თბილისის თავისუფალი უნივერსიტეტი

კომპიუტერული მეცნიერებების, მათემატიკისა და ინჟინერიის სკოლა (MACS[E])

ელექტრო და კომპიუტერული ინჟინერიის პროგრამა

ვადაქარია თეკლა
ფრანგულაშვილი საბა

ჯუნიორ პროექტი

8 Bit Computer

ხელმძღვანელი:
ვარდიაშვილი გუგა
სულაბერიძე ზვიად

თბილისი
2025

ანოტაცია

პროექტის მიზანი იყო სრულად გაგვეგო, როგორ მუშაობს კომპიუტერი, მისი აწყობით ძირითადი ლოგიკური კომპონენტებიდან. ეს იყო სასწავლო პროექტი, რომელიც შექმნილია იმისათვის, რომ თვალსაჩინოდ წარმოვაჩინოთ და პრაქტიკულად განვახორციელოთ თეორიული ცოდნა, რომელიც მივიღეთ საგანში *ციფრული წრედები*. გვინდოდა შეგვექმნა პროექტი, რომელიც ნათლად აჩვენებდა როგორ მუშაობს კომპიუტერის თითოეული ნაწილი.

სანამ ფიზიკურად ავაწყობდით სისტემას, პირველ რიგში გავაკეთეთ მისი სიმულაცია პროგრამაში Logisim Evolution, რაც საშუალებას გვაძლევდა შეგვემოწმებიდა თითოეული მოდულის მუშაობა. ლოგიკურიდან ბრედბორდებზე გადასვლა მარტივი იყო, მცირე შეცდომები კი მარტივად აღმოვფხვერით.

პროექტის მიმდინარეობისას ორი ძირითადი პრობლემა შეგვექმნა: EEPROM-ები და PCB-ები. საბოლოო ვადის მოახლოებისას შემთხვევით დავამოკლეთ უკვე დაპროგრამირებული EEPROM, ხოლო ახალი ჩიპების შეკვეთა დროში ვერ მოვასწარით, ისინი არც საქართველოში იყო ხელმისაწვდომი. შედეგად, პროექტის საბოლოო ვერსიაში EEPROM-ს გარეშე მუშაობს მხოლოდ. ასევე, ამის გამო, ჩვენი Output მოდული შედეგს აჩვენებს ორობით ფორმატში და არა ათობითში, როგორც დაგეგმილი იყო.

მეორე პრობლემა უკავშირდებოდა PCB დიზაინს. ყველა ნაწილისთვის შევქმენით PCB, მაგრამ RAM-ის დაფა არ მუშაობდა სწორად. მიუხედავად იმისა, რომ რამდენჯერმე გადავამოწმეთ სქემა, შეცდომის პოვნა ველარ მოვასწარით და გადავწყვიტეთ გაგვეგრძელებინა მუშაობა ბრედბორდებზე.

საბოლოოდ, შევძელით აგვეშენებინა მოქმედი კომპიუტერი, რომელსაც მომხმარებელი ხელით პროგრამირებს. მომავალში ვგეგმავთ ახალი EEPROM-ების შეკვეთას და სრული წრედის დასრულებას. მიუხედავად სირთულეებისა, ძალიან ვამაყობთ ჩვენი პროექტით და იმ ცოდნით, რაც ამ გამოცდილებით მივიღეთ.

Annotation

The goal of this project was to fully understand how computers work by building one from basic logic components. It was designed as an educational project to visualize and apply the theoretical concepts we learned in the university course *Digital Circuits*. We wanted to create a project that would clearly demonstrate how each part of a computer operates physically, beyond the theory taught in class.

Before assembling the hardware, we first simulated the entire system using Logisim Evolution, which allowed us to test and verify the logical behavior of each module. The transition from simulation to breadboards was smooth, and although we encountered minor bugs in the circuits, they were easy to identify and fix.

We faced two main challenges during the project: EEPROMs and PCBs. Close to the project deadline, we accidentally shorted a working and programmed EEPROM. Unfortunately, we could not replace it in time because the specific chips were unavailable in Georgia and ordering new ones would have taken too long. As a result, the final version of our computer could not include the EEPROM module. Because of this, our computer can only be programmed by hand and our output display currently shows data in binary format instead of base 10, which would normally be decoded by the EEPROM.

The second issue was related to PCB design. We created and tested PCBs for every module, but the RAM PCB did not function correctly. Despite repeated checks, we could not find the exact mistake in the schematic before the deadline. Therefore, we continued working with breadboards instead.

In the end, we successfully built a working computer that can be programmed manually by the user. In the future, we plan to order new EEPROMs and complete the full implementation. Despite the challenges, we are very proud of the result and the deep understanding we gained from this experience.

