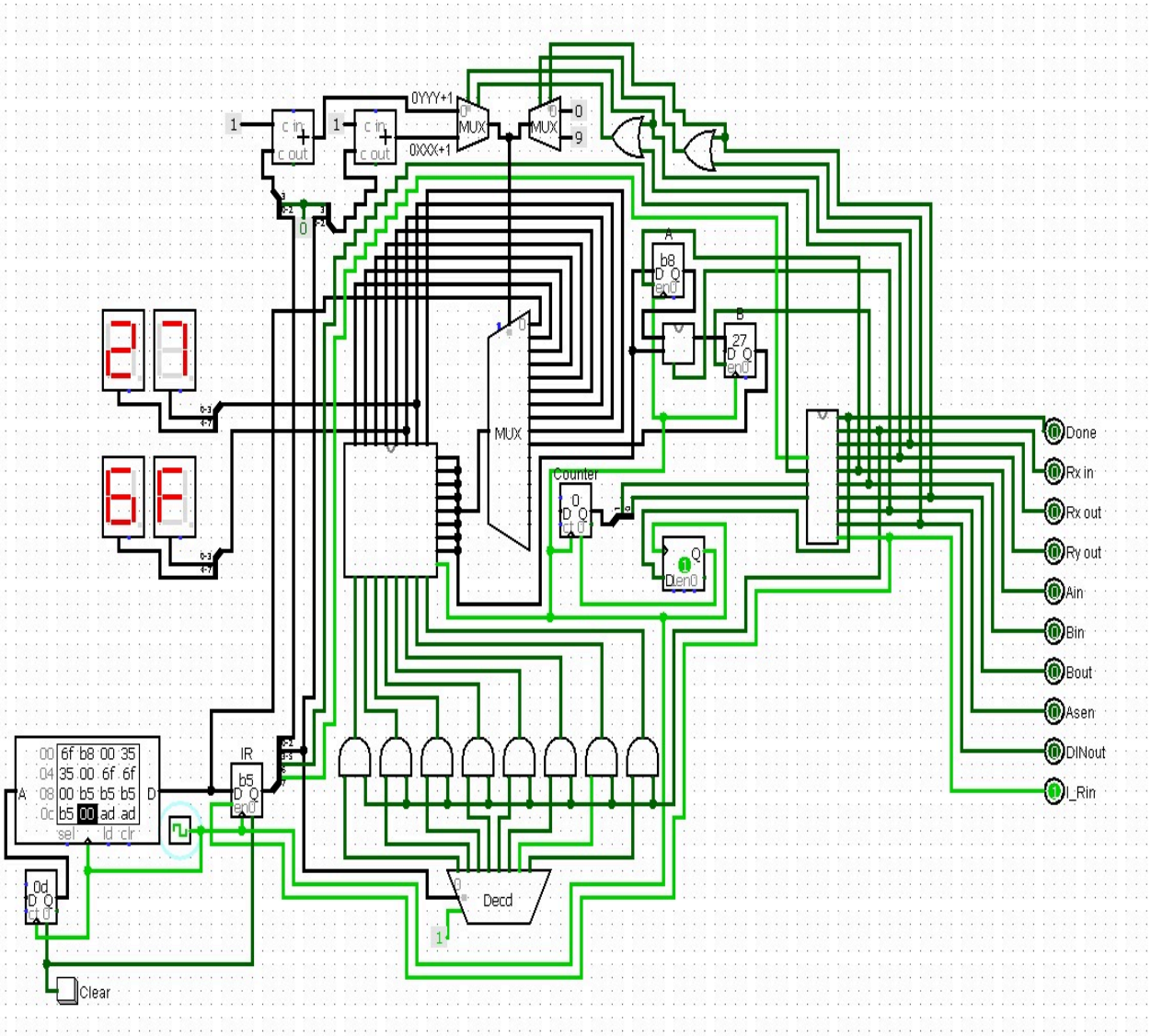
Third Operation: mvi R5, 0x6F | R5 ← 0x6F

- In the Third Operation, we need to fill **R5** with the value **0x6F**.
- We use the **mvi** command. When we complete our Third Operation, we write **6F** (that is, our IR) in the 7th space of the RAM. After that we write **6F** (that is, the value we want to fill) in the 8th space of the RAM.



