Snowiee -8-bit Computer

SAP-1

Table of Contents

Adding two number -

- Step-by-step execution of instructions (LDA, ADD, etc.)
- Role of micro-instructions and control signals

Conclusion

- Summary of 8-bit computer as a foundational model

Abstract

An **8-bit computer** is a simple computing system where data is processed in 8-bit chunks, meaning the arithmetic logic unit (ALU), registers, and data bus all handle 8-bit values. It consists of several essential components working together to execute instructions. At its core, the **clock circuit** provides timing signals to synchronize operations, often controlled by a manual button for debugging or an oscillator for continuous execution. The **registers**, including the accumulator (A), instruction register (IR), and program counter (PC), temporarily store and manage data during computation. The **ALU** performs basic arithmetic and logic operations, such as addition, subtraction, AND, OR, and XOR, directly manipulating 8-bit values.

The **control unit** is responsible for decoding instructions and coordinating the activities of different components, following the **fetch-decode-execute** cycle. Memory is divided into **RAM**, which stores temporary data and running programs, and **ROM**, which holds permanent instructions like a bootloader or microcode. The **bus system** allows communication between these components, with the data bus transferring data, the address bus selecting memory locations, and the control bus managing read and write operations. **Input and output (I/O) devices**, such as switches, LEDs, and displays, provide interaction with the user.

The computer operates by fetching an instruction from memory, decoding it in the control unit, executing it in the ALU or other relevant components, and then repeating the cycle. Building an 8-bit computer on a breadboard involves using simple logic chips like the **74LS series**, along with **EEPROM** for storing microcode and **SRAM** for memory. A manual clock can be used for step-by-step debugging, while an oscillator allows continuous program execution.

Introduction-

A. Transistor-

A transistor is a semiconductor device that acts as a switch or amplifier. It is one of the most fundamental components in digital electronics. In the context of digital logic and 8-bit computers built on breadboards, Bipolar Junction Transistors (BJTs) are widely used due to their availability, simplicity, and educational value.

A **BJT** is a three-terminal device consisting of:

- **Base (B)** – the control terminal
- **Collector (C)** – where current flows out
- **Emitter (E)** – where current flows in

There are two types of BJTs:

- **NPN** – Most commonly used in digital logic
- **PNP** – Not used commonly

In digital circuits, we mostly use **NPN transistors**, where:

- A small current into the **base** allows a much larger current to flow from **collector to emitter**.
- When base current is applied → transistor is **ON** (saturation mode).
- When base current is absent → transistor is **OFF** (cutoff mode).

This **switching behavior** allows BJTs to represent binary **1 (on)** and **0 (off)** states.
BJTs can be arranged with resistors to form logic gates. Below is a breakdown of how common gates (NAND)are built using NPN transistors:

Here in Fig-1 shown how we can from a NAND gate using the BJT transistor; A NAND gate using two NPN bipolar junction transistors (BJTs) operates by arranging the transistors in series between the output and ground. The base of each transistor is connected to one of the input signals (A and B), and the collector of the first transistor (Q2) is connected to the output, which is also tied to a pull-up resistor connected to the power supply (Vcc). When either A or B is LOW (0), at least one of the transistors remains OFF, breaking the path to ground. As a result, the output is pulled HIGH through the resistor. Only when both A and B are HIGH (1) do both transistors turn ON, creating a continuous path from Vcc through the output and transistors to ground, causing the output to drop to LOW. Thus, the gate only outputs LOW when both inputs are HIGH, which matches the behavior of a NAND gate — the inverse of an AND gate. This setup demonstrates how BJTs can be used as electronic switches to implement logic functions at the fundamental level.
Transistors form the foundation of logic gates—the basic units that make decisions in digital systems. With just a handful of transistors, one can build AND, OR, NOT, NAND, NOR, XOR gates and more, which in turn can be combined to build counters, memory, arithmetic units, and control logic.

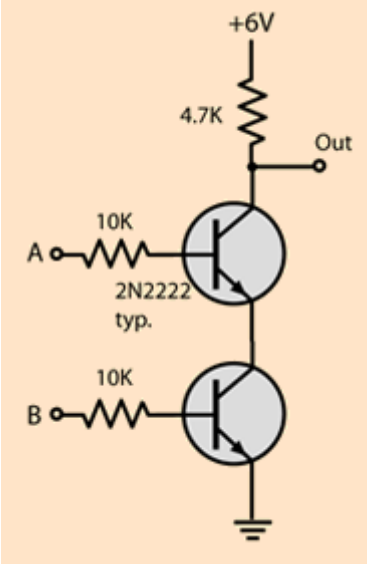


Fig-1(NAND Gate from transistor)[1]

Its truth table (Fig-2) can be verified by above circuit in Fig1

Input	Input	Output
A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

Fig-2 (NAND Gate truth table)

Likewise, every other IC we will use in this project will be made up of gates which in turn are made up of transistor[2].

B. Why 8-bit computer?

An 8-bit computer is an excellent project for a subject like Digital Electronics because it brings together fundamental concepts of digital logic, sequential circuits, memory organization, and data processing in a hands-on and visual way. By building an 8-bit computer from scratch—using components like flip-flops, logic gates, multiplexers, and memory chips—students gain practical experience in how real digital systems operate. It transforms abstract topics like binary arithmetic, clock pulses, instruction cycles, and bus systems into tangible, observable processes. Moreover, it provides a clear and simplified model of how all computers fundamentally work, making it a perfect bridge between theoretical learning and real-world applications.
Analogously, an 8-bit computer serves as a miniature version of modern computers. While today's computers operate on 64-bit architectures with highly complex CPUs and massive memory systems, the core principles remain the same: fetch-decode-execute cycles, data buses, ALUs, registers, and control units. Understanding how these elements interact in an 8-bit machine gives a strong foundational insight into how modern processors work—just on a smaller, simpler scale. This grassroots understanding helps demystify the inner workings of computers, making it easier to comprehend more advanced topics like microprocessor design, system architecture, and embedded systems. Ultimately, the project encourages critical thinking, problem-solving, and system-

C. Modules of an 8-bit computer-

1. Clock -

The **clock module** is a crucial component of an 8-bit computer, responsible for synchronizing all operations by generating a steady pulse signal. It determines the speed at which the computer processes instructions by providing timing signals
between memory and the ALU. Registers operate much faster than memory (RAM) because they are directly integrated into the CPU, reducing delays in data access.

2. Registers -

Registers are small, high-speed storage units within a computer's central processing unit (CPU) that temporarily hold data and instructions during processing. In an **8-bit computer**, registers are designed to store and manipulate **8-bit values**, playing a crucial role in executing instructions efficiently.

The most important register is the **accumulator (A)**, which stores intermediate results from arithmetic and logical operations performed by the arithmetic logic unit (ALU).

[1] hyperphysics.phy-astr.gsu.edu/

[2] We will not use the transistor to make our circuits instead we will be using the suitable IC as this will led to wastage of time and efforts considering the magnitude of this project.

3. ALU -

The **Arithmetic Logic Unit (ALU)** is a fundamental component of an **8-bit computer**, responsible for performing arithmetic and logical operations on data. As part of the CPU, the ALU executes essential tasks such as **addition, subtraction, bitwise operations (AND, OR, XOR, NOT), and comparison operations**, all of which are crucial for processing instructions. It operates on **8-bit values**, meaning it can handle numbers ranging from **0 to 255 (unsigned)** or **-128 to 127 (signed, using two's complement representation)**.

4. RAM -

RAM is a temporary storage unit used by an 8-bit computer to hold data and instructions that are currently being used or processed. It allows quick read and write access, enabling the CPU to retrieve and store information efficiently during program execution. Unlike ROM, RAM is volatile, meaning its data is lost when power is turned off.

5. Program Counter –

The Program Counter is a specialized register in an 8-bit computer that keeps track of the address of the next instruction to be executed from memory. After each instruction is fetched, the Program Counter automatically increments, ensuring a smooth sequential flow of the program. It plays a crucial role in the fetch-decode-execute cycle, as it tells the control unit where to find the next instruction. In cases of jumps, branches, or subroutine calls, the PC is updated to a new address, allowing non-linear execution of code.

6. Output Register –

The Output Register acts as a bridge between the CPU and external output devices like LEDs, seven-segment displays, or serial interfaces. It holds processed data from the ALU or memory that is intended to be shown or transmitted to the outside world. This register ensures that the output remains stable and accessible until the next value is written. In an 8-bit computer, this register typically drives visual indicators and can be connected directly to hardware for real-time feedback of computation results.

7. Control Logic-

Control Logic is the central component that manages and orchestrates all operations within the 8-bit computer. It interprets the binary instructions fetched into the Instruction Register and generates a series of control signals to activate the appropriate hardware components at the right time. These signals coordinate tasks like data transfers between registers, ALU operations, memory reads/writes, and instruction sequencing. The control logic can be implemented using finite state machines, microcode stored in EEPROM, or hard-wired logic, depending on the computer's design. It ensures each instruction is executed accurately and efficiently, step by step.

We will briefly discuss about all these modules in the upcoming subsequent section.

D. Analogy with Modern Computer-

1. Fetch-Decode-Execute Cycle

- 8-bit Computer:
  The CPU follows a simple fetch-decode-execute cycle. It fetches instructions from ROM, decodes them using control logic, and executes them through the ALU or by interacting with memory.
- Modern Computer:
  Modern CPUs still use the same cycle, though optimized with features like pipelining, instruction prefetch, and branch prediction. The principle remains the same: instructions are fetched, decoded, and executed in sequence or parallel.

2. Arithmetic Logic Unit (ALU)

- **8-bit Computer:**
  A basic ALU performs 8-bit operations like addition, subtraction, AND, OR, etc., often built with ICs like the 74LS181.
- **Modern Computer:**
  ALUs are embedded in each core of a multi-core CPU, performing 64-bit (or higher) operations, vector processing, floating-point math, and more.

3. Registers

- 8-bit Computer:
  Includes a few general-purpose registers, an accumulator, a program counter (PC), and an instruction register (IR), often implemented using flip-flops or register chips.
- Modern Computer:
  Modern CPUs have a large set of registers (x86 has 16 general-purpose registers, ARM has 31+), including special ones like stack pointers, instruction pointers, and flags.

4. Program Counter (PC)

- 8-bit Computer:
  The PC holds the address of the next instruction and increments after each fetch.
- Modern Computer:
  The same concept exists—called the Instruction Pointer (IP) in x86 architecture. It's central to control flow, branching, and instruction tracking.

5. Control Logic

- 8-bit Computer:
  Built using combinational logic or microcode in EEPROM to decode opcodes and issue control signals to all components.
- Modern Computer:
  Modern CPUs have complex control units managing micro-operations, pipelining, out-of-order execution, and more, but still based on opcode decoding.

6. Memory (RAM & ROM)

- 8-bit Computer:RAM stores running data; ROM stores hardcoded instructions or microcode. Address and data buses are typically 8-bit and limited to 256 bytes or a few KB.
- Modern Computer:
  RAM is volatile high-speed memory (GBs), while ROM has evolved into firmware like BIOS/UEFI or embedded bootloaders in flash memory.

7. Output Register & I/O

- 8-bit Computer:
  The output register stores data to be shown on LEDs or displays. I/O is typically memory-mapped or port-mapped and very basic.
- Modern Computer:
  Output goes through video cards, sound cards, USB controllers, etc., using protocols like HDMI, PCIe, and USB—but at their core, still rely on output registers and memory-mapped I/O.

8. Clock Circuit

- 8-bit Computer:
  Provides timing pulses using a 555 timer or oscillator. May allow manual stepping for debugging.
- Modern Computer:
  High-frequency crystal oscillators drive GHz-level clock speeds. CPUs use PLLs (phase-locked loops) for dynamic frequency scaling.

In a modern computer, everything is miniaturized, integrated, and optimized, but it still boils down to a CPU processing, instructions via a control unit, using registers and ALUs, reading/writing memory, and interfacing with I/O—all of which are visible and manageable in an 8-bit computer. By building or simulating an 8-bit computer, students get a clear, visual model of how all components collaborate in harmony—something that's hard to see in today's multi-layered architectures. It builds intuition for debugging, writing efficient code, designing digital systems, and understanding hardware/software interaction from the ground up.

The CPU is the part of a computer that executes instructions, The Central Processing Unit (CPU) is the "brain" of a computer. It performs all the computations, logic operations, and manages the flow of data through a system.

Inside a CPU, **wiring** is microscopic and modern CPUs can have **12 to 15+ metal layers ,24 cores, over 20 billion transistors.**

Fun fact: If you could stretch all the microscopic wires inside a modern CPU end-to-end, they'd span **kilometers** in total length!

You might be wondering, *how are billions of transistors and microscopic wires packed into such a tiny chip?*

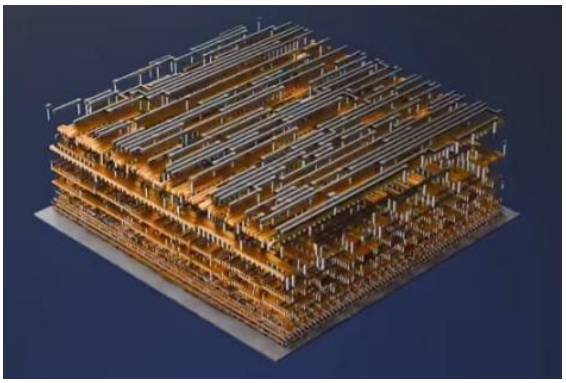


Fig-3 (Metal Interconnects in a cpu core)

CPUs are made from silicon the second most abundant element on earth using semiconductor fabrication, built on silicon wafers using photolithography, where billions of transistors are etched in microscopic detail.

