# INDY 04 - - - 8-Bit CPU - - - Final Report

---

**CS4850 - Section 01, Spring 2026**

**Kyran Day, John Dislen, Samuel Hoerner, Aryan Merchant**

**Professor: Sharon Perry**

**Date: 5/4/2026**

**Website:** https://www.8-bitcpu.com/

**Github:** https://github.com/8-bitcpu/8-bitcpu

**Statistics:**

- Around 650 Transistors
- 30 Breadboards
- 4 months to learn
  - Electrical Engineering
  - HDLS / FPGA Programming
  - Embedded C and Embedded Assembly
- Components:
  - Raspberry
  - CMOD-A7-35t FPGA (Verilog HDL, circuit design, digital design)
  - Raspberry Pi Pico (embedded Python, C, ARM ASM)
  - Electrical components: transistors, resistors, jumper wires, breadboards
  - OLED screen
- **Total man-hours:** ~ 425 hours
- **Status:** 95% Completion

# Table of Content

**1.0 Introduction** **2**
**2.0 Challenges and Assumptions** **2**
Project Goals 2
Assumptions 2
**3.0 Requirements** **3**
3.1 Environment 3
3.2 System 4
3.3 Receiver 4
3.4 Functional Requirements 4
**4.0 Analysis / Design** **6**
4.1 System Overview 6
4.2 Modularity Strategy 7
4.3 Architecture 7
4.4 Software Design 7
4.5 Instruction Processing Flow 8
**5.0 Development** **8**
5.1 Implementation Details 8
5.2 Data and Connectivity 14
5.3 Project Setup 15
5.4 Limitations/Issues 16
**6.0 Software Testing** **16**
6.1 Scope of Testing 17
6.2 Test Cases 19
6.3 Software Test Report (STR) 24
**7.0 Version Control Plan** **25**
**8.0 Conclusion** **25**
**Appendix A - Project Plan** **26**

# 1.0 Introduction

This project presents the design and implementation of a custom 8-bit CPU built to demonstrate how computer architecture concepts work in real hardware. The CPU will be in the style of the 1970s. The system combines physical logic components like drain logic and Kirchhoff's law, registers, an ALU, a FPGA-driven Control Unit, a shared data bus, and an external memory system supported by a Raspberry Pi Pico. The CPU is designed to perform basic operations such as loading, storing, adding, subtracting, comparing, jumping, and handling input/output behavior. The project focuses not only on building the CPU components, but also on using our custom Instruction Set Architecture (ISA) or assembly to execute programs and display them on the OLED screen.

The goal of this design is to show how instructions move through a processor using the Fetch, Decode, and Execute cycle. During execution, the Program Counter tracks the next instruction, the Instruction Register holds the current instruction, the Control Unit decides which signals should be activated, and the ALU performs arithmetic or logic operations. The project also emphasizes modular design, meaning each major component can be designed, tested, and improved separately before being connected into the full CPU system. Overall, this report will explain the requirements, architecture, development process, testing methods, and final design of the 8-bit CPU as both an educational proof of concept and a prototype.

# 2.0 Challenges and Assumptions

## Project Goals

- Design ALU
- Design 8-bit registers (read/write)
- Design datapath (bus/mux routing between regs and ALU)
- Design control interface
- Define software/hardware interface (memory map)
- Write software driver

These are just assumptions we have at the beginning of the project.

# Assumptions

## Hardware

- All registers will be 8-bit
- Inputs and outputs are 8-bit
- Results wider than 8 bits will be wrapped around

## ALU

Functions of the ALU must include:

- ADD
- SUB
- AND
- OR
- NOT
- EQUAL

## Registers

Registers must be able to write and output.

There will be 4 registers total.

# 3.0 Requirements

## 3.1 Environment

The software will be run on a Raspberry Pi PICO. We will use the pins on the board to communicate with the “CPU”. From a software perspective, the CPU we are communicating to a “black box”; there is no way for the software sending signals to ever know if what is being sent is properly interpreted correctly. Likewise, the software made to interpret the output signal won't know when a signal is coming and must always listen and parse incoming signals.

## 3.2 System

The system we run our software on will be bare metal C/C++; there is no OS, and it must be precompiled before being installed on the board.

## 3.3 Receiver

The receiver must be able to sit in a perpetual state of listening, parse signals as they arrive correctly, and display the output.

## 3.4 Functional Requirements

The software will be two independent programs, one to send signals to the CPU and one to receive signals from it.

### Sender Software:

Software must be able to take input from a keyboard and translate user input into binary commands for the CPU to process. Sender must be able to translate the following commands into signals as well.

0000 — LOAD

Load value at memory location into accumulator (AC).

AC ← M[MAR]

0001 — STORE

Store value from accumulator into memory location.

M[MAR] ← AC

0010 — ADD

Add value at memory location to accumulator.