Table of Contents

List of tables, graphs, formulas and images

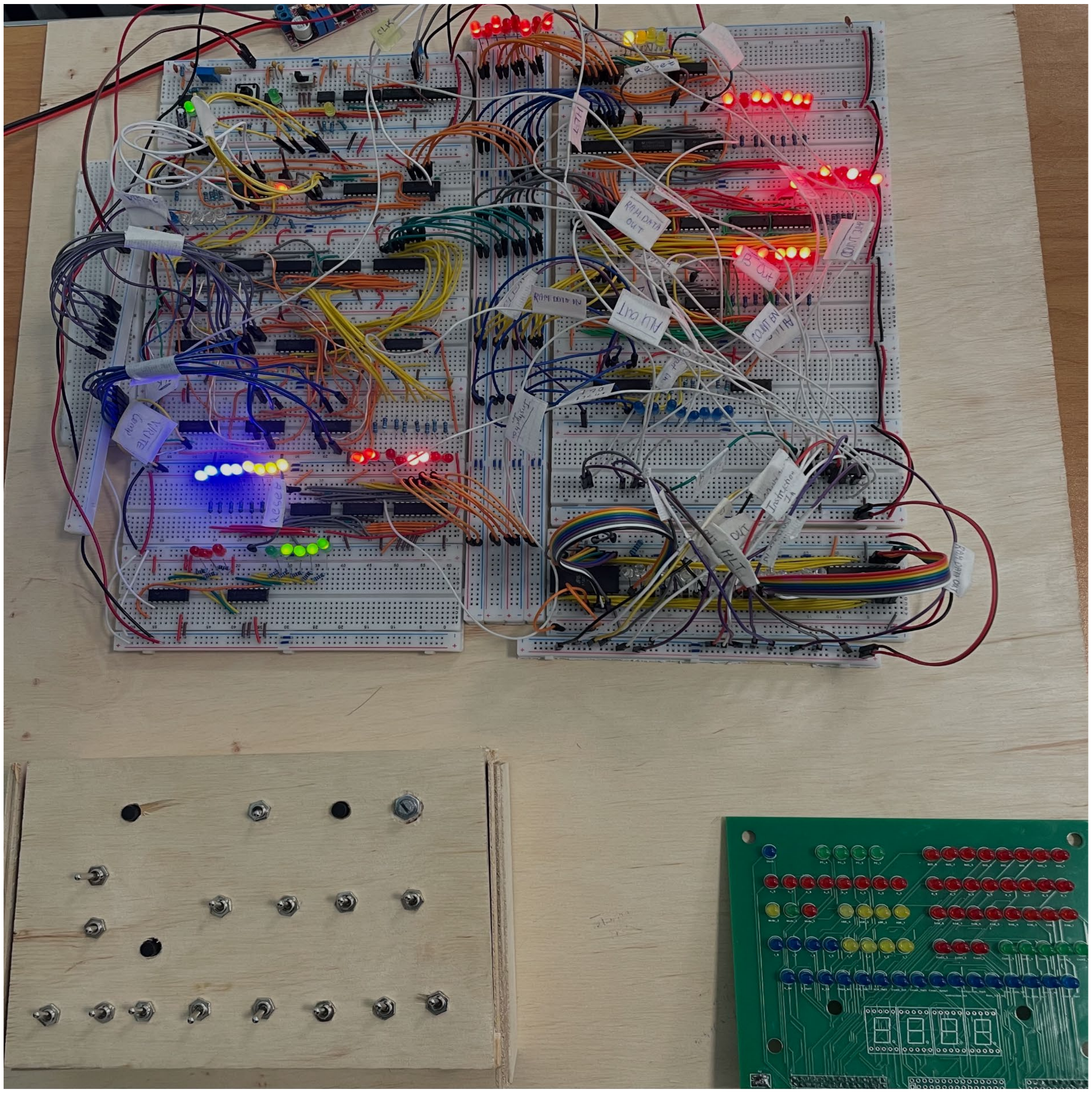


Image 1. Our Breadboard Circuit of the Computer

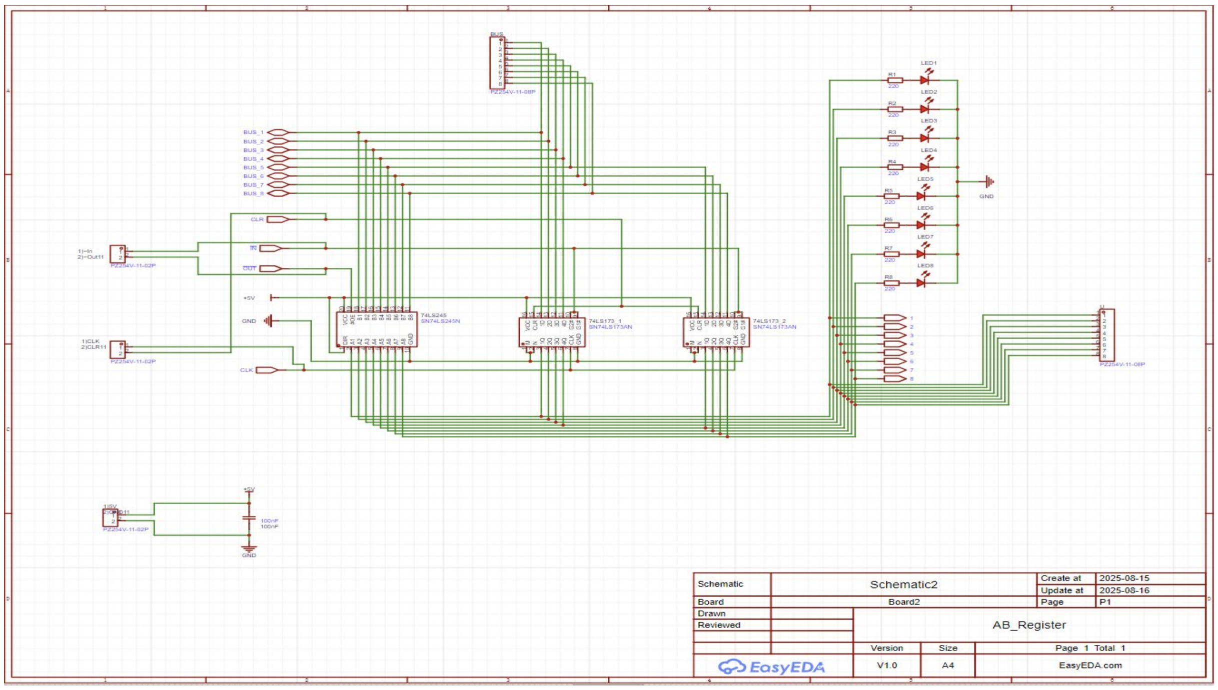


Image 2. A/B Registers Schematic in EasyEDA Pro

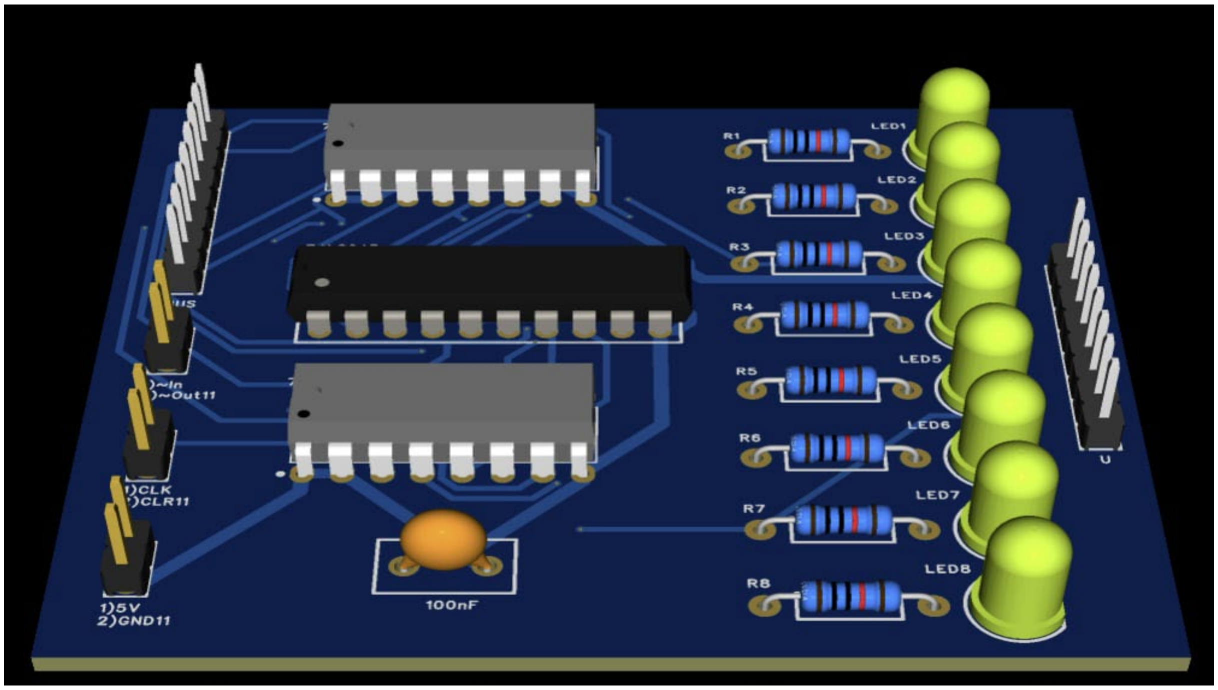


Image 3. A/B Registers PCB in EasyEDA Pro

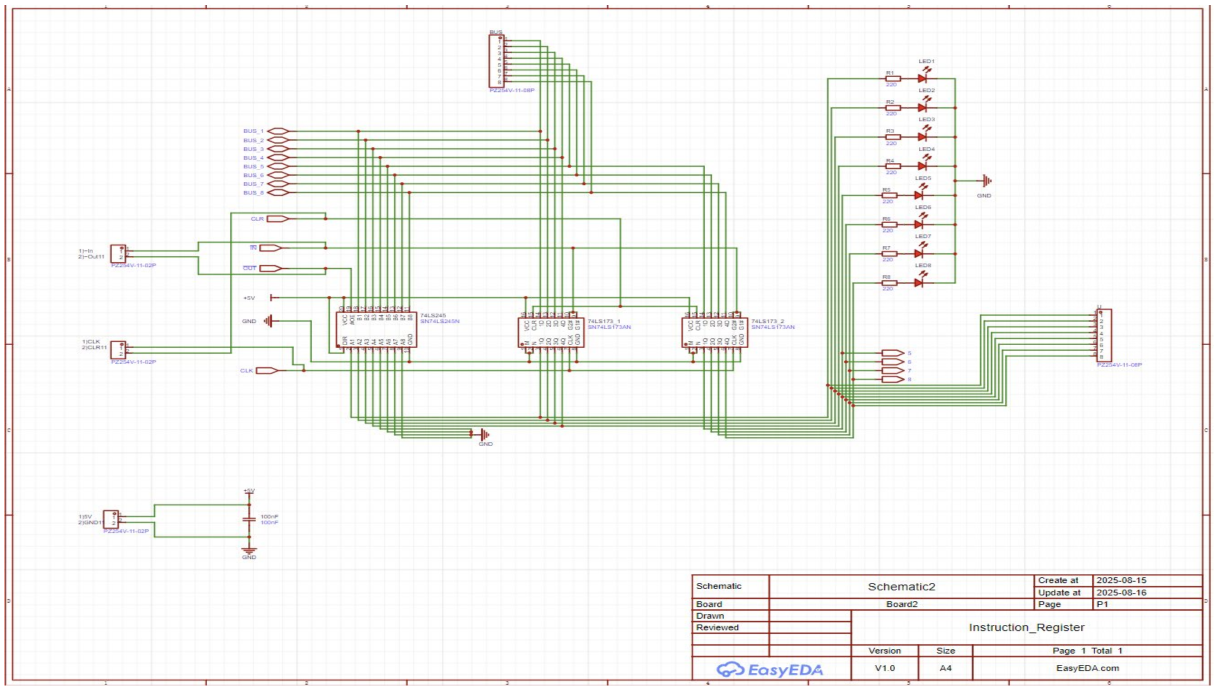


Image4. Instruction Register Schematic in EasyEDA Pro

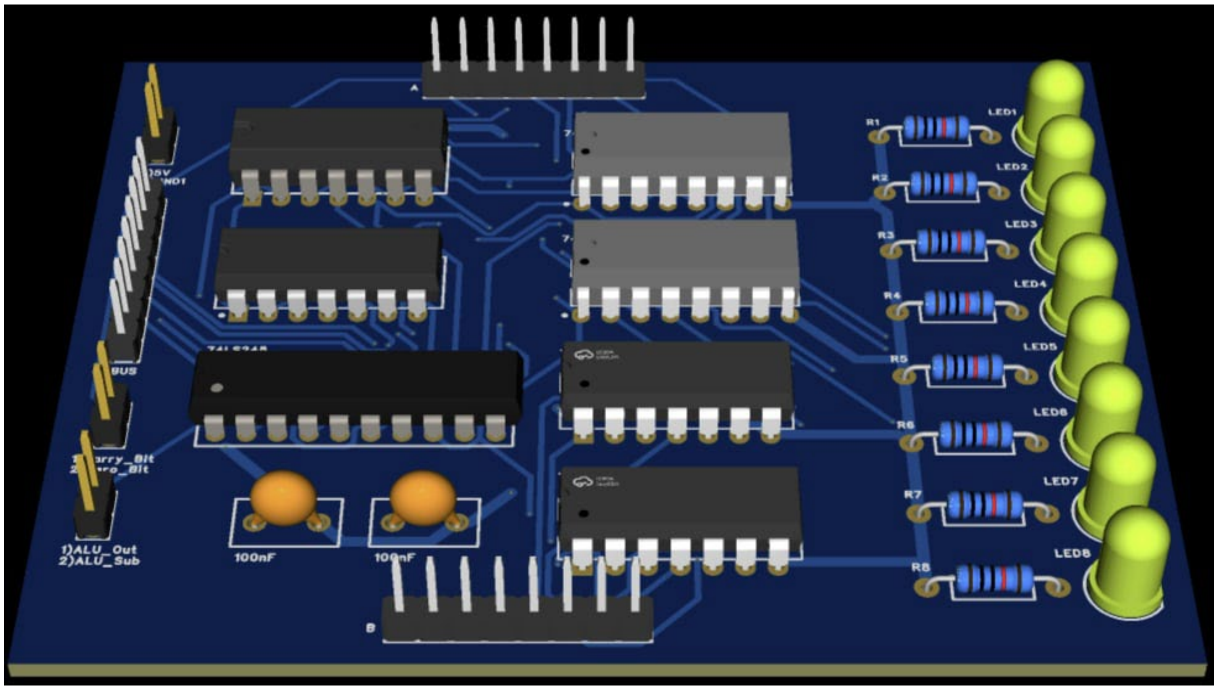


