Simple Multi-Cycle Central Processing Unit (CPU) Design

EE 203 Digital Systems Design

Date Assigned:	Dec. 18, 2024
Teaching Assistant:	Ayşenaz Ezgi Ergin
Teaching Assistant:	Bora Kayaoğlu
Instructor:	Dr. Mustafa Tanış
Institution:	MEF University

Authors:
Mustafa Garip, 042201173
Uygar Özbakır, 042302007
Yavuz Selim Kaygusuz, 042202040
Mehmet Fatih Yalçınkaya, 042202004
Team 18.

1 Objective

The objective of this project assignment is to help you design and implement a simple multi-cycle central processing unit (CPU) that has a datapath unit and a hard-wired control unit which can process 8-bit data through various register transfers and Arithmetic Logic Unit (ALU) operations over a common 8-bit bus[1].

2 Work Organization

In this project you shall work in groups of two (three in exceptional circumstances like in case of odd/even enrollments in the class etc). You will have to report what you have gone through/performed just like you did in previous lab assignments. The report and your performance evaluations by TAs are going to be counted as the project's "written" part. Please use Lab Report Preparation Guidelines while writing your report for the project. The "action" part of the project is going to be your demonstration session which is mandatory for the successful completion of your project. The time of this session will be announced soon but most probably be during Finals time.

3 Description and The Design Work

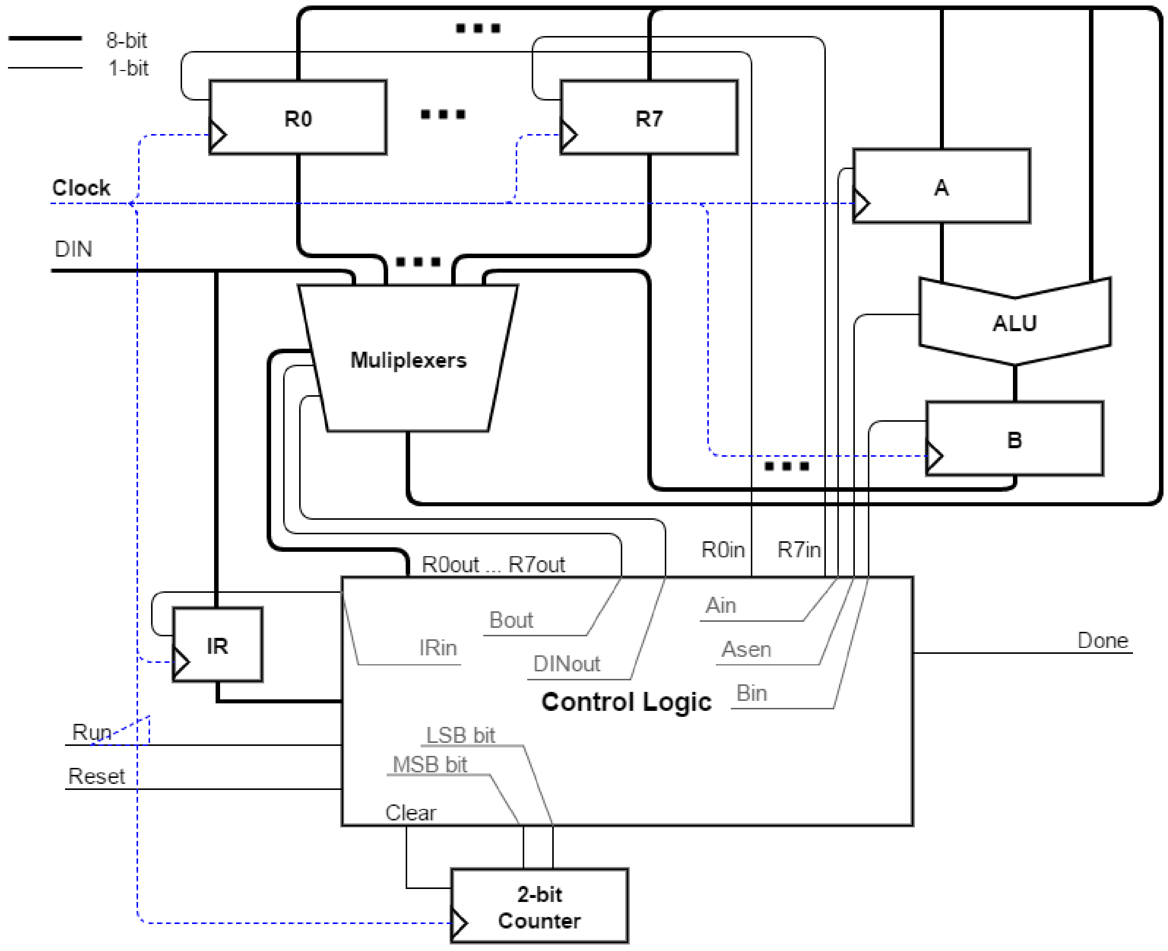
There are three parts to this project. In the first phase, you will have to understand the description of the block diagram given to you. This phase also includes the design of the instruction set architecture (ISA) however, the design of the ISA is usually challenging hence I am going to provide you a simple set of ISA instructions. You can add your own instructions later if you want to. Keep in mind however adding more instructions may have its own challenges such as the size of the bus increase etc. The second phase involves the design and implementation of the major components of the simple multi-cycle CPU such as register file, arithmetic logic unit (ALU) and a counter in our default software Multisim. It is recommended that you use "subcircuits" to make your design look cleaner and easy to deal with. The final phase involves the design of the control logic and its implementation. Here is the formal definition of our simple multi-cycle central processing unit design (CPU).