MBR ← M[MAR]

AC ← ALU ADD

0011 — SUBTR

Subtract value at memory location from accumulator.

MBR ← M[MAR]

AC ← ALU SUBTR

0100 — GT

Set boolean flag if accumulator value is greater than value at memory location; otherwise clear flag.

AC[0] ← ALU GT

0101 — EQ

Set boolean flag if accumulator value equals value at memory location; otherwise clear flag.

AC[0] ← ALU EQ

0110 — NOT

Negate boolean flag bit.

AC[0] ← ALU NOT

0111 — JMP

Set program counter to value in memory address register.

PC ← MAR

1000 — SKIF

Skip next instruction if boolean flag is set.

If AC[0] = 1 → PC ← PC + 1

1001 — NEXT

Select next 16-byte memory bank and set program counter to memory address register.

PC ← MAR

Memory[Next_Pin] ← High

10100000 — BACK

Select previous 16-byte memory bank and set program counter to memory address register.

PC ← MAR

Memory[Back_Pin] ← High

11110000 — HALT

Reset program counter to zero.

PC ← 000

# 4.0 Analysis / Design

## 4.1 System Overview

The system is meant to be an instruction-driven 8-bit processing architecture that executes a minimal instruction set. At a high level, the system will operate by

- fetching instructions from memory
- decoding those instructions
- executing the operations

These operations will consist of arithmetic and logical computation. Additionally, the flow of operations will consist of reading data from memory and writing it to memory to perform computation or alter the flow of execution.

The system will consist of

- Control Unit (CU)
- Internal Register (IR)
- Program Counter (PC)
- Accumulator Register (AC)
- Memory Buffer Register (MBR)
- Arithmetic and Logical Unit (ALU)
- External memory subsystem

Inputs to the system include instruction data and memory values, while outputs consist of computed results or memory contents. Overall behavior of the system will follow a structured execution cycle that ensures consistent and predictable operation.

## 4.2 Modularity Strategy

A significant aspect of the design is its modularity for each component with a phased implementation strategy. Development follows a phased strategy:

- Initial design in Multisim
- Physical implementation using transistors
- Final integration and testing with the PI-pico

Major subsystems, such as the ALU, control logic, registers, and memory interface, are designed to be

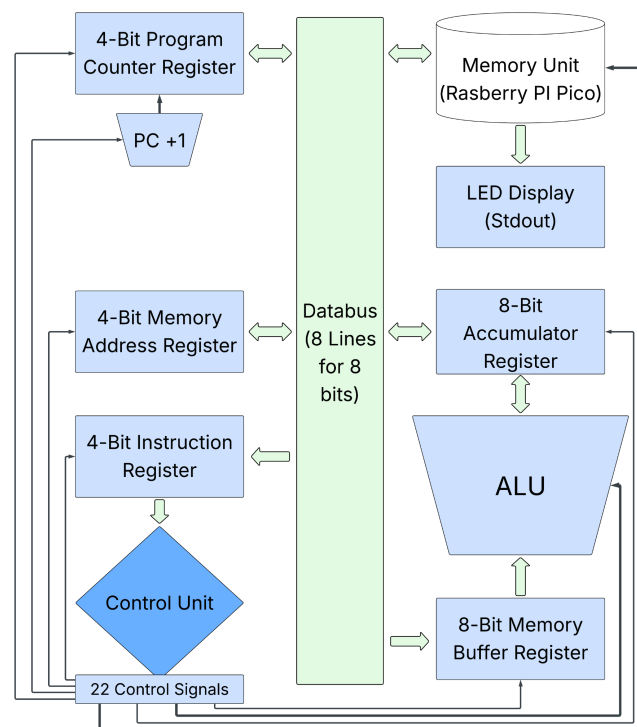
- constructed independent
- tested independently.

This modular approach will have multiple benefits:

- minimizes risk during physical construction
- supports iterative development and refinement

## 4.3 Architecture

The architecture of the system will be organized into several modularized components, which will communicate through a controlled signal to achieve the corresponding execution. At the center of the architecture, CU will determine which operations are performed at each stage. The CU will interact with the IR, which holds the current executing instruction. The PC will track the location of the next instruction to be fetched. There will also be an external memory system using a PI-Pico that will provide storage for instructions and data.




Regarding the external memory, the Pi-Pico will be used as an interface that allows the system to read from and write to memory locations, as well as navigate between different memory regions. The ALU performs computational operations such as addition, subtraction, and logical comparisons. Registers, such as the AC and MBR, are used to temporarily hold intermediate values during execution. Custom byte-sized assembly language will be used to translate to the machine code.