Image5. Instruction Register PCB in EasyEDA Pro

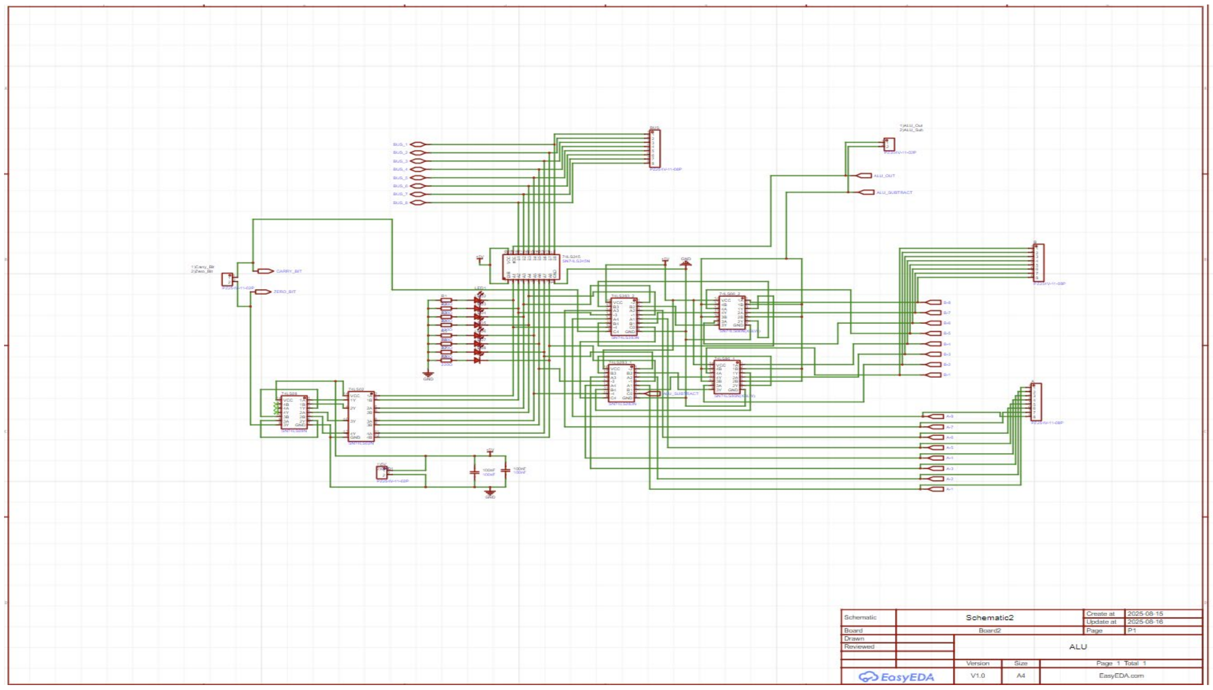


Image 6. ALU Schematic in EasyEDA Pro

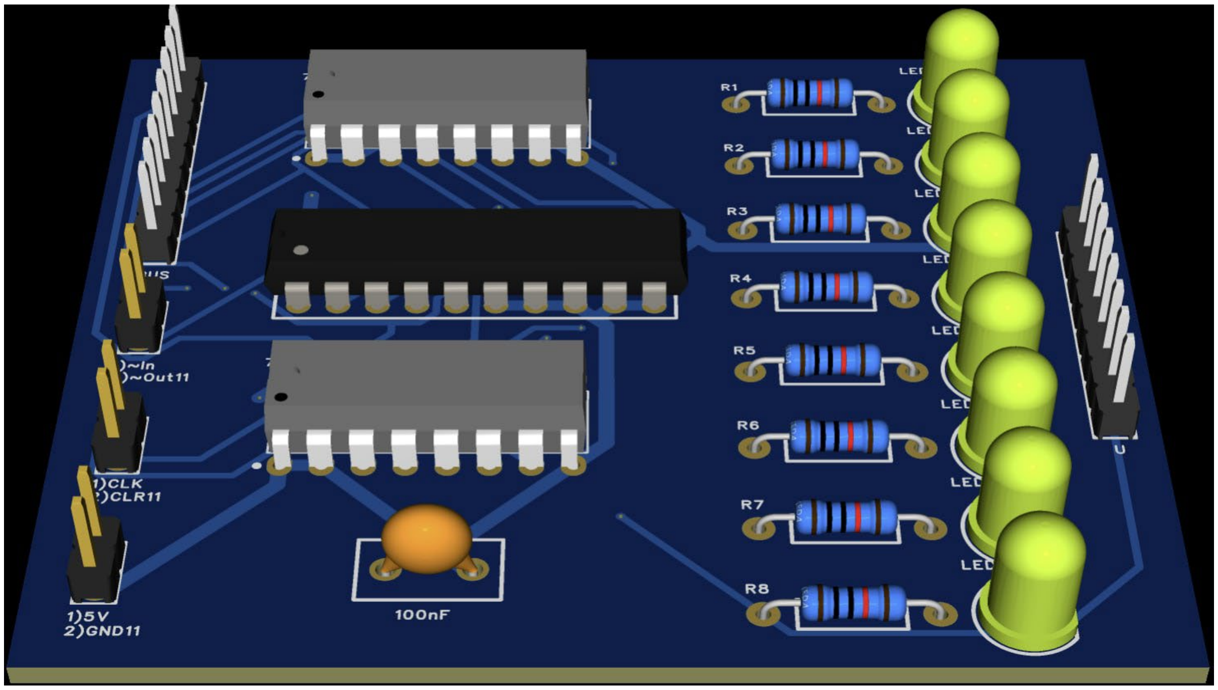


Image 7. ALU PCB in EasyEDA Pro

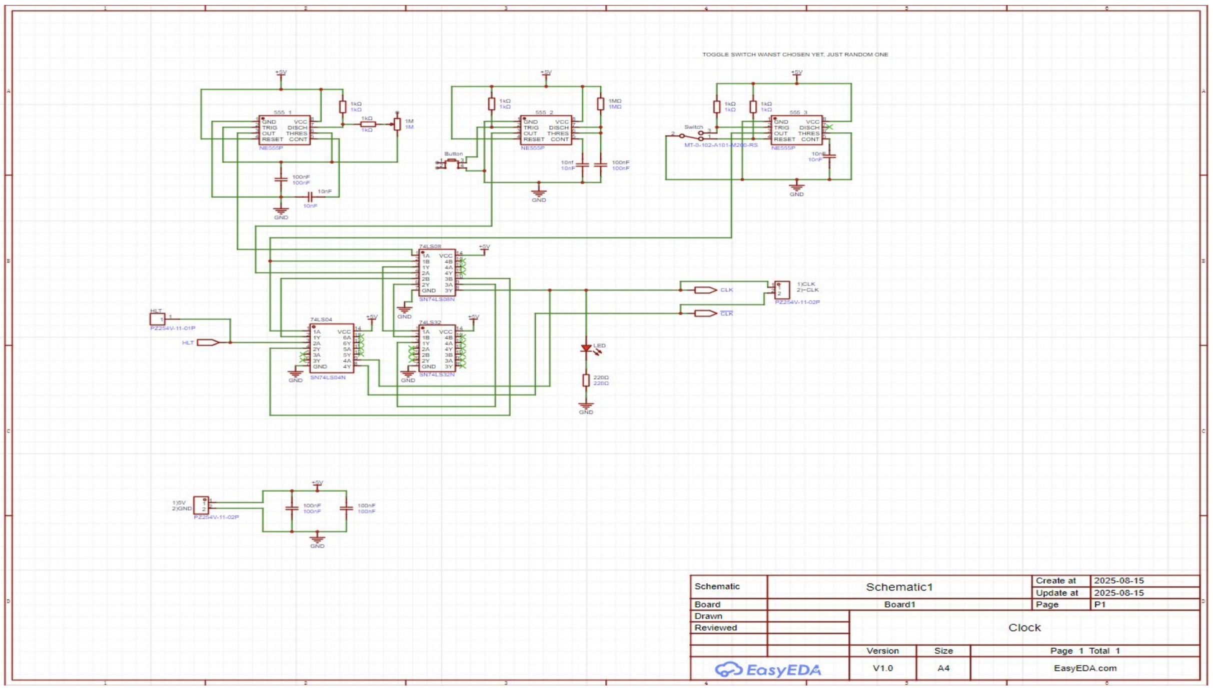


Image 8. Clock Schematic in EasyEDA Pro

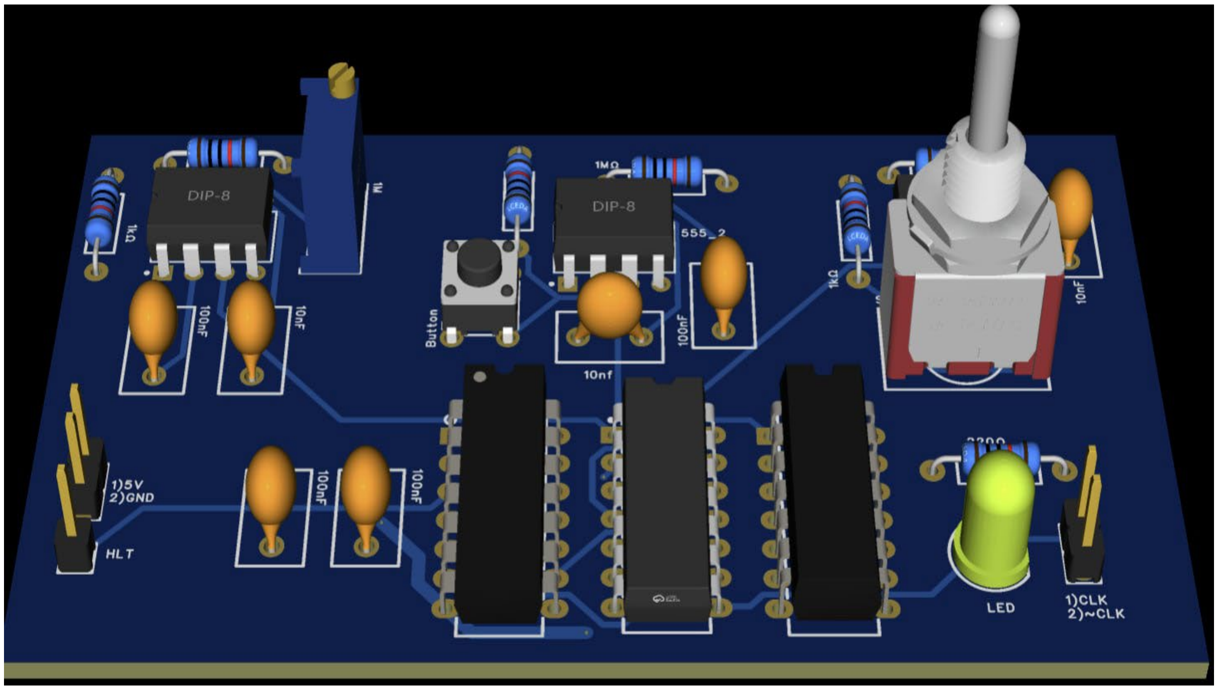


Image 9. Clock PCB in EasyEDA Pro

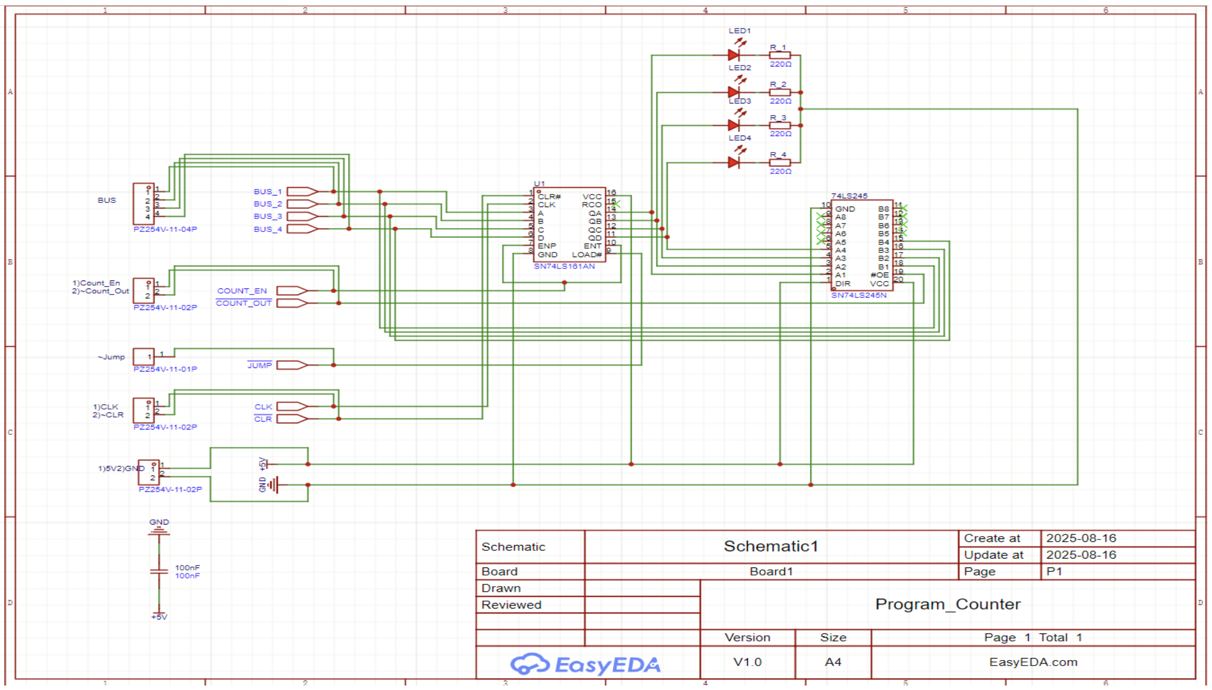


Image 10. Program Counter Schematic in EasyEDA Pro