Let us start with Figure 1 which shows a digital computer system that contains eight 8-bit registers indexed by *R*0 through *R*7, a bank of multiplexer, an adder/subtractor unit, a counter, and a control

[1]This project is majorly inspired by few of the end-term projects of the online courses whose content topics show dramatic similarities to that of EE203. I appreciate all the contributors who has graciously made their course materials online and open-source, in particular special thanks should go to University of California, Riverside, Stanford University and Marquette University.

unit. Data is input to this system via the 8-bit DIN input. This data, provided through DIN input, can be loaded through the 8-bit wide multiplexer unit into the various registers, such as *R*0, . . . , *R*7 and *A*. The multiplexer should also allow data to be transferred from one register to another. In this digital system, the wires can carry 1-bit information except if its drawn bold which would mean that it is a **bus** that can carry 8-bits of information in parallel. In case of addition or subtraction operations, the multiplexer first places one 8-bit number onto the bus wires and load this number into register *A*. At the next clock edge, a

Figure 1: Block diagram of a simple multi-cycle 8-bit processor.



second 8-bit number is placed onto the bus and therefore the adder/subtractor unit executes the required operation, and the result can finally be loaded into register *B* (upon an enable signal for that register to store the appropriate value). Finally, in the next clock edge, the data in register *B* is transferred to one of the other registers as required.

As can be understood, the digital system can perform different operations in each clock cycle, mainly governed by the control signals of the control logic. This unit determines when is a particular data placed onto the bus and also which of the registers should be loaded with the data. For instance, if the control unit asserts the signals *R7out* and *Ain* (which gives logical “1”), then the multiplexer will place the contents of register *R*7 onto the bus and this data will be automatically loaded by the next active clock edge (could be positive or negative) into register *A*.

The set of executable operations are specified in the form of instructions. Table 1 lists the instructions that the CPU is intended to support for this project. The left column shows the name of an instruc- tion and its operand/s. We use indexes $x, y \in \{0, 1, 2, 3, 4, 5, 6, 7\}$ to indicate one of the registers.
When the contents of register *Ry* are loaded into register *Rx*, we use a syntax $Rx \longleftarrow [Ry]$. Hence, the

Table 1: The instructions of the processor

OP code	Operation and Operands	Executed Function
00	*mv Rx, Ry*	*Rx* ⟵ [*Ry*]
01	*mvi Rx, D*	*Rx* ⟵ *D*
10	*add Rx, Ry*	*Rx* ⟵ [*Rx*] + [*Ry*]
11	*sub Rx, Ry*	*Rx* ⟵ [*Rx*] − [*Ry*]

Table 2: The set of control signals asserted for each instruction and time step.

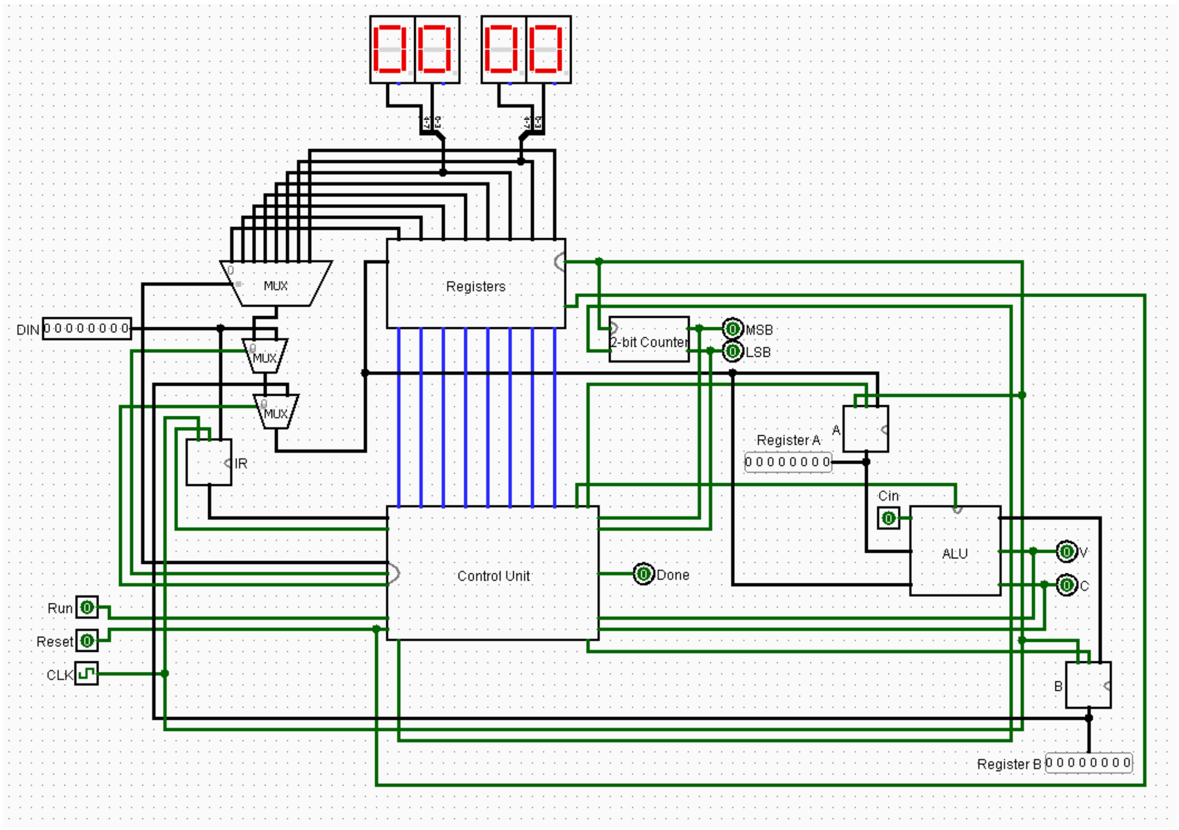
Instruction Type		T_0	T_1	T_2
mv	*IRin*	*Rxin*, *Ryout*, *Done*		
mvi	*IRin*	*Rxin*, *DINout*, *Done*		
add	*IRin*	*Ain*, *Rxout*	*Bin*, *Ryout*	*Rxin*, *Bout*, *Done*
sub	*IRin*	*Ain*, *Rxout*	*Bin*, *Ryout*, *Asen*	*Rxin*, *Bout*, *Done*

right column of the table shows the operation corresponding to the given instruction. The *mv* (move) instruction allows data to be copied from one register to another. For the *mvi* (move immediate) instruction, the expression *Rx* ⟵ *D* indicates that the 8-bit constant *D* is loaded into register *Rx*. There are two other operations *add* and *sub* to represent binary addition and subtraction as indicated in the same table.

