International Journal of Enhanced Research in Science, Technology & Engineering
ISSN: 2319-7463, Vol. 14 Issue 6, June-2025

Exploring Microcode CPU Instructions with RISC 8-Bit Breadboard Computer

Ayushman Bhattacharya[1], Anwesha Chakraborty[2], Vivek Yadav[3], Piyali Sarkar[4], Sonali Bhowmik[5], Dr. Dipankar Misra[6], Prof. (Dr.) Sumit Nandi[7]

[1,2,3,4] Student, Department of Computer Science and Engineering, JIS University, West Bengal
[5]Assistant Professor, Department of Computer Science and Engineering, JIS University, West Bengal
[6] Professor, Department of Computer Science and Engineering, JIS University, West Bengal
[7] Principal, Harishchandrapur College, Malda, West Bengal

ABSTRACT

This research explores the implementation of a transistor-based 8-bit CPU inspired by the RISC architecture and built entirely using TTL logic from the 74HC/LS series. The system features a minimal instruction set architecture with 8-bit wide instructions comprising a 4-bit opcode and a 4-bit address, enabling basic operations such as addition, subtraction, data transfer, and control flow. A total memory of 128 bits is accessed via an 8-bit data bus, with a program counter and instruction register facilitating sequential execution. Control logic [6 4] is driven through ARDUINO-based microcode, while arithmetic operations are handled by a discrete ALU supporting two's complement logic, supported by direct access to two 8-bit registers. Output is visualized using a 4-digit seven-segment display connected via an output register. The project demonstrates a fundamental understanding of digital systems, instruction sequencing, and low-level computing using discrete logic components [2 15]

Keywords: 8-Bit CPU, TTL Logic, RISC, ISA, EEPROM, Arduino, Breadboard, Low-level computing.

INTRODUCTION

The 8-bit CPU [1] described in this work serves as a foundational platform for understanding the principles of TTL (Transistor-Transistor Logic) design, computer architecture, and digital logic, drawing significant inspiration from the Reduced Instruction Set Computer (RISC) philosophy. Built entirely on a breadboard using 74HC/LS-series ICs [8 7], the system is designed to illustrate the working of a minimalist CPU with fundamental computational capabilities. The CPU is operated using a synchronized clock signal, ensuring that all modules function in coordination. Clock frequencies range from a minimum of approximately 0.1 Hz to a maximum of 300 Hz, allowing both step-by-step debugging and rapid execution. A memory address register, implemented using the 74HC157 [2-3] multiplexer, operates in two modes: Program Mode and Run Mode. The main memory is constructed from 74HC189 ICs, allowing a total of 128 bits of data storage across 16 address locations, selected via 4-bit addressing. Memory values are programmed through 8-position DIP switches, and signal inversion is achieved using 74HC04 [4-6] hex-inverters.

Instructions are 8 bits long, comprising a 4-bit opcode and a 4-bit address field. These are passed over the data bus to control both memory addressing and instruction execution. A timing controller, built using the 74HC161 [7-9] 4-bit binary counter, generates six microsteps (T0–T5) per clock cycle to manage sequential operations. The program counter, also a 4-bit 74HC161-based module [10-12], allows traversal of the 16 memory addresses.Two general-purpose 8-bit registers, A and B, are implemented using pairs of 74HC173 [13-21] ICs. The Arithmetic Logic Unit (ALU) consists of two 74HC183 4-bit parallel adder/subtractor ICs, along with 74HC86 [22] XOR gates to implement two's complement subtraction. The ALU supports three status flags: Zero, Overflow, and Negative. The Zero Flag is set by AND-ing all output bits and checking for zero. The Overflow Flag is derived directly from the carry-out (C_out) pin of the most significant 74HC183. The Negative Flag is activated by detecting whether subtraction is in progress and the ALU's MSB is set, confirming a negative result via a logical AND operation between the subtract signal and MSB.
Output display is managed by an EEPROM 28C64 [23-27], used as a lookup table for seven-segment decoding. Although the EEPROM supports 12 address lines, only the lower 8 are used, providing a 256-byte programmable space. To distinguish between positive and negative outputs, the EEPROM is partitioned: addresses 0–127 store segment patterns for positive results, and 128–255 handle negative values. This mapping allows signed two's complement values to be visualized in real time without additional logic. Display output is multiplexed using a NE555 timer, cycling rapidly across four seven-segment common cathode displays to create a continuous visual output through persistence of vision.Control logic is handled externally by an Arduino Nano, which emulates a microcode controller. It transmits control signals via two 74HC595 [28-40] shift registers, allowing compact serial-to-parallel communication.

The opcode-to-control-line mapping is preloaded into the Arduino, while the RAM is pre-programmed with instruction sequences prior to entering Run Mode. The Arduino is also capable of halting the clock upon program completion, enabling controlled execution and debugging.

MATERIALS

Core ICs (TTL Logic – 74 Series)

1. **74HC/LS157** – Quad 2-to-1 multiplexer (used for Memory Address Register)
2. **74HC/LS189** – 64-bit RAM (used for main memory, multiple units for 128-bit total)
3. **74HC/LS04** – Hex inverter (used for signal inversion)
4. **74HC/LS173** – 4-bit register (2x for A and 2x for B register to make 8-bit registers)
5. **74HC/LS183** – 4-bit binary adder/subtractor (2x for 8-bit ALU)
6. **74HC/LS86** – Quad XOR gate (2x for two's complement logic in subtraction)
7. **74HC/LS161** – 4-bit binary counter (used for program counter and timing controller)
8. **74HC/LS595** – 8-bit serial-in, parallel-out shift register (2x for control signal expansion)
9. **EEPROM 28C64** – 8Kx8 non-volatile memory (used for seven-segment display decoding)

Display and Output

10. **Seven-Segment Displays (Common Cathode)** – 4 units (for visual output of 8-bit results)
11. **NE555 Timer IC** – (used for fast multiplexing of display segments)

Microcontroller and Control

12. [8 9 2] **Arduino Nano** – (drives control logic, handles instruction decoding and timing)
13. **DIP Switches (8-position)** – (used to program main memory manually in Program Mode)

Supporting Components

14. **Resistors** – Basically pull-down to not cause override issues mostly 1k, 100k, 220, 10k (ohms)
15. **Capacitors** – For power smoothing, debounce circuits, and timing circuits
16. **LEDs** – Indicator LEDs (status, output monitoring, clock pulse indicator)

METHODS

The Clock Module

The clock module serves as the primary timing source that synchronizes all computational modules of the CPU. It is designed to generate a square wave signal that drives the sequential operation of the control logic, registers, memory, and arithmetic logic unit (ALU). The module is implemented using a standard NE555 timer integrated circuit configured in **astable mode** to produce continuous pulses.The frequency of the output clock is adjustable via an external resistor-capacitor (RC) network.

In the default configuration, the system supports a clock frequency range of approximately **0.1 Hz to 300 Hz**, enabling both manual and automatic operation. For slower debugging purposes, a **manual clock pulse button** is provided in parallel with the NE555 timer's output, allowing single-step instruction execution for educational or diagnostic purposes.

The generated clock signal is distributed via a **common clock rail** and buffered through inverters from a **74HC04 hex inverter** [10] IC to maintain signal integrity and prevent loading effects across the system. This ensures uniform propagation delay and consistent timing across all sequential elements such as the program counter, instruction register, and control signal step counters.

The flexibility of the clock module allows the system to be operated in two primary modes:

- **Continuous Mode**: Automated execution at the selected frequency.
- **Single-Step Mode**: Manual pulse advancement, useful for observing state transitions at each clock cycle.