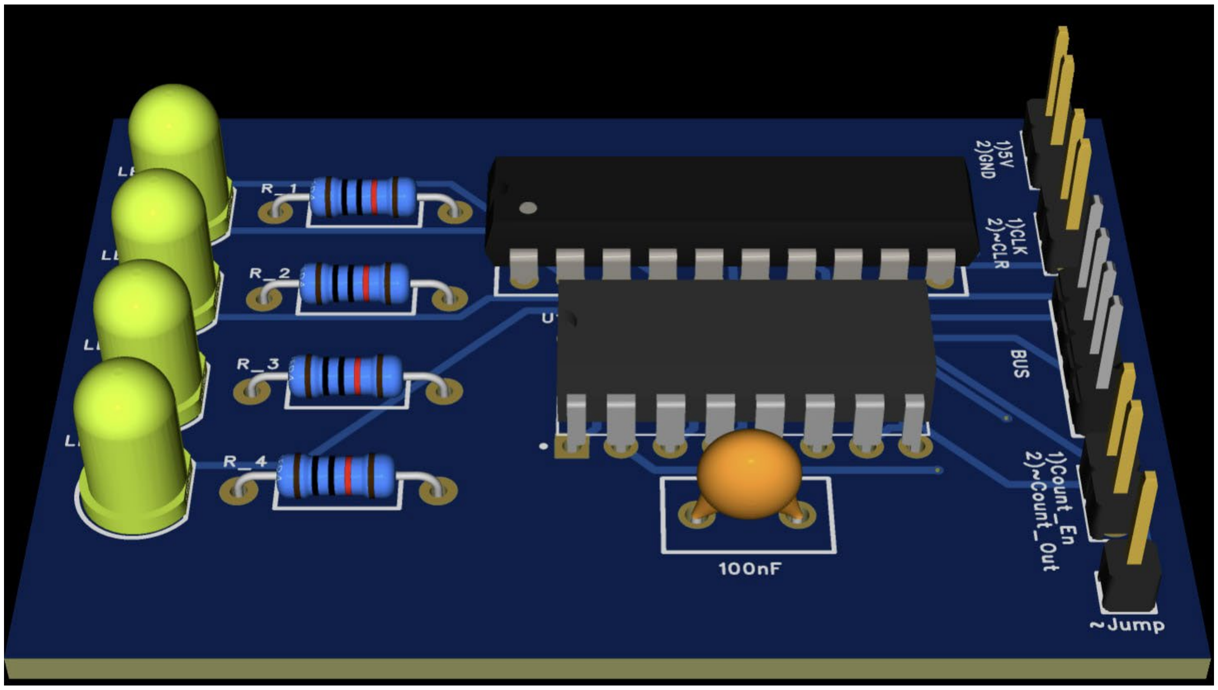


Image 11. Program Counter PCB in EasyEDA Pro

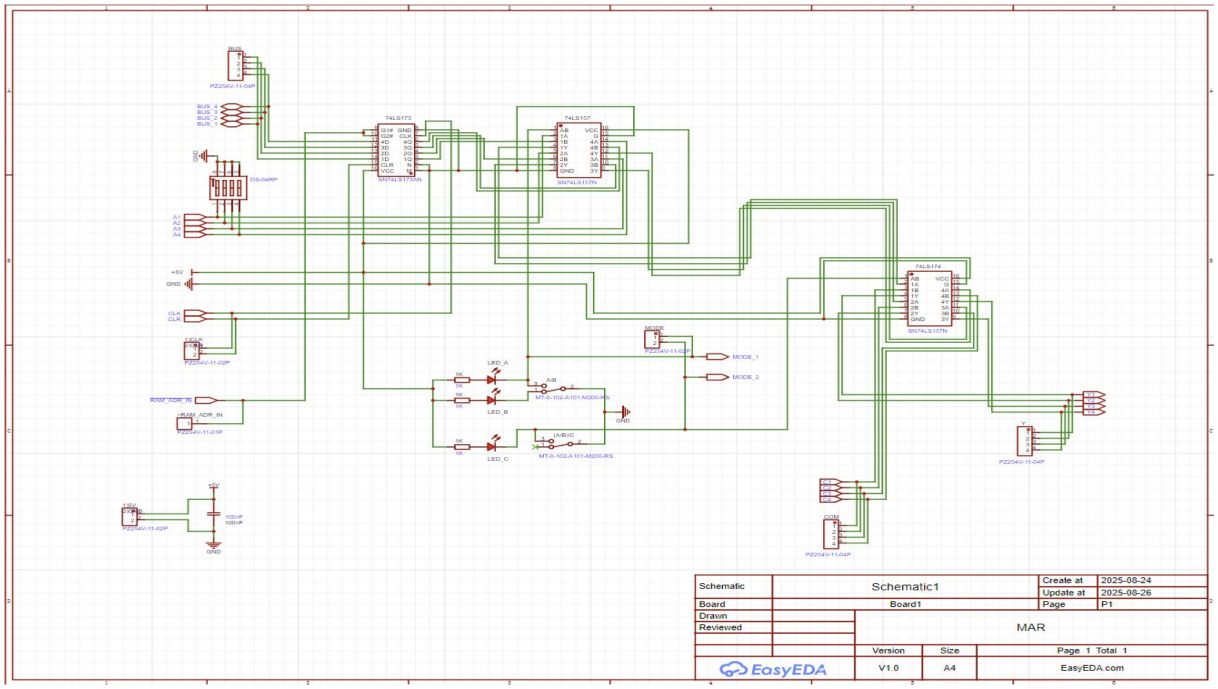


Image 12. Memory Address Register (MAR) Schematic in EasyEDA Pro

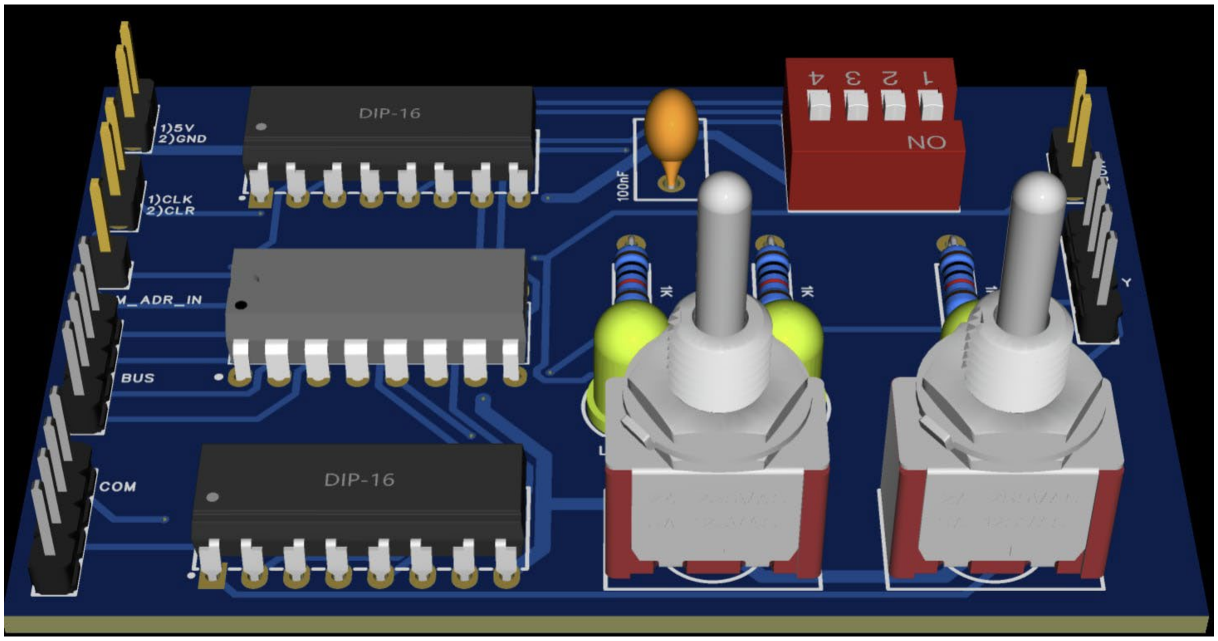


Image 13. Memory Address Register (MAR) PCB in EasyEDA Pro

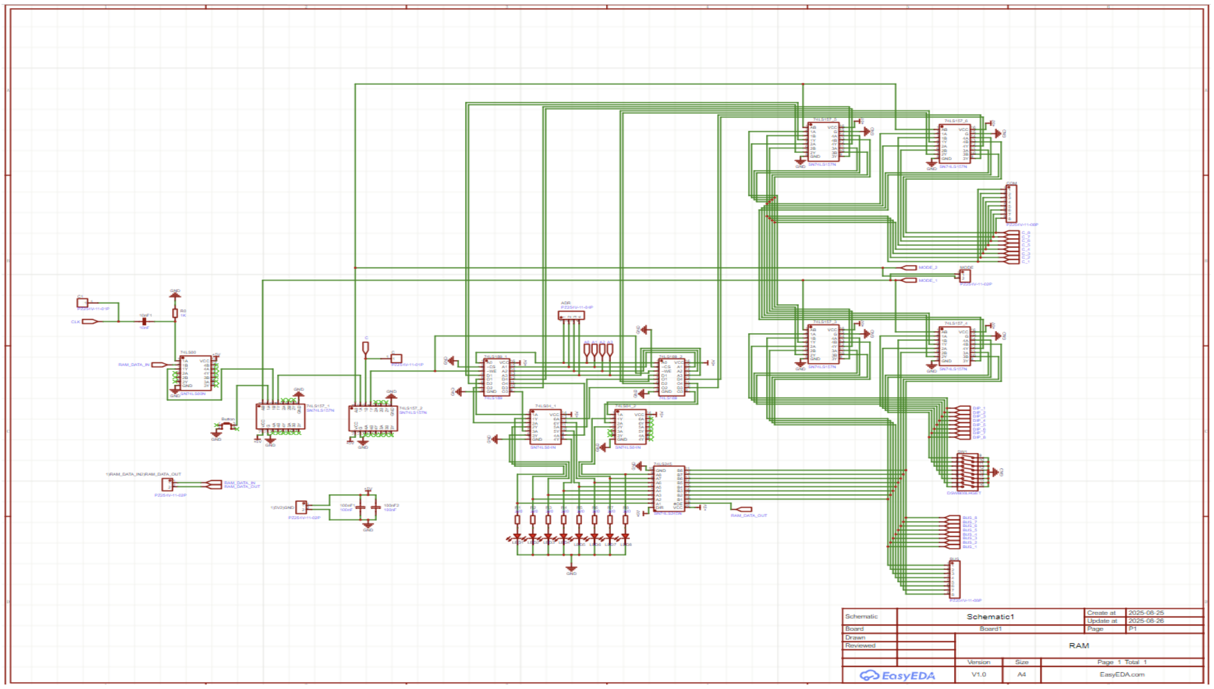


Image 14. Random Access Register (RAM) Schematic in EasyEDA pro

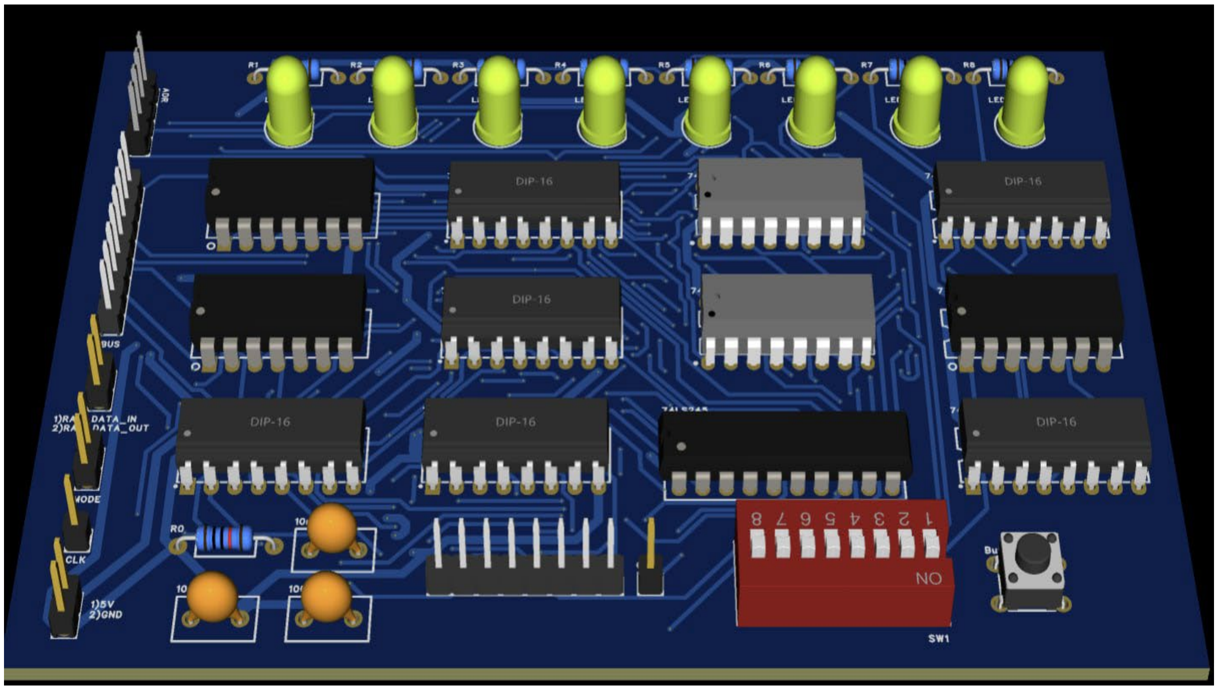