The data input DIN can either be an 8-bit instruction or an 8-bit signed number. In other words, you can think of a memory pre-written with 8-bit words some of them are instructions and sum of them are data itself (signed numbers). Each instruction can be loaded and stored in the instruction register (*IR*) using the 8-bit format "IIXXXYYY", where II represents the instruction type or so called OP code (2 bits, denoted as OP[0] and OP[1]), XXX gives the *Rx* register address (using simple binary addressing $(x)_2$), and YYY gives the *Ry* register address ($((y)_2$). Since DIN also provides the instructions for the processor, *IR* uses all eight bits of the DIN input, as indicated in Figure 1. The control unit has the corresponding decoding engine that decodes the instruction and produces the right set of control signals. For the *mvi* instruction, the YYY field bears no meaning, and the immediate data *D* has to be supplied on the DIN input after the *mvi* instruction word is loaded into *IR* in the previous active clock edge. Some instructions, such as an *add* or *sub*, take more than one clock cycle to fully complete, because multiple transfers have to be performed across the bus. Please look at Figure 1 and observe that particular phenomenon. The control unit uses the two-bit counter to enable multi-cycle operations to "step through" such types of instructions. The CPU starts executing the instruction on the DIN input when the *Run* signal is asserted and the CPU asserts the *Done* output when the instruction is finished executing. Table 2 indicates the control signals that may be asserted in each time step to implement the instructions in Table 1. Note that since the only control signal that needs to be asserted in time step 0 is *IRin*, in which the control unit shall know the instruction and hence the appropriate set of control signals to generate for the next few clock cycles, we start counting right after loading the instruction to IR (and hence T_0). Once the datapath components are implemented using the appropriate blocks/gates/wires in Multisim, your design task boils down to handling the following three work packages.

2.1 **(30pts)** Draw the datapath and corresponding registers/counters and ALU as shown in Figure 1. You are more than welcome to use your previous ALU design (Lab 2) in your circuit.

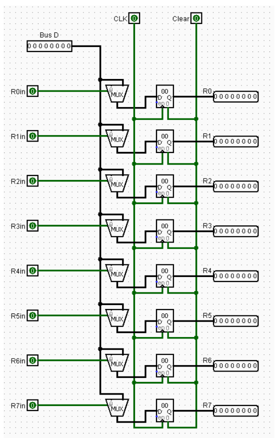
Here's the our multi cycle CPU:



Quick Explanation:

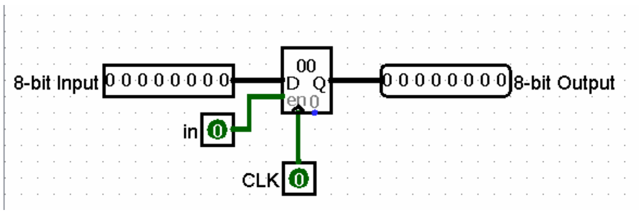
The datapath was created by Mustafa Garip and Uygar Özbakır. The data processes 8-bit instructions and data. The registers block stores values in eight 8-bit registers. The control unit generates control signals based on the opcode in the Instruction register (IR) and the current state of the CPU from the 2-bit counter that tracks the cycles (T0, T1, T2). The ALU performs arithmetic operations (add & sub) using inputs from Register A and Register B (from MUX, not directly). The MUXes select which data or control signals (Registers (already stored), B (operation output) or DIN (data input)) will be sent to the cycle based on the control unit's decision.

Registers:



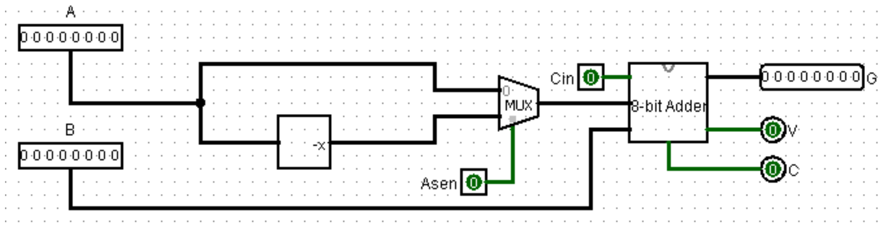
The registers subcircuit was created by Mustafa Garip. There are eight 8-bit registers in it. And the Bus D is written in which register the control unit decided.

Mini Registers (A, B and IR):



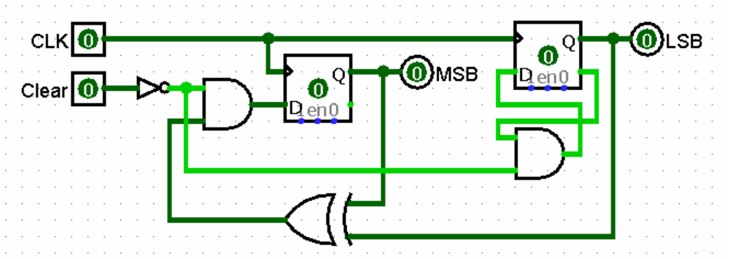
The mini register was created by Mustafa Garip due to organizing the circuit. The circuit makes the datapath clear and organized. There is no extra feature in a mini register. It is exactly a stock 8-bit register. Register A, Register B and Register Instruction use this element.

ALU:



The ALU was created by Mustafa Garip. We used previous Lab 7’s Arithmetical Unit to build this ALU. The Lab 2 ALU would have to be edited a lot, so we decided to use this circuit. The ALU has the ability to add or subtract 2 variables via Asen (Subtraction enabled) bool. If it is 1, The negator takes the A’s 2’s complement and the adder adds it to B.