## 4.4 Software Design

- The PI-pico will serve as an external memory, which represents the active byte address
- There will be a continuous stream of 8-bit array values to represent memory
- The pico will operate in a continuous loop, interpreting control signals and updating memory state that samples control inputs, evaluates the requested memory action, updates internal state, and produces output data when required
- When both software signals are connected, the pico updates the memory pointer using the input data
- During the read operation, the pico outputs the byte code stored at the current address to the processor
- While in the write process, the other software will store the incoming data at the current location

## 4.5 Instruction Processing Flow

Instruction execution will follow the FDR cycle

- Fetch
- Decoding
- Execution cycle

During the fetch stage, the system retrieves the next instruction from memory using the PC and stores it in the IR. The Program Counter is then updated for the next instruction. In the decode stage, the CU will interpret the instruction to determine the operation. During the execute stage, the system performs the specified operation. This may involve reading from or writing to memory, performing an arithmetic or logical operation, or modifying control flow. Upon completion, the system updates its internal state as needed and prepares to fetch the next instruction.

# 5.0 Development

## 5.1 Implementation Details

### System Overview

This project is a custom 8-bit Central Processing Unit (CPU) implemented from a logic-based design using Multisim to a hardware-based design, inspired by early computer architectures from the 1960s-1970's, where:

- Instructions are executed sequentially
- Memory is external
- Data is followed through a shared bus system
- Breadboards, transistors, and resistors are commonly used
- The Control Unit (CU) is handled via micro-operation
- Registers store data, instructions, and intermediate results
- Registers also enable synchronous data flow over the shared bus
- The Arithmetic Local Unit (ALU) performs all arithmetic and logical operations

## Big Picture Integration

The 8-bit CPU is mainly hardware-based, integrated with a software architecture composed of:

- Hardware Components
  - ALU
  - Registers
  - CU
  - Data Bus
  - Memory
- Software Components
  - MicroPython
  - Pi Pico W(Memory)

Communication flow between all of the components:

- The CU gives control signals through 4 pins: Read, Write, Next, and Back
- The Data Bus communicates 8-bit values through multiple components using GPIO pins
- Pi Pico acts as RAM, storing and retrieving data using GPIO input/output
- Each component interacts through synchronized control signals + shared bus lines

## Fetch Execution (FDE)

The FDE cycle is the process by which the 8-bit CPU operates and executes each operation. It describes how each instructions stored in memory are continuously processed where:

- In the Fetch stage, the CPU retrieves an instruction from memory using the program counter
- In the Decode stage, the CU interprets the instruction to determine the required operation
- In the Execute stage, the CPU operates using components like the ALU, registers, and data bus.

## Design Decisions

Overall design follows practical and easily implemented steps that prioritize simplicity, ease of implementation, and components that can be implemented individually. Strategies that give us the advantage of gliding through the project easily are:

- CU design
    - Utilizes an FPGA to avoid the number of transistors that would be used
- Memory design
    - Using external Memory via PI Pico
    - Replaces physical memory chips
- Modular strategy
    - Each component is built and tested independently
    - If any errors are detected, the whole CPU does not have to be re-implemented
    - If any future changes we make, like user input on a keyboard to make future improvements, then we can make those advancements

## Control Unit (CU)

The CU is the brain of the CPU, deciding how the data flows and how each component operates during instructions. Responsibility of the CU is generating the control signals, managing the FDE cycle, and ensuring all components of the system work synchronously, like the register, ALU, and memory. Micro-operations are designed per instruction. An example of these instructions for addition would include:

- MBR <- Memory[MAR]
- AC <- ALU[Addition]
- Control signals map
    - AC[Write]
    - ALU[Add]
    - Memory[Write]

The instruction format for this 8-bit CPU is divided into two distinct 4-bit fields: opcode and instruction. The upper 4 bits (bits 7–4) represent the opcode, which defines the operation to be performed by the CPU:

- ADD
- SUB

- LOAD
- STORE
- JUMP

These instructions allow for up to 16 unique instructions. The lower 4 bits (bits 3–0) represent:

- address or operand, typically pointing to a memory location 0–15

This instruction format simplifies decoding, as the Control Unit (CU) can directly split the instruction byte into operation and operand without additional parsing logic. It also aligns with the system's shared bus design, where the address field is routed to the Memory Address Register (MAR), and the opcode is decoded to generate appropriate control signals for execution.

All of these executions are done using an FPGA, which is a programmable hardware device that is used to design circuits using the Verilog language.

## Arithmetical Logical Unit (ALU)

The purpose of the ALU is to perform all of the arithmetical and logical computations:

- Addition
- Substraction
- Increment
- Decrement

Logical operations include:

- AND
- OR
- NOT
- Equal
- Greater

The hardware implementation of this build is using the main big components that include:

- Adders (for arithmetical operations)
- Logic gates (AND, OR, NOT, Equal, Greater)
- Multiplexer (operational selection)

Operational selection happens via opcode and decode. ALU is its own component, like others, but it only performs the computation and does not store data. There are several operations that are controlled by the CU signals. Each operation updates the flag like carry -> overflow, in addition, negative -> MSB = 1, and overflow -> signed overflow.