Image 15. Random Access Register (RAM) PCB in EasyEDA pro

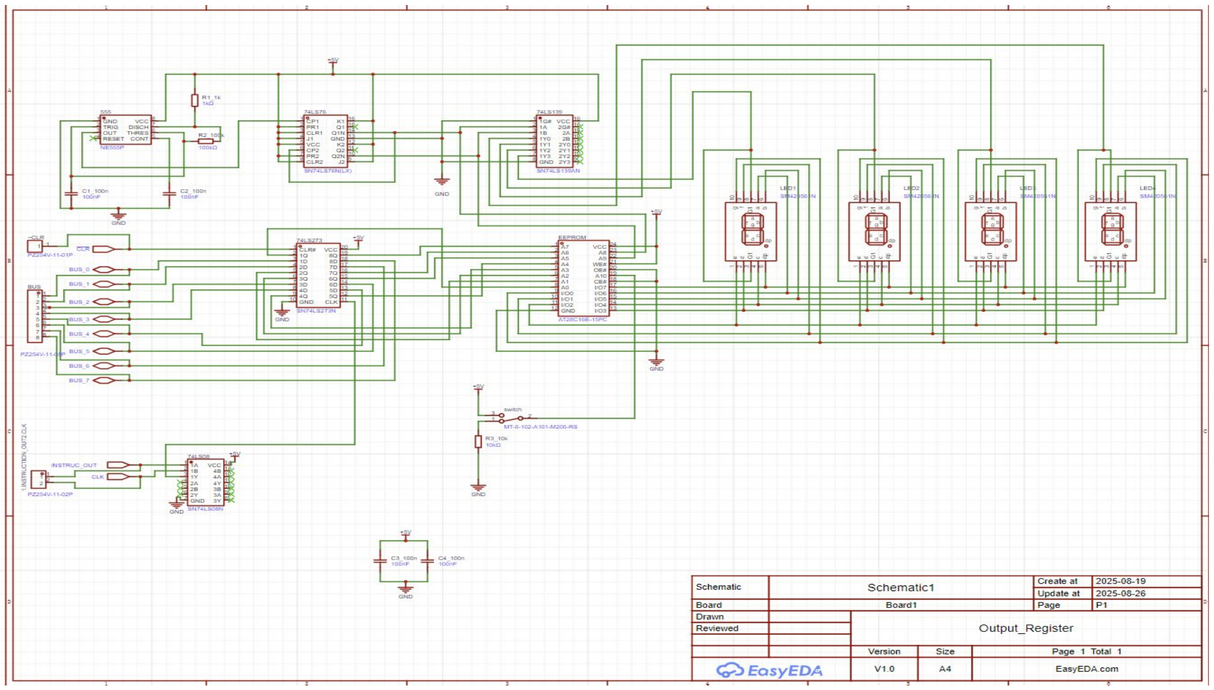


Image 16. Output Register Schematic in EasyEDA Pro

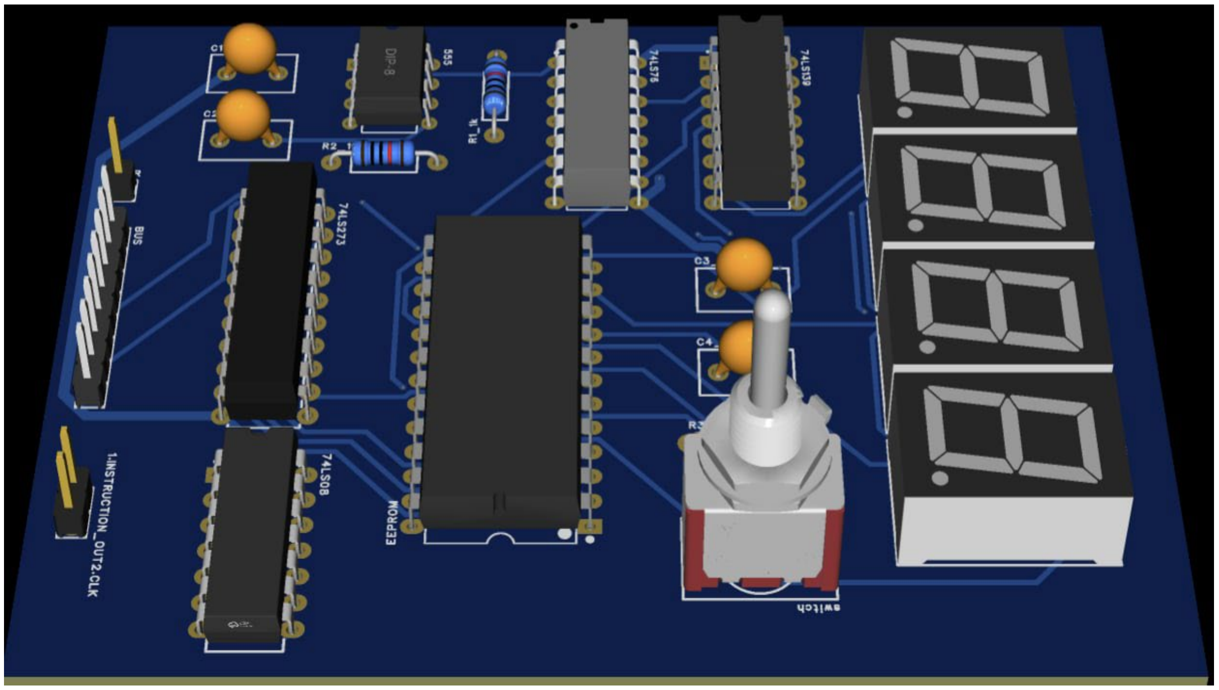
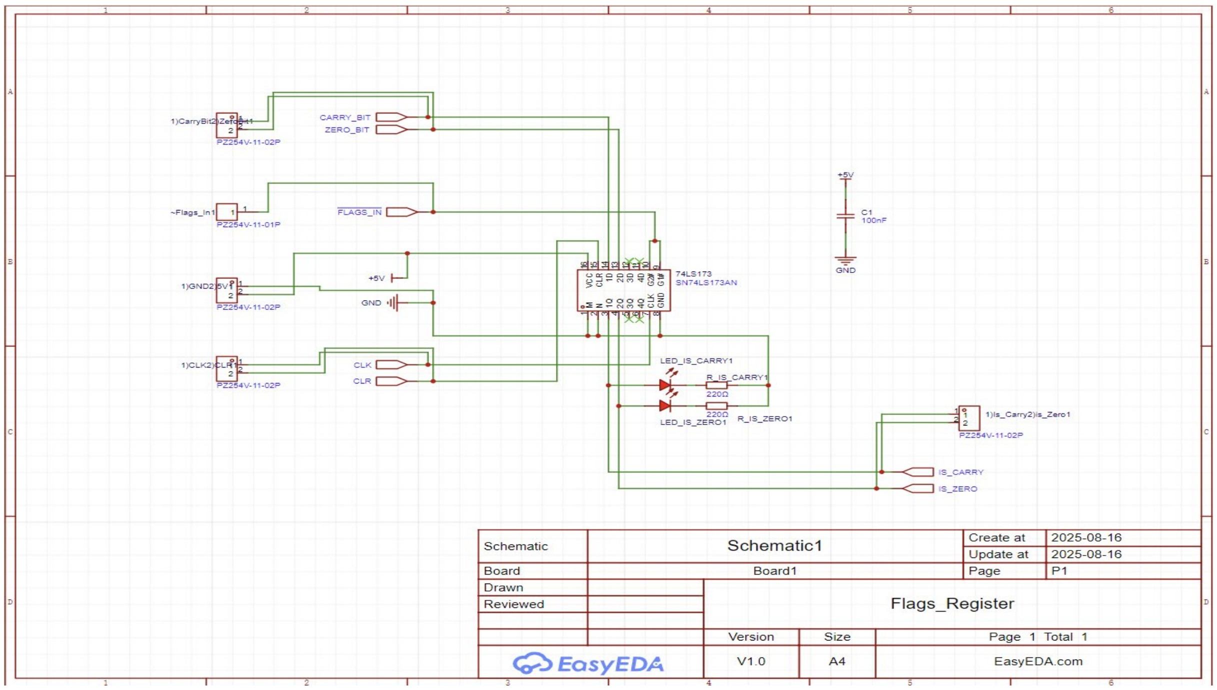


Image 17. Output Register PCB in EasyEDA Pro



Schematic	Schematic1	Create at	2025-08-16
		Update at	2025-08-16
Board	Board1	Page	P1
Drawn		Flags_Register	
Reviewed			
	Version	Size	Page 1 Total 1
EasyEDA	V1.0	A4	EasyEDA.com