2-bit Counter:



The 2-bit Counter was created by Mustafa Garip. The circuit counts the clock's rising edges, until clear takes 1. The operation helps the control unit to understand which phase the CPU is in. Master D flipflop gives us the least significant bit of the 2-bit. We used an XOR gate to convert the D flipflop to a JK flipflop that. The slave JK- flipflop gets his master’s output and gives us the most significant bit of the 2–bit. The control unit gets LSB and MSB from 2-bit Counter.

The control unit will be held on 2.2.

2.2 **(20pts)** Design the control logic unit in Figure 1 by drawing the appropriate truth tables and performing simplifications. In order to ease your work, we provided Table 3, at the end of the document for you to fill in.

The control unit was created by Mehmet Fatih Yalçınkaya and Yavuz Kaygusuz.

We have begun constructing the truth table for our logic inputs and outputs as a foundation for developing the Control Logic Component, as outlined below.

OP[0]	OP[1]	T_0	T_1	T_2	Done	Rxin	Rxout	Ryout	Ain	Bin	Bout	Asen	DINout
0	0	1	0	0	1	1	0	1	0	0	0	0	0
0	1	1	0	0	1	1	0	0	0	0	0	0	1
1	0	1	0	0	0	0	1	0	1	0	0	0	0
1	0	0	1	0	0	0	0	1	0	1	0	0	0
1	0	0	0	1	1	1	0	0	0	0	1	0	0
1	1	1	0	0	0	0	1	0	1	0	0	0	0
1	1	0	1	0	0	0	0	1	0	1	0	1	0
1	1	0	0	1	1	1	0	0	0	0	1	0	0

Done - Rxin

When T0 is set to 1, the "Done" and "Rxin" signals are high for OP values "00" and "01." This leads to the equation "T0 and OP[0]’." Similarly, when T2 is set to 1, the "Done" and "Rxin" signals are high for OP values "10" and "11," resulting in the equation "T2 and OP[0]." By combining these two equations using an OR gate, the final expression is derived.

- ❖ Done = OP[0]′ .T0 + OP[0].T2
- ❖ Rxin = OP[0]′ .T0 + OP[0].T2

Ryout

When the T0 signal is active, the Ryout signal becomes high for the OP value “00”, which can be expressed by the equation T0 and OP[0]′ and OP[1]′. However, the Ryout signal does not go high for the OP value "01." On the other hand, when the T1 signal is active, the Ryout signal is high for OP values "10" and "11," represented by the equation T1 and OP[0]. By combining these equations using an OR gate, the complete expression can be derived.

- ❖ Ryout = OP[0]′ .OP[1]′ .T0 + OP[0].T1

Rxout - Ain

An examination of the input combinations that result in a high signal for both the Rxout and Ain outputs reveals that, in contrast to other input combinations, the OP[0] and T0 values are both set to 1. Therefore, it can be inferred that the equation "OP[0] and T0" governs the generation of the Rxout and Ain outputs.

- ❖ Rxout = OP[0].T0
- ❖ Ain = OP[0].T0

Bin

It can be observed that, in contrast to the other input combinations, the "OP[0] and T1" values are 1 for the inputs when the "Bin" output produces a strong signal. This leads to the assumption that the output "Bin" is dependent on an equation of the type "OP[0] and T1".

- ❖ Bin = OP[0].T1

Bout

The "OP[0] and T2" values are 1, in contrast to the other input combinations, when the inputs where the "Bout" output produces a strong signal are checked. Because of this, it is presumably that the output "Bout" is dependent on an equation of the type "OP[0] and T2".

- ❖ Bout = OP[0].T2

Asen

Only the high value in a circumstance is offered by the "Asen" value. In this instance, the OP[0] and OP[1] are both 1, and the operation occurs at time T1. It is therefore assumed that an equation of the type "OP[0] and OP[1] and T1" governs the output "Asen".

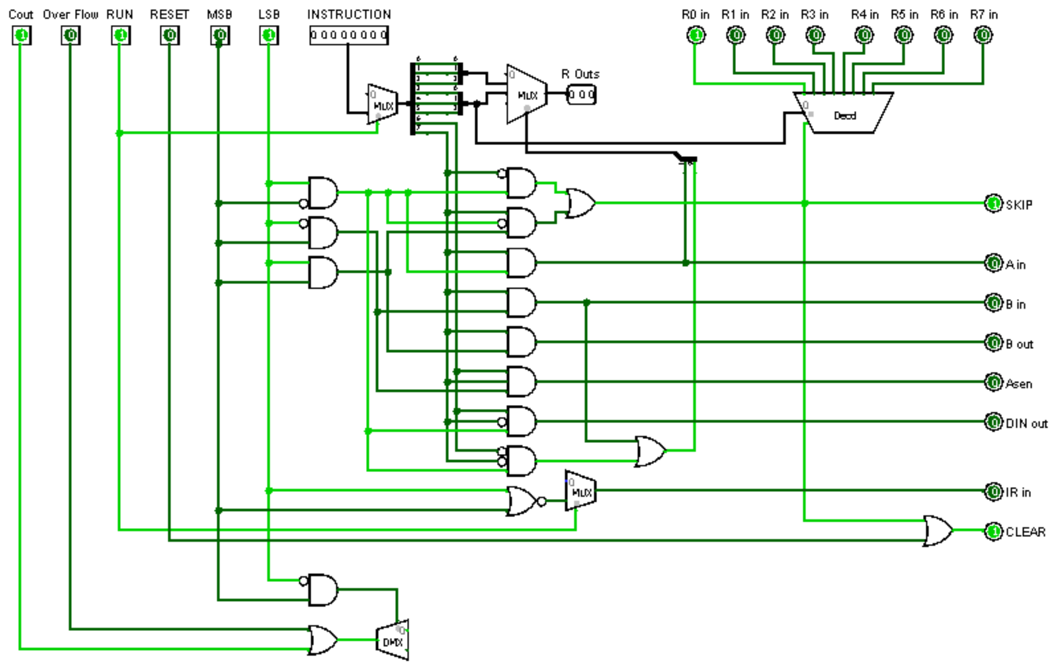
- ❖ Asen = OP[0].OP[1].T1

DINout

Only the high value in a circumstance is provided by the "DINout" value. In this instance, the OP[0] is equal to 0, the OP[1] is equal to 1, and the operation occurs at time T0. It is therefore assumed that an equation of the type "OP[0]′ and OP[1] and T0" governs the output "DINout".