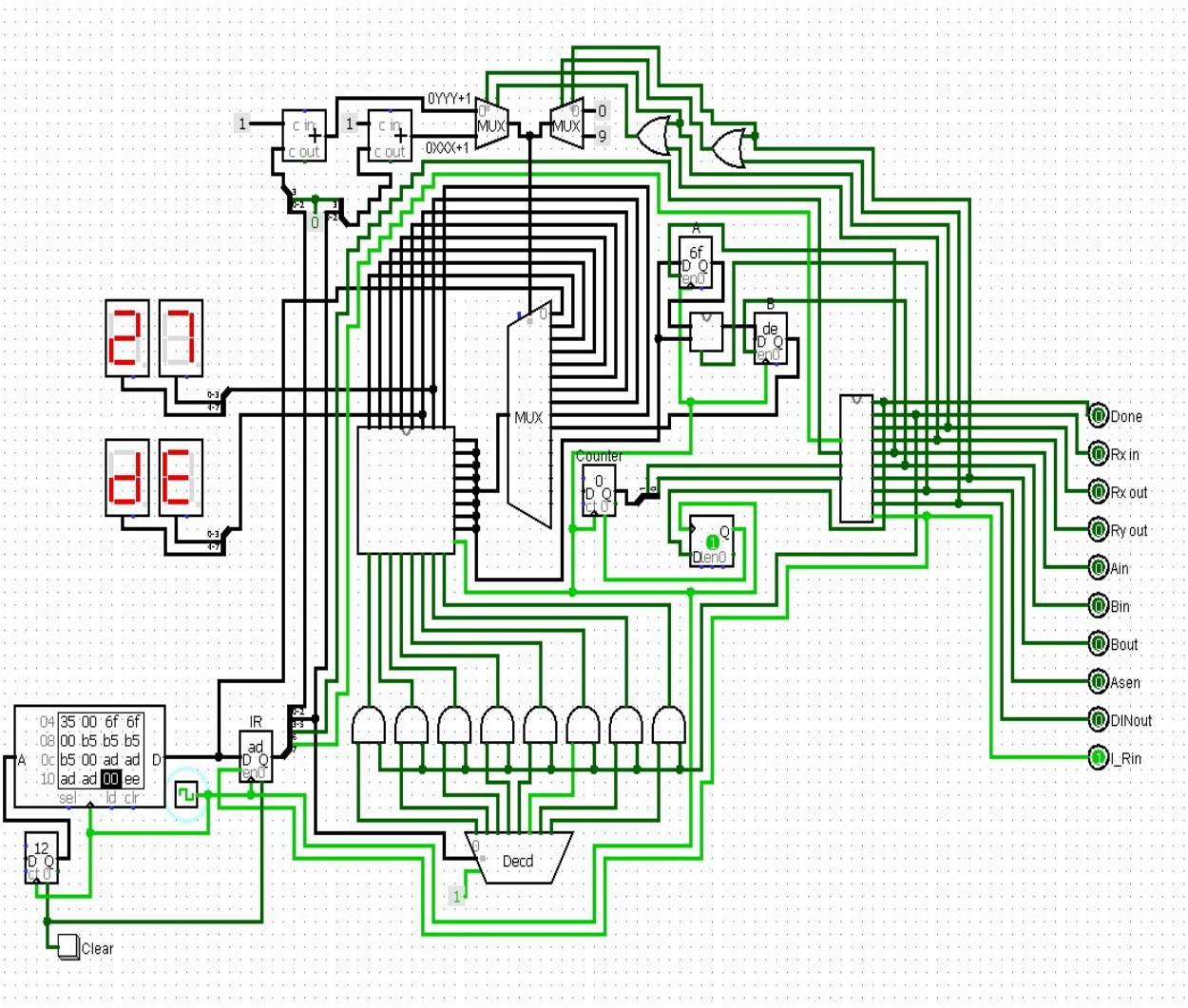
Fourth Operation: add R6, R5 | R6 ← R6 + [R5]

- In the Fourth Operation, we need to add **R5** to **R6**.
- The **add** command is completed 4 times and this command allows us to add by taking values with two registers. When we complete our Fourth Operation, we write **b5** (that is, our IR) in the 10th, 11th, 12th, 13th space of the RAM.
- When we finished the operation we found the result **27**. We added this value to **R6**.