Image 18. Flags Register Schematic in EasyEDA Pro

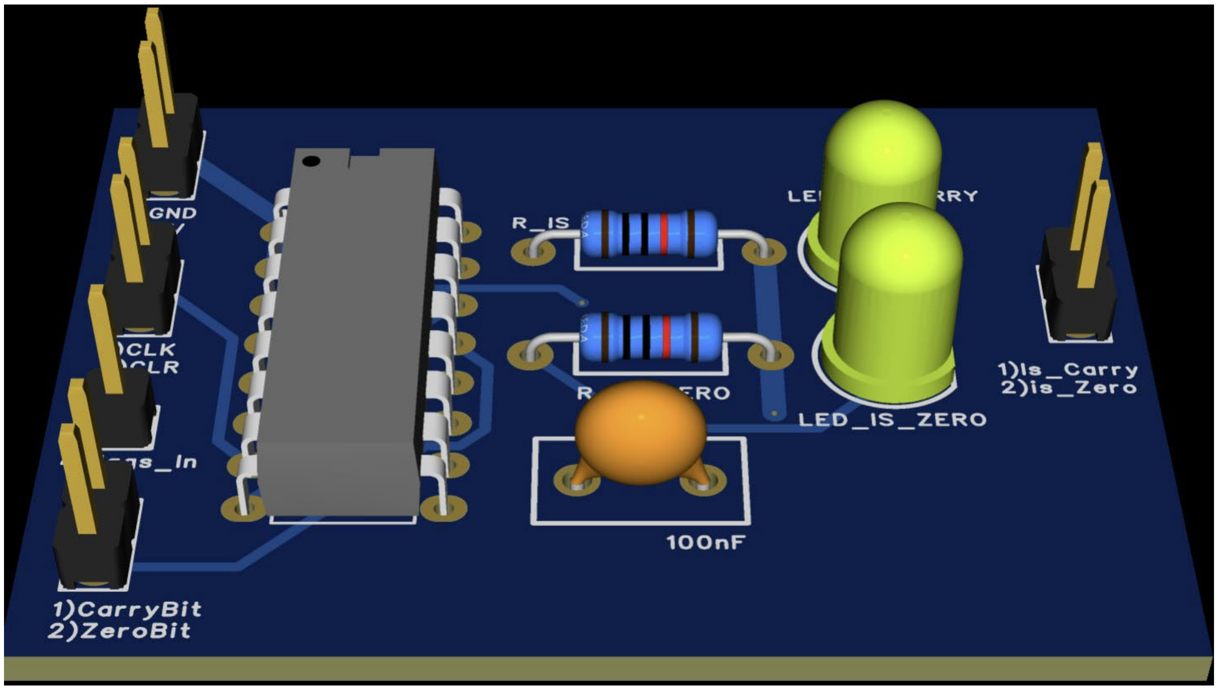


Image 19. Flags Register PCB in EasyEDA Pro

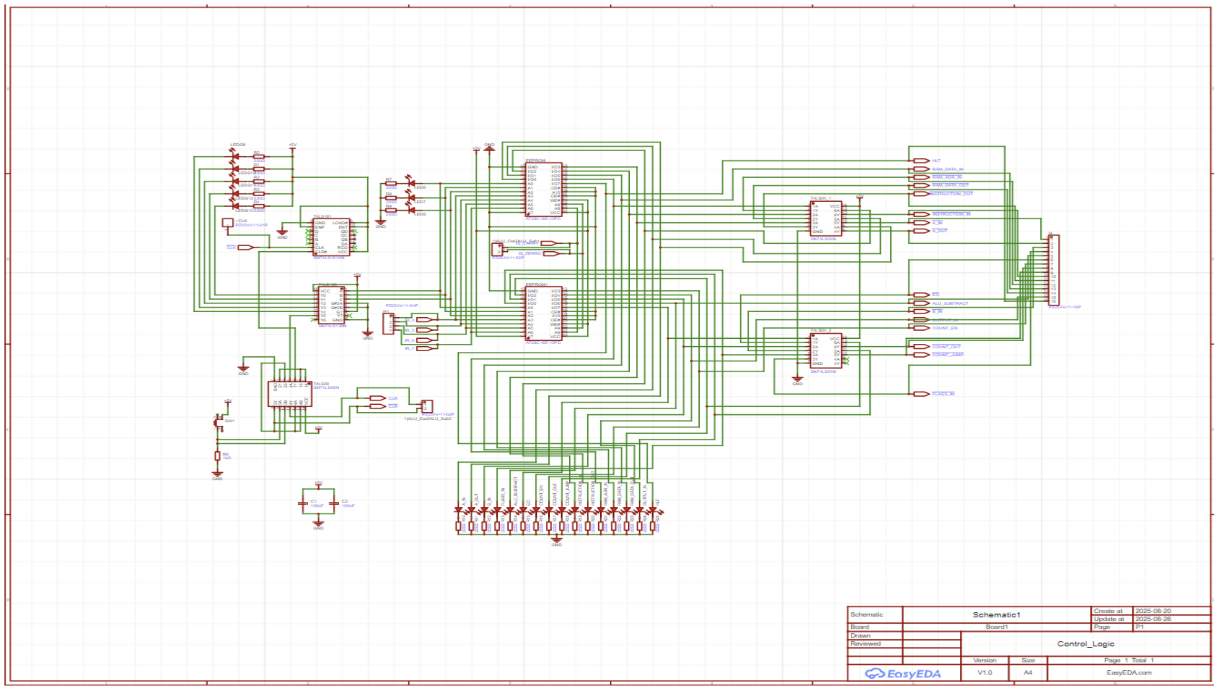


Image 20. Control Logic Unit Schematic in EasyEDA Pro

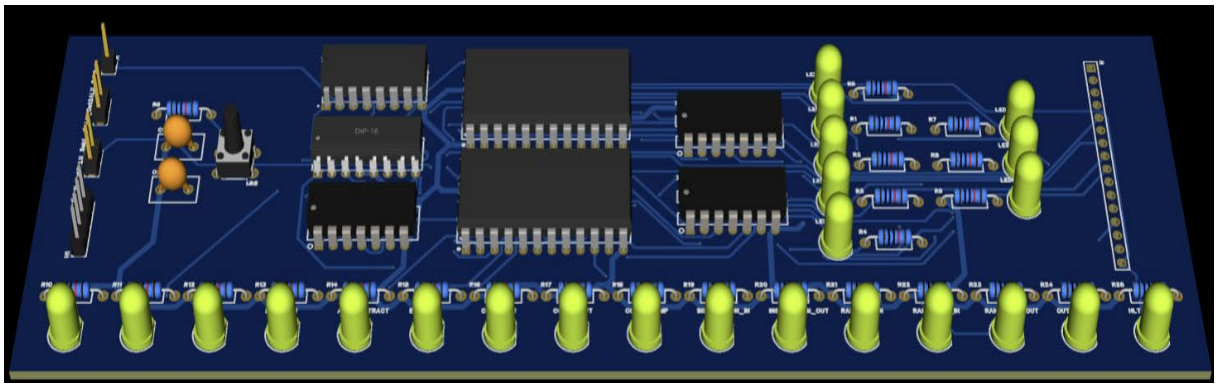


Image 21. Control Logic Unit Schematic in EasyEDA Pro

Introduction

We chose to build an 8-bit computer for our project because we have always been curious about the inner workings of computers and wanted to deepen our understanding beyond theory. Modern computers are incredibly complex, but at their core they follow simple digital logic principles. By constructing a computer from basic components, we aimed to explore how data is processed, how instructions are executed, and how fundamental modules interact to perform meaningful computation. This hands-on approach allowed us to bridge the gap between theoretical learning and practical understanding.

This project is not intended to be a marketable product. Instead, it is an educational project designed to reinforce concepts we learned in the Digital Circuits course. Throughout the semester, we studied logic gates, flip-flops, counters, registers, and finite state machines. We wanted to demonstrate how these elements come together to form a functioning computer system. Our goal is to show that anyone who understands digital logic theory can use that knowledge to build a real, working machine.

Our 8-bit computer is inspired by Ben Eater's educational architecture and consists of several core components, each performing an essential role in the system:

Two General-Purpose Registers (A and B)

Arithmetic Logic Unit

Clock Module

Program Counter

Instruction Register

Memory Address Register (MAR)

Random Access Memory (RAM)

Output Register

Flags Register

Control Logic Unit

Together, these modules form a simple but complete computing system capable of executing machine-level instructions stored in memory. Through our work, we gained practical experience in digital design, circuit debugging, sequential logic, and system-level thinking. Most importantly, we developed a much deeper appreciation for how computers operate at the most fundamental level.

Problem Statement and Solution Method

Modern computers are extremely complex systems, making it difficult for students to understand how they actually work at the fundamental level. Most learning focuses on abstract theory — binary numbers, logic gates, and machine instructions — without showing how these concepts come together to form a working computer. As a result, students often struggle to connect theory with real hardware operation.
To address this problem, we decided to design and build an 8-bit computer from basic electronic components. By constructing each module we can demonstrate how data is processed and instructions are executed step by step. This hands-on approach provides a tangible understanding of how a computer operates at the hardware level and bridges the gap between theoretical knowledge and practical application.

Technical Side

A/B Register

Used Integrated Circuits:

IC	Quantity
74LS173	2
74LS245	1

The A/B register pair serves as temporary data storage units within the computer. Their main role is to hold variables that the Arithmetic Logic Unit (ALU) will later use for computations. The information stored in these registers can be transferred to or received from the common data bus as needed.

Each register is constructed using two 4-bit 74LS173 latch ICs, which together form an 8-bit register. Both registers are connected to a 74LS245 bidirectional transceiver that controls the direction of data flow between the register and the shared bus. This allows data to be flexibly loaded into the register or output to the bus depending on the control signals.

Inputs:

- Clock (1 bit): Synchronizes loading operations with the system clock; new data is latched on the rising edge.
- A/B In (1 bit): Determines whether data from the bus is stored in the register.
- A/B Out (1 bit): Controls whether the register's stored data is placed on the bus.
- Bus (8 bits): The shared communication channel for data transfer between modules.

Outputs:

- A/B (8 bits): The current value stored in the register, directly available to the ALU.
- Controlled A/B Output (8 bits): The output that passes through the 74LS245 to the bus, enabled only when the output signal is active.

Instruction Register

Used Integrated Circuits:

IC	Quantity
74LS173	2
74LS245	1

The Instruction Register holds the current instruction fetched from memory. It is structurally similar to the A/B registers but serves a different purpose — it temporarily stores the instruction so it can be decoded and executed by the control logic.
The IR is composed of two 74LS173 ICs, forming an 8-bit register. It is connected to a 74LS245 transceiver that manages data flow between the IR and the common bus. Only the lower 4 bits (least significant nibble) of the IR can be output to the bus, while the upper 4 bits are sent to the control unit to determine which operation should be executed.
The instruction's upper 4 bits represent the opcode (the command), while the lower 4 bits usually represent an operand or address.
Inputs:

- Clock (1 bit): Data is latched on the rising edge of the clock.
- Instruction In (1 bit): Controls whether a new instruction from the bus is loaded.
- Instruction Out (1 bit): Controls whether the lower nibble is output to the bus.
- Bus (8 bits): The shared data channel.

Outputs:

- Instruction (8 bits): The complete instruction currently stored.
- Controlled Output (4 bits): The operand part of the instruction that can be placed on the bus when required.

This register connects memory with the control unit, enabling the computer to fetch an instruction, decode it, and execute the corresponding operation through the control signals.