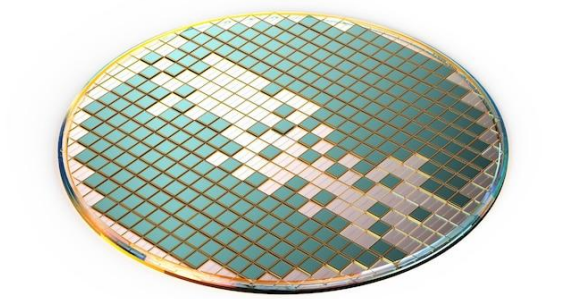


Fig-4 (Silicon Wafer with chips)

The manufacturing process begins with a silicon wafer, which serves as the base for building integrated circuits (see Fig-4). Each wafer serves as the base on which **thousands of CPU dies** will be built with nanoscopic transistors (see Fig-5).

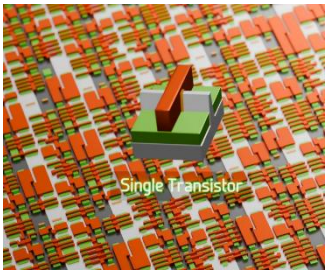


Fig-5 (Transistor)

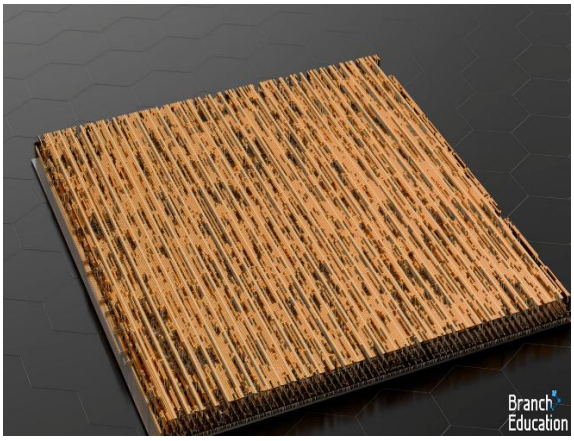


Fig-6 (Masked connection by photolithography*)*

Figure-6 illustrates the astonishing precision of photolithography—one of the most critical steps in CPU manufacturing. As shown, ultraviolet light passes through a **photomask**, projecting complex circuit patterns onto a silicon wafer coated in light-sensitive **photoresist**. This process is repeated **layer by layer**, with each pattern aligned down to the nanometer scale.

Steps:

1.Coating with Photoresist: A light-sensitive chemical layer is applied.

2.Exposure: Ultraviolet (UV) light shines through a **photomask**, which has the circuit patterns.

3.Developing: Light-exposed photoresist is washed away, leaving the pattern.

4.Etching: The exposed silicon is etched away using plasma or chemicals.

5.Photoresist Removal: Remaining photoresist is stripped off.

Zooming out a single core will look like the Figure-7

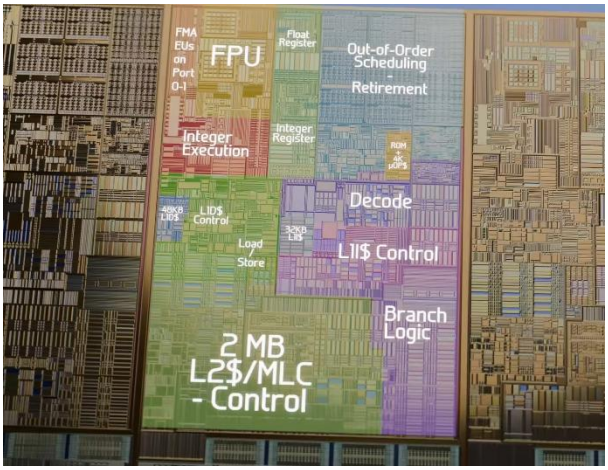


Fig-7 (CPU core die shot)

Here (Figure-6) shows the different parts inside a CPU core and how they work together. Each block represents a part of the chip that has a special job. For example, the **FPU** handles complex math, while the **Integer Execution** unit deals with regular number calculations. The **Load/Store** block helps move data in and out of memory, and the **L1 and L2 caches** are small, fast memory areas that store data the CPU uses often. Other parts like **Decode** and **Branch Logic** help understand instructions and decide what the CPU should do next. Everything works together to make sure the CPU runs programs quickly and efficiently.

In conclusion, the 8-bit computer acts as a simplified mirror of modern computing systems, revealing the foundational architecture that underpins even the most advanced processors today. By exploring its components—such as the ALU, registers, control unit, memory, and clock—we gain direct insight into how modern computers process information, albeit on a much larger and faster scale. This analogy bridges the gap between theory and real-world application, helping us understand complex digital electronics concepts through hands-on experimentation and visual logic. Ultimately, studying an 8-bit computer cultivates a deep, intuitive understanding of how computers work at the hardware level, making it an invaluable project for grasping the inner workings of modern digital systems.

"Let us now examine each unit of the 8-bit computer in detail to understand how they work together to execute instructions."

1. Clock Module-

The clock module acts as the heartbeat of the computer, ensuring that each operation (like fetching, decoding, or executing an instruction) occurs at the right moment.

Components used - 555 Timer IC, Capacitor, Potentiometer, Push Button, Switch, Logic Gates

555 Timer:

The LM555 is a highly stable device for generating accurate time delays or oscillation. Additional terminals are provided for triggering or resetting if desired. In the time delay mode of operation, the time is precisely controlled by one external resistor and capacitor. For a stable operation as an oscillator, the free running frequency and duty cycle are accurately controlled with two external resistors and one capacitor. The circuit may be triggered and reset on falling waveforms, and the output circuit can source or sink up to 200 mA or drive TTL circuits.

PIN diagram-

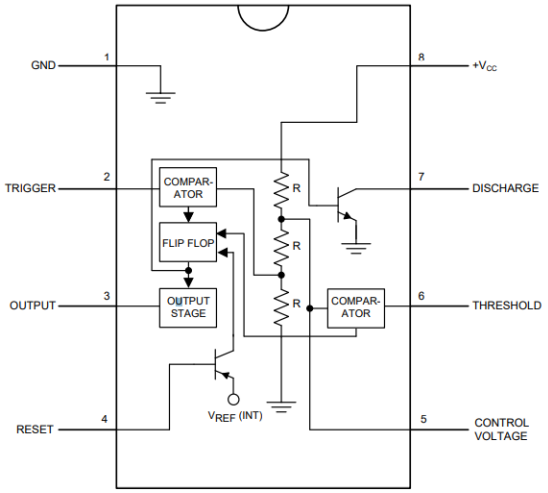


Fig-9 (Internal Circuit 555 timer)

Pin		DESCRIPTION
NO.	NAME	
1	GND	Ground Reference Voltage
2	Trigger	Responsible for transition of SR flip-flop
3	Output	Output driven waveform
4	Reset	A negative pulse on reset will disable or reset the timer
5	Control Voltage	Controls the width of the output pulse by controlling the threshold and trigger levels
6	Threshold	Compares the voltage applied at the terminal with a reference voltage of 2/3
7	Discharge	Connected to open collector of a transistor which discharges a capacitor between intervals.
8	V_{CC} Supply	Supply voltage

Fig-10(PIN configuration 555)

Working –

The 555 timer IC generally operates in 3 modes:

1. Astable Mode - In Astable mode(Fig-11), the 555 timer continuously switches between HIGH and LOW states on its output pin (Pin 3), creating a square wave (digital clock pulses). There is no stable state—the timer is always oscillating.
This mode is ideal for creating a free-running clock signal to drive the operations of a digital system like the 8-bit computer.

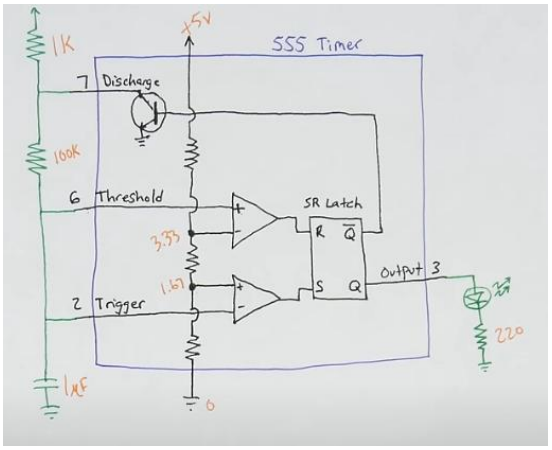


Fig-11(Astable Configuration)

The oscillation is driven by the charging and discharging of a capacitor through two resistors. Initially, the capacitor charges through both resistors R1 and R2, and while its voltage is below 2/3 of the supply voltage (Vcc), the output remains HIGH. Once the voltage reaches 2/3 Vcc, the internal threshold comparator activates, turning on a discharge transistor connected to Pin 7. This causes the capacitor to discharge through R2, and the output switches to LOW. When the capacitor's voltage drops below 1/3 Vcc, the trigger comparator resets the output to HIGH again, and the capacitor begins charging. This repeating cycle of charge and discharge generates a continuous square wave at the output.

Waveform-

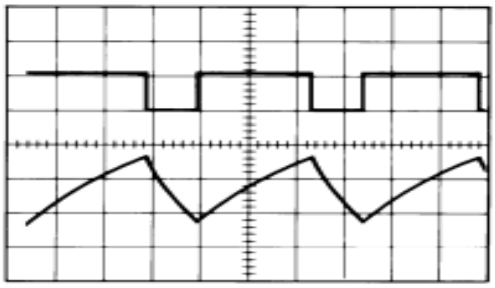


Fig-12(Astable Configuration Waveform)

Charging and Discharging Time –

The charge time (output high) is given by:
$t_1 = 0.693 \times (R1 + R2) \times C$
And the discharge time (output low) by:
$t_2 = 0.693 \times R2 \times C$

The frequency of oscillation is:

f = 1 / T = 1.44 / [(R1 + 2 × R2) × C]

The duty cycle is:

D = R2 / (R1 + 2 × R2)

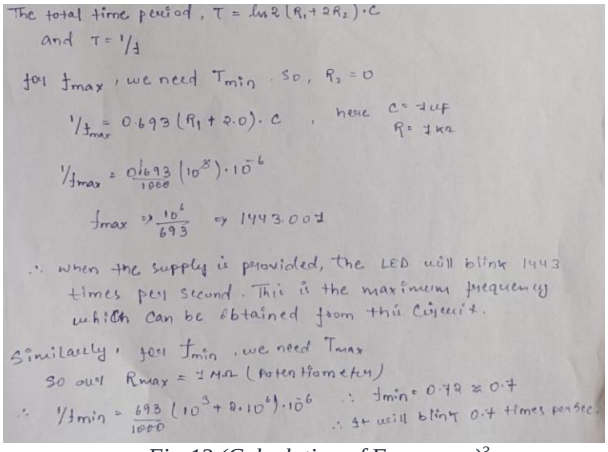


Fig-13 (Calculation of Frequency)[3]

Recommendations –

1. Connect the pin 5 of the 555 timer via capacitor to ground so that the noise when the output pulse is transitioning from 0v to 5v can be removed.

Fig-14 ()

2. Connect the pin 4 of the 555 timer always to +5v, If you **don't** connect it to +5V (and just leave it floating), electrical noise or interference could accidentally pull it low That would randomly **reset your timer circuit**, causing erratic behavior.

3. To control the speed of the clock pulses, a potentiometer is added to the circuit. By adjusting its resistance, the frequency of the clock can be slowed down for debugging or sped up for normal operation.

Connect:
1.one outer terminal of the potentiometer to Vcc.
2.The middle wiper pin to Pin 7 (Discharge) of the 555 timer.
3.The other outer terminal to R2, which then goes to Pins 6 and 2 and then to capacitor C to ground.

2. Monostable Mode - In monostable mode, the 555 timer produces a single output pulse of fixed duration in response to a trigger signal, rather than generating continuous pulses like in astable mode. This mode is useful in the 8-bit computer clock module when precise, manual control of clock pulses is needed. For example, when debugging or stepping through each instruction cycle, the monostable configuration allows the user to press a push button to generate one clean clock pulse at a time.

Thus the total period is:
$T = t_1 + t_2 = 0.693 \times (R1 + 2 \times R2) \times C$
trigger pin (Pin 2). Once triggered, the output (Pin 3) goes HIGH and stays high for a fixed time determined by the resistor and capacitor connected to the timer. After this time, the output automatically returns to LOW, ready for the next trigger. The duration of the output pulse is given by the formula:
T = 1.1 × R × C,
where R is the resistor and C is the capacitor used in the timing circuit.
You might be thinking, "Why not simply use a push button and an LED to control the clock manually?"
When we press a push button, two internal metal contacts come together to close the circuit and send a signal. However, due to the mechanical nature of the button, those contacts can bounce—meaning they rapidly touch and release several times before settling. This results in multiple quick pulses instead of a single clean one. So, during debugging, when we only need one pulse to step through and observe what's happening in the 8-bit computer, the button might accidentally generate multiple pulses. This can cause the system to skip steps or behave unpredictably, which can be frustrating and make it difficult to identify the problem accurately.

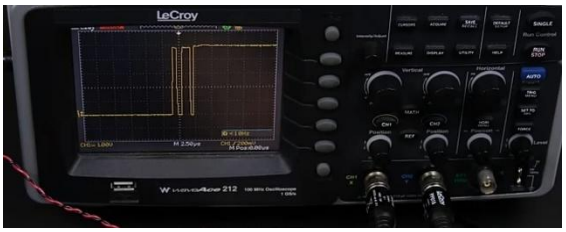


Fig-15 (Bouncing when a switch is pressed)

Here the button is pressed one time but due to **bouncing**, we are getting multiple pulses. So to make it a debouncing circuit we will use 555 timer.