The clock module is essential in maintaining temporal order in the CPU's operation, ensuring each micro-operation aligns with the correct time step (T0–T5) defined by the control logic sequencing.

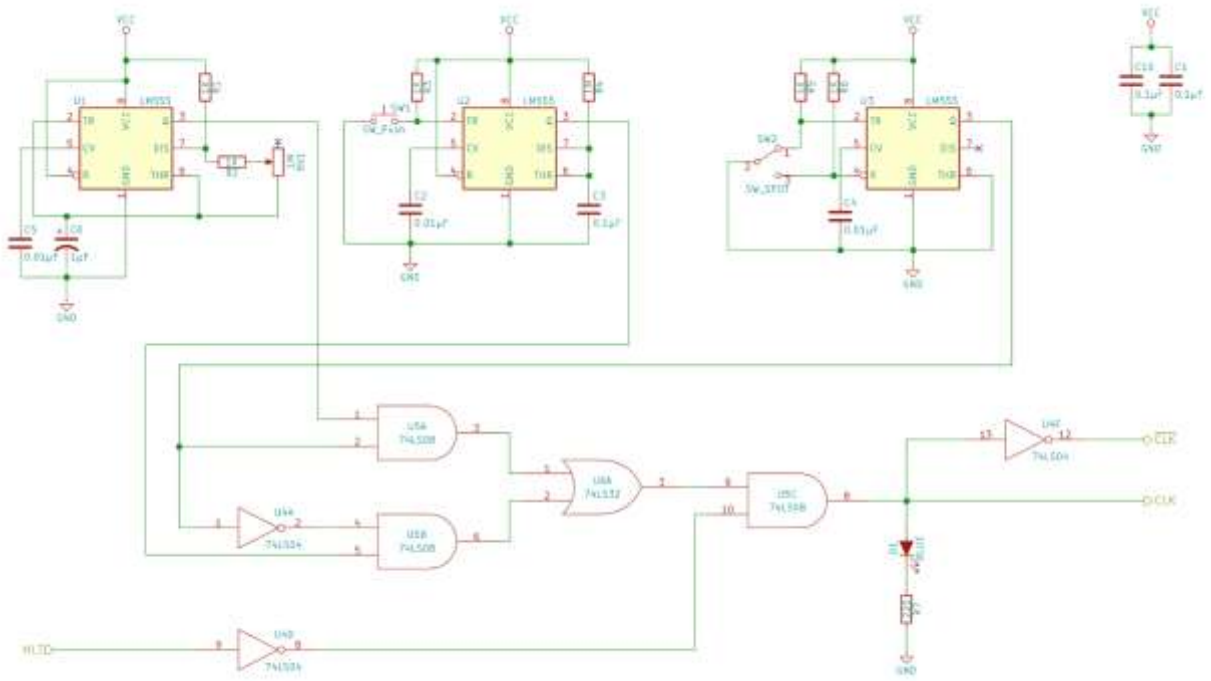

Fig.1: Diagram showing the circuit-layout of the CLOCK module circuit of the CPU

Program Counter

The **Program Counter (PC)** is a sequential logic module responsible for storing and advancing the address of the next instruction to be executed. It plays a critical role in maintaining the linear execution flow of instructions within the CPU.

The program counter is implemented using a **74HC161** [34] **4-bit binary counter**, which allows for address space ranging from **0 to 15**, corresponding to 16 instruction locations. This is aligned with the CPU's **4-bit memory addressing scheme**, enabling traversal through the entire 128-bit primary memory divided across 16 addresses (each holding 8-bit data or instructions).

The PC increments on every positive edge of the system clock, receiving its timing input from the clock module. It outputs a 4-bit binary value representing the current instruction address, which is sent to the **Memory Address Register (MAR)** and subsequently to the main memory.

To allow greater flexibility in execution control, the module supports:

- **Automatic Incrementing**: In normal operation, the PC automatically increments after every instruction cycle.
- **Manual Reset**: A reset line is connected to allow reinitialization of the PC to 0000 at system startup or after halting.
- **Conditional Jumps**: The PC supports overrides from the control logic in response to jump or branch instructions, where the counter can be loaded with a specific value using the parallel load feature of the 74HC161.

The 74HC161's synchronous load and clear functionalities are exploited to enable both unconditional and conditional jumps, critical for implementing flow-control operations such as loops and subroutine calls.

This module ensures that the instruction sequence proceeds deterministically unless deliberately altered by jump instructions, thereby maintaining logical coherence in the instruction execution cycle.

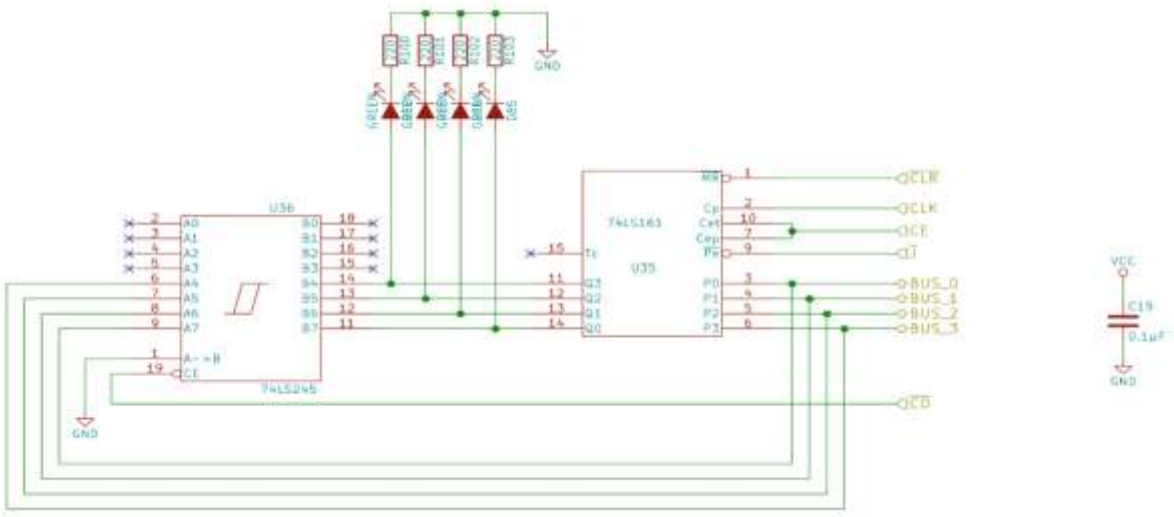

Fig. 2: Diagram showing the program counter module circuit for the CPU

Memory Address Register & Main Memory

The MAR is implemented using a **74HC157** [31] **quad 2-to-1 multiplexer**, enabling the selection between two operational modes: **Program Mode** and **Run Mode**. In Program Mode, the memory address is supplied externally via DIP switches to allow manual writing of data into memory. In Run Mode, the address is sourced from the **Program Counter (PC)** or **Instruction Register**, enabling the automated execution of instructions.

The MAR outputs a 4-bit binary address to the main memory system and acts as the address selector for read or write operations. The use of the 74HC157 allows dynamic switching between input sources without additional complex logic, which is essential for implementing a dual-mode memory interface.