- ❖ DINout = OP[0]′ .OP[1].T0

Here's the control unit:

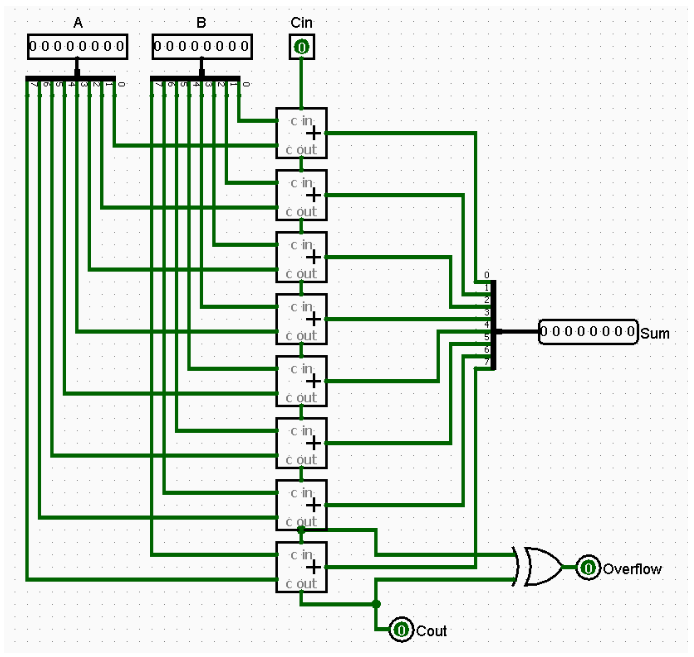


This circuit features five inputs: Instruction Code, Least Significant Bit (LSB), Most Significant Bit (MSB), Run, and Reset. The Instruction Code is an 8-bit value, with the first two bits specifying the operation code (OP Code) and the subsequent three bits identifying individual registers. The LSB and MSB values of the counter are mapped to T0, T1, and T2, representing the current time. The Run input initiates circuit operation, while the Reset input restores the Control Logic to its initial state, as shown in Figure 7.

The Control Logic utilizes "Rin" inputs and a decoder to select the "Rxin" input. The outputs "Rxout" and "Ryout" are merged into a single "Rout" output, managed by a multiplexer based on the "Rxout" and "Ryout" inputs. The instruction register is controlled by the "IRin" output, which signals high only when both the LSB and MSB are "00" and the Run input is active. Additionally, the circuit includes a Clear output, which becomes high either when the "Done" signal indicates an instruction has been completed or when the circuit is reset.

2.3 **(Bonus+5pts)** As an additional step, you may like to add the overflow and carry-out outputs of the ALU as two additional inputs for the Control unit while making decisions.

Here's the 8-bit Adder with overflow and carry-out outputs on Logisim:



The 8-bit adder was created by Mustafa Garip. We used previous Lab 7's 4-bit adder to build this 8-bit adder. The Cout can be easily obtained from the last adder in the chain. We used an XOR operation between the MSB's Cin and Cout to detect overflow, enabling the control unit to make accurate decisions.

4 Demonstration

In this project, you are going to have to demonstrate your operational CPU in our prearranged session in order to get full credit.

In demonstration session, for automating the following instruction execution and data processing, it might be advisable to use a RAM, load it with the appropriate set of data (both instructions as well as data) and use the counter to jump to the appropriate location to load information on DIN input of the CPU. Although this is not required, it can certainly expedite your project demonstration. **I recommend you look at built-in RAM capabilities of Multisim and get yourself familiar with it.**

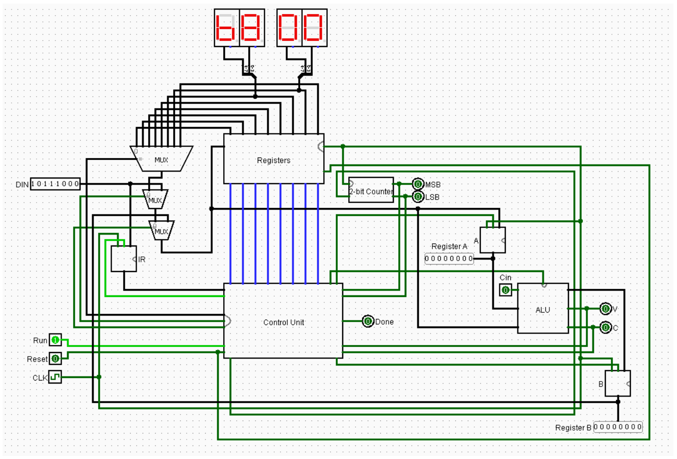
Note: Demonstration section of the paper was filled by teamwork, by Mustafa Garip (Logisim), Uygar Özbakır (Organization), Fatih Yalçınkaya (Writing), Yavuz Selim Kaygusuz (Writing).