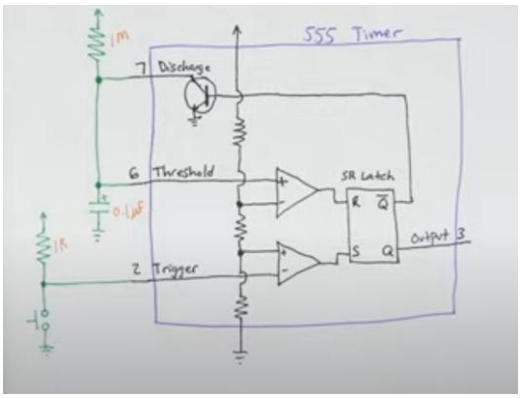


Fig-16 (Monostable Configuration)

contacts inside the button may rapidly make and break contact several times in just a few milliseconds. Each of

[3] The maximum frequency our 8-bit computer will achieve with these components will be 1449.007 Hz. While the modern computer have frequencies in GHz.

In this setup, the 555 timer remains in a stable LOW state until it is triggered by a negative pulse at the voltage across it rises exponentially from 0V. While this voltage is below 2/3 of the supply voltage (Vcc), the output (Pin 3) stays HIGH. This high output state continues for a fixed duration, determined by the time constant of the resistor-capacitor combination, calculated by the formula T = 1.1 × R × C. Once the capacitor's voltage reaches 2/3 Vcc, the threshold comparator resets the flip-flop, switching the output back to LOW. At the same time, the discharge transistor turns back on and quickly discharges the capacitor to ground, preparing the circuit for the next trigger

Waveform-

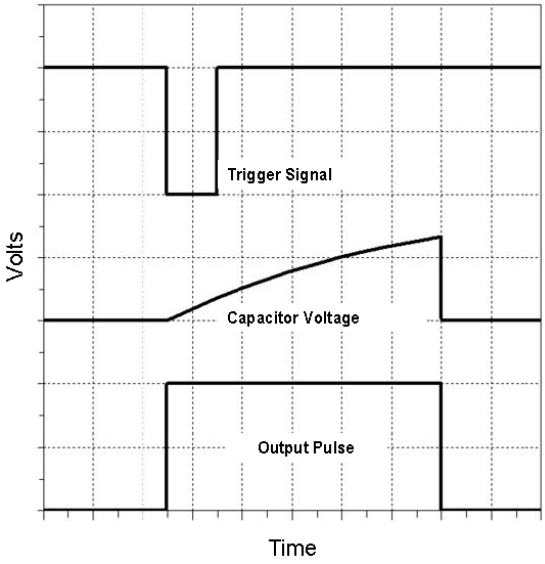


Fig-16 (Monostable Configuration Waveform)

Recommendations –

1.Connect the pin 5 of the 555 timer via capacitor to ground so that the noise when the output pulse is transitioning from 0v to 5v can be removed.

2.Connect the pin 4 of the 555 timer always to +5v, If you don't connect it to +5V (and just leave it floating), electrical noise or interference could accidentally pull it low That would randomly reset your timer circuit, causing erratic behavior.

3. Bistable Mode - In bistable mode, the 555 timer functions as a flip-flop, meaning it has two stable states—HIGH and LOW—and it stays in either state until it receives an external trigger to switch. Unlike astable or monostable modes, there is no automatic timing or oscillation involved. One input (usually Pin 2) is used to set the output HIGH, while another input (usually Pin 4 or Pin 6 depending on the configuration) is used to reset the output to LOW. In the context of a clock module, the bistable mode is not commonly used to generate clock pulses directly, but it can be useful as a manual toggle switch—for instance, turning the clock on and off, enabling or disabling pulse generation, or controlling the mode (manual vs automatic) of the clock.

A bistable circuit can be built using a 555 timer, a pair of push buttons, and an LED but like in monostable, bouncing also happens here because when you press a push button to set or reset the output in a bistable 555 timer, the mechanical

these little bounces can be interpreted by the 555 timer as multiple pulses (Refer figure-15)**.**
In monostable mode, the 555 timer remains in a stable LOW output state until it is triggered by a negative pulse at Pin 2. When triggered, the internal flip-flop is set, turning off the discharge transistor connected to Pin 7 and allowing the timing capacitor (C) to begin charging through a resistor (R). As the capacitor charges, the

trigger/reset signals, causing the output to flicker or behave unpredictably.

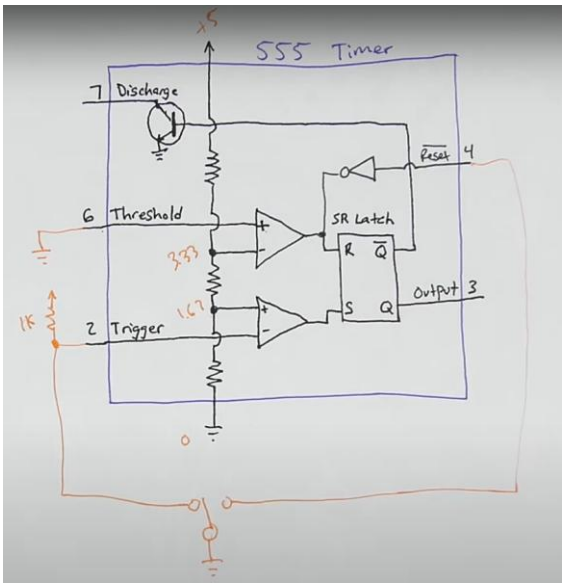


Fig-17 (Bistable Configuration)

In bistable mode, the 555 timer does not rely on any capacitor charging or discharging to control the output timing as it does in monostable or astable modes. Instead, it operates like a flip-flop with two stable voltage states: HIGH and LOW. When a LOW (0V) signal is applied to the trigger pin (Pin 2), the lower comparator sets the internal flip-flop, causing the output (Pin 3) to go HIGH. In this state, the output remains HIGH indefinitely, as long as there is no input on the reset pin. When a LOW signal is applied to the reset pin (Pin 4), the upper comparator resets the flip-flop, and the output switches to LOW. The voltage at the output changes immediately without depending on capacitor charging or discharging, which makes the bistable mode different from the timing-based operations of other modes.

Waveform-

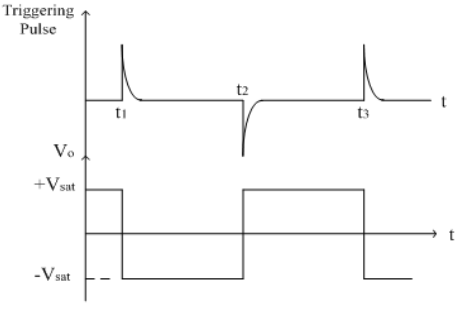


Fig-18 (Bistable Configuration Waveform)

Circuit for the desired signals is:

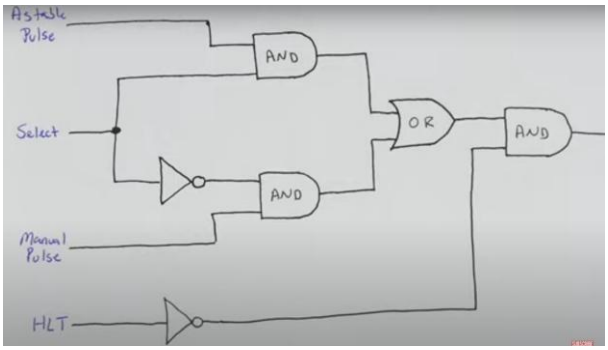


Fig-19 (Circuit to control output Clock which will be fed to the all the parts of computer)

Logic Flow:

1. Select Line:

Directly feeds one AND gate (with the Astable Pulse).

Goes through a NOT gate (inverter) before going to the other AND gate (with the Manual Pulse).

This means:

If Select = 1 → Astable Pulse is enabled.
If Select = 0 → Manual Pulse is enabled.

2. HLT Line:

Also goes through a NOT gate before entering the final AND gate.

If HLT = 1 → NOT(HLT) = 0 → disables output (halts it).

If HLT = 0 → NOT(HLT) = 1 → allows output.

3. AND-OR-AND Structure:

Two AND gates for selecting between Astable and Manual Pulse (depending on Select).

Their outputs go to an OR gate, combining the selected pulse.

Output of OR gate goes to final AND gate, which is controlled by NOT(HLT).

Now what we need to do is to combine all three signals to make a single clock and we want to either output Astable signal or Manual signal depending on which one is selected.

Truth table:

Select	HLT	Output Source	Output Enabled ?
1	0	Astable Pulse	Yes
0	0	Manual Pulse	Yes
1	1	Astable Pulse	No
0	1	Manual Pulse	No

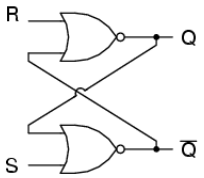
Fig-20 ()

2. Registers -

Since we're using **74LS173** ICs—which are **quad D-type flip-flops with 3-state outputs and common clock**—it's a great idea to **first explain the foundational concepts**: **SR latch → D latch → D flip-flop**, and then connect them to how the **74LS173** works to store 4-bit values in our 8-bit computer.

SR Latch[4]-

A bistable multivibrator has *two* stable states, as indicated by the prefix *bi* in its name. Typically, one state is referred to as *set* and the other as *reset*. The simplest bistable device, therefore, is known as a *set-reset*, or S-R, latch. To create an S-R latch, we can wire two NOR gates in such a way that the output of one feeds back to the input of another, and vice versa, like this(Figure-21):

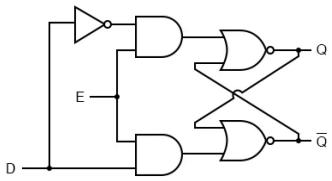


S	R	Q	$\overline{Q}$
0	0	latch	latch
0	1	0	1
1	0	1	0
1	1	0	0

Fig-21 (SR-Latch using NOR gates)

D Latch-

Since the enable input on a gated S-R latch provides a way to latch the Q and not-Q outputs without regard to the status of S or R, we can eliminate one of those inputs to create a multivibrator latch circuit with no "illegal" input states.
Such a circuit is called a D latch, and its internal logic looks like this(Figure-22):



E	D	Q	$\overline{Q}$
0	0	latch	latch
0	1	latch	latch
1	0	0	1
1	1	1	0

Fig-22 (D-Latch using NOR gates)

Note that the R input has been replaced with the complement (inversion) of the old S input, and the S input has been renamed to D. As with the gated S-R latch, the D latch will not respond to a signal input if the enable input is 0—it simply stays latched in its last state. When the enable input is 1, however, the Q output follows the D input.
Since the R input of the S-R circuitry has been done away with, this latch has no "invalid" or "illegal" state. Q and not-Q are *always* opposite of one another.

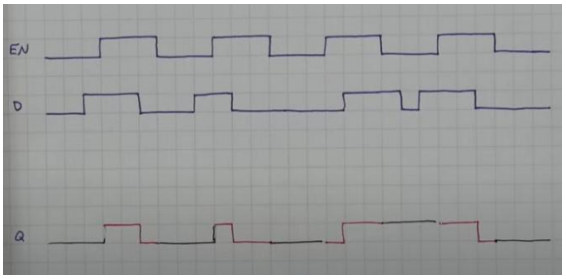


Fig-23 (D-Latch waveform with EN)

D Flip Flop[7]-

The only difference here will be at place of enable we will add edge-triggered enable as shown in Figure-24

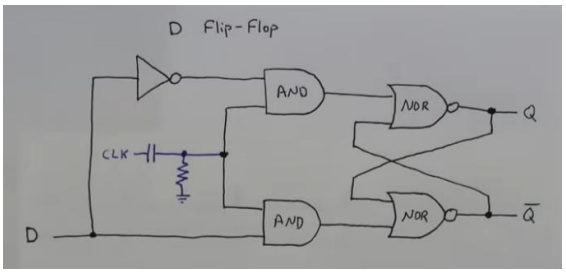


Fig-24 (D-Flipflop)

And the related waveform can be seen in Figure-25

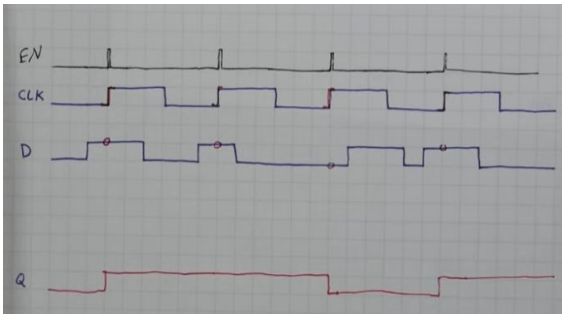


Fig-25 (D-Flipflop waveform with EN)

Now let's break down **how to use the 74LS173 IC to store data** in your 8-bit computer. Since each 74LS173 can store **4 bits**, we'll need **two 74LS173 chips** to store a full **8-bit value** in a register like A, B, or IR.

The **74LS173** is a quad D-type flip-flop integrated circuit used to store 4 bits of data, commonly employed in 8-bit computer designs by pairing two of these chips. Each 74LS173 chip features four data inputs (1D to 4D), four corresponding outputs (1Q to 4Q), and control pins that determine when data should be loaded or cleared.

[4] A latch is level-triggered, meaning it's responsive to input signals when an "enable" input is active—during the logic-high pulse of a clock signal, for example.
[7] A flip-flop is edge-triggered, meaning that it's responsive to input signals when an "enable" input changes state—for example, at the rising edge of a clock signal.