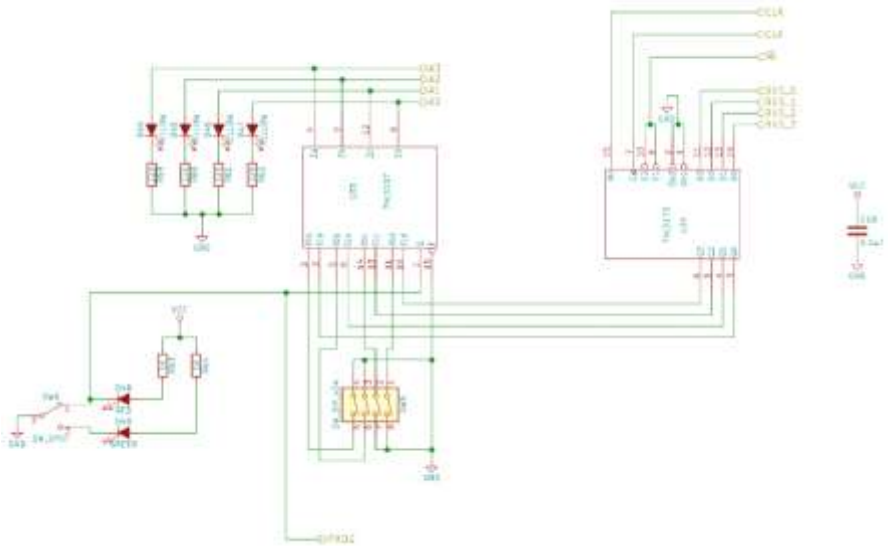

Fig. 3: Diagram showing the Memory Address Register module circuit for the CPU

The primary memory is constructed using **74HC189** 64-bit static RAM ICs. Each IC provides 16 memory locations of 4 bits each. To support 8-bit instruction width, two 74HC189 ICs are used in parallel—one handling the upper 4 bits and the other the lower 4 bits—thus forming a complete 8-bit wide memory cell per address.

The total addressable space comprises **16 locations × 8 bits = 128 bits**, consistent with the 4-bit address width (2^4 = 16). Data is loaded into memory through **DIP switches** in Program Mode, with input lines buffered and latched accordingly. During Run Mode, instructions and operands are fetched based on the address provided by the MAR.

Data read and write operations are controlled using the **Write Enable (WE)** and **Chip Enable (CE)** signals derived from the control logic. Inverters from the **74HC04** hex inverter IC are used to manage these control signals and ensure proper logic levels.

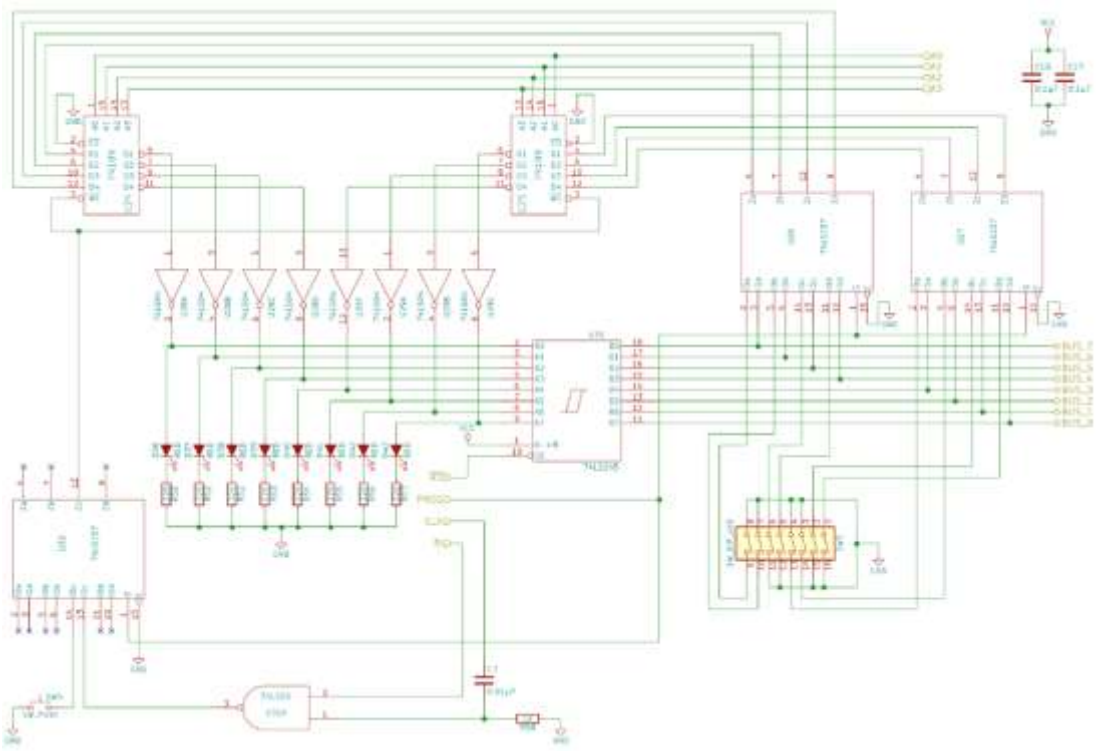

Fig. 4: Diagram showing the Random-Access Memory circuit for the CPU.

Instruction Register

The **Instruction Register (IR)** serves as the intermediate storage for the instruction currently being executed by the CPU. It is responsible for decoding the instruction into its constituent components: the **opcode** and the **memory address**, both 4 bits in length.

The IR is implemented using **two 74HC173** [20 1] **4-bit D-type register ICs**, allowing storage and buffering of the full 8-bit instruction fetched from main memory. These ICs provide tri-state outputs, enabling seamless integration with the common data bus while preventing signal conflicts during parallel communication with other modules.

When a memory read operation is performed during the instruction fetch phase, the 8-bit instruction data is latched into the IR under control of the **Load Instruction** signal generated by the control logic. Once latched, the instruction remains stable for the duration of the execution cycle, allowing its components to be routed to different parts of the system:

- The **upper 4 bits (bits 7–4)** are interpreted as the **opcode**, used by the **control logic** to determine which sequence of micro-operations to perform.
- The **lower 4 bits (bits 3–0)** represent the **memory address**, which is passed to the **Memory Address Register (MAR)** for operand fetching during execution.

This separation of instruction components allows the CPU to follow a clear **fetch-decode-execute** cycle, critical for the deterministic operation of any RISC-based processor.

Additionally, the IR supports synchronous loading and clearing, which ensures instructions are captured cleanly in sync with the clock and can be reset when required during system initialization or program halting.By isolating instruction decoding into its own dedicated module, the architecture promotes modularity and clarity, facilitating debugging, educational demonstration, and expansion of the instruction set in future iterations.

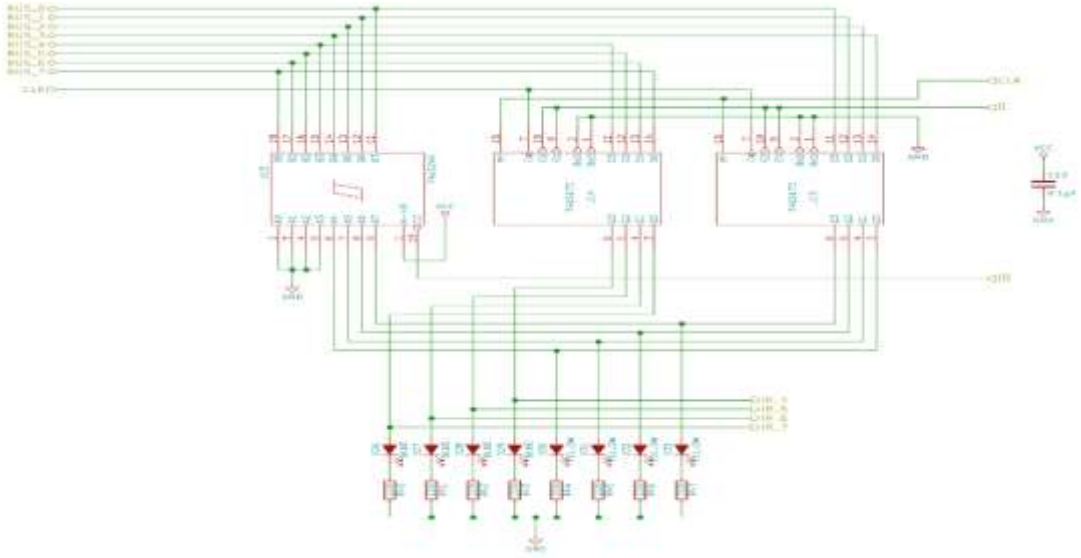

Fig. 5: Diagram showing the Instruction Register module circuit for the CPU.