Example execution flow:

- R1 → ALU input
- R2 → ALU input
- ACC ← ALU result
- Update flags

## Registers

Registers are implemented as the primary working storage within the CPU, allowing temporary storage of operands and intermediate computation results. These registers enable efficient data manipulation without frequent memory access. Each register is designed to interface directly with the shared system bus.

The Instruction Register (IR) is designed to hold the current instruction being executed. Once an instruction is fetched from memory, it is loaded into the IR, where it remains stable during decoding and execution. This allows the Control Unit to generate the appropriate micro-operations. IR is implemented as an 8-bit register, which:

- Loads during fetch cycle: IR ← MDR
- Holds: Opcode Operand bits (depending on instruction format)
- Ensures instruction stability during the execution phase
- Output connected to: Control Unit (decoder logic)
    - Controlled by: IR_LOAD

The Program Counter (PC) is implemented to track the address of the next instruction to be executed. It enables instruction execution by automatically incrementing after each fetch cycle, while also supporting control flow changes through jumps and branches. This register is critical for maintaining the execution order of the program.

- Implemented as an 8-bit register with incrementation
- Connected to the address bus via MAR
- Supports operations: Increment (PC ← PC + 1) and Load (for jumps/branches)
- Used in instruction fetch: MAR ← PC

The MAR is designed to hold the address of the memory location being accessed. It acts as the interface between the CPU and external memory by driving the address bus. Since memory is external, MAR plays a crucial role in all read and write operations. This is implemented as an 8-bit register, connected directly to the address bus.

- Receives input from: Program Counter (during fetch) and Bus (during data access)
- Controlled by: MAR_LOAD
- Used in operations: MAR ← PC MAR ← Address
- Ensures the correct memory location is accessed

The Memory Buffer Register (MBR) is implemented to temporarily hold data being transferred between the CPU and memory. It acts as a buffer that isolates the CPU from direct memory timing constraints. Additionally, MBR ensures stable data transfer during read and write operations.

- Used in: MDR ← Memory[MAR] and Memory[MAR] ← MDR
- Helps synchronize CPU and memory communication
- Connected between:
    - Data bus
    - External memory data

## Memory System

Memory is external, serving as the RAM using the Pi Pico programmed in Python. MicroPython is used to program GPIO pins to listen to the data coming from the data bus and output the data to the bus. GPIO pins are listed as follows:

- In [0-7]
- Out [0-7]
- Write_Pin - Write Control Pin
- Read_Pin - Read Control Pin
- Next_Pin - NEXT Pin
- Back_Pin - BACK Pin
- Output_Pin - Reserved for future OUT operation
- Input_Pin - Reserved for future IN operation

The memory will have the following local variables: UINT8[]  Data[0-n], where n is evenly divisible by 16, Int Memory Pointer: A positive number that holds a memory address. Operations for these pins are:

- Set Memory Location Condition: Write = HIGH and Read = HIGH
  The memory pointer is updated using the input data bus. Memory Pointer ← IN[0–7]
- Output Data (Read Operation) Condition: Write = LOW and Read = HIGH
  The value stored at the current memory location is placed on the output bus. OUT[7–0] ← DATA[Memory Pointer]
- Input Data (Write Operation) Condition: Write = HIGH and Read = LOW
  The value from the input bus is written into the current memory location. DATA[Memory Pointer] ← IN[0–7]

- Bias to Next 16 Bytes Condition: Next = HIGH
  The memory pointer moves forward by 16 addresses. Memory Pointer ← Memory Pointer + 16
- Bias to Last 16 Bytes Condition: Back = HIGH
  The memory pointer moves backward by 16 addresses. Memory Pointer ← Memory Pointer - 16
- Reset Memory Pointer Condition: Back = HIGH and Next = HIGH
  The memory pointer is reset to 0. Memory Pointer ← 0

## Instruction Set

| Opcode | Instruction |
| --- | --- |
| 0000 | LOAD |
| 0001 | STORE |
| 0010 | ADD |
| 0011 | SUB |
| 0111 | JUMP |
| 1000 | SKIP |
| 1011 | IN |
| 1100 | OUT |

### Data Bus

The bus of the 8-bit CPU is designed around a shared communication system that allows all core components to exchange data and control information efficiently. The system utilizes an 8-bit data bus, which serves as the primary pathway for transferring values between registers, the ALU, and memory. The entire system operates using standard digital voltage levels, where 0 volts represents a logic low (0) and 5 volts represents a logic high (1). This clear distinction in voltage levels ensures reliable signal interpretation across all hardware components, enabling synchronized and accurate execution of instructions.