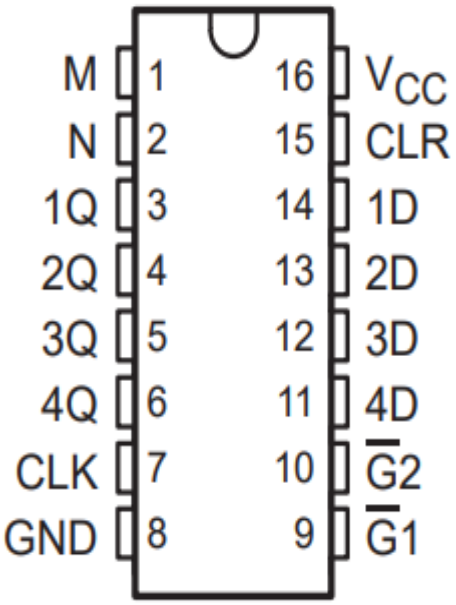


Fig-26 (74LS173)[5]

The key to storing data in the 74LS173 lies in the clock input (CLK) and two active-low load enable pins (G1 and G2). When both G1 and G2 are held low (can be seen at Figure-28(in Figure we considered the load as active high while the IC has active low)), and a rising edge is applied to the CLK pin, the data present on the D inputs is latched into the flip-flops and becomes available on the Q outputs. These outputs represent the stored bits and can be connected to other components in the computer, such as the data bus or LEDs for visualization.

To ensure stable operation, the clear pin (CLR), which is also active-low, should be held high during normal use. If CLR is pulled low, it instantly resets all stored bits to zero regardless of the clock or enable inputs. The chip is powered by connecting Vcc (pin 16) to a 5V supply and GND (pin 8) to ground. In the context of an 8-bit computer, two 74LS173 chips are used together—each handling 4 bits—to form a full 8-bit register. For example, when implementing general-purpose registers like A and B or the instruction register (IR), each register would use one 74LS173 for the upper nibble (4 bits) and another for the lower nibble. The synchronized clock ensures that both halves of the 8-bit value are stored simultaneously, allowing the CPU to process full bytes of data efficiently.

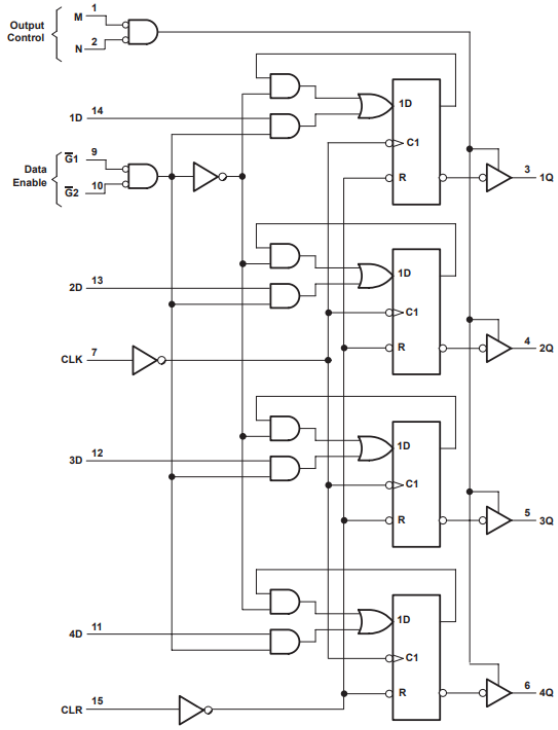


Fig-27 (Internal Circuit 74LS173)

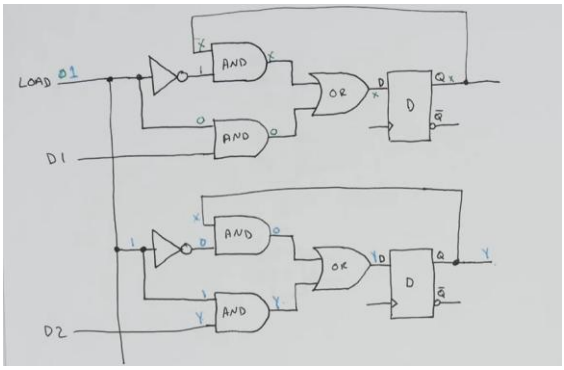


Fig-28 (Load [G2+G1]) working of 74LS173)

INPUTS					OUTPUT Q
CLR	CLK	DATA ENABLE $\overline{G1}$	DATA ENABLE $\overline{G2}$	DATA D	
H	X	X	X	X	L
L	L	X	X	X	Q_0
L	↑	H	X	X	Q_0
L	↑	X	H	X	Q_0
L	↑	L	L	L	L
L	↑	L	L	H	H

When either M or N (or both) is (are) high, the output is disabled to the high-impedance state; however, sequential operation of the flip-flops is not affected.

Fig-29 (Function table of 74LS173)

[5] Here both M and N will be tied to low i.e. what we will be receiving at D will directly appears at Q; the reason behind doing this is we will control the enable function for these chips by a Tri-state buffer.

In our simple 8-bit breadboard CPU, three primary registers—A, B, and IR (Instruction Register)—play crucial roles in the operation of the processor. These are all 8-bit registers, meaning they can store values ranging from 0 to 255 (unsigned), or -128 to 127 (signed, using two's complement representation). Each of these registers has a specific function and works closely with the Arithmetic Logic Unit (ALU), control logic, and memory.

The A Register (Accumulator)-

The **A register**, also known as the **accumulator**, is perhaps the most important general-purpose register in a simple CPU. It serves as the default location for storing the result of operations performed by the ALU. For instance, when the CPU adds two numbers together, one of the operands usually resides in the A register, and the result is often placed back into the A register itself.
In a minimalistic 8-bit computer, most instructions are designed to operate on the accumulator. This simplifies the instruction set and the design of the control logic. For example, instructions like ADD B would mean "add the value in the B register to the value in A, and store the result in A." The accumulator reduces complexity by centralizing many operations into a single register. It works as the go-to location for arithmetic, logic, data manipulation, and even some control operations. Because it interfaces directly with the ALU, the A register is often wired such that its contents can be passed to or from the ALU with minimal control signaling. This also makes it easier to observe and debug when building or testing the CPU.

The B Register (General-Purpose Operand Register)-

The **B register** serves as a **secondary general-purpose register**, typically used as the second operand in operations involving the ALU. While the accumulator (A) might hold one value, the B register would hold the other, and together they form the input pair for the ALU to process.
For example, in the instruction SUB B, the value in the B register is subtracted from the value in the A register, and the result is stored back in A. This method of storing operands in A and B simplifies the design and makes the instruction set straightforward to implement.
The B register is also useful for temporary data storage and movement. For instance, data fetched from memory may be temporarily loaded into the B register before being operated on or transferred elsewhere. Its general-purpose nature allows it to act as a buffer or staging area in multi-step instructions.
Another benefit of having a separate B register is to allow pipelined or staged execution. While one operation is being executed on the current contents of A and B, the next value can be fetched into B without affecting A. This concept becomes especially valuable in more advanced CPUs but is also a practical benefit in breadboard computers.

The IR Register (Instruction Register)-

The **Instruction Register (IR)** holds the **current instruction** being executed by the CPU. It is a specialized register distinct from the general-purpose registers, designed solely for control and sequencing purposes.
When the CPU begins an instruction cycle, it fetches an instruction from memory using the **Program Counter (PC)** to know where to look. Once the instruction is fetched, it is stored in the IR. From there, the control unit decodes the bits in the IR to determine what operation needs to be performed and which other components—such as the ALU, registers, or memory—need to be activated.
The IR plays a pivotal role[8] in the **fetch-decode-execute cycle**:

1. **Fetch:** The CPU retrieves the instruction from memory.
2. **Decode:** The bits in the IR are interpreted by the control unit.
3. **Execute:** Signals are sent to the necessary components (e.g., ALU, A/B registers, RAM) to carry out the instruction.

In simple 8-bit CPUs, the IR is typically an 8-bit register that, stores both the operation code (opcode) and, in some designs, a small part of the operand or address. This simplifies control logic and helps the CPU execute instructions efficiently with minimal decoding overhead.

[8] This will play a crucial role in control logic handling.

3. ALU -

In an 8-bit computer, the Arithmetic Logic Unit (ALU) is a crucial component responsible for performing basic arithmetic operations on 8-bit binary numbers. It typically takes two inputs—one from the A register and one from the B register—and outputs the result of operations such as addition and subtraction.

Components used – 74LS83, 74LS86, 74LS273

To understand how subtraction works in binary, especially within an 8-bit computer, it's essential to learn about two's complement, which is the standard method used in digital systems to represent negative numbers. Instead of designing a separate circuit for subtraction, most computers use two's complement to convert subtraction into addition, which simplifies the design of the Arithmetic Logic Unit (ALU). The key idea is that subtracting a number is the same as adding its negative. For example, to perform A - B, we convert B into its negative form using two's complement, and then add it to A. The process of finding the two's complement of a number involves two steps: first, invert all the bits of the number (turn 0s into 1s and 1s into 0s), and second, add 1 to the result. This new value is the negative equivalent of the original number in binary. For instance, if we want to subtract 5 from 10, we first write 5 as 00000101 in 8-bit binary. Then, we invert it to get 11111010, and finally add 1 to get 11111011, which is -5 in two's complement. We can now add this to 10 (00001010) using an adder circuit, and the result is 00000101, which is 5—the correct answer for 10 - 5. This approach makes subtraction efficient and allows the ALU to handle both addition and subtraction using the same logic circuitry. It's a fundamental concept that not only simplifies hardware design but also enables handling of both positive and negative numbers in binary systems.

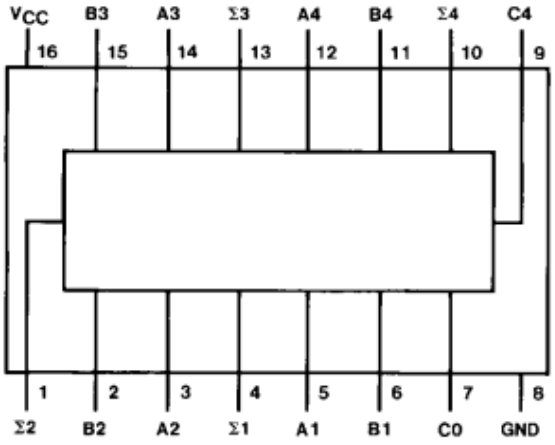


Fig-30 (Function table of 74LS283)

In our 8-bit computer, to perform addition and subtraction using binary numbers, we will be cascading two 4-bit adders to build an 8-bit Arithmetic Logic Unit (ALU). Since each 4-bit adder can only handle 4 bits at a time, we connect two of them in series—one for the lower 4 bits and one for the upper 4 bits. The carry-out from the lower 4-bit adder is connected to the carry-in of the upper 4-bit adder, allowing proper calculation across all 8 bits.

	A				B				Sum				Carry
Cin	A3	A2	A1	A0	B3	B2	B1	B0	S3	S2	S1	S0	Cout
0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	1	0	0	1	0	0
0	0	0	1	0	0	0	1	0	0	1	0	0	0
0	0	0	1	1	0	0	1	1	0	1	1	0	0
0	0	1	0	0	0	1	0	0	1	0	0	0	0
0	0	1	0	1	0	1	0	1	1	0	1	0	0
0	0	1	1	0	0	1	1	0	1	1	0	0	0
0	0	1	1	1	0	1	1	1	1	1	1	0	0
0	1	0	0	0	1	0	0	0	0	0	0	0	1
0	1	0	0	1	1	0	0	1	0	0	1	0	1
0	1	0	1	0	1	0	1	0	0	1	0	0	1
0	1	0	1	1	1	0	1	1	0	1	1	0	1
0	1	1	0	0	1	1	0	0	1	0	0	0	1
0	1	1	0	1	1	1	0	1	1	0	1	0	1
0	1	1	1	0	1	1	1	0	1	1	0	0	1
0	1	1	1	1	1	1	1	1	1	1	1	0	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1

Fig-31 (Function table of 74LS283)

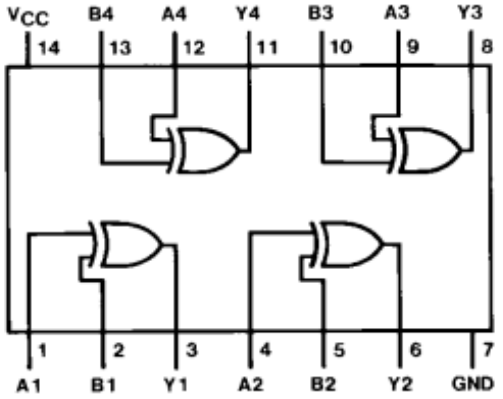


Fig-32 (74LS86)

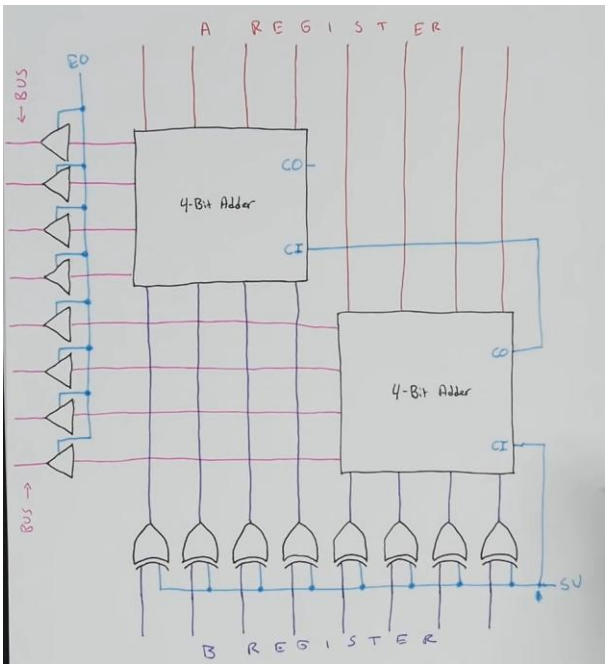
Inputs		Output
A	B	Y
L	L	L
L	H	H
H	L	H
H	H	L

Fig-33(Function table of 74LS86)

In our 8-bit computer, subtraction is performed using the same 8-bit adder circuit that handles addition, but with a smart twist using XOR gates and a control signal called SUBTRACT[9]. This method takes advantage of how two's complement subtraction works, allowing us to subtract by

[9] Each bit of the **B input** is passed through an XOR gate, and the **SUBTRACT signal is connected to the other input of all these XOR gates**.

simply inverting the bits of the B input and adding 1. To do this in hardware, each bit of the B register is passed through an XOR gate, with the SUBTRACT signal as the other input. When SUBTRACT is low (0), the XOR gates pass the B bits unchanged, so the ALU performs normal addition (A + B). But when SUBTRACT is high (1), the XOR gates invert each bit of B (just like the first step of two's complement). Then, the SUBTRACT signal is also connected to the carry-in of the least significant 4-bit adder. This effectively adds the "+1" part of the two's complement, completing the conversion of B to -B. The result is that the adder now calculates A + (-B), which is the same as A - B, without needing a separate subtractor circuit. This technique keeps the ALU simple and compact, while still supporting both addition and subtraction using the same hardware.