Step Counter Module (T0-T5)
The **Step Counter** module is responsible for orchestrating the internal timing sequence of micro-operations that occur within a single instruction cycle. By subdividing each clock pulse into discrete steps, it ensures that the various modules of the CPU activate in a precisely defined order, facilitating proper instruction execution.

This functionality is achieved using a **74HC161** [34 1] **4-bit synchronous binary counter**, configured to count from **T0 to T5**, thereby enabling a total of **six unique timing states**. Each timing state (T0–T5) corresponds to a specific micro-operation such as instruction fetch, decode, memory read/write, ALU operation, and result storage.

The step counter is incremented with each clock pulse, and its output is used to control the activation of other modules via the control logic. After reaching **T5**, the counter automatically resets to **T0**, thereby initiating the next instruction cycle. This cyclic behavior ensures that every instruction is allocated a fixed and consistent number of micro-steps, simplifying control logic design and predictability.The counter receives its clock input from the main clock module, ensuring synchronization across all components of the CPU. Its outputs are decoded using combinational logic to generate timing-specific enable signals, which are distributed to modules like the Instruction Register, Program Counter, Memory Address Register, and ALU.By enforcing a structured step-by-step execution pipeline, the Step Counter plays a pivotal role in maintaining temporal control over the entire CPU, supporting the deterministic behavior expected in a RISC-inspired architecture.

General Register A & B
The CPU features two general-purpose 8-bit registers, **Register A** and **Register B**, which serve as the primary operands for arithmetic and logical operations carried out by the Arithmetic Logic Unit (ALU). These registers act as temporary data storage points during the execution cycle, particularly between the fetch and execute stages.

Each register is implemented using **two 74HC173** [20 1 2] **ICs**, with each IC capable of storing 4 bits, resulting in full 8-bit storage per register. The 74HC173 is a 4-bit D-type register with tri-state outputs and independent control signals for load and enable operations, allowing these registers to interact flexibly with the system's common data bus.

- **Register A** functions as the primary operand holder, typically storing the data fetched from memory or loaded during instruction execution.
- **Register B** holds the secondary operand and is often used in conjunction with Register A during ALU operations such as addition or subtraction.

Data is loaded into the registers from the data bus under the control of the step counter and the control logic. The **Load A** and **Load B** signals determine when data from memory is latched into Register A or B, respectively. Similarly, the **Output Enable** lines of the 74HC173 chips allow selective access to the bus when the content of either register needs to be read or transferred.These registers are directly connected to the input lines of the ALU, enabling fast and synchronous operation without additional buffering or signal routing. Their use of tri-state logic ensures that they do not interfere with bus communication when not actively engaged.By incorporating dedicated general-purpose registers, the CPU architecture supports efficient instruction execution and maintains modular clarity, both of which are hallmarks of the RISC philosophy.

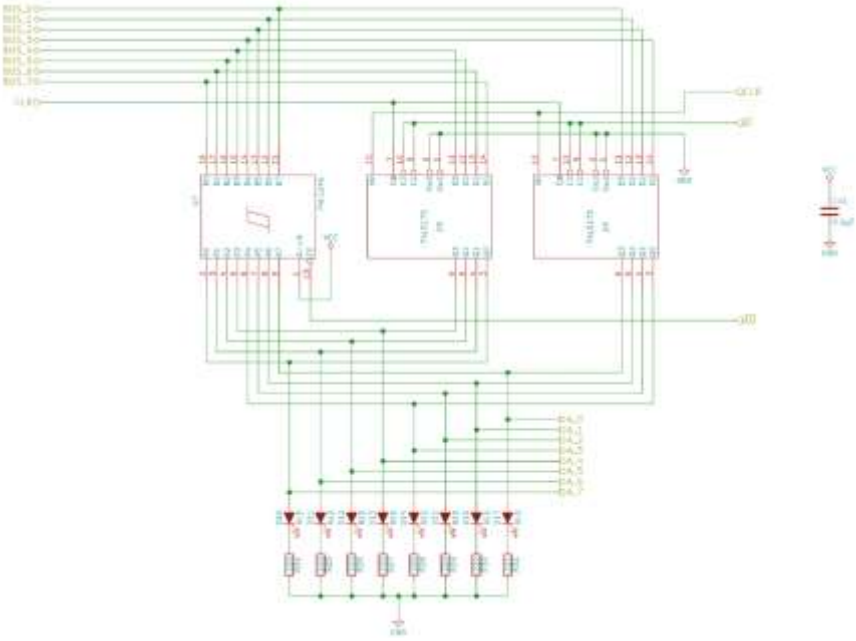

Fig. 6: Diagram showing the A and B register circuit for the CPU.

Flag Registers
The CPU has been accompanied with 3 Flag Registers and the functionalities are being discussed thus: -

Zero Flag
The **Zero Flag [1 8 18]** is set when the result of an arithmetic or logical operation equals zero. This detection is accomplished by performing a logical OR across all 8 output bits of the ALU and then inverting the result using a NOT gate. If all output bits are low (logic 0), the OR operation yields 0, and the inversion produces a logic 1, thereby setting the Zero Flag.

Negative Flag
The **Negative Flag[1 8 18]** is used to determine if the ALU result represents a negative number, assuming two's complement notation. It is set when two conditions are simultaneously satisfied:

1. The **Subtract** control signal is active (indicating a subtraction operation), and
2. The **Most Significant Bit (MSB)** of the ALU output is high, signifying a negative value in two's complement form.

These two signals are ANDed together, and the result is used to set the Negative Flag. This design ensures that the flag is not falsely triggered during addition or unsigned operations.

Carry Flag
The **Carry or Overflow Flag[1 8 18]** reflects whether an arithmetic operation has exceeded the maximum representable value in 8 bits. It is directly connected to the **carry-out (C_out)** pin of the **MSB 74HC183** IC used in the ALU. For addition operations, a carry-out of 1 indicates that the result exceeded 8-bit capacity, setting the Carry Flag. In subtraction operations, it can similarly indicate underflow depending on implementation logic.Each of these flags is updated in real-time with every ALU operation and plays a central role in decision-making within control flow, such as implementing conditional jumps or halts based on computational results.

Arithmetic Logical Unit (ALU)
The **Arithmetic Logic Unit (ALU)[1 8 18]** is the core computational component of the CPU, responsible for performing arithmetic operations such as **addition** and **subtraction**, as well as generating output flags based on result conditions. The ALU in this system is designed to handle **8-bit parallel operations [1 8 18 4]** and follows a modular hardware design based on **TTL logic**.

Hardware Implementation
The ALU is constructed using:

- **2× 74HC183** – 4-bit full adders with carry-in and carry-out support.
- **2× 74HC86** [22] – XOR gates used for enabling subtraction via two's complement.

To achieve 8-bit operations, two 74HC183 chips are cascaded. The first handles the lower 4 bits, and the second handles the upper 4 bits. Carry chaining is employed between the two to maintain arithmetic consistency across all 8 bits.