## 5.2 Data and Connectivity

### Connection Overview

- No database

### Configuration Details

- No API endpoints or anything related

### Integration Logic

Software communicates with the hardware components using the Raspberry Pi Pico with MicroPython. All of the implementation logic is in the Memory component on the previous page. Important components of the software include:

- GPIO Pins: GP0–GP7 → Data input
- GP12–GP19 → Data output
- Control pins → READ / WRITE / NEXT / BACK
- The CPU sends control signals
- Pico reads signals
- Pico performs:
  - Memory read/write
  - Outputs data onto the bus

## 5.3 Project Setup

### Prerequisites

- Software
    - Python installed on Pi Pico
    - Thonny IDE (optional) for debugging
    - MicroPython firmware (optional)

- Hardware
    - Raspberry Pi Pico
    - Breadboard
    - LEDs + resistors
    - FPGA

### Installation Steps

- Flash Python code onto Pi Pico
- Have the CPU all wired up with the hardware components and the GPIO pins on Pi Pico.

### Verification

- System works if:
    - Memory outputs correct values
    - ALU produces correct results
    - PC increments correctly
    - Instructions execute in order

### Current State

- An 8-bit CPU is logically implemented using Multisim
- All of the individual hardware-based components are in progress
- Registers are being debugged and tested
- Memory implementation using the Pi Pico is done
- Including user input by implementing the memory is being focused on
- ALU is done
- Mainly, the components need to be tested connectively

## 5.4 Limitations/Issues

- There were synchronization issues in the CU section
- There is a learning curve to using an FPGA for the CU
- An edge case error has been detected right now about the CU being talked through using the data bus that was not discovered before, and the solution for the edge case is planned out and needs to be implemented
- GPIO pins are limited, so maybe another Pi Pico might be implemented
- Wiring issues that can happen anytime, which we are consciously being careful about

# 6.0 Software Testing

Software testing is conducted to explain the instructions to test the operation of the 8-bit CPU. These operations include the memory module of the Pi-Pico, control-signal behavior, instruction execution, and output functionality. Verifying these operations will also make sure the hardware part of the CPU is working correctly.

Testing is designed to ensure the correctness of the software:

- Unit level (individual memory module)
- Signal level (GPIO control behavior)
- Instruction level (CPU execution correctness)
- System level (end-to-end program execution)

Additionally, this document includes a Software Test Report (STR) section to record:

- Pass/fail results
- Observed defects
- Severity levels

## 6.1 Scope of Testing

### In-Scope Testing

The scope of testing includes all components and features that are essential to the correct operation of the 8-bit CPU system. Testing is designed to verify both individual functionality and system-level integration using the software. Each component is evaluated and seen using the Raspberry Pi Pico and the output on the OLED screen on how it interacts with other parts of the system.

The following testing areas include:

- Memory System: Testing ensures that the external RAM behaves correctly, including initialization, reading and writing data, and pointer navigation. This is critical because memory is the foundation for instruction storage and execution
- Control Signals: GPIO-based signals will be tested to confirm that they correctly trigger memory and CPU operations. Since the system relies heavily on signal combinations (e.g., READ and WRITE together), verifying these interactions is essential
- CPU Execution: The fetch and execute cycles will be tested to ensure that instructions are properly decoded and executed. This includes validating that the CPU follows the correct sequence of operations
- Instruction Set: Core instructions such as LOAD, STORE, ADD, SUB, JMP, conditional operations, and HALT will be tested individually to ensure the correct decimal for the instruction is implemented
- Input/Output Behavior: Testing will verify that input signals correctly capture data and that output signals correctly display or transmit results. This includes validating string-building or display mechanisms.
- Program Execution: Preloaded programs (such as printing sequences or running Fibonacci logic) will be tested to confirm that the system performs complete workflows correctly with the whole CPU

## Out-of-Scope-Testing

We aim to make the testing as detailed as possible. However, certain areas, such as detailed isolation of hardware testing, are excluded due to project constraints. These exclusions help keep the testing focused on core software output functionality rather than going into hardware testing in detail. Although testing the hardware components is also on our mind. But, hardware testing, like the Memory Address Register, will be making sure the representation of the address LEDs, which should simplify for that operand. This step is taken because testing can also be done using the software, which will be visible on the OLED screen.

More out-of-scope testing includes:

- Long-term hardware reliability
- Hardware performance optimization
- Physical hardware durability

## Testing Environment

Testing will be done individually for the memory module in a controlled environment using toggled pins for control pins and the input bus. The overall testing of the CPU will be done by pre-loading a program into the memory and getting the expected output on the “OUT” control signal.