4. Program Counter-

The Program Counter (PC) is a small register in a computer's control unit that keeps track of which instruction to execute next. It holds the memory address of the next instruction in RAM.
Think of it like a bookmark in a book—always pointing to the next line to read.

It has a 4-bit program counter, even though the rest of the computer is 8-bit. Here's how it works:

- Holds 4-bit values: So it can address up to 16 memory locations that's same as our storage of RAM.
- It's connected to the address bus, and it tells the RAM which memory location to fetch the next instruction from.
- It can increment automatically (go to the next instruction), or it can load a new value (like during a jump instruction).

Components used – 74LS161, 74LS76, 74LS273

J-K flipflop - A J-K flip-flop is a type of sequential logic circuit that has two inputs, J and K, and two outputs, Q and Q'. It's a modification of the SR flip-flop and addresses the undefined state of the SR flip-flop.

CLK	J	K	Q_{next}	Comment
Rising edge	0	0	Q	Hold state
Rising edge	0	1	0	Reset
Rising edge	1	0	1	Set
Rising edge	1	1	$\overline{Q}$	Toggle
Non-rising	X	X	Q	No change

Fig-35(Truth Table of J-K flip flop)

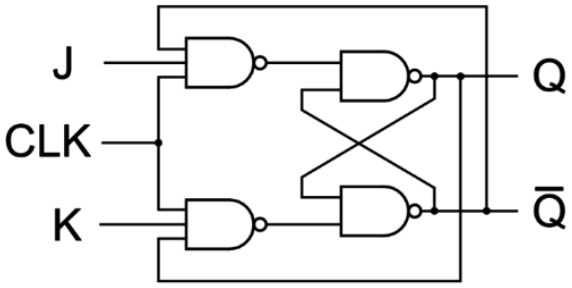


Fig-36(J-K flipflop)

J-K flipflop Racing- J-K Flip-Flop Racing (or Race Condition) refers to an undesirable situation that can occur in J-K flip-flops when both J and K inputs are set to 1 simultaneously. This means that each clock pulse causes the output to change state (from 0 to 1 or from 1 to 0).
If the flip-flop's clock pulse is not clean or has glitches, there could be race conditions where the output toggles unpredictably, leading to an unstable state. The flip-flop might toggle multiple times within a single clock pulse or not behave as expected.

Fig-38 (J-K Flip-Flop racing)

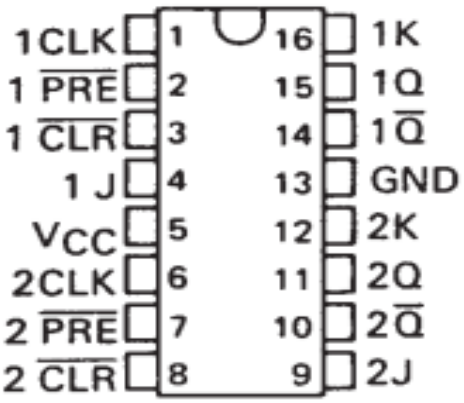


Fig-37 (74LS76)

Master-Slave J-K Flip-Flop - The Master-Slave J-K Flip-Flop configuration is commonly used to prevent race conditions. In this configuration, the master flip-flop is triggered by the clock's rising edge, and the slave flip-flop is triggered by the falling edge, ensuring that the flip-flop changes states only on one clock edge.

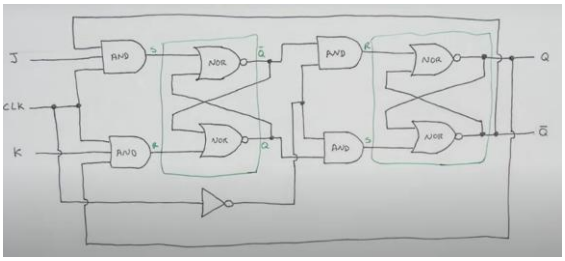


Fig-38(Master-Slave J-K Flip-Flop)

The PC can be made using individual JK flip-flops. JK flip-flops are versatile because, when configured with J = K = 1, they toggle on every clock pulse, making them suitable for constructing simple binary counters.

It can be made by cascading two 74LS76, The Q (pin 15) output of the first flip-flop is connected to the clock input of the second flip-flop. Similarly, the Q output of the second flip-flop is connected to the clock input of the third, and so on, until the fourth flip flop.

But the problem is each flip-flop triggers the next flip-flop's clock. This means the output of each flip-flop is delayed by the propagation time of the previous flip-flop, causing a cumulative delay. The flip-flops do not change their state simultaneously; there is a slight delay between each stage in the counter. For example, the first flip-flop

might toggle on the first clock pulse, the second on the second clock pulse, and so on.

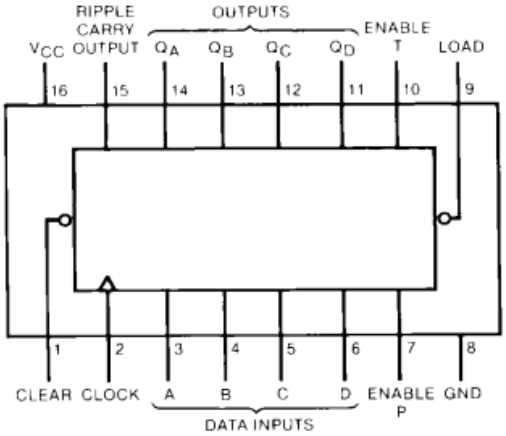


Fig-39(74LS161)

The 74LS161 is a 4-bit synchronous binary counte[6]r IC. It's designed to count from 0 to 15 (0000 to 1111 in binary) and is widely used in digital electronics for timing, counting, and sequencing operations.

Program counter has three signals:

1.Count enable – This signal will enable(pin7) the program counter to increment on the next clock pulse to jump to a different instruction If we don't increment, the CPU would keep fetching the same instruction over and over.

2.Count out – This signal will output the program counter's value to the bus, it allows the system to transfer control from the program counter to memory to fetch the next step.

To Implement the Count out:

1. As the output of 74LS161 is connected to 74LS273.
2. Enable the of 74LS273 to load the output to bus.
3. On the next rising edge of the clock, the memory register loads the new address from the BUS.

3.Jump - The **Jump signal** enables the **load input** of the Program Counter so that it **loads a specific address** (usually from the bus), instead of incrementing.

To Implement the Jump:

4. Put the new address (destination of jump) onto the data bus.
5. Connect 74LS273 to the inputs of the 74LS161.
6. Activate the Jump signal, by connecting LOAD to GND.
7. On the next rising edge of the clock, the PC loads the new address from the BUS.

[6] It will keep the track of which instruction should be executed next

5. RAM –

Random Access Memory (RAM) serves as a temporary storage medium used to hold both the program currently being executed and the data it operates on. It plays a crucial role in enabling the CPU to perform computations by providing fast and direct access to memory locations.

For this 8-bit breadboard computer, the RAM has been designed with a 4-bit address bus, allowing it to address 2^4 = 16 memory locations. Each location can store 8 bits (1 byte) of data, resulting in a total capacity of 16 bytes.

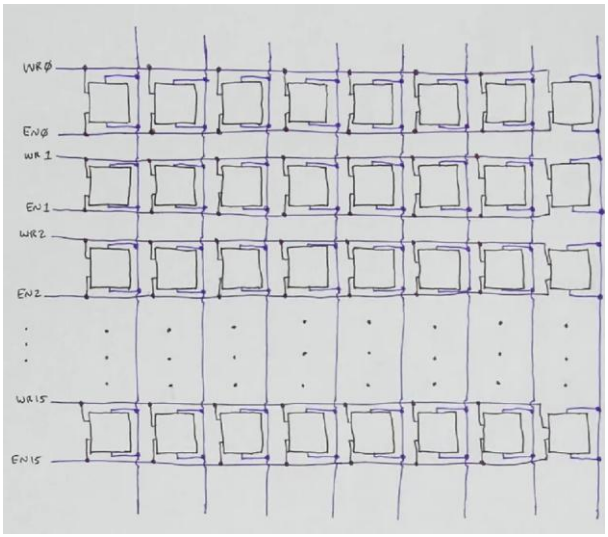


Fig-40(RAM)

The RAM in this computer is constructed using a grid of latching elements (most likely D flip-flops or SR latches), where each latch stores a single bit of data. This custom RAM is organized in a matrix form, with 16 rows (WR0 to WR15) and 8 columns, providing 16 × 8 = 128 bits, or 16 bytes of storage. Each row represents a memory address, and each column corresponds to a bit in the 8-bit data word.

Addressing Mechanism:

- Write Lines (WR0 to WR15): Each row has a dedicated write enable line. When a particular write line is activated (e.g., WR3), the latches in that row are enabled to capture and store the input data bits.
- Enable Lines (EN0 to EN15): These lines are used for reading data. When a particular enable line is activated (e.g., EN3), the contents of that row's latches are allowed to appear on the shared data bus

This two-line system per row (WR and EN) enables selective read and write access to individual memory addresses. Only one row is written to or read from at a time, preventing data corruption or bus conflicts.

To implement the RAM in this 8-bit breadboard computer, the 74LS189 IC was selected. The 74LS189 is a 16-word by 4-bit memory chip, meaning it can store 16 memory locations (addresses), each holding 4 bits. For an 8-bit system, two 74LS189 chips are used in parallel to store the full byte (8 bits) at each memory address.
The address pins for both the ICs will get same input so that both IC should output correct value at a particular address.

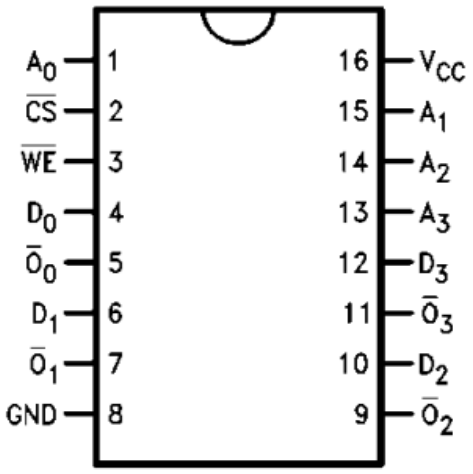


Fig-41(74LS189)

As the output pins are active low so we need to connect the output pins to a NOT gate.

Now, to access data stored at different memory locations, a **Memory Address Register (MAR)** is introduced. The MAR allows the CPU to specify which address in RAM should be accessed for either reading or writing operations.

For this 8-bit breadboard computer, the RAM uses a 4-bit address bus(**74LS173**), which allows for 16 unique memory locations (2^4 = 16). Therefore, the MAR is implemented as a 4-bit register, capable of holding addresses from 0000 to 1111 in binary. The output of this register is directly connected to the address inputs of the RAM (A0–A3), determining which memory location is accessed during read or write operations.

In this design, manual switches are used to set both the memory address and the data manually. This allows the user to directly program the RAM by selecting the desired address via the MAR and inputting the corresponding data through toggle switches. This method is particularly useful for testing and initializing memory contents during development.

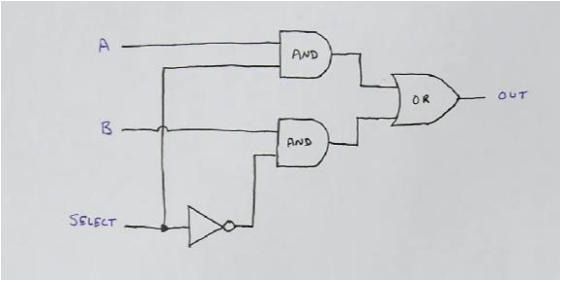


Fig-42(RAM)

To allow the user to choose between manual programming and automatic execution, a push-button switch is used as the control input to a simple logic circuit acting as a 2-to-1 multiplexer. This circuit selects between two sources: manual input (from DIP switches) and automatic input (from the CPU).
When the button is not pressed, the circuit selects the manual input, allowing the user to program data or

addresses using switches. When the button is pressed, the circuit switches to automatic mode, allowing the CPU to control memory access and instruction flow.