Addition
When performing an addition, both input operands (from Register A and Register B) are directly fed into the ALU. The **Carry-in (C_in)** for the lower nibble is typically set to 0, while the carry-out from the lower nibble feeds into the upper nibble adder.

Subtraction
Subtraction is implemented using the **two's complement method**. The operand from Register B is passed through XOR gates (74HC86) [22] to invert its bits conditionally when the **Subtract** control line is active. Simultaneously, the carry-in is set to 1 to complete the two's complement addition logic, allowing the subtraction to be handled using the same adders.

This shared hardware strategy simplifies design while supporting full-range signed arithmetic.

Integration with Flags
The ALU's output directly affects the **Zero**, **Negative**, and **Carry** flags:

- **Zero Flag[1 8 18]** is set if all 8 output bits are 0.
- **Negative Flag[1 8 18]** is triggered if subtraction is active and the MSB of the result is 1.
- **Carry Flag[1 8 18]** reflects overflow, indicated by the carry-out from the MSB 74HC183.

Bus Connection
The ALU is connected to the system data bus via tri-state logic, allowing the result to be placed on the bus when the **Output Enable** signal is activated. The result can then be routed to the Output Register, RAM, or any other module requiring the computed value.This ALU design, although minimalistic, aligns well with RISC principles and offers a clear, pedagogical representation of digital arithmetic on a breadboard-level CPU.

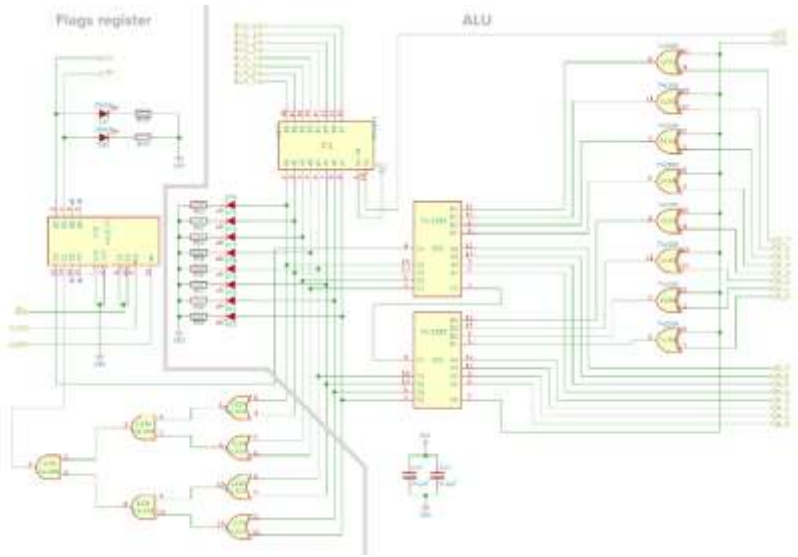

Fig. 7: Diagram showing the ALU module circuit for the CPU.

Display and Output Register
The **Output Register and Display Module** is responsible for converting 8-bit binary outputs from the ALU into human-readable decimal values, which are then shown on **seven-segment displays**. This module integrates a memory-based lookup mechanism and dynamic display control to visually represent signed numerical results.

Output Register Implementation
The output register is implemented using**2× 74HC173** – Quad D-type registers with tri-state outputs.
These registers are configured as an 8-bit latch to hold the ALU output. Upon activation of the **load signal**, the current bus value is latched into the output register and made available to the EEPROM input lines.

Decoder EEPROM (28C64)
An **EEPROM 28C64** is used as a hardware lookup table to translate binary values into seven-segment display patterns:

- It features **12 address lines**, but only the lower **8 address lines** are utilized.
- These 8 lines represent all 256 possible 8-bit values.

The EEPROM is programmed in two halves:

- **Addresses 0–127**: Correspond to positive binary values (0 to +127).
- **Addresses 128–255**: Represent negative values (using two's complement from -128 to -1).

Each EEPROM address stores a pre-defined byte pattern that directly maps to the correct segment activation for the displays. Negative results include an additional segment pattern to indicate a minus sign, enabling accurate signed output without requiring additional processing logic.

Seven-Segment Display Control

- The display is driven using **four seven-segment common-cathode displays**.
- Display multiplexing is performed via a **NE555 timer**, which generates a high-frequency clock signal.
- The timer sequentially enables each display digit at a speed imperceptible to the human eye (persistence of vision), giving the illusion of continuous illumination.

Segment Refresh Logic

The EEPROM output is fed to **BCD-to-seven-segment decoding logic[1 8 18]** or directly to the segment lines, depending on the bit configuration stored. The system cycles through each digit rapidly while maintaining segment data using synchronized enable lines, ensuring stable and flicker-free output.This module allows real-time binary-to-decimal output rendering using a purely hardware-based, programmable memory system, adhering to low-level digital design principles.

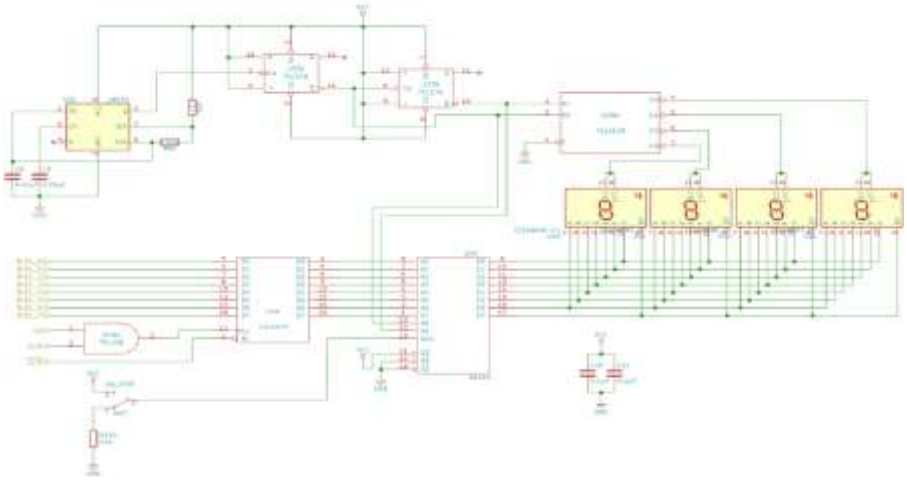

Fig. 8: Diagram showing the Output Register module circuit for the CPU.

Control World

The **Control Logic[1 8 18]** module governs the overall behavior of the CPU by orchestrating timing and data flow across various components using a **control word**. This control word is a 16-bit signal line distributed via **shift registers (2× 74HC595)**, [40] and is driven by an **Arduino Nano**, which interprets the current instruction and emits the corresponding sequence of control signals.Each bit in the control word represents a specific enable or control signal responsible for activating a part of the circuit during a particular step of the instruction execution cycle **[1 8]**. The defined control signals are as follows:

- **SCK** – Stop Clock: Halts clock pulses at the end of execution or upon certain instructions.
- **MARI** – Memory Address Register In: Loads the address lines from the bus into the Memory Address Register.
- **RIN** – RAM In: Enables data input into the RAM from the bus.
- **ROT** – RAM Out: Pushes data from RAM onto the bus.
- **IRO** – Instruction Register Out: Places the instruction register contents on the bus.
- **IRI** – Instruction Register In: Loads the current bus value into the instruction register.
- **ARO** – A Register Out: Transfers the contents of Register A onto the bus.
- **ARI** – A Register In: Loads data from the bus into Register A.