3.1 **(30pts)** Please demonstrate the operation of your simple multi-cycle computer by executing the following instructions. You need to find the appropriate set of *DIN* values to be able to use your computer. It is advised that you find the sequence of machine code that generates these instructions/data and load it up onto a RAM.

$$mvi\ R5,\ \text{0xB8} \mid R5 \leftarrow \text{0xB8}, \tag{1}$$
$$mv\ R6,\ R5 \mid R6 \leftarrow [R5], \tag{2}$$
$$mvi\ R5,\ \text{0x6F} \mid R5 \leftarrow \text{0x6F}, \tag{3}$$
$$add\ R6,\ R5 \mid R6 \leftarrow R6 + [R5], \tag{4}$$
$$add\ R5,\ R5 \mid R5 \leftarrow R5 + [R5], \tag{5}$$
$$sub\ R5,\ R6 \mid R5 \leftarrow R5 - [R6] \tag{6}$$

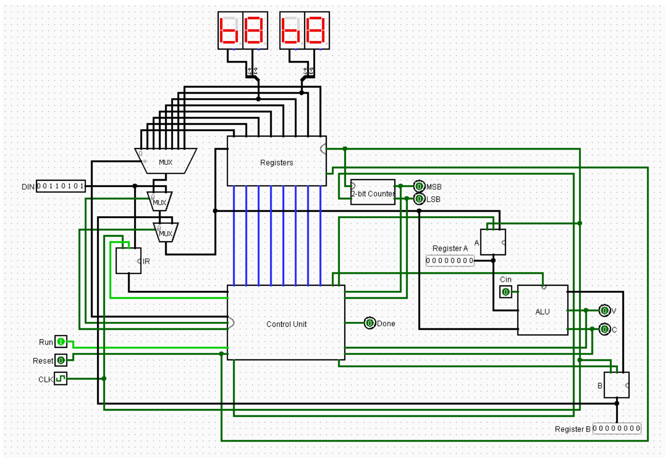
Step-by-Step Explanation with images:

Operation 1 (*mvi R*5, 0xB8 | *R*5 ← 0xB8):



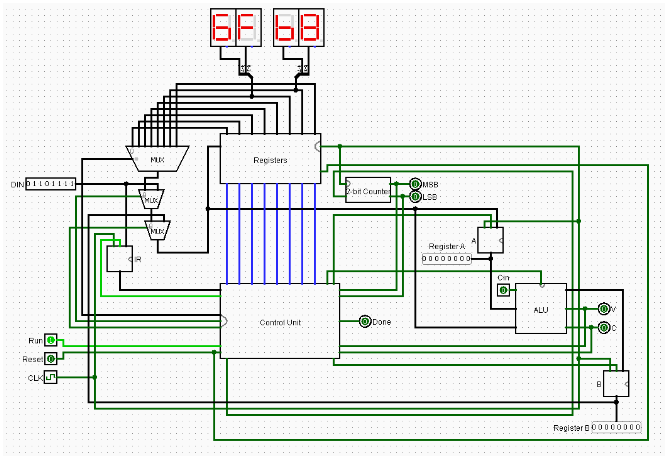
DIN (Instruction): 01 (mvi) 101 (R5) 000 (void) **DIN (Data):** 1011 1000

Operation 2 (*mv R6, R5* | *R6* ← [*R5*]):



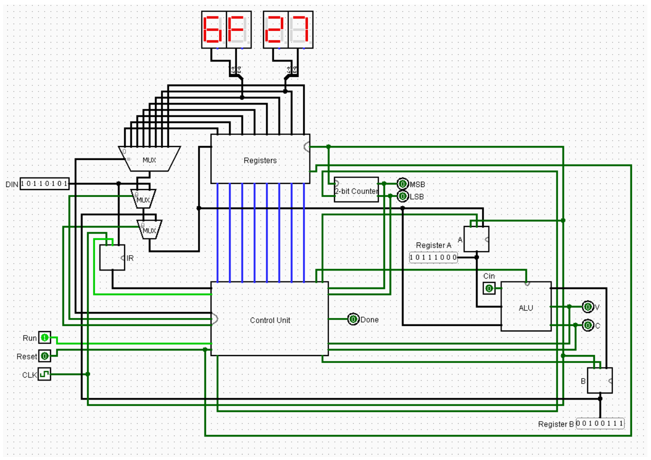
DIN (Instruction): 00(mv) 110(R6) 101(R5)

Operation 3 (*mvi R5*, 0x6F | *R5* ← 0x6F):



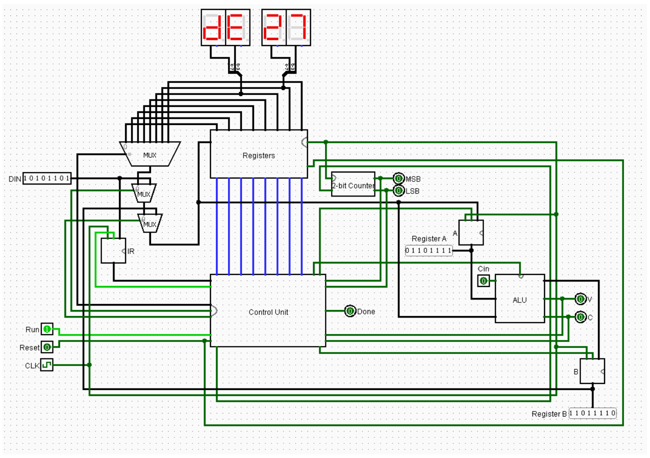
DIN (Instruction): 01(mvi) 101(R5) 000(void) **DIN(Data):** 0110 1111

Operation 4 (*add R6, R5* | *R6* ← *R6* + [*R5*]):



DIN (Instruction): 10(add) 110(R6) 101(R5)

<u>Operation 5 (*add R*5, *R*5 | *R*5 ← *R*5 + [*R*5]):</u>

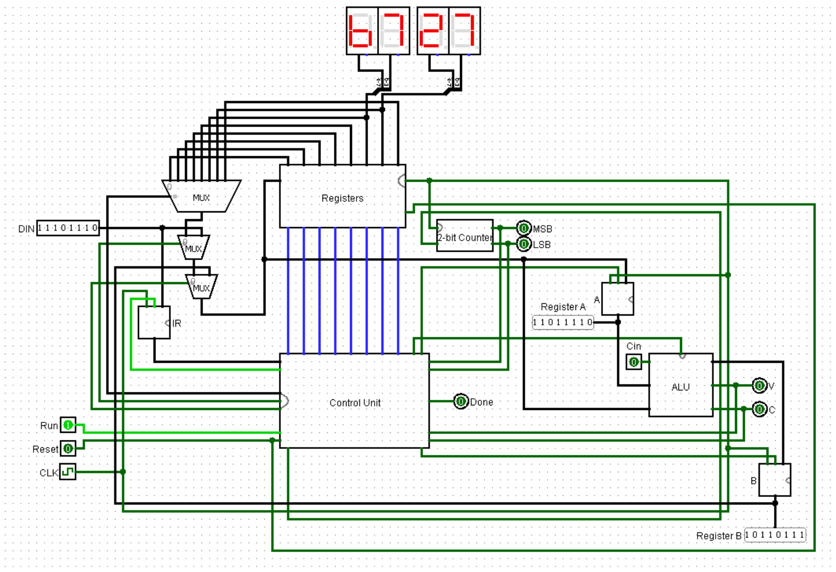


<u>Instruction:</u> 10(add) 101(R5) 101(R5)