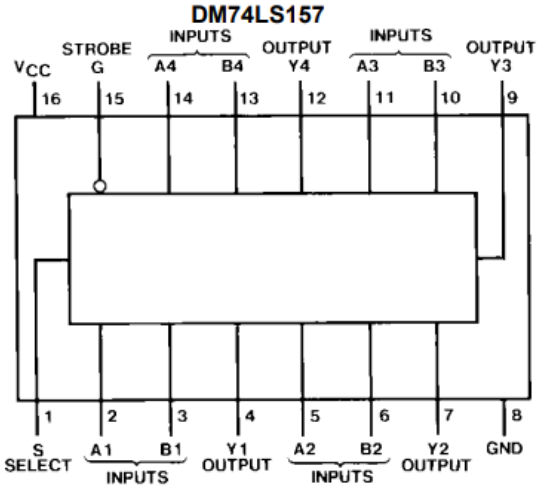


Fig-42(74LS157)

A inputs are connected to DIP switches and B inputs are connected to Bus and as we toggle the push button the mode will switch to either automatic or manual.
The output pins (Y) will be connected to the address pins of the RAM.

To enable manual programming of the RAM, DIP switches are used to input both the memory address and the data. This allows users to directly write values to specific memory locations without needing a microcontroller or CPU to handle the process.

- A 4-way DIP switch is used to select the memory address. Since the RAM uses a 4-bit address space (0 to 15), four switches are sufficient to manually choose any of the 16 memory locations.
- An 8-way DIP switch is used to provide the data input. This allows the user to set any 8-bit value (from 0 to 255) to be written to the selected address.

By combining these DIP switches with a write control button and the Memory Address Register (MAR), the user can manually load data into memory. When the write button is pressed, the system stores the value from the data DIP switch into the memory location selected by the address DIP switch.

6. Output register –

The Output Register is responsible for holding the final result produced by the CPU or the value retrieved from memory, and displaying it to the user. Functionally, it behaves just like other 8-bit registers used in the system, such as the A or B registers. However, unlike those registers—which typically display their contents as binary values using 8 individual LEDs—the output register is connected to a common cathode 7-segment display to present its contents in a more human-readable decimal format.

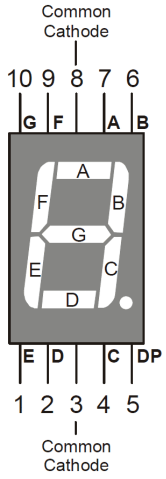


Fig-45(Common Cathode)

Inputs				Segments							
A	B	C	D	a	b	c	d	e	f	g	
0	0	0	0	1	1	1	1	1	1	0	For display 0
0	0	0	1	0	1	1	0	0	0	0	For display 1
0	0	1	0	1	1	0	1	1	0	1	For display 2
0	0	1	1	1	1	1	1	0	0	1	For display 3
0	1	0	0	0	1	1	0	0	1	1	For display 4
0	1	0	1	1	0	1	1	0	1	1	For display 5
0	1	1	0	1	0	1	1	1	1	1	For display 6
0	1	1	1	1	1	1	0	0	0	0	For display 7
1	0	0	0	1	1	1	1	1	1	1	For display 8
1	0	0	1	1	1	1	1	0	1	1	For display 9
1	0	1	0	1	1	1	0	1	1	1	For display A
1	0	1	1	0	0	1	1	1	1	1	For display b
1	1	0	0	1	0	0	1	1	1	0	For display C
1	1	0	1	0	1	1	1	1	0	1	For display d
1	1	1	0	1	0	0	1	1	1	1	For display E
1	1	1	1	1	0	0	0	1	1	1	For display F

Fig-46(Truth Table of Common cathode 7-segment display)

Initially, the logic required to convert a 4-bit binary value into its corresponding 7-segment display pattern was implemented using discrete logic gates, as shown in the figure. While functionally correct, this approach results in a highly complex circuit, requiring a large number of AND, OR, and NOT gates for each of the seven segments (a through g).
To significantly simplify the design, this logic is replaced with a pre-programmed EEPROM.

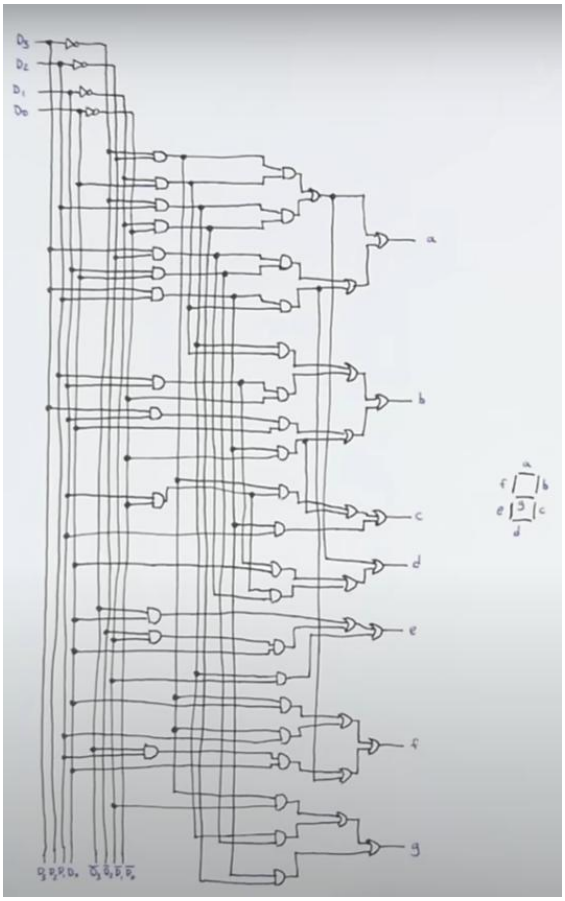


Fig-47(Combinational logic for 7-segment display)

As the complexity of the digital computer increases, certain components and operations become difficult to implement using only basic logic gates. To overcome this limitation and enhance flexibility, we introduce the use of an EEPROM (Electrically Erasable Programmable Read-Only Memory) in the system.
EEPROM acts as a small, programmable memory chip that can store and output data based on given input addresses. It allows us to replace complex combinational logic circuits with a much simpler and programmable alternative. By writing a predefined set of outputs for specific inputs, EEPROM can effectively behave as a custom logic device or a lookup table.We will be using AT28C64B(64-Kbit).

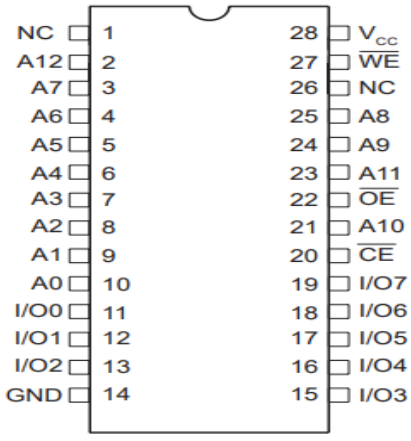


Fig-48(AT28C64B)

Pin Name	Function
A0 - A12	Addresses
$\overline{CE}$	Chip Enable
$\overline{OE}$	Output Enable
$\overline{WE}$	Write Enable
I/O0 - I/O7	Data Inputs/Outputs
NC	No Connect

Fig-49(Pin Specifications)

To simplify the logic required for driving a 7-segment display, we use a pre-programmed EEPROM that stores the necessary segment patterns. This EEPROM is programmed externally using an Arduino, and this step is performed only once, before the EEPROM is integrated into the main computer.
The binary value to be displayed is sent to the EEPROM address lines, and the output — a 7-bit segment pattern — is read from the data lines.
To interface this efficiently with the display hardware, we use a 74HC595 shift register, which allows us to serially shift out the EEPROM output and drive the 7-segment display with minimal pin usage.

Multiplexed 7-Segment Display Using EEPROM and Control Logic:

To display full 8-bit values (0–255) in a human-readable format, we use four 7-segment displays, each representing one decimal digit (hundreds, tens, units, and optional leading zero suppression or placeholder). Driving all four displays simultaneously would require a large number of I/O pins, which is not efficient on a breadboard computer. To solve this, we implement a multiplexed display system.
The following components are used in this setup:

- EEPROM: Stores the 7-segment patterns for digits 0–9. The EEPROM acts as a lookup table, outputting the correct segment configuration for a given input digit.
- 2-bit Binary Counter: It is made by dual Jk flip flop (74LS76), it Continuously cycles through values 00 to 11 (0 to 3), representing which of the four displays is currently active.
- 2-to-4 Line Decoder / Multiplexer: Converts the 2-bit counter output into a 4-line control signal, enabling only one display at a time.
- Clock Signal: Drives the binary counter at a frequency fast enough that, due to persistence of vision, all displays appear to be lit simultaneously.

We will make a separate clock for multiplexing and the counter is driven by a clock signal, which increments its value on each pulse. We will use 74LS76 to make a 2-bit binary counter.

The output of the binary counter will be connected to the two extra address pins, to access the correct segment data for each digit, we use the 2-bit binary counter output to control the higher address lines of the EEPROM. This allows us to split the EEPROM memory into four separate blocks, each corresponding to one digit of the display.

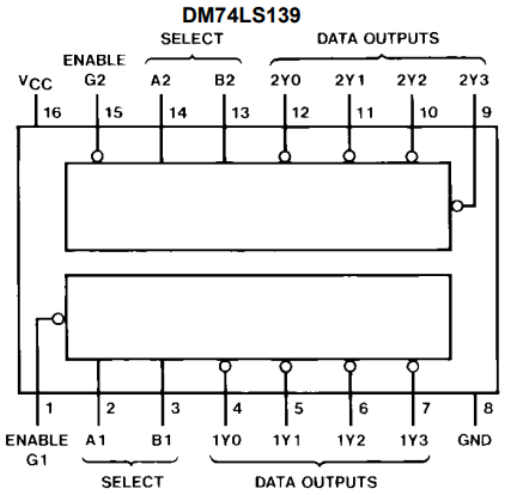


Fig-50(74LS139)

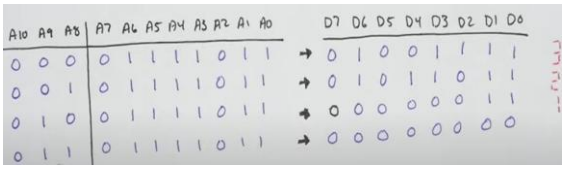
Inputs			Outputs			
Enable	Select					
G	B	A	Y0	Y1	Y2	Y3
H	X	X	H	H	H	H
L	L	L	L	H	H	H
L	L	H	H	L	H	H
L	H	L	H	H	L	H
L	H	H	H	H	H	L

Fig-51(Truth Table of 74LS139)

In the multiplexed display system, only one 7-segment display should be active at any given time to avoid overlapping or ghosting. To achieve this, we use a 2-to-4 line decoder(74LS139), also commonly referred to as a 2-to-4 multiplexer or demultiplexer, which plays a crucial role in digit selection.
The 2-bit binary counter generates a sequence of binary values: 00, 01, 10, and 11. These values are connected to the select inputs of the decoder. Based on this input, the decoder activates only one of its four outputs at a time, corresponding to the currently selected display digit.

Counter Output	Decoder Output (Active Line)	Active Display
00	D0	Units
01	D1	Tens
10	D2	Hundreds
11	D3	Thousands

Role of the Fast-Paced Clock in Display Multiplexing:
The fast-paced clock plays a critical role in ensuring that all digits appear to be lit simultaneously and smoothly, even though only one digit is active at any given moment.



Why a Fast Clock Is Needed:

Multiplexing works by rapidly turning each display on and off in a sequence. However, the human eye has a property known as persistence of vision, where any image shown for a short time (typically less than 20 milliseconds) appears to remain on the screen.

7.Connecting the Bus –

In any computer system, especially in a breadboard-based 8-bit computer, the bus acts as a shared communication highway that allows various components (such as the registers, RAM, ALU, and output display) to exchange data efficiently.
Now that individual modules like memory and registers are built, the next critical step is to interconnect them using a common bus. This allows us to transfer 8-bit data between components using a single set of data lines, rather than dedicated wires for every connection. To safely and efficiently connect multiple components to a common 8-bit bus, we make use of tri-state logic using the 74LS245 octal bus transceiver and the enable pins on the registers.

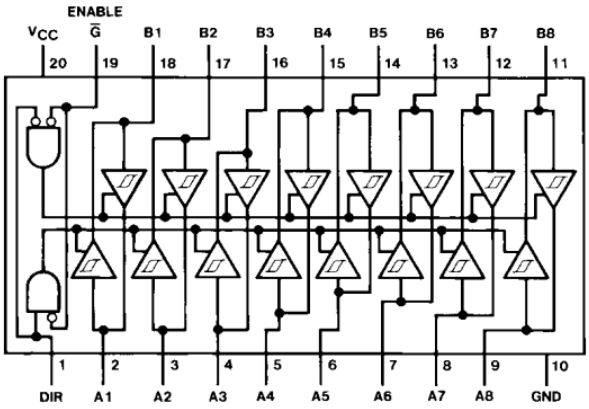


Fig-53(74LS245)

Enable $\overline{G}$	Direction Control DIR	Operation
L	L	B Data to A Bus
L	H	A Data to B Bus
H	X	Isolation

Fig-54(Function table of 74LS245)

In a shared bus system, only one component can drive the bus at a time. If two outputs try to place different values on the same line, it leads to bus contention, which can damage components or result in incorrect data. To prevent this, we use tri-state buffers, like the 74LS245.

Role of 74LS245:

- The 74LS245 is placed between a component (e.g., RAM, register, ALU output) and the bus.
- Its direction (DIR) pin determines the direction of data flow (to or from the bus).
- Its Enable pin (Active low) controls whether it actively drives the bus or stays disconnected.
- When E is LOW, the chip outputs data to the bus. When E is HIGH, its outputs are in high-impedance state.

Role of Register Enable Pins:

- Registers (like the A and B registers, or output register) have input enable and output enable controls.
- When reading from a register to the bus, the output enable is activated.
- When writing to a register from the bus, the input enable is activated.

In your bus system, only one component should output (write to the bus) at a time, while one or more components can read (input from the bus) at that moment.