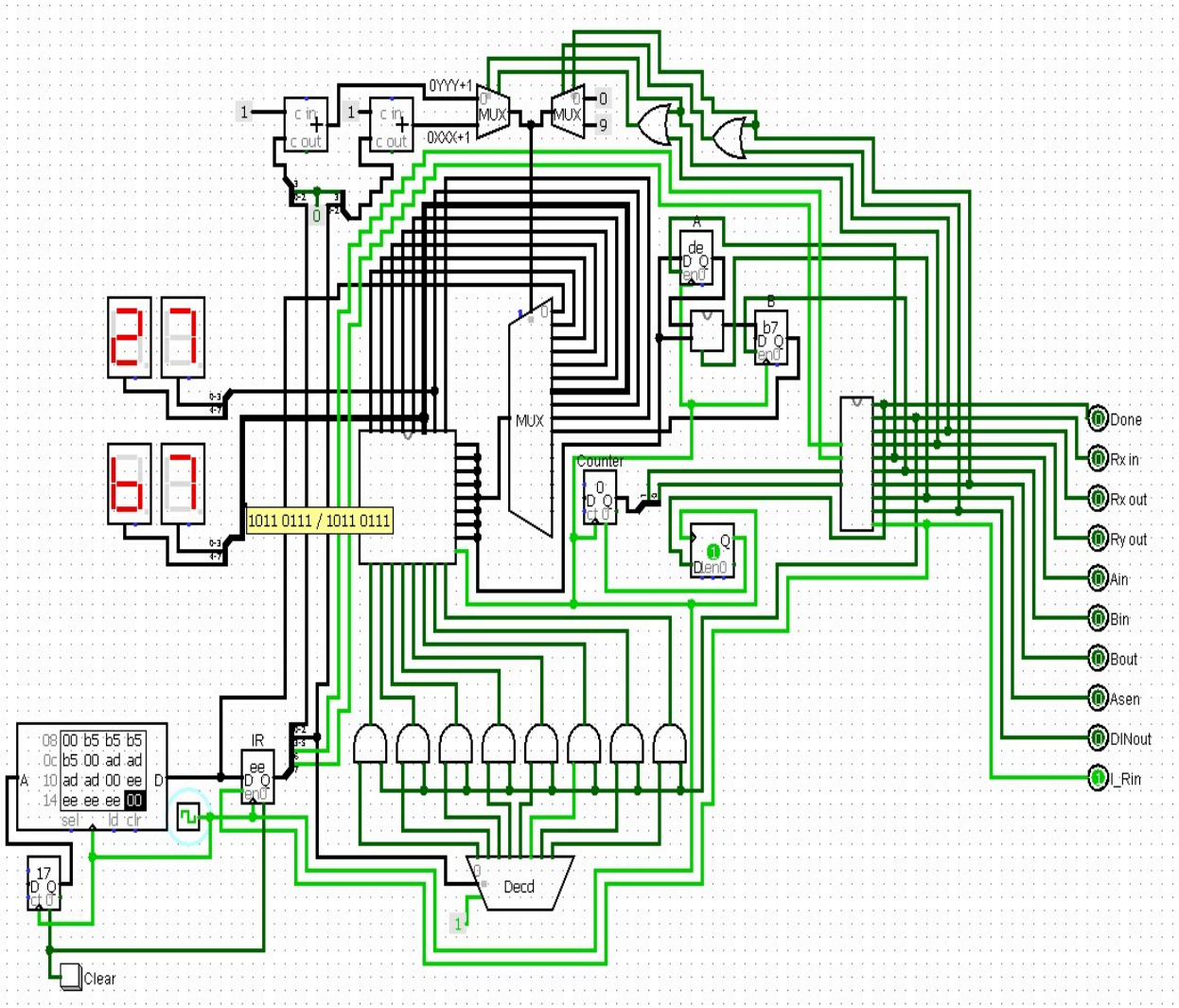
Fifth Operation: add R5, R5 | R5 ← R5 + [R5]

- In the Fifth Operation, we need to add **R5** to **R6**.
- We use the **add** command for this operation. When we complete our Fifth Operation, we write **Ad** (that is, our IR) in the 15th, 16th, 17th, 18th space of the RAM.
- When we finished the operation we found the result **dE**. We added this value to **R5**.