The last operation will be held on 3.2, FYI.

3.2 **(10pts)** What are the contents of *R*5 and *R*6 at the end of executing above instructions? Use hex displays and LEDs to demonstrate the functionality.

<u>Here's the last operation's (Operation 6, sub R5, R6 | R5 ← R5 − [R6]) output on Logisim:</u>



<u>R5:</u> 0xB7, 1011 0111, -73. **<u>R6:</u>** 0xDE, 1101 1110, -34.

3.3 **(10pts)** We do not have a special instruction to execute other operations that may come handy. However, it might be possible to execute those operations using multiple registers and the available instruction types. Please generate the appropriate instructions to perform the following steps and comment on the function we are trying to make our CPU execute. Suppose we have three registers *Ra, Rb* and *Rc*. We would like to manipulate the contents of *Rb* and transfer it to *Ra*.

- *Rb* ← 0x1F
- *Rc* ← [*Ra*]
- *Ra* ← *Ra* − [*Rb*]
- *Ra* ← *Ra* − [*Rc*]

- *Ra* ⟵ *Ra* − 1

What's the final content of register *Ra*? How many clock cycles it takes to execute this operation? Note that It would have been much more convenient to add this new instruction to our ISA to run the corresponding operation faster!

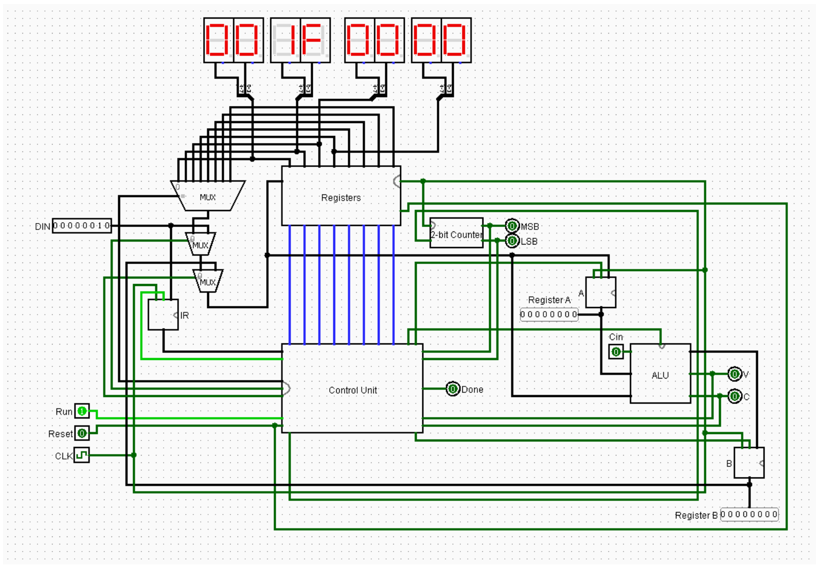
Let a = 0, b = 1 and c = 2.

Unfortunately, we were not able to add a new instruction due to lack of time and overwork in the project week, so we will record the 1 into a register, then we will operate the subtraction (last op.). We also will handle the 1 as both 0x01 and 0xFF on R3.

So, we will work on R0, R1, R2 and R3 in this CPU for this section.

Step-by-Step Explanation with Images:

Operation 1 (Rb ⟵ 0x1F):



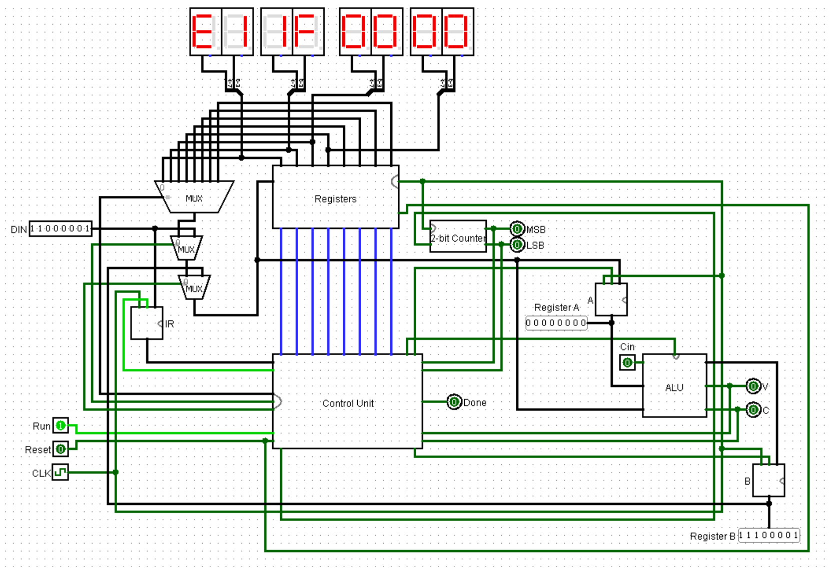
T0 => 1 Clock Cycle

DIN (Instruction): 01 (mvi) 001 (Rb) 000 (void) **DIN (Data):** 0001 1111

Operation 2 (Rc ⟵ [Ra]):
T0 => 1 Clock Cycle

Ra and Rc already are 0x00, so the outputs is the same as operation 1's outputs.
DIN (Instruction): 00 (mv) 010 (Rc) 000 (Ra)