- **SUM OUT** – Sum Out: Enables ALU result (add or subtract) onto the bus.
- **SUB** – Subtract: Activates subtraction mode within the ALU using two's complement logic.
- **BRO** – B Register Out: Transfers Register B contents onto the bus.
- **BRI** – B Register In: Loads data from the bus into Register B.
- **OURI** – Output Register In: Transfers the bus contents into the output register.
- **CKOT** – Clock Out: Forwards the current clock signal to Memory Address Register input logic.
- **CKEN** – Clock Enable: Allows the clock pulses to propagate through the system.
- **CKLD** – Clock Load: Loads the program counter or step counter with the current clock cycle step.

This control scheme provides complete modular control of data movement, arithmetic processing, and instruction decoding, enabling the system to execute operations deterministically across each micro-instruction step.

Instruction Overlay Execution Model
The execution of instructions within the CPU is orchestrated by an **Arduino Nano**, which sends 16-bit control words serially to **two 74HC595 shift registers**. Each instruction is broken down into **micro-instruction steps (T0–T5)**, though most instructions execute fully within the first four (T0–T3). The system follows a RISC-like simplicity where instruction decoding and execution are compact and deterministic.

Common Starting Sequence (Applicable to All Instructions)
Every instruction cycle begins with a **two-step preamble**, regardless of the opcode. This ensures correct memory addressing and instruction retrieval:

- **T0 (000)**: MARI ← PC (Memory Address Register In)
- **T1 (001)**: ROT → IR (Instruction fetched from memory and loaded into Instruction Register)

Instruction: LDA (Load A Register)
Opcode: 0001
Loads the data from memory into Register A.

- **T2 (010)**: MARI + IRO – The lower nibble (address) from IR is placed into MAR.
- **T3 (011)**: ROT + ARI – Data from memory is fetched and stored in Register A.
- **T4 (100)**: No operation (NOP) — reserved for timing padding.

Instruction: ADD (Add with A Register)
Opcode: 0010
Adds the value from memory to Register A using Register B and ALU.

- **T2 (010)**: MARI + IRO – Address operand from IR sent to MAR.
- **T3 (011)**: ROT + BRO – Data fetched and stored in Register B.
- **T4 (100)**: SUM OUT + ARI – ALU adds A and B, result written back to Register A.

Instruction: OUT (Output to Display Register)
Opcode: 1110
Sends the content of Register A to the output display.

- **T2 (010)**: IRI + OURI – Data from Register A transferred to Output Register.
- **T3 (011)**: No operation.
- **T4 (100)**: No operation.

Instruction: HLT (Halt Execution)
Opcode: 1111
Stops the system clock, ending execution.

- **T2 (010)**: SCK – The Stop Clock signal is issued.
- **T3 (011)**: No operation.
- **T4 (100)**: No operation.

INSTRUCTION TO CONTROL WORLD MAPPING

Table1: Displaying the Control Abbreviations

Bit	Signal	Description
0	SCK	Stop Clock
1	MARI	Memory Address Register In
2	RIN	RAM In
3	ROT	RAM Out
4	IRO	Instruction Register Out
5	IRI	Instruction Register In
6	ARO	A Register Out
7	ARI	A Register In
8	SUM OUT	ALU Sum Out
9	SUB	ALU Subtract Mode
10	BRO	B Register Out
11	BRI	B Register In
12	OURI	Output Register In
13	CKOT	Clock Out to MARI
14	CKEN	Clock Enable
15	CKLD	Clock Load

Instruction Table Control Sequence
This is a hypothetical code which is being assumed for execution, the idea is to
LOAD A – Load the A Register with some value from the M address of the RAM
ADD B – Add some value from the N address of RAM with the A Register
OUT – Let the output register display the results of the RAM
SCK – Stop the clock after the execution is complete.

Table 2: Displaying the control sequence for a sample program

Step	LDA (0001)	ADD (0010)	OUT (1110)	HLT (1111)
T0	MARI	MARI	MARI	MARI
T1	ROT + IRI	ROT + IRI	ROT + IRI	ROT + IRI
T2	IRO + MARI	IRO + MARI	IRI + OURI	SCK
T3	ROT + ARI	ROT + BRI	—	—
T4	—	SUM OUT + ARI	—	—

The instruction cycle in this CPU operates in a sequential, step-by-step process, driven by the Arduino through shift registers. Upon receiving an instruction, the cycle begins with the **MARI** signal, which loads the memory address from the Program Counter (PC) into the Memory Address Register (MAR). In the first two steps, **ROT** sends the instruction stored at that address to the Instruction Register (IR). From there, based on the opcode, the cycle proceeds to perform different tasks. For example, with the **LDA (0001)** instruction, the address from the IR is passed to the memory, and data from the memory is fetched into Register A (via **ARI**). For the **ADD (0010)** instruction, the operand is fetched from memory into Register B, and the ALU performs the addition with the value in Register A, storing the result back in Register A. The output is handled by transferring data to the output register (via **IRI** and **OURI**) for display. After the execution of an instruction, the **SCK** (Stop Clock) signal is triggered in the case of a halt instruction (**HLT**), stopping the system clock and terminating the cycle. Each instruction follows a set pattern of control signals, ensuring that the CPU operates in a highly coordinated and systematic manner, with each step of the cycle triggering specific memory and register operations, leading to the final output or halt.

Arduino Code Breakdown
The Arduino driver code [19 8 9] is structured to control a hypothetical CPU that uses shift registers and step sequences to execute simple programs. The code performs a sequence of operations that fetch and execute instructions, leveraging shift registers to communicate between the Arduino and the control world. Here's a breakdown of the key components:

Pin Definitions and Setup

- **Shift Registers:**SHIFT_DATA, SHIFT_CLK, SHIFT_LATCH: These pins are connected to a shift register, which controls the data flow for instruction handling. The Arduino sends instructions to the control world via these pins using serial data transfer.The shift register helps the Arduino communicate with the control world (memory, registers, etc.) by shifting out data bit by bit.
- **Step Pins:**STP22, STP21, STP20: These pins represent the 3-bit step counter (T1-T5), controlling the step sequence and determining the current phase of the instruction cycle.
- **Instruction Pins:**INS23, INS22, INS21, INS20: These pins read the instruction bits from the instruction register. The binary value of the instruction is determined by reading these pins and combining the results to form a full instruction (4-bit in this case).

Instruction Decoding

- The instruction bits (from INS23 to INS20) are read and combined to form a 4-bit binary value, representing the current instruction.
- Based on the instruction bit value, specific actions are performed during the T3-T5 phases. The instructions correspond to operations such as Load Accumulator (LDA), Add, Output (OUT), and Halt (HLT).

Register In (IRI) to retrieve the instruction from memory and load it into the instruction register.

T3-T5 Instruction Steps
These functions handle the actions taken during steps T3, T4, and T5 of the instruction cycle. Depending on the opcode (the instruction), different operations are performed:

- **T3 (step 010)**: Controls the initial processing of the instruction. For example, LDA and ADD fetch data from memory and pass it to registers (like Register A).
- **T4 (step 011)**: Handles data movement, such as passing the data from Register A to Register B or executing specific actions related to instructions like ADD.
- **T5 (step 100)**: Completes the instruction by performing final operations like storing the result in the accumulator (Register A) or sending output to the output register.

Instruction Handling
For each instruction, the code specifies actions for each step of the cycle:

- **LDA (0001)**: Loads data from memory into Register A.
- **ADD (0010)**: Adds data from memory to Register A.
- **OUT (1110)**: Sends data from Register A to the output register.
- **HLT (1111)**: Halts the execution by triggering the **SCK (Stop Clock)** signal.