7.Control Logic –

Now that all the core components of the 8-bit computer are in place — including registers, memory, the bus, and the ALU — we need a system that coordinates when and how each component operates. This system is called the control logic. The control logic is responsible for generating the right control signals at the right time, telling each component what to do during each step of an instruction. It decides when data should be moved onto the bus, when a register should read or write data, when the ALU should perform an operation, and when the output should be updated.

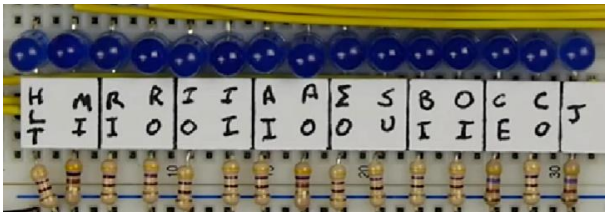


Fig-54(Control signals)

we first gather all the control signals from various parts of the computer — including registers, the ALU, RAM, program counter, and jump logic. Each signal represents a specific control action, such as loading data into a register, enabling output onto the bus, or performing an arithmetic operation. Since many of these control signals are active-low (meaning the signal is active when LOW), we connect them through NOT gates (inverters) to convert them into active-high signals for LED display.

This makes it easier to visually track which signals are currently active — an LED lights up when the signal is active (logic high after inversion). If the LED is glowing, that means the signal is active, regardless of whether it's active-high or active-low.

Now we can implement automatic instruction execution.

In this phase, the computer is no longer manually controlled using switches for each step. Instead, a clock signal drives the execution automatically, and a binary counter keeps track of the current micro-instruction step.

In our 8-bit breadboard computer, we use a binary counter (like a 3-bit counter) to automatically sequence through the steps of each instruction. Each instruction, such as LDA or ADD, is broken down into a series of micro-instructions — small steps that control the flow of data between registers, memory, and the ALU.

Rather than manually triggering each micro-instruction, the binary counter advances automatically with each clock pulse, allowing the control logic to progress through instruction steps in a timed and organized way.

The binary counter's output (e.g., 3 bits) is fed into three address lines of the EEPROM (used for control logic), telling it which step we are on and the other four address lines will be connected to the 4 MSB of the instruction register telling which instruction to perform.

We are using two EEPROMs as we have 16 memory signals, These EEPROMs are programmed in such a way that for each combination of instruction and micro-instruction step, they output a specific 16-bit control word. This control word determines which components are active at that moment — such as enabling registers, loading data, or triggering the ALU. The outputs of the EEPROMs are directly connected to the control inputs of the CPU, allowing it to automatically execute instructions step-by-step based on the clock and current instruction.

NOTE: We will give inverted clock to the program counter because we want our control signals to get loaded before the main clock pulse so that the control signals must be stable and available before the rising edge of the main system clock, If the control logic were to update after the clock pulse, components like registers might respond to old or invalid signals.

A 3:8 decoder(74LS138) is connected to the three output bits of the binary counter, which represents the current microinstruction step. The decoder translates these 3-bit binary values (ranging from 000 to 111) into eight individual active output lines, with only one line active at a time.

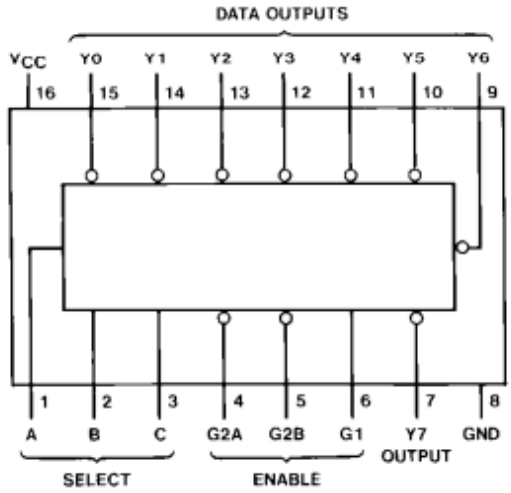


Fig-54(74LS138)

Inputs					Outputs							
Enable		Select										
G1	G2 (Note 1)	C	B	A	YO	Y1	Y2	Y3	Y4	Y5	Y6	Y7
X	H	X	X	X	H	H	H	H	H	H	H	H
L	X	X	X	X	H	H	H	H	H	H	H	H
H	L	L	L	L	L	H	H	H	H	H	H	H
H	L	L	L	H	H	L	H	H	H	H	H	H
H	L	L	H	L	H	H	L	H	H	H	H	H
H	L	L	H	H	H	H	H	L	H	H	H	H
H	L	H	L	L	H	H	H	H	L	H	H	H
H	L	H	L	H	H	H	H	H	H	L	H	H
H	L	H	H	L	H	H	H	H	H	H	L	H
H	L	H	H	H	H	H	H	H	H	H	H	L

Fig-51(Truth Table of 74LS138)

our design requires only five steps per instruction (Steps 0 to 4). To ensure that the control logic doesn't proceed beyond Step 4, we use the decoder's output corresponding to Step 5 (Y5) as a reset trigger.

The Y5 output is connected to the STEP RESET logic, which in turn activates the CLEAR pin of the binary counter (via a NOT gate, if needed for active-low CLEAR). This means that as soon as Step 5 is reached, the counter is automatically reset, returning it to Step 0, and the next instruction cycle can begin.

Reset circuit - Reset circuit is essential to ensure all components start from a known, predictable state. Our reset circuit performs this task by generating a logic

-

signal that resets counters, registers, and other sequential elements to their initial states (usually 0).

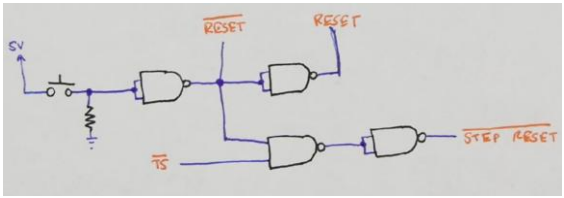


Fig-52(Reset using using 74LS00)

The NOT RESET signal is connected to components that require an active LOW reset, ensuring they are triggered when the reset line goes LOW. On the other hand, the RESET signal (without inversion) is connected to components that require an active HIGH reset, enabling them when the line goes HIGH. This configuration allows the same reset circuit to effectively handle both types of reset inputs across different components in the system.

The NOT STEP RESET signal is connected to the CLEAR pin of the binary counter that manages the microinstruction steps. Since the CLEAR pin is typically active LOW, it must receive a LOW signal to reset the counter. By inverting the STEP RESET signal using a NOT gate, we generate the NOT STEP RESET signal, which ensures that the counter is properly cleared when a step reset is required. This reset mechanism ensures that each instruction always begins from step 0, maintaining the correct execution sequence within the control logic.

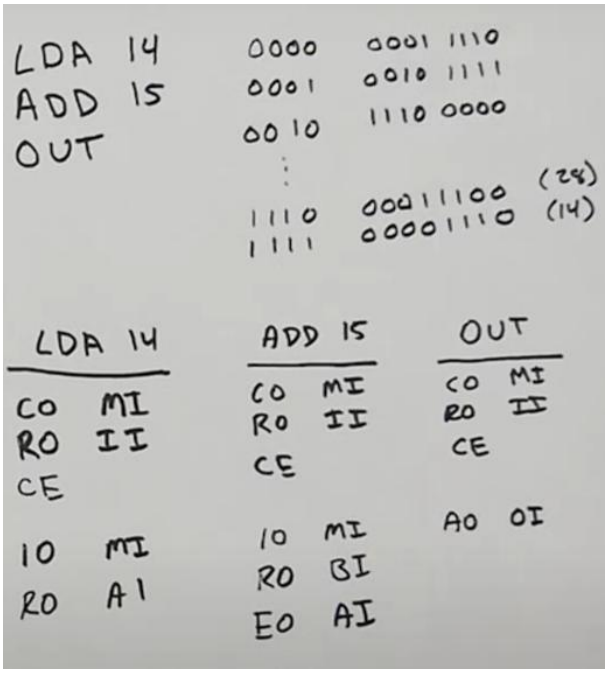


Fig-53(Assembly code for our 8-bit computer to add two numbers)

In our 8-bit breadboard computer, each machine-level instruction is broken down into a series of **micro-instructions**, which are executed step-by-step using control signals. The EEPROM-based control logic determines which signals to activate at each step based on the current instruction and step count.

The image above shows the micro-instruction breakdown for three instructions while we have these set of instructions coded in our eeprom:

- ADD- loads value in B and then what's in ALU reg loads back to Reg A (also does latch the flags value on last step)
- NOP- Does nothing
- LDA – Loads value in Reg A
- SUB- loads data in Reg B and on the Sub signal so the ALU will have A-B (also does latch the flags value on last step)
- STA- loads reg A value to the RAM
- LDI- loads Instruction Reg data to Reg A (note IR takes 8 bits from bus only push least 4 bits so LDI command only can load value from 0 to 15)
- JMP- reset the program count to a particular address
- JC- only reset the program counter to a particular address if the carry flag was on
- JZ- only reset the program counter to a particular address if the zero flag was on
- OUT- displays reg A value to the seven-segment display (we have three so we can display 255 as max)
- HLT- stops the master clock

The micro-instruction sequence {CO MI, RO II, CE} is common to every instruction because it represents the instruction fetch cycle. In this phase, the computer retrieves the next instruction from memory.

The different micro-instruction has their own working, either to take signal from the bus or send signal to the bus or anything. All of the micro-instruction are;

- CO - Counter Out sends the value of the counter to the bus.
- MI – Memory In takes the value of the bus and sends it in memory address.
- RO – Ram out sends the value of the ram to the bus.
- II - Instruction In takes the value of the bus and sends it in the instruction register.
- CE – Counter Enable increases the counting of the counter by 1.
- IO – Instruction Out sends the value of the instruction register to the bus.
- AI – The value of bus it stored in register A.
- BI – The value of bus is stored in register B.

Different instructions (LDA, ADD etc.) has a combination of different set of micro-instructions (MI, IO etc.). This combination of micro-instruction for time T_0 and time T_1 is called fetch cycle, which is same for all OpCode like LDA, RAM etc.
For every instruction, the first two combinations of the micro-instruction are same. These micro-instructions are MI, CO, RO, II, CE. The first two instructions (MI, CO)

are executed at the first clock input or at time T_0 and the other three instruction (RO, II, CE) are executed at the second clock input or at time T_1. The time T_2, T_3 and T_4 has different set of micro-instruction for different instructions (OpCode).

Adding Two Numbers –

The 8-bit computer and the x86-64 architecture approach the same computational goal—adding two numbers and outputting the result—in fundamentally different ways, reflecting the contrasting design philosophies of minimalist versus general-purpose computing. In the 8-bit computer, the entire program is composed of just four simple instructions: LDA 14, ADD 15, OUT, and HLT(see figure-54). Here, LDA 14 tells the computer to load the value at memory address 14 into the accumulator, which is the only general-purpose register available in such designs. ADD 15 adds the value stored at address 15 to the accumulator. OUT sends the accumulator's value to an output device (such as LEDs or a display register), and HLT stops the program. These instructions are each 1 byte long, and the entire program uses only 4 bytes of code and operates directly on hardcoded memory addresses without abstraction. The simplicity of this model allows each instruction to be executed in one or two clock cycles, making the data flow transparent and easy to trace. There is no stack, no function calls, and no memory management—just direct, low-level control over the processor's basic operations.

```
LDA 14      ; Load the value at address 14 into the accumulator
ADD 15      ; Add the value at address 15 to the accumulator
OUT         ; Output the accumulator value
HLT         ; Halt the program
```

Fig-54(Assembly code for our 8-bit computer to add two numbers with explanation)

```c
int main() {
    int a = 7;
    int b = 3;
    int c = a + b;
    printf("%d\n", c);
    return 0;
}
```

Fig-55(C code for adding two numbers)

In contrast, the x86-64 assembly output generated by compiling a simple C program like int a = 7; int b = 3; int c = a + b; printf("%d\n", c); return 0; (see figure-55) involves around 20 lines of low-level instructions. The compiled code begins by setting up a stack frame with push rbp and mov rbp, rsp, (see figure-56) followed by allocating stack space for local variables (sub rsp, 0x10).

The values 7 and 3 are stored in local memory slots on the stack using mov DWORD PTR [rbp-0x4], 0x7 and mov DWORD PTR [rbp-0x8], 0x3. These values are then loaded into registers (mov edx, DWORD PTR [rbp-0x4]) and added together (add edx, DWORD PTR [rbp-0x8]). The result is moved into eax, the register used for return values and parameters, and passed to the printf function through a call instruction. The function call follows the System V AMD64 calling convention, which means registers must be saved and parameters passed in specific registers (like rdi for the first argument). After printing the result, the program cleans up with leave and returns with ret.