The software tools support development, execution, and debugging. These tools are necessary to load programs, monitor outputs, and analyze system behavior during memory module testing and the overall testing:

- Pico SDK (C/C++)
- Visual Studio Code with CMake for debugging
- OLED screen for debugging output
- Correct precompiled CMake build file (.uf2) for flashing the Raspberry Pi-Pico

The system relies on a defined GPIO (listening Raspberry Pi-Pico pins) mapping that simulates buses and control signals. Correct implementation is essential for accurate testing:

- Input bus mapped to GPIO pins (e.g., GP0–GP7), giving a byte
- Control signals assigned to dedicated pins (READ, WRITE, NEXT, BACK, IN, OUT)

- Output bus mapped to GPIO pins, bits that will go to the registers (e.g., GP12–GP19)
- Control signals assigned to dedicated pins:
    - READ (GP9 pin)
    - WRITE (GP10 pin)
    - NEXT (GP11 pin)
    - BACK (GP12 pin)
    - IN (GP20 pin)
    - OUT (GP21 pin)

## 6.2 Test Cases

Test cases include testing the memory using the Raspberry Pi Pico.

1. Tc1
    - Description: Verify that memory initializes correctly when the system starts
    - Expected Result: All memory locations are set to 0
2. TC2
    - Description: Verify that writing a value to memory stores the correct data at the current pointer location when WRITE = 1 and READ =1
    - Expected Result: The value written to memory is stored correctly at the specified address
3. TC3
    - Description: Verify that reading from memory returns the correct value stored at the current pointer location when WRITE = 0 and READ = 1
    - Expected Result: The value retrieved from memory matches the value previously written
4. TC4
    - Description: Verify that setting the memory pointer updates the pointer to the correct address when READ pin = 1 and WRITE pin = 1
    - Expected Result: The pointer is updated to the specified address within the valid memory bounds of 0-255 using

5. TC5
    - Description: Verify that the NEXT operation increments the pointer by the defined page size when NEXT = 1
    - Expected Result: The pointer increases by the page size (e.g., +16) and wraps correctly if needed
6. TC6
    - Description: Verify that the BACK operation decrements the pointer by the defined page size when BACK = 1
    - Expected Result: The pointer decreases by the page size and wraps correctly within memory limits
7. TC7
    - Description: Verify that the RESET operation sets the pointer back to the starting address when NEXT = 1 and BACK = 1
    - Expected Result: The pointer is set to 0
8. TC8
    - Description: Verify that the bits being high are translated to the correct value, which can be used for different operations
    - Expected Result: When loading the instruction STORE, the GP4 pin will be high, and the other GP0 - GP7 pins will be low and will be translated as 16 on the “IN” of the OLED screen
9. TC9
    - Description: Verify that the IN signal captures input data and appends it to the string builder or buffer
    - Expected Result: The input value is correctly appended to the buffer in the correct sequence when the IN pin is high
10. TC10
    - Description: Verify that the OUT signal outputs the contents of the buffer to the display or output bus
    - Expected Result: The correct data from the string builder is displayed in the correct sequence
11. TC11
    - Description: Verify that the system displays a HALT message when execution stops
    - Expected Result: The display shows “HALT
12. TC12
    - Description: Verify that a preloaded program correctly prints the intended output using the system workflow
    - Expected Result: The output displays the right output in the correct order.

13. TC13
    - Description: Verify that the system remains stable after executing a program and reaching HALT
    - Expected Result: No unintended memory changes or operations occur after HALT

## Test Procedures

Each test case is executed using a consistent and repeatable set of steps. Clearly defined procedures ensure that testing can be performed reliably and that results can be compared across multiple runs. These procedure steps are executed in a toggling pins functionality that will operate the behavior that will happen in the final working scenario:

1. TC1
    a. Step 1: Power on the Raspberry Pi Pico with the memory firmware loaded
    b. Step 2: Allow system initialization to complete
    c. Step 3: Read multiple memory addresses using READ = 1
    d. Step 4: Observe output values via the output bus or debug serial
    e. Expected Outcome Check: All memory values should be 0

2. TC2 – Memory Write Operation (WRITE = 1, READ = 1)
    a. Step 1: Set input bus (GP0–GP7) to desired address value
    b. Step 2: Set WRITE = 1 and READ = 1 to update pointer
    c. Step 3: Change the input bus to the desired data value
    d. Step 4: Set WRITE = 1 and READ = 0
    e. Step 5: Trigger operation
    f. Step 6: Set READ = 1 and WRITE = 0
    g. Expected Outcome Check: Value written matches value read

3. TC3 – Memory Read Operation (WRITE = 0, READ = 1)
    a. Step 1: Set pointer using WRITE = 1 and READ = 1
    b. Step 2: Ensure the memory location contains a known value
    c. Step 3: Set WRITE = 0 and READ = 1
    d. Step 4: Observe the output bus
    e. Expected Outcome Check: Output matches stored memory value

4. TC4 – Pointer Set Operation (READ = 1, WRITE = 1)
    a. Step 1: Place address value (0–255) on the input bus
    b. Step 2: Set WRITE = 1 and READ = 1
    c. Step 3: Trigger operation
    d. Step 4: Read the pointer value using debug output or perform a read
    e. Expected Outcome Check: Pointer matches input bus value