Arithmetic Logic Unit (ALU)

Used Integrated Circuits:

IC	Quantity
74LS86	2

74LS283	2
74LS245	1
74LS08	1
74LS02	1

The ALU is the computational core of the computer. It performs all arithmetic and logical operations required by the processor. Its primary purpose is to take inputs from the A and B registers, perform operations such as addition or subtraction, and send the result back to the bus or registers.

The ALU uses 74LS283 binary adders for addition, 74LS86 XOR gates to invert bits for subtraction, and 74LS08/74LS02 gates to generate logic conditions for the Zero and Carry flags. The 74LS245 transceiver controls whether the ALU's output is placed on the bus.

Operating Principle:

When the ALU Subtract signal is 0, the circuit performs addition (A + B).

When the signal is 1, the B input is inverted via XOR gates, and a carry-in of 1 is provided to the adder, effectively performing subtraction using two's complement arithmetic:

$A + NOT(B) + 1 = A - B$.

Inputs:

- A (8 bits): Data from the A register.
- B (8 bits): Data from the B register.
- Alu Subtract (1 bit): Control signal that selects between addition (0) and subtraction (1).
- Alu Out(1 bit): Determines whether the ALU places its result on the system bus via the 74LS245 transceiver.

Outputs:

- Result (8 bits): The computed result of the operation.
- Carry Flag (1 bit): Indicates overflow during arithmetic operations.
- Zero Flag (1 bit): Indicates whether the output result equals zero.

- Bus Output (8 bits): Data passed to the bus via the 74LS245, active when enabled.

Clock Module

Used Integrated Circuits:

IC	Quantity
555 Timer IC	3
74LS04 Hex Inverter	1
74LS08 Quad AND Gate	1
74LS32 Quad OR Gate	1

The clock module provides the timing pulses that synchronize all operations within the computer. Every process — from incrementing the Program Counter to performing ALU operations — depends on these periodic signals. Without a stable clock, modules would not coordinate correctly, leading to timing errors and incorrect data transfers.
This clock is built around a 555 timer configured as an astable multivibrator, generating a continuous square wave output. The oscillation frequency is determined by the connected potentiometer and capacitor values, which can be adjusted to modify the computer speed. This allows the system to run slowly for observation or faster for normal operation.
A mode switch lets the user choose between:
Automatic mode: The clock runs continuously, advancing the computer at a steady rate.

Manual mode: A pushbutton produces single pulses, enabling step-by-step control over the computer’s operation.

In manual mode, a debouncing circuit ensures that each button press produces only one clean pulse, preventing unintended multiple triggers due to switch bounce. LED indicators display the current clock state (on/off) and selected mode, which helps during testing and demonstrations.
The clock’s output drives all synchronous components through a shared line, ensuring every register, counter, and control signal operates in harmony. In essence, this module acts as the heartbeat of the 8-bit computer, coordinating every micro-operation and maintaining precise execution timing across the system.
Input:

- Halt (1 bit): This signal stops the operation of the computer when activated. When the Halt line is high, the clock circuit is disabled, preventing further clock pulses from being sent to the system. As a result, all sequential components such as registers and counters stop updating, effectively freezing the computer’s state.

Outputs:

- Clock (1 bit): generated clock pulse.

- ~Clock (1 bit): inverse of the generated clock pulse. It ensures that certain components that require operations on the opposite clock edge (for example, counter for micro-instructions in Control Logic Unit) can function correctly.

Program Counter (PC)

Used Integrated Circuits:

IC	Quantity
74LS161	1

The Program Counter (PC) is an essential component that keeps track of which instruction in memory should be executed next. It automatically increments after each instruction fetch and can be reset or manually set to a specific address for jumps or branches.

Built with a 74LS161 4-bit binary counter, the PC can count from 0 to 15, representing the 16 memory addresses in the system. For 8-bit addressing, two counters can be cascaded, but in this project, one was sufficient due to the small memory size.
Inputs:

- Clock (1 bit): Increments the counter on the rising edge.
- Counter Out (1 bit): Places the current address on the bus. Active Low.
- Counter Enable (1 bit): Allows counting when active.
- Jump (1 bit): Loads a new address into the counter for jumps. Active Low.
- Bus (8 bits): Used when loading a jump address.

Outputs:

- Bus Output (4 bits): The state of the counter/address placed on the bus when PC Out is active.

Memory Address Register (MAR)

Used Integrated Circuits:

IC	Quantity
74LS173	1
74LS157 multiplexer	1

The Memory Address Register (MAR) temporarily stores the address of the memory location that the computer wants to access. When the Program Counter outputs an address, the MAR latches it so the memory module can use it for data retrieval or storage.
The MAR in this design operates in two distinct modes, selectable by a switch connected to a multiplexer:

- Manual Mode: In this mode, the address is chosen manually by the user. This is useful for debugging or testing specific memory locations but mostly used to put code and variables inside RAM. When manual mode is active, the chosen address is not stored in the register and is directly applied to ram. a red LED indicator lights up, showing that the current address is being set by hand rather than by the computer.

- Program Mode: In this mode, the address is automatically provided by the bus, typically from the Program Counter or Instruction Register during normal program execution and than stored in register. A green LED indicates that the MAR is receiving its address input from the system bus, and thus operating under computer control.

The mode selection switch determines which address source is active by controlling the multiplexer input. This setup provides flexibility—allowing both automated program execution and manual inspection of memory contents without reconfiguring the circuit.
In addition, the MAR outputs a Program Mode signal, which connects to the RAM's multiplexer. Since the RAM module also supports two modes (manual and program), this signal ensures that both components operate in synchronization, preventing conflicts between manual and automatic addressing.
Inputs:

- Clock (1 bit): Latches the address on the rising edge of the system clock.

- RAM Address In (1 bit): Enables loading of an address from the bus when active. Active Low.

- Bus (8 bits): Carries the address data lines from the computer to the MAR.

- Mode Switch (1 bit): Determines whether the MAR operates in manual or program mode.
- Addresses (4 bits) : Chosen by the user

Outputs:

- Address (4 bits): The currently selected address, sent directly to the RAM module.

Program Mode (1 bit): Indicates the active mode and connects to the RAM multiplexer to maintain synchronized address selection.

Random Access Memory (RAM)

Used Integrated Circuits:

IC	Quantity
74LS189 RAM	2
74LS157 multiplexer	3
74LS04 Hex inverter	2

The Random Access Memory (RAM) module serves as the main data storage for the 8-bit computer. It holds both instructions and data that the computer reads and writes during program execution. The design uses two 74LS189 memory ICs, each providing 16 addresses of 4-bit storage, which together form an 8-bit wide memory by connecting their address lines in parallel.

All address inputs are shared between the two RAM chips so that each memory location represents a single 8-bit value — the high 4 bits stored in one IC and the low 4 bits in the other. This configuration effectively doubles the data width while maintaining the same address space.

The RAM supports two operating modes—Manual Mode and Program Mode—allowing both testing and normal program execution. Mode selection is controlled using three 74LS157 multiplexers:

Two multiplexers handle switching between manual data input and bus data for the 8 data lines. The select signal comes from MAR.

The third multiplexer selects the source for the write signal to RAM, choosing between the manual input and signal controlled by the control logic unit during program execution.

Mode Descriptions:

- Manual Mode
  In this mode, data and addresses are entered manually using switches. This is useful for loading test values or inspecting memory without running a program but primarily used to put code that the computer should execute after putting it on runtime mode.

- Runtime Mode
  In program execution, the RAM interacts automatically with the computer over the system bus. The MAR provides the memory address, and data transfer is managed by the control logic. The Program Mode signal selects the correct inputs on the multiplexers, ensuring synchronized operation with other modules.

The RAM connects to the system bus through a 74LS245 bidirectional bus transceiver. This component controls whether RAM outputs are allowed to place data onto the bus. It prevents bus conflicts by enabling data output only when required, based on the RAM Data Out control signal. This ensures clean and safe data transfer between RAM and other modules.

Inputs to the RAM

- Clock (1 bit): Synchronizes write operations to memory.

- Address (4 bits): Selects the memory location. Comes from the MAR.

- Program Mode (1 bit): Selects whether the RAM receives address and data from manual inputs or from the system bus during program execution.

- Bus (8 bits): Carries data between the RAM and computer.

- Manual Data Input (8 bits): Used only in Manual Mode to load data directly into memory through two 74LS157 multiplexers.

- RAM Data In (1 bit, active high): Write enable signal that stores the input data into the selected memory address on the rising clock edge.

- RAM Data Out Enable (1 bit, active low): Enables memory output, allowing stored data to be placed on the bus via the 74LS245.

- 74LS245 Output Enable (1 bit, active low): Controls whether RAM is allowed to drive the system bus, preventing bus conflicts.

Outputs from the RAM

- Data Output to Bus (8 bits): When RAM Data Out is low, the stored data from the selected address is placed on the system bus.

- Through this setup, the RAM module provides a flexible and efficient data storage system. The dual 74LS189 configuration not only extends the data width to 8 bits but also allows seamless switching between manual and automated operations — an essential feature for testing, debugging, and program execution in the 8-bit computer.

Output Register

Used Integrated Circuits:

IC	Quantity
74LS273 Octal D-type Register	1
28C16 (2816) EEPROM	1
74LS76 JK Flip-Flop	1
LM555 Timer	1
74LS08 Quad AND Gate	1

The Output Register module displays the 8-bit value from the system bus as a decimal number on four 7-segment displays. A single 74LS273 stores the 8-bit output, and a 28C16 EEPROM converts that value into segment data for each display digit.
The EEPROM uses 11 address lines:
A0–A7 come from the 74LS273 (the binary value). A8 and A9 select which decimal digit (hundreds, tens, or ones) is currently displayed. A10 selects between signed and unsigned display modes. In unsigned mode, the leftmost display is blank and the other three digits show values from 0–255.
In signed mode, the leftmost display shows a minus sign (–), and the range becomes 0–128.
A 555 timer and 74LS76 generate a blinking sequence that rapidly switches A8 and A9, cycling through each digit. The blinking signal is also connected to the common grounds of the displays, so only one display is active at a time, giving the illusion that all digits are on simultaneously.
Inputs

- Clock: Latches data into the 74LS273.
- Data Bus (8 bits): Binary value from CPU.
- Mode Select (A10): Chooses signed or unsigned mode.

Outputs

- Visual Output: Four 7-segment displays showing the sign and three decimal digits.

Bus System

The bus acts as the shared 8-bit communication channel connecting all modules. At any given moment, only one device can place data onto the bus, while others read it. This is managed through control signals like X Out (to output to the bus) and X In (to read from it).
The bus ensures synchronization and prevents data conflicts. The 74LS245 transceiver chips are essential for this process, as they isolate and control the flow of data, ensuring stable operation.

Flags Register

Used Integrated Circuits:

IC	Quantity

74LS173N	1

The Flags Register is a small but essential component of the 8-bit computer, for conditional jumps. It stores information generated by the Arithmetic Logic Unit (ALU). These status indicators, known as flags, are used by control logic to make decisions during program executions.
The module uses a 74LS173 quad D-type register to store two flag bits:
Carry Flag (is_carry): Indicates whether an arithmetic operation generated a carry out of the most significant bit, useful for multi-byte addition or detecting unsigned overflow.