While the 8-bit computer operates with fixed, transparent memory and a single accumulator, the x86-64 code is much more structured and modular, supporting dynamic memory, multiple functions, recursion, interrupts, and more. The x86-64 processor uses a rich instruction set, general-purpose registers, and a stack-based calling convention to support complex application logic and operating system interactions. This added complexity ensures portability, reusability, and integration with higher-level software like standard libraries, but also makes the code harder to follow manually compared to the 8-bit version. In summary, the 8-bit computer emphasizes minimalism and direct control, making it ideal for learning and simple tasks, whereas the x86-64 code prioritizes generality, scalability, and interoperability, enabling the execution of complex programs in modern computing environments.

```
yusyii@apollo:~$ cd c
yusyii@apollo:~/c$ nvim add.c
yusyii@apollo:~/c$ objdump -d -M intel ./add | awk '/<main>:/,/^$/'
0000000000400466 <main>:
  400466:       55                      push   rbp
  400467:       48 89 e5                mov    rbp,rsp
  40046a:       48 83 ec 10             sub    rsp,0x10
  40046e:       c7 45 fc 07 00 00 00    mov    DWORD PTR [rbp-0x4],0x7
  400475:       c7 45 f8 03 00 00 00    mov    DWORD PTR [rbp-0x8],0x3
  40047c:       8b 55 fc                mov    edx,DWORD PTR [rbp-0x4]
  40047f:       8b 45 f8                mov    eax,DWORD PTR [rbp-0x8]
  400482:       01 d0                   add    eax,edx
  400484:       89 45 f4                mov    DWORD PTR [rbp-0xc],eax
  400487:       8b 45 f4                mov    eax,DWORD PTR [rbp-0xc]
  40048a:       89 c6                   mov    esi,eax
  40048c:       bf 80 11 40 00          mov    edi,0x401180
  400491:       b8 00 00 00 00          mov    eax,0x0
  400496:       e8 d5 fe ff ff          call   400370 <printf@plt>
  40049b:       b8 00 00 00 00          mov    eax,0x0
  4004a0:       c9                      leave
  4004a1:       c3                      ret
```

Fig-56(C language machine code for adding two numbers)

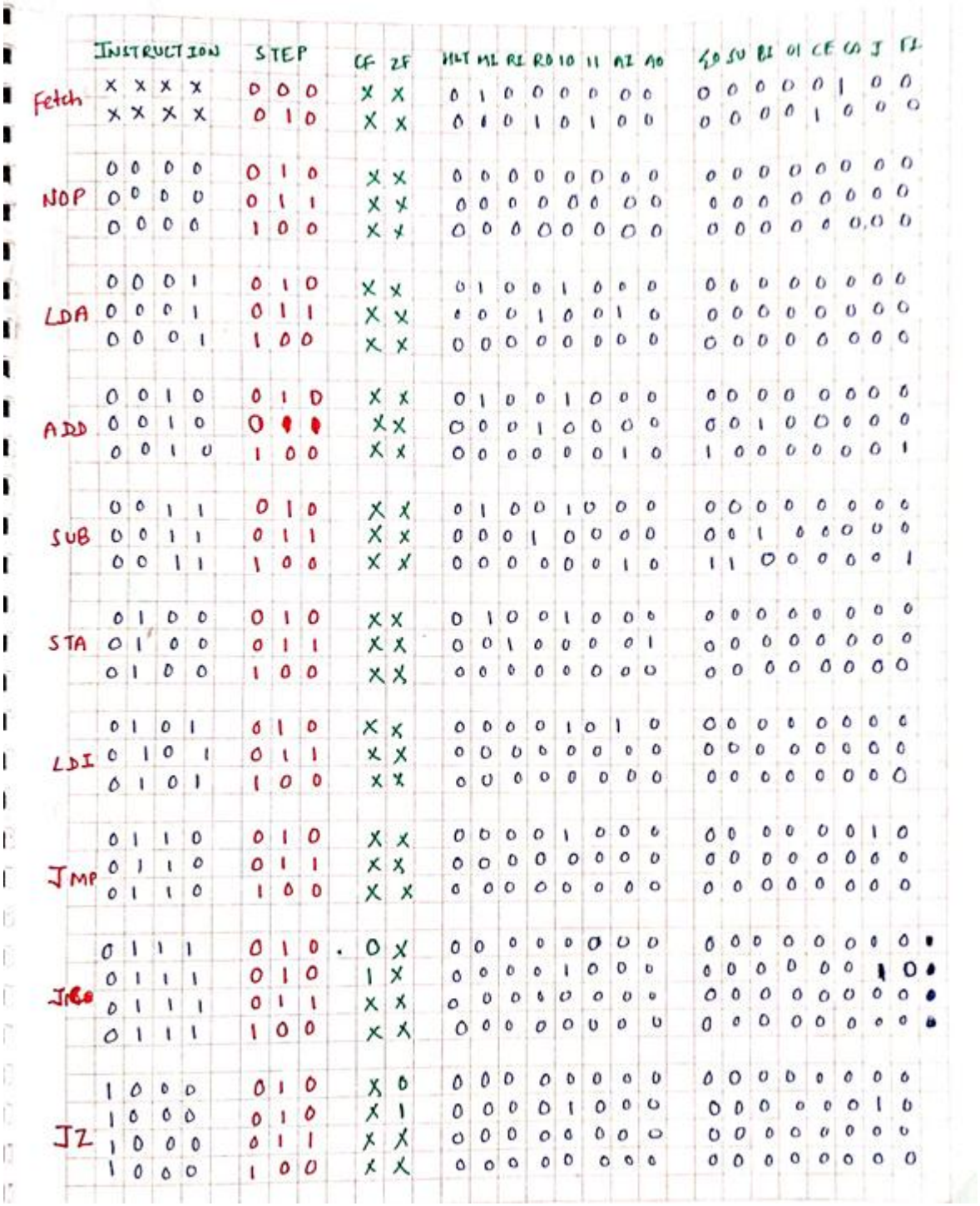


Fig-58(Binary code corresponding to each instruction)

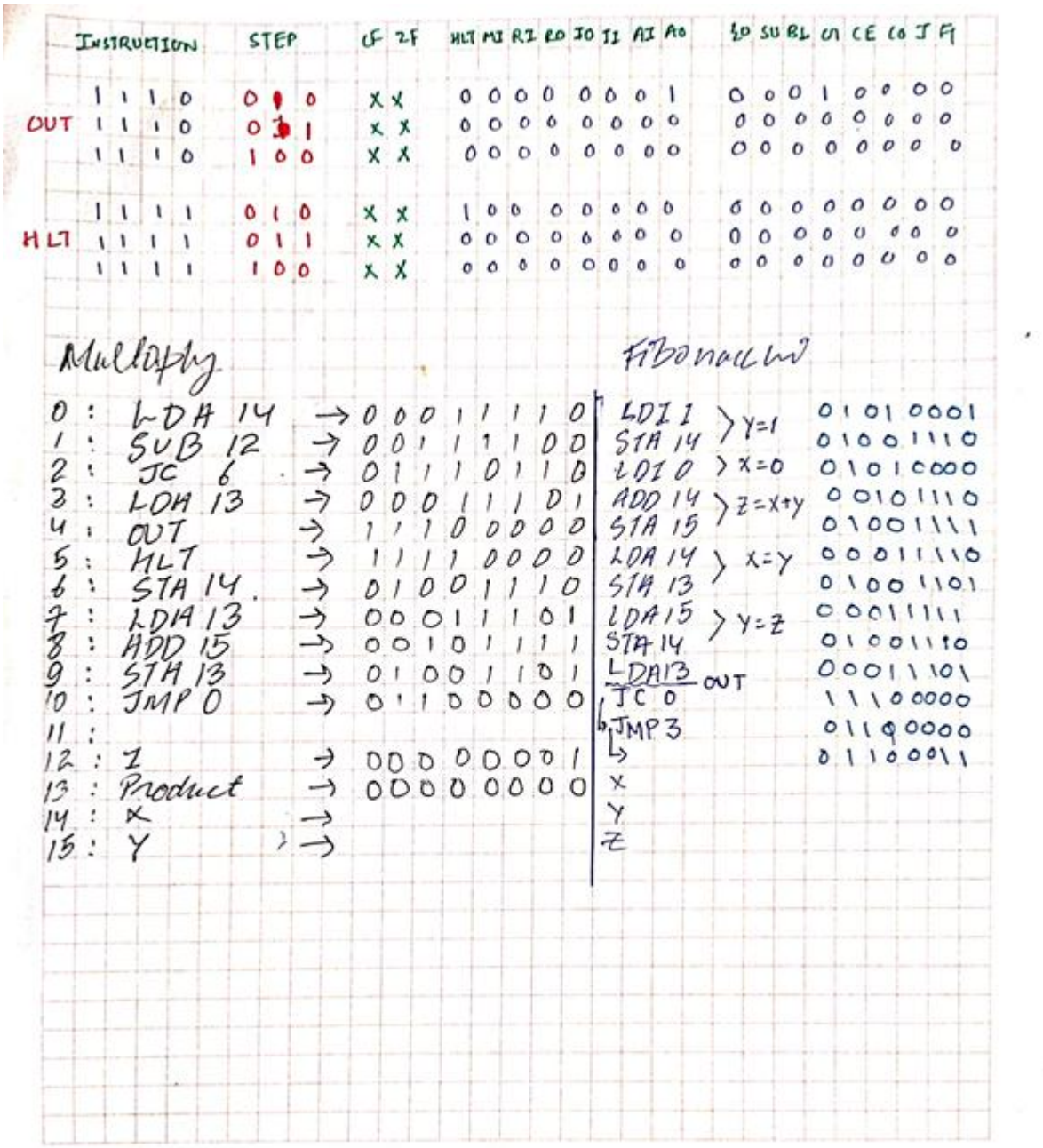


Fig-59(Binary code corresponding to each instruction with assembly language to multiply two number and fibonacchi series)

Fig-60(Binary code corresponding to each address (similar to instruction) that is programed into control logic EEPROM's)

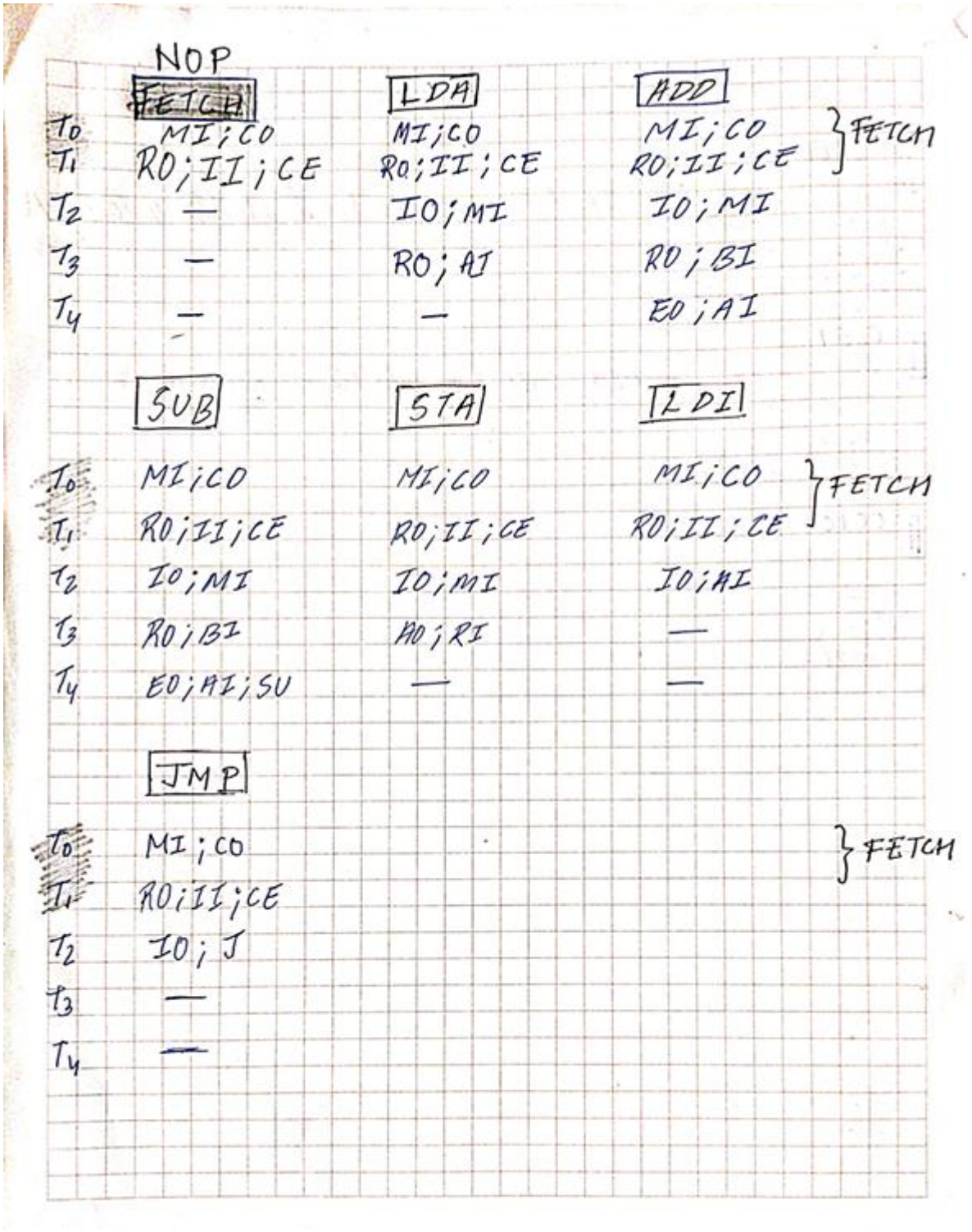


Fig-61(Fetch cycle, micro-instructions that constructs an instruction)

References:

- **Ben Eater (https://www.youtube.com/@BenEater)**
- **555 timer**
- **74LS04 HEX inverter**
- **74LS08 QUAD AND gate**
- **74LS32 QUAD OR gate**
- **74LS173 4bit – D register**
- **74LS86 Quad XOR gate**
- **74LS245 Octal bus transceiver**
- **74LS283 4- bit adder**
- **74LS00 QUAD NAND gate**
- **74LS157 QUAD 2- to -1 selectors/multiplexer**
- **74189 64 bit RAM**
- **74LS161 4-bit Binary counter**
- **74LS107 Dual J/K flip-flop**
- **74LS139 Dual 2- to – 4 line decoder/demultiplexer**
- **74LS273 8-bit D register**
- **28C64 64K EEPROM**