Sixth operation: sub R5, R6 | R5 ← R5 − [R6]

- In the Sixth Operation, we need to subtract R6 from R5.
- The **sub** command is completed in 4 times and this command allows us to subtract by taking values with two registers. When we complete our Sixth Operation, we write **EE** (that is, our IR) in the 20th, 21th, 22th, 23th space of the RAM.
- When we finished the operation we found the result **b7**. We added this value to **R5**.



3.2

The last set of hex LED displays in Figure is meant to provide a result similar to 0x6F-0xB8, however it is unable to perform the procedures correctly because of an overflow scenario in the addition operations. Finally, the LEDs linked to R5 and R6 produce the hex numerals **b7** and **27**, respectively.

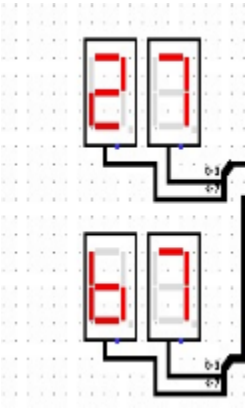


Figure Final display of LEDs

3.3

First of all, before starting we decided registers which Ra, Rb, Rc and we wrote them to below:

- Ra = R5
- Rb = R6
- Rc = R7

Before we start operations we check them about:

- Which commands are they doing ?
- Which IR`s should they need ?
- Which values should we integrate ?

We solve this question by type 8 bit number and seeing output in HEX Led`s and separating 8 bit IR for finding exact registers and doing exact operation.

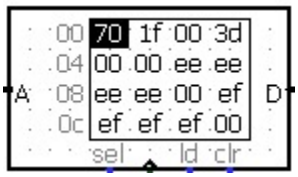
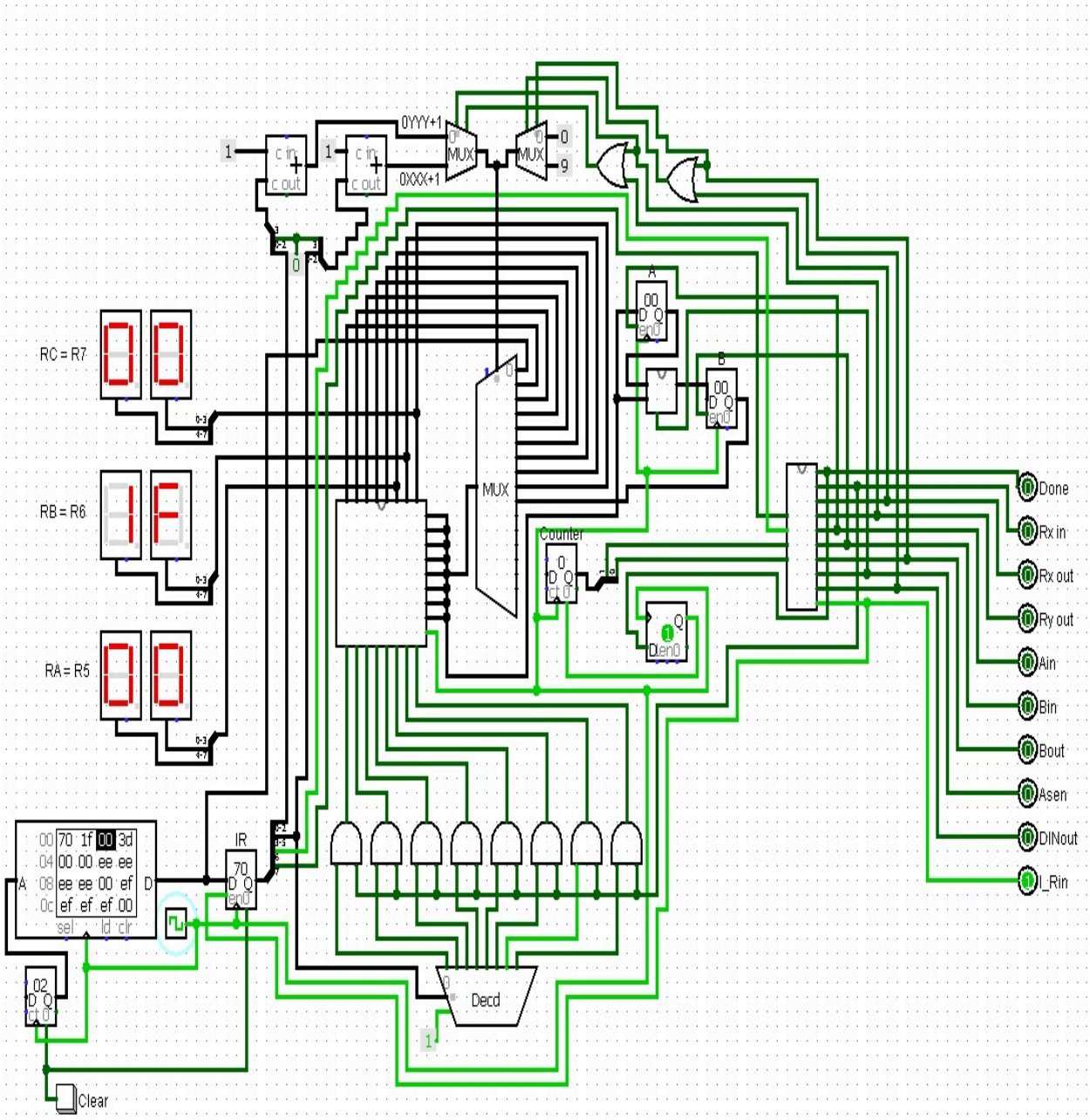