Operation 3 (Ra ⟵ Ra − [Rb]):



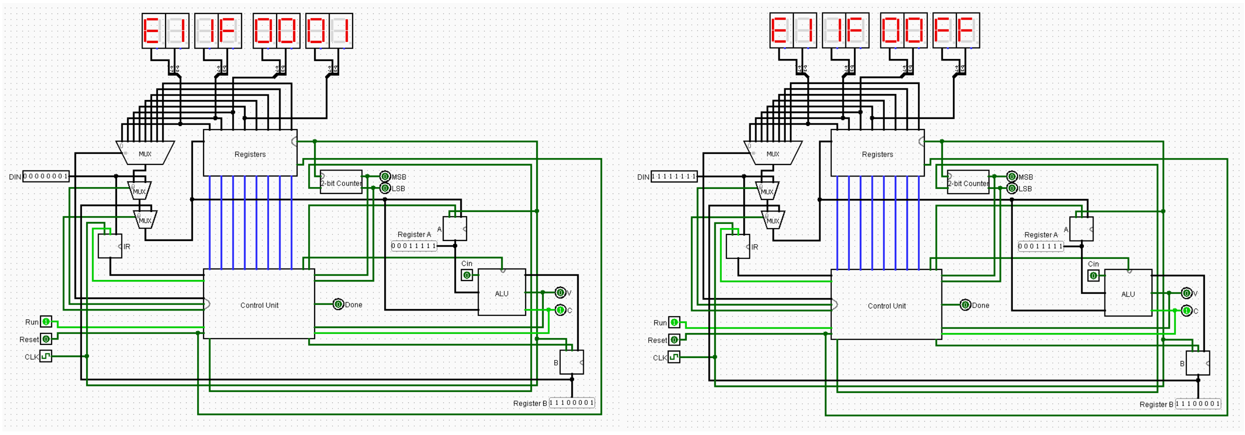
T0, T1, T2 => 3 Clock Cycle

Output: 0xE1 = -31.
DIN (Instruction): 11(sub) 000 (Ra) 001(Rb)

Operation 4 (Ra ⟵ Ra − [Rc]):
T0, T1, T2 => 3 Clock Cycle

We know that 0xE1 - 0x00 (Decimal, -31 - 0) equals 0xE1 (Decimal, -31). So the outputs is the same as operation 3's outputs.
DIN (Instruction): 11(sub) 000 (Ra) 010 (Rc)

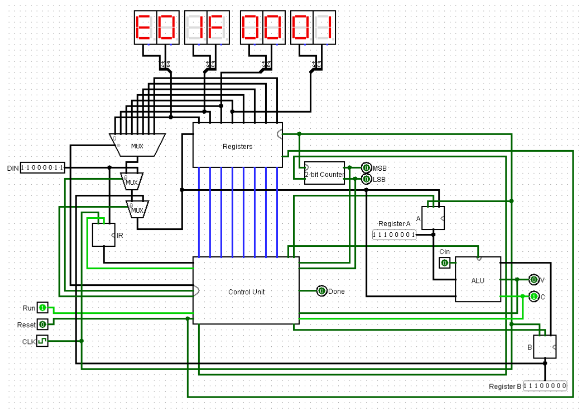
Operation 5 (R3 ⟵ 0x01 (Left) and R3 ⟵ 0xFF (Right), Optional):



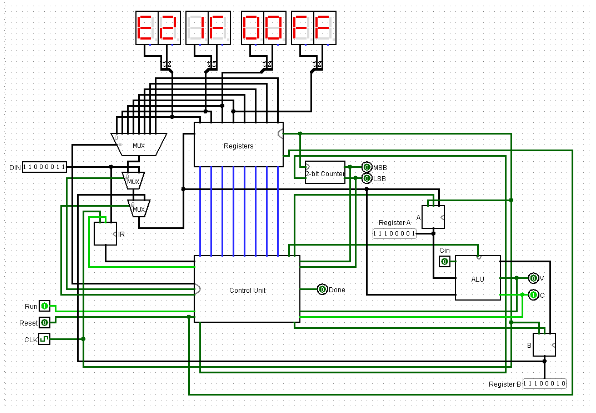
T0 => 1 Clock Cycle (both)

DIN (Instruction) (R3 ⟵ 0x01): 01 (mvi) 011 (R3) 000 (void) **DIN (Data):** 0000 0001
DIN (Instruction) (R3 ⟵ 0xFF): 01 (mvi) 011 (R3) 000 (void) **DIN (Data):** 1111 1111

Operation 6 (Ra ⟵ Ra − 1(R3), 0x01 Left, 0xFF Right):



0x01's Output (Ra): 0xE0, 1110 0000, -32.



0xFF's Output (Ra): 0xE2, 1110 0010, -30.

T0, T1, T2 => 3 Clock Cycle (both)

DIN (Instruction) (Ra ⟵ Ra − 0x01): 11 (sub) 000 (Ra) 011 (R3, 0x01 = 1)
DIN (Instruction) (Ra ⟵ Ra − 0xFF): 11 (sub) 000 (Ra) 011 (R3, 0xFF = 1)

5 Organization and Ethics Statement

In your project, you are advised to take into account the following seriously.

- You are allowed to work in groups of three or four .
- You must provide information about the distribution of labour by specifying which group member did what.
- You must provide electronic signatures to attest the completion of the project work and agree to the distribution of labor so claimed.
- You will be asked detailed questions during an online demonstration session to check whether you really have done the project. So you must show up in the demonstration session in order for your project to be evaluated.
- **You cannot copy from old projects (last year etc. from senior students). We have advanced tools to spot any overlap with previous works. In case of an extreme similarity, you will receive zero credit without any prior notice and your case will be delegated to Ethics commission for further sanctions.**
- Having completed the circuit design and requirements of the this document does not mean you will receive full credit unless you successfully present your work.