Zero Flag (is_zero): Signals that the result of an ALU operation is zero.

Only the flags relevant to this architecture are used; the remaining bits of the 74LS173 are unused and grounded or left stable.
Inputs to the Flags Register:

- Carry Bit (1 bit): Comes from the ALU and indicates whether a carry was generated during an addition or subtraction.

- Zero Bit (1 bit): Also from the ALU, it signals that the ALU result is zero.

- FLAGS_In (1 bit, active low): Load enable signal. When active (low), it stores the current Carry and Zero bits into the 74LS173 register on the next clock edge.

- Clock (1 bit): Synchronizes the loading of flag values into the register.

Outputs from the Flags Register:
- is_carry (1 bit): Output of the Carry flag, used by control logic for conditional operations such as "Jump if Carry."

- is_zero (1 bit): Output of the Zero flag, used for instructions like "Jump if Zero."

Control Logic Unit

Used Integrated Circuits:

IC	Quantity
28C16 (2816) EEPROM	2
74LS161 4-bit Binary Counter	1
74LS138 3-to-8 Decoder	1
74LS00 Quad NAND Gate	1

The Control Logic module orchestrates all CPU operations by generating 16 control signals that manage data flow between registers, memory, the ALU, and I/O. It uses two 28C16 EEPROMs programmed with microcode sequences. Each machine instruction is broken down into a series of microinstructions (time steps), with the EEPROMs acting as lookup tables that map instruction opcodes, execution states, and processor flags to specific control signals.
The module uses 9 of the 11 available address lines on each EEPROM to create a 512-entry (2^9) control store:
Address Line Allocation:
A5–A8: Instruction opcode from the Instruction Register (16 possible instructions)
A2–A4: Microinstruction step from 74LS161 counter (up to 8 time steps per instruction)
A0–A1: Processor flags for conditional execution (4 flag combinations)
Step Counter (74LS161)
The 74LS161 4-bit synchronous counter generates the microinstruction sequence (T0, T1, T2, ... T7). Only its lower 3 outputs (Q0–Q2) are used, providing 8 possible steps per instruction. The counter:
Advances on each CLKD (gated/divided clock) pulse
Resets after completing an instruction cycle
Sequences through each phase of instruction execution
Output Decoder (74LS138)

The 74LS138 3-to-8 line decoder monitors the counter outputs and generates a reset signal when the count reaches a specific value (typically after T5 or when the microinstruction sequence completes).
Inputs
Clock (CLKD): Synchronized clock signal that advances the microinstruction counter
Instruction Register (A5–A8): 4-bit opcode from current instruction
Flags (A0–A1): Direct processor status inputs (Zero, Carry, or other ALU flags)
Outputs: The two EEPROMs provide 16 control signals (8 bits each) that coordinate all CPU operations:
Register Control Signals:
A_In: Load data into Register A
A_Out: Enable Register A output to bus
B_In: Load data into Register B
Flags_In: Store ALU flags into flag register
ALU Control Signals:
ALU_subtract: Configure ALU for subtraction operation
ALU_Out: Enable ALU result onto data bus
Program Counter Control:
Count_En: Enable program counter increment
Count_Out: Output program counter value to address bus
Count_Jump: Load program counter with new address (for jumps)
Memory Interface Signals:
Ram_Adr_In: Load RAM address register
Ram_Data_In: Write data from bus to RAM
Ram_Data_Out: Read data from RAM to bus
System Control Signals:
Instruction_In: Load instruction register from memory
Instruction_Out: Output instruction operand to bus (immediate values)
Output_In: Load output display register
HLT: Halt CPU execution
The EEPROM contents define the computer's instruction set. Each address location stores a 16-bit control word, with each bit corresponding to one control signal. By reprogramming the EEPROMs, the instruction set can be modified without changing hardware, demonstrating the flexibility of microprogrammed control units. For more information about our control signals you can look up Figure 1.

Logisim Evolution Simulation

Before assembling the physical circuits, the entire computer was designed and tested using Logisim Evolution, a digital logic simulation software. This step was crucial in verifying the correctness of the architecture before committing to hardware construction.

Each module — including the ALU, registers, memory, program counter, control logic, and clock — was implemented and connected virtually within Logisim. The simulation allowed observation of data flow across the bus, the timing of control signals, and the interaction between modules during each clock cycle.

The modular approach made it possible to test components individually before integrating them into the complete system. Errors such as incorrect wiring, inverted control signals, or timing mismatches were identified and fixed at this stage, saving significant time later during physical implementation.

Logisim's visual interface provided an excellent platform to demonstrate the computer's fetch-decode-execute cycle in real time. Control signals were animated, and binary data changes could be observed directly on virtual LEDs, giving a clear picture of how the computer operated internally.

The full schematic in Logisim included all core modules — the A and B registers, ALU, Program Counter, Instruction Register, Memory Address Register, RAM, Output Register, Flags Register, and Control Logic Unit — all connected through a shared 8-bit bus.

Overall, the Logisim Evolution simulation played a key role in validating the design, ensuring logical correctness, and preparing for successful hardware assembly.

Assembly Instructions

Fetch		
000	Counter out Ram adr In	
001	Ram data out	

	Instruction In Counter Enable	

Operation	Name	Instruction	Micro Instruction	Carry Flag	Zero Flag	
No operation	NOP	0000	010	X	X	
			011	X	X	
			100	X	X	
Load adr into A	LDA(adr)	0001	010	X	X	Instruction out Ram adr in
			011	X	X	Ram data out A in
			100	X	X	
Add adr to A and store answer in A	ADD(adr)	0010	010	X	X	Instruction out Ram adr in
			011	X	X	Ram data out B in
			100	X	X	ALU out A in Flags_In
Subtract adr to A and store	SUB(adr)	0011	010	X	X	Instruction out Ram adr in
			011	X	X	Ram data out B in

Carry Flag	Zero Flag	

answer in A			100	X	X	ALU_Subtract ALU out A in Flags_In			
Stores A in adr	STA(adr)	0100	010	X	X	Instruction out Ram adr in			
			011	X	X	A_Out Ram_Data_In			
			100	X	X				
Loads value into A	LDI(val)	0101	010	X	X	Instruction_Out A_In			
			011	X	X				
			100	X	X				
Jumps to this adr in ram	JMP(adr)	0110	010	X	X	Instruction_Out Count_Jump			
			011	X	X				
			100	X	X				
jump to adr if carry of ALU(A B) is 1	JMC(adr)	0111	010	0	X		1	X	Instruction_Out Count_Jump
			011	X	X				
			100	X	X				
jumt to adr if	JMZ(adr)	1000	010	X	0		X	1	Instruction_Out Count_Jump

ALU(A B) is 0			011	X	X		
			100	X	X		
		1001	010				
			011				
			100				
		1010	010				
			011				
			100				
		1011	010				
			011				
			100				
		1100	010				
			011				
			100				
		1101	010				
			011				
			100				
Put A into output register	OUT	1110	010	X	X	A out Out in	
			011	X	X		
			100	X	X		
Halt the clock	HLT	1111	010	X	X	HLT	
			011	X	X		
			100	X	X		

For assembly code example see listings 2 and 3.

Results and Evaluation

Most of the hardware modules were successfully implemented, tested, and interconnected. The computer was able to execute simple instructions such as loading data into registers, performing addition and subtraction, and outputting results.
The control unit malfunction limited full automation of the instruction cycle, but manual testing verified correct data flow and ALU operations. The project effectively demonstrated how low-level electronic components combine to perform computational logic and how timing coordination is crucial for system stability.
This project provided deep insight into hardware-level computing and improved our understanding of computer design, synchronization, and debugging of complex digital systems.

Contribution of the team members

The project was completed collaboratively, with each team member responsible for specific modules of the 8-bit computer. Tekla worked on the RAM, Clock Module, and Program Counter, ensuring proper memory access, timing synchronization, and instruction sequencing within the system. Saba developed the A and B Registers, Instruction Register, Arithmetic Logic Unit (ALU), and Output Register, which together form the data path and processing core of the computer.
Both team members jointly designed and implemented the Control Logic Unit, which coordinates the entire system and ensures that all components operate in synchronization.
In addition to circuit design, each of us created the PCB layouts for our respective modules and performed the necessary testing and debugging.
Before assembling the physical circuits, we collaboratively developed a full Logisim Evolution simulation of the 8-bit computer. This allowed us to verify the logical operation of the system, debug timing issues, and confirm that all components interacted correctly before hardware implementation.
Through this division of work, both members contributed equally to the success of the project, combining individual strengths and teamwork to achieve a functioning and educational 8-bit computer.

Conclusion

The development of our 8-bit computer was an ambitious educational project designed to transform theoretical knowledge from our *Digital Circuits* course into a working, physical system. Our main objective was to understand, in detail, how a computer operates internally — how instructions are fetched, decoded, and executed at the hardware level using binary signals and basic logic gates. Through this project, we not only achieved that goal but also gained practical experience in designing, debugging, and integrating digital systems.

Our 8-bit computer successfully performs several fundamental operations essential to a central processing unit. It can add and subtract data, execute conditional jumps (such as “if” operations), and store and retrieve information from RAM. These capabilities allow it to run small but meaningful programs that demonstrate the principles of computation.

We also designed a simple assembly-like language specifically for our architecture. By writing short sequences of binary instructions or assembly-style code and loading them into memory, the computer can execute structured algorithms such as for loops, while loops, and conditional branches. This provided a tangible way to observe how high-level programming constructs are ultimately represented and handled by hardware.

The computer’s modular design includes distinct units for arithmetic logic, control, memory, registers, and input/output, all synchronized by a clock signal. Each component was first simulated in Logisim Evolution and later built on breadboards using TTL logic ICs. The simulation stage allowed us to verify correctness before moving to hardware, while the breadboard implementation helped us understand practical challenges like timing synchronization and wiring management.

Although we faced obstacles — most notably the EEPROM failure and RAM PCB issue — we still achieved a fully operational system capable of manual programming and real instruction execution. Because of the EEPROM damage, our output is currently displayed in binary form rather than decimal, but this does not affect the system’s core functionality. We plan to order new EEPROMs and finalize this part in the future.

Overall, this project allowed us to bridge the gap between theory and practice. We gained a profound understanding of how a CPU processes data and instructions, learned to work through real engineering challenges, and deepened our appreciation for computer architecture. Despite its simplicity, our 8-bit computer represents a complete and functioning computational system — a significant milestone in our learning journey as future engineers.

References

B. Eater, “Building an 8-bit breadboard computer,” *Ben Eater Tutorials*, 2018. [Online]. Available: https://eater.net/8bit

Listings

Our github project: https://github.com/sababa27/8-Bit-Computer

Code N1:

```
15 - num(10)
14 - num(01)
00 - lda(15)
01 - out
02 - jmz(12)
03 - sub(14)
04 - sta(15)
05 - jmp(00)
12 - hlt
13 - jmp(12)
```

Code N2:

```
15 - jmp(14)
14 - hlt
13 - num(05)
12 - num(09)
11 - num(01)
00 - lda(13)
01 - out
02 - lda(12)
03 - sub(13)
04 - jmc(14)
05 - lda(13)
06 - add(11)
07 - sta(13)
```

08 - jmp(00)