5. TC5 – NEXT Operation (NEXT = 1)
    a. Step 1: Set the pointer to a known starting address
    b. Step 2: Activate NEXT = 1
    c. Step 3: Trigger operation
    d. Step 4: Check updated pointer value
    e. Expected Outcome Check: Pointer increases by page size (e.g., +16)

6. TC6 – BACK Operation (BACK = 1)
    a. Step 1: Set the pointer to a known address greater than the page size
    b. Step 2: Activate BACK = 1
    c. Step 3: Trigger operation
    d. Step 4: Check updated pointer value
    e. Expected Outcome Check: Pointer decreases by page size correctly

7. TC7 – RESET Operation (NEXT = 1 and BACK = 1)
    a. Step 1: Set the pointer to a non-zero value
    b. Step 2: Activate NEXT = 1 and BACK = 1 simultaneously
    c. Step 3: Trigger operation
    d. Step 4: Check pointer value
    e. Expected Outcome Check: Pointer resets to 0

8. TC8 – Bit-to-Value Translation
    a. Step 1: Set the input bus such that only GP4 = HIGH and others LOW
    b. Step 2: Trigger READ or IN operation to capture value
    c. Step 3: Observe value on OLED or debug output
    d. Expected Outcome Check: Value displayed equals 16

9. TC9 – IN Signal Behavior
    a. Step 1: Set the input bus to bits that will translate to an ASCII (e.g., ‘C’ = 67)
    b. Step 2: Set IN = 1
    c. Step 3: Trigger operation
    d. Step 4: Repeat for multiple characters
    e. Step 5: Observe string builder content
    f. Expected Outcome Check: Characters are appended in the correct sequence

10. TC10 – OUT Signal Behavior
    a. Step 1: Ensure the string builder contains a known sequence
    b. Step 2: Set OUT = 1
    c. Step 3: Trigger operation
    d. Step 4: Observe the display or output bus
    e. Expected Outcome Check: Output matches stored sequence

11. TC11 – HALT Display
    a. Step 1: Load the program containing the HALT instruction
    b. Step 2: Execute the program
    c. Step 3: Monitor execution until HALT is reached
    d. Step 4: Observe the display
    e. Expected Outcome Check: Display shows “HALT.”

12. TC12 – Program Execution (Preloaded Program)
    a. Step 1: Preload memory with program instructions and data
    b. Step 2: Start CPU execution
    c. Step 3: Allow the program to run fully
    d. Step 4: Observe output
    e. Expected Outcome Check: Output matches intended program result

13. TC13 – Post-HALT Stability
    a. Step 1: Execute the program until HALT
    b. Step 2: After HALT, continue monitoring the system
    c. Step 3: Attempt additional reads without triggering new operations
    d. Step 4: Observe memory and output behavior
    e. Expected Outcome Check: No changes occur after HALT

## Test Data

The test data used in this project consists of predefined memory values, input bus configurations, instruction codes, and program sequences. These data sets are carefully selected to verify the correct behavior of memory operations, control signals, instruction execution, and input/output functionality. Using known values ensures that system outputs can be compared against expected results for accurate validation. If our testing cases work with sample data, then real CPU program scenarios will also work.

- Predefined memory addresses and values are used to verify correct read and write operations, including initial states and modified memory locations
- A range of address values within valid memory bounds (0–255) is used to test pointer setting, navigation, and boundary conditions
- Binary input values applied through GPIO pins (GP0–GP7) are used to simulate data input and verify correct bit-to-decimal translation
- Specific bit patterns with individual pins set HIGH (e.g., single-bit activation) are used to validate the correct interpretation of control and data signals
- Known decimal values are used to confirm accurate storage, retrieval, and output behavior across memory and bus operations
- ASCII character values are used as input data to test string-building functionality and verify the correct sequencing of input operations
- Sequences of ASCII values are used to validate correct output behavior through display or output bus mechanisms
- Know instruction codes (opcodes) that are used to simulate CPU operations and verify correct instruction decoding and execution behavior, like LOAD, ADD, STORE, OR, AND, EQ, and more
- Combinations of control signals (READ, WRITE, NEXT, BACK, IN, OUT) are used to trigger specific system operations and validate signal handling
- Preloaded program data is used to simulate real execution workflows, including simple output sequences and iterative logic
- Known expected outputs are used as reference points to validate system behavior during integration testing
- Stable post-execution states are used to confirm that no unintended changes occur after program completion or HALT