Timing and Delay
The loop runs with a delay(20) to ensure the cycle doesn't run too fast, with a total rate of 50Hz (1000/20ms). This provides a stable timing for the instruction cycle and prevents excessive clock speed.

Control Signal Management
Each instruction uses different control signals that are sent through the shift registers. The instructions like MARI, ARO, CKOT, SCK, etc., define the exact behavior of each operation, from reading memory to writing to registers or halting the system.

RESULTS AND DISCUSSIONS

The following microcode was run in the hardware-model which reads –

1. LDA 1110
2. ADD 1111
3. OUT
4. HALT

The value of 43 with the binary equivalent $(00101011)_2$ was loaded into the memory address of 1110 and the value of 100 with the binary equivalent of $(01100100)_2$ was kept in the memory address 1111.

Memory Configuration

Address	Data (Decimal)	Description
14	43	Value to Load
15	100	Value to Add

The detailed summary of the breakdown of the steps taken from the **STEP COUNTER** to run through the instructions are thus being broken down in the table:

Time Step	Instruction	Action Performed	Register/Bus Values
T0	LDA 14	Load address into MAR	PC = 0, MAR = 14
T1		Load value at address 14 into A Register	A-Reg = 43
T2	ADD 15	Add value at address 15 to B Register	B-Reg = 100
T3	OUT	Output value of ALU	Output = 143
T4	HLT	Halt program	Execution Stopped

The equivalent of the addition was performed by the ALU and following which the result was OUT through the 8-Bit BUS to the OUTPUT Register. The last clock cycle had set the sum result 143 $(10001111)_2$ to the 7 Segment Display as Displayed in **Fig9**. Thus, the computer performed a simple addition operation with 2 numbers. The observed clock cycle suggests that the entire operation was performed with just **11 Clock Cycles**, which when run using the actual hardware was just **0.11s at 100Hz**.

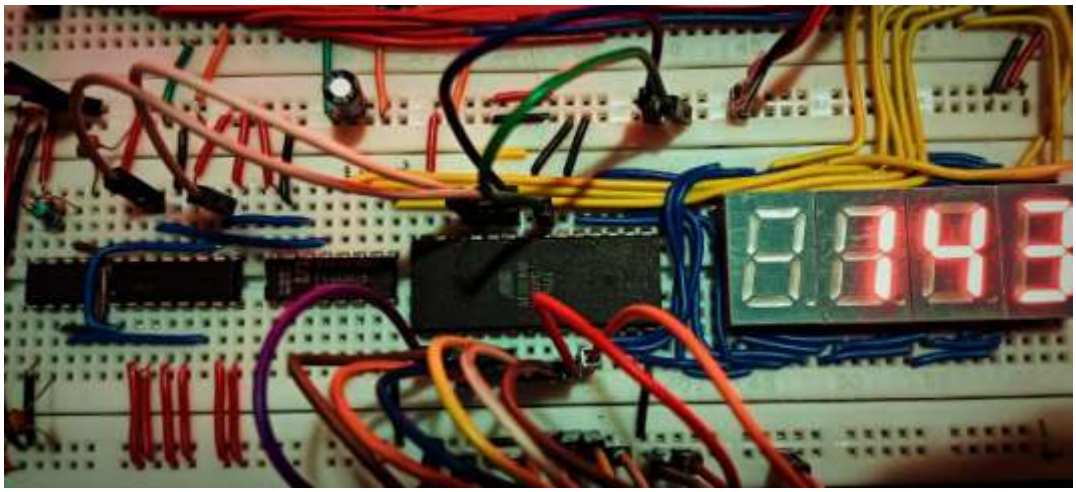

Fig. 9: Displays the Output Register Showing The Output Of 143 From A Sample Computation Of Addition Operation

CONCLUSION

The design and implementation of the RISC computer system using Arduino and shift registers demonstrates an effective approach to creating a simple CPU that can execute basic instructions in a controlled, step-by-step manner. By leveraging a 3-bit step counter and a 4-bit instruction register, this system simulates the core functionalities of a basic computer architecture, including memory addressing, data movement, arithmetic operations, and output handling.

Key takeaways from the implementation include:

1. **Step-by-Step Instruction Execution**: The system is organized around a sequential execution cycle, with each instruction being broken down into multiple steps (T1 to T5). This structure mirrors real-world CPU architectures, where different stages of instruction fetch, decode, and execute occur.

2. **Shift Register Communication**: The use of shift registers for communication between the Arduino and the control world allows for scalable, efficient bit manipulation. It enables the transfer of instruction data in a controlled manner, ensuring precise timing and synchronization with external hardware.
3. **Instruction Handling**: The instruction set, while simple, demonstrates fundamental operations like load, add, output, and halt (LDA, ADD, OUT, HLT). [19] These operations are executed through a combination of control signals sent to the shift registers, allowing the system to interact with memory and registers.
4. **Arduino as the Controller**: The Arduino functions as the central controller, managing the instruction cycle and issuing commands to the control world through the shift registers. By encoding instructions in binary and using control signals for different operations, the Arduino orchestrates the flow of the entire system.
5. **Modular and Expandable Design**: The code structure allows for easy expansion, both in terms of adding new instructions and enhancing the timing mechanism. The current system can be adapted for more complex instruction sets or additional components, such as more registers or peripheral devices.
6. **Timing and Synchronization**: With the use of a delay mechanism (set to 50Hz), the system ensures that instructions are executed at a stable and predictable rate. This timing control is crucial for ensuring that each step of the cycle is completed before moving to the next, preventing errors due to race conditions or misalignment.

In conclusion, this project provides a foundational understanding of how a basic CPU [1] can be built and controlled using simple hardware components like an Arduino **[1 8]** and shift registers. The modularity of the design allows for easy upgrades and the addition of new features, making this system a versatile starting point for building more complex computing systems. It demonstrates not only how basic instructions can be executed sequentially, but also how the principles of computing—such as data movement, arithmetic operations, and program control—can be implemented in a tangible, low-cost platform.