Figure. values for completing operations

First operation: Rb ← 0x1F

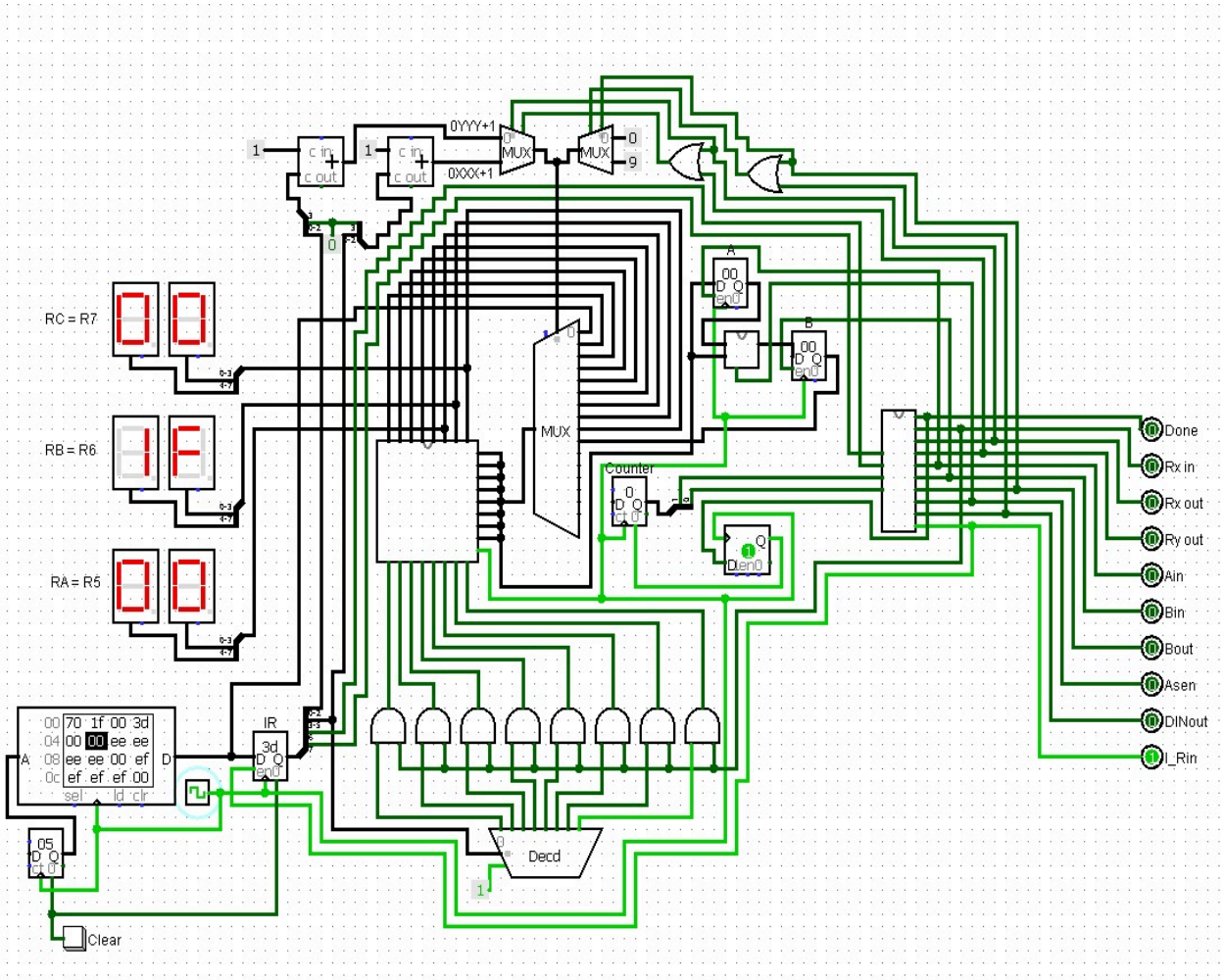
- In the First Operation, we need to fill **Rb** (R6) with **0x1F**.