## 6.3 Software Test Report (STR)

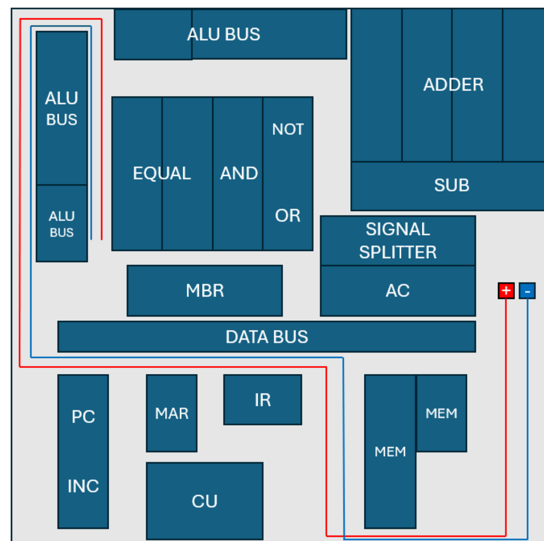
| Requirements | Pass/Fail | Severity | Notes |
|---|---|---|---|
| TC1 | Pass | None | |
| TC2 | Pass | None | |
| TC3 | Pass | None | |
| TC4 | Pass | None | |
| TC5 | Pass | None | |
| TC6 | Pass | None | |
| TC7 | Pass | None | |
| TC8 | Pass | None | |
| TC9 | Fail | Moderate | IN pin is not able to have a functioning string builder. Correct sequences cannot be executed. |
| TC10 | Pass | None | |
| TC11 | Pass | None | |
| TC12 | Fail | Critical | This is where the hardware components are not working, failing in the integrated CPU test |
| TC13 | Fail | Critical | The issue is with the software. |

# 7.0 Version Control Plan

The software written for this project is version-controlled with GitHub. Because this is a hardware project, all of the version control is done using the modular strategy.

# 8.0 Conclusion

Overall, this 8-bit CPU project shows how the major parts of a processor work together through a shared data bus, control signals, registers, memory, and the ALU. As shown in the diagram, the Control Unit directs the movement of data between the PC, MAR, IR, MBR, AC, memory, and ALU so the CPU can follow the Fetch, Decode, and Execute cycle. The design also shows the importance of modularity because each component can be built, tested, and improved separately before full integration. Although the project had challenges with moving around the CPU, synchronization, and system-level testing, it successfully demonstrated the connection between computer architecture theory and physical hardware implementation.



# Appendix A - Project Plan

## Project Overview

In this project, members will design and simulate the digital logic behind a custom 8-Bit CPU. Once the abstract digital design is finalized and deemed functional, it will be translated into electrical designs, which will be implemented using transistors.

This project is designed to provide educational benefits to members and observers on the topics of Computer Architecture and Circuit Design. It is also intended to serve as a proof-of-concept experiment for the topics.

The end product of this project is a functioning 8-Bit CPU with all basic operations either hardwired or simulated, including: Addition, Subtraction, Loops, and Memory Manipulation.

## Roles Description

**Head Documentationalist**
- Documents the meeting minutes
- Manages shared documents and reports
- Verifies completion of all documents and reports
- Designated person to submit team reports

**Head Digital Designer**
- Verifies digital logic IO design specification accuracy
- Oversees component implementation and testing

**Head Electrical Designer**
- Oversees the electrical implementation of digital components
- Creates base circuit templates
- Verifies electrical component IO

**Head Program Designer**
- Makes a website to present the required information
- Writes the memory simulation program
- Responsible for the machine code to assembly translator
- Responsible for CPU Assembly documentation

## Project Website

8-BitCPU.com

## Deliverables

1. The CPU must be of at least an 8-bit architecture.
2. The CPU must have the following basic operations within the architecture or synthesizable with operations:
    a. Add
    b. Subtract
    c. Not
    d. Or
    e. And
    f. Load
    g. Store
    h. Skip if
    i. Jump
3. The CPU must be able to run a basic program that demonstrates all functionality listed above.
4. A final report that must satisfy the above requirements with a website, GitHub link, and a video.

## Group Meeting Schedule Date/Time

Tuesdays at 10 am every week in-person, in the Kennesaw State Marietta Campus Library. Additional meetings will be held as needed.

## Collaboration and Communication Plan

As discussed in the previous section, we have a weekly meeting from 10 am-12 pm on Tuesdays in the Library. Additionally, we may choose to host various meetings throughout the semester during various other portions of the week. Next, in terms of asynchronous communication, we have set up both a texting chat and a Discord server for communications throughout the week. Lastly, in terms of resource sharing, we have elected to create a GitHub account, which shall serve as both version control and a central cloud sharing platform.

## Risk Assessment

We will be working with electrical components and soldering irons. Safety and caution will be needed when using the soldering iron. We will ensure that each team member using one is educated on their dangers and limitations. Because some team members have and will be sponsoring the purchase of the electrical components, caution will be warranted when working with these components so as not to damage them. We will be working with low voltages in this project, but caution will still be used to verify safe contact with the tools and electrical components.