REFERENCES

[1]. B. Eater, *Build an 8-bit computer from scratch*, Eater.net, 2017. [Online]. Available: https://eater.net/8bit/
[2]. Arduino, *Serial to Parallel Shifting-Out with a 74HC595*, Arduino.cc, 2015. [Online]. Available: https://www.arduino.cc/en/Tutorial/ShiftOut
[3]. Codebender.cc Team, *How to Use a Shift Register - Arduino Tutorial*, Instructables, 2015. [Online]. Available: https://www.instructables.com/How-to-use-a-Shift-Register-Arduino-Tutorial/
[4]. Upcycle-Electronics, *8-Bit Breadboard Computer*, GitHub, 2020. [Online]. Available: https://github.com/Upcycle-Electronics/8-Bit-Breadboard-Computer
[5]. D. Megías and J. Prieto, "Computer Architecture Software-Based Simulation," *Journal of Digital & Analog Circuits & Systems*, vol. 1, no. 1, pp. 1–10, 2023. doi: 10.1109/JDAC.2023.00001.
[6]. J. Djamaluddin et al., "Teaching Computer Architecture by Designing and Simulating Processors," *International Journal of Computer Applications*, vol. 182, no. 24, pp. 7–13, 2019. doi: 10.5120/ijca2019918574.
[7]. D. Malvino and A. Brown, *Digital Computer Electronics*, 3rd ed. New York, NY, USA: McGraw-Hill Education, 1992.
[8]. R. S. Sedha, *A Textbook of Applied Electronics*, New Delhi, India: S. Chand Publishing, 2008.
[9]. Texas Instruments, "74LS08 Quad 2-Input AND Gates," *SN74LS08 Datasheet*, Rev. F, Apr. 1996. [Online]. Available: https://www.ti.com/lit/ds/symlink/sn74ls08.pdf
[10]. Texas Instruments, "74LS04 Hex Inverter," *SN74LS04 Datasheet*, Rev. D, Jan. 1998. [Online]. Available: https://www.ti.com/lit/ds/symlink/sn74ls04.pdf
[11]. Texas Instruments, "74LS32 Quad 2-Input OR Gates," *SN74LS32 Datasheet*, Rev. E, Apr. 1996. [Online]. Available: https://www.ti.com/lit/ds/symlink/sn74ls32.pdf
[12]. Texas Instruments, "74LS173 4-Bit D-Type Register with 3-State Outputs," *SN74LS173 Datasheet*, Rev. C, Nov. 1999. [Online]. Available: https://www.ti.com/lit/ds/symlink/sn74ls173.pdf
[13]. Microchip Technology Inc., "AT28C256 – 32K x 8 EEPROM," *AT28C256 Datasheet*, Rev. G, 2007. [Online]. Available: https://ww1.microchip.com/downloads/en/DeviceDoc/doc0006.pdf
[14]. Intel Corp., "Intel 8085 Microprocessor Architecture," *Intel 8085 Hardware Manual*, 1982.
[15]. LTspice Analog Devices, *LTspice XVII - Circuit Simulation Software*, Analog Devices, 2020. [Online]. Available: https://www.analog.com/en/design-center/design-tools-and-calculators/ltspice-simulator.html
[16]. Logisim, *Digital Circuit Simulator*, Version 2.7.1, 2011. [Online]. Available: http://www.cburch.com/logisim/
[17]. Proteus Design Suite, *Proteus Simulation Software*, Labcenter Electronics, 2021. [Online]. Available: https://www.labcenter.com/
[18]. KiCad Project, *KiCad EDA - Open Source PCB Design Tool*, Version 7, 2023. [Online]. Available: https://www.kicad.org/
[19]. Arduino UNO Rev3, *Microcontroller Board Based on ATmega328P*, Arduino.cc, 2021. [Online]. Available: https://store.arduino.cc/products/arduino-uno-rev3
[20]. Raspberry Pi Foundation, "Raspberry Pi Pico Datasheet," RP2040 Documentation, Jan. 2021. [Online]. Available: https://datasheets.raspberrypi.com/pico/pico-datasheet.pdf

[21]. Texas Instruments, "SN74LS00: Quad 2-Input NAND Gates," *SN74LS00 Datasheet*, Rev. E, May 2003. [Online]. Available: https://www.ti.com/lit/ds/symlink/sn74ls00.pdf
[22]. Texas Instruments, "SN74LS02: Quad 2-Input NOR Gates," *SN74LS02 Datasheet*, Rev. B, Mar. 2000. [Online]. Available: https://www.ti.com/lit/ds/symlink/sn74ls02.pdf
[23]. Texas Instruments, "SN74LS86: Quad 2-Input Exclusive-OR Gates," *SN74LS86 Datasheet*, Rev. D, Apr. 1996. [Online]. Available: https://www.ti.com/lit/ds/symlink/sn74ls86.pdf
[24]. Texas Instruments, "SN74LS244: Octal Buffers and Line Drivers With 3-State Outputs," *SN74LS244 Datasheet*, Rev. B, Oct. 1999. [Online]. Available: https://www.ti.com/lit/ds/symlink/sn74ls244.pdf
[25]. Texas Instruments, "SN74LS245: Octal Bus Transceivers," *SN74LS245 Datasheet*, Rev. C, Jun. 2003. [Online]. Available: https://www.ti.com/lit/ds/symlink/sn74ls245.pdf
[26]. Texas Instruments, "SN74LS273: Octal D-Type Flip-Flops With Clear," *SN74LS273 Datasheet*, Rev. A, Jan. 1999. [Online]. Available: https://www.ti.com/lit/ds/symlink/sn74ls273.pdf
[27]. Texas Instruments, "SN74LS181: 4-Bit Arithmetic Logic Unit," *SN74LS181 Datasheet*, Rev. D, May 2000. [Online]. Available: https://www.ti.com/lit/ds/symlink/sn74ls181.pdf
[28]. Texas Instruments, "SN74LS193: 4-Bit Synchronous Binary Counter," *SN74LS193 Datasheet*, Rev. A, Apr. 2000. [Online]. Available: https://www.ti.com/lit/ds/symlink/sn74ls193.pdf
[29]. Texas Instruments, "SN74LS147: 10-Line to 4-Line Priority Encoders," *SN74LS147 Datasheet*, Rev. A, Mar. 1997. [Online]. Available: https://www.ti.com/lit/ds/symlink/sn74ls147.pdf
[30]. Texas Instruments, "SN74LS138: 3-Line to 8-Line Decoder/Demultiplexer," *SN74LS138 Datasheet*, Rev. C, Feb. 2000. [Online]. Available: https://www.ti.com/lit/ds/symlink/sn74ls138.pdf
[31]. Texas Instruments, "SN74LS157: Quad 2-to-1 Multiplexers," *SN74LS157 Datasheet*, Rev. B, Mar. 1996. [Online]. Available: https://www.ti.com/lit/ds/symlink/sn74ls157.pdf
[32]. Texas Instruments, "SN74LS32: Quad 2-Input OR Gates," *SN74LS32 Datasheet*, Rev. E, Apr. 1996. [Online]. Available: https://www.ti.com/lit/ds/symlink/sn74ls32.pdf
[33]. Microchip Technology Inc., "AT28C64B: 8K x 8 EEPROM," *AT28C64B Datasheet*, Rev. C, Jun. 2005. [Online]. Available: https://ww1.microchip.com/downloads/en/DeviceDoc/doc0006.pdf
[34]. Fairchild Semiconductor, "DM74161: 4-Bit Synchronous Binary Counter," *Datasheet*, Rev. J, Jan. 1996. [Online]. Available: https://media.digikey.com/pdf/Data%20Sheets/Fairchild%20PDFs/DM74LS161.pdf
[35]. Texas Instruments, "CD4511B: BCD to 7-Segment Latch/Decoder/Driver," *CD4511B Datasheet*, Rev. E, Jan. 2022. [Online]. Available: https://www.ti.com/lit/ds/symlink/cd4511b.pdf
[36]. Texas Instruments, "SN74LS47: BCD to 7-Segment Decoder/Driver," *SN74LS47 Datasheet*, Rev. C, Feb. 1998. [Online]. Available: https://www.ti.com/lit/ds/symlink/sn74ls47.pdf
[37]. Texas Instruments, "SN74LS90: Decade Counter," *SN74LS90 Datasheet*, Rev. D, Nov. 1996. [Online]. Available: https://www.ti.com/lit/ds/symlink/sn74ls90.pdf
[38]. Texas Instruments, "SN74LS93: 4-Bit Binary Counter," *SN74LS93 Datasheet*, Rev. A, Aug. 1998. [Online]. Available: https://www.ti.com/lit/ds/symlink/sn74ls93.pdf
[39]. Microchip Technology Inc., "AT28C010: 128K x 8 Parallel EEPROM," *AT28C010 Datasheet*, Rev. G, 2005. [Online]. Available: https://ww1.microchip.com/downloads/en/DeviceDoc/doc0336.pdf
[40]. Texas Instruments, "SN74HC595: 8-Bit Serial-In, Parallel-Out Shift Registers," *SN74HC595 Datasheet*, Rev. L, Nov. 2015. [Online]. Available: https://www.ti.com/lit/ds/symlink/sn74hc595.pdf