- We use the **mvi** command. When we completing our First Operation, we type **70** (that is, our IR) in the 1st space of the RAM and type **1f** (that is, our value which we wanted to integrate to Rb) in the 2nd space of the RAM.
- Finally we see that **1f** is integrated to **Rb** register



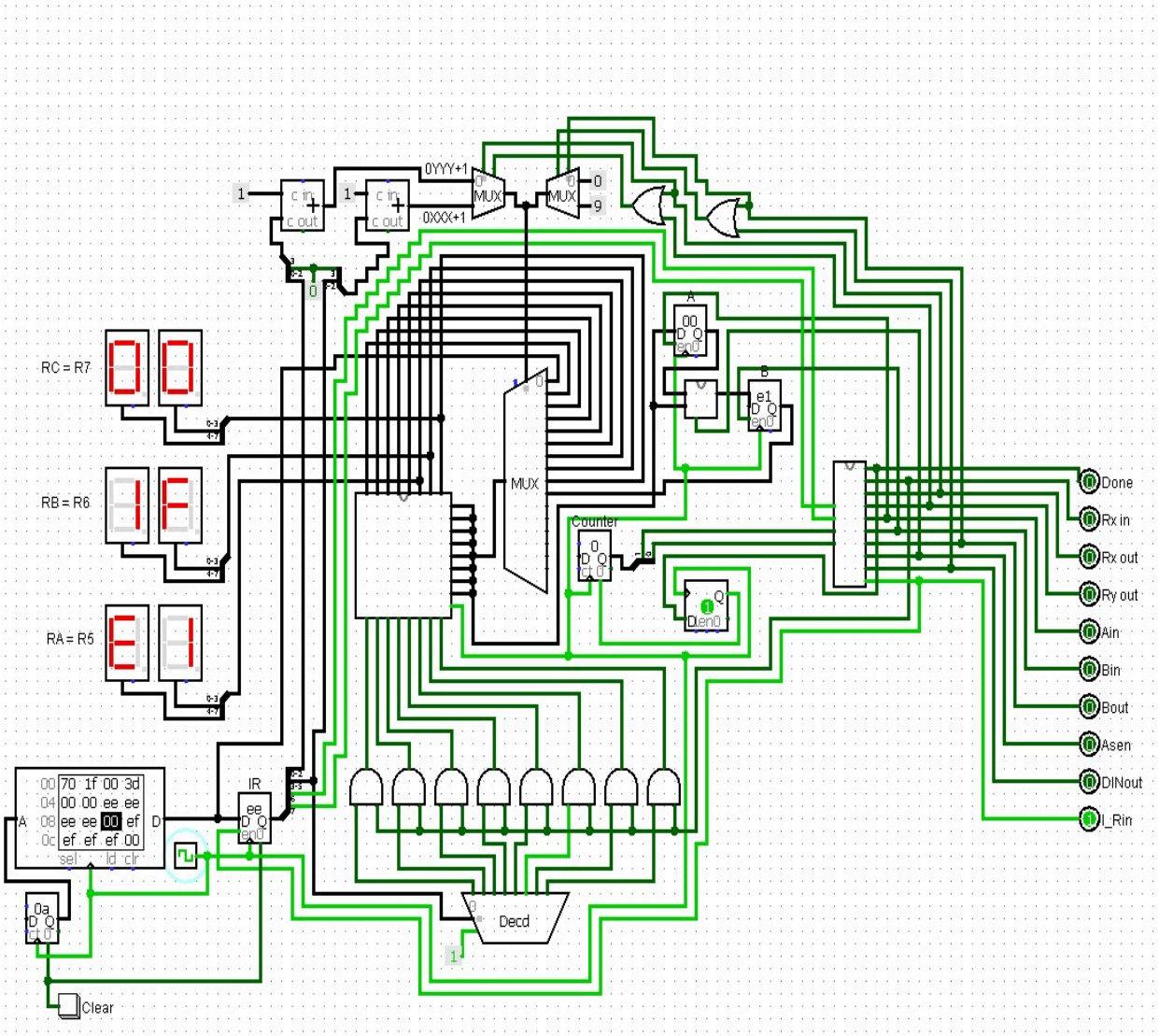
Second operation: Rc ← [Ra]

- In the Second Operation, we need to copy **Ra** (R5) to **Rc** (R7).
- We use the **mv** command. When we completing our Second Operation, we type **3d** (that is, our IR) in the 4th space of the RAM and type **any value** (because we don't get an 8 bit number from RAM) in the 5th space of the RAM.
- Finally we are doing operations in order and when we came to that operation our register both equal to 00. In fact Program copied value from Ra to Rc but we couldn't see them in HEX Led`s because nothing changed. They equal each other in the beginning .
-



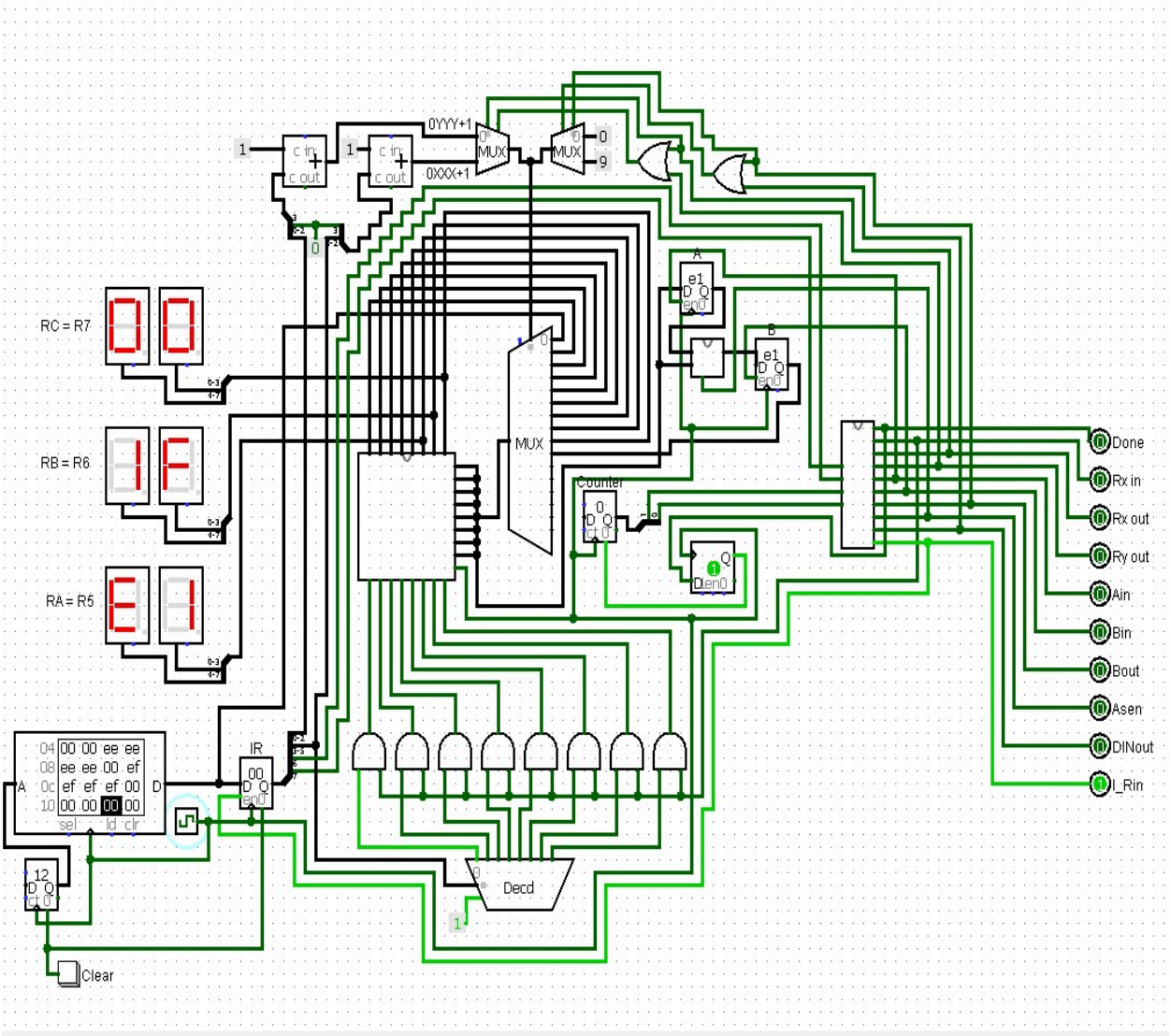
Third operation: Ra ← Ra − [Rb]

- In the Third Operation, we need to subtract **Rb** (R6) from **Ra** (R5).
- We use the **sub** command. When we complete our Third Operation, we type **EE** (that is, our IR) in the 7th space of the RAM and type **any value** (because we don't get an 8 bit number from RAM) in the 8th, 9th, 10th space of the RAM.
- Finally we are doing operations in order and when we came to that operation our Ra equal to 00. So when we did the subtraction calculation we subtract Rb from 00 and we get the result **E1.** We can see it in **Ra**.



Fourth operation: Ra ← Ra − [Rc]

- In the Fourth Operation, we need to subtract **Rc** (R7) from **Ra** (R5).
- We use the **sub** command. When we completing our Fourth Operation, we type **EF** (that is, our IR) in the 12th space of the RAM and type **any value** (because we don't get an 8 bit number from RAM) in the 13th, 14th, 15th space of the RAM.
- Finally when we came to that operation our Rc equal to 00. So when we did the subtraction calculation, we subtract **00** from **E1** and we get the result **E1**.
- We can see in Ra again.



Ethics Statement and Experimental Results

One of the difficulties we had in this experiment was connecting the Control Logic, the register, and the MUX choices because we had to utilize the Control Logic outputs to control the MUX inputs, but it was difficult to comprehend how that worked. To the 16-input MUX, we had to add 10 inputs. Other than that, the items that presented the most challenges to us and which we eventually figured out included control logic, truth tables, ram tables, and the difficulty of pulling connecting wires in a general circuit.

Organization

Baha Karadağlı did report writing, 3.1, 3.2 , 3.3, 2.1

Semih Yılmaz did report writing, 2.2, 2.1, 3.2

Emir Önalan did report writing, researching, 3.2, 2.3