University of the Philippines Cebu
College of Science
Department of Computer Science
Gorordo Avenue, Lahug, Cebu City

Design and Implementation of a Medical Triage Classification System on a Custom 8-bit CPU Using Logisim

CMSC 133: Introduction to Computer Organization, Architecture, and Assembly Language

Presented to
ENGR. MARK ANTHONY CABILO
Professor

Presented by
CAESAR ISIDRO VA-AY
202255068

30 May 2025

Abstract

This project was implemented to address the need for rapid and automated medical triage in situations where timely assessment of patient urgency is critical. Using Logisim, a custom 8-bit CPU was designed and programmed to classify patients based on binary-encoded inputs such as vital signs, symptom severity, and elapsed time since onset. The CPU executed a set of assembly-level instructions to determine whether a patient required Critical, Urgent, or Non-Urgent care using simple conditional logic and memory-mapped I/O. The system successfully demonstrated that meaningful health decision-making processes can be modeled and executed on low-resource hardware, confirming the potential of basic embedded systems to support essential functions in healthcare, particularly in constrained or mobile environments.

Table of Contents

I. Problem and Specifications

1.1 Background

Efficient and timely patient prioritization, or triage, is a crucial aspect of emergency medical care, where resources and personnel are often limited. Delays or errors in triage can lead to severe health outcomes, making it essential to identify high-risk patients as early as possible. Traditionally, triage decisions are performed manually by trained professionals using standardized guidelines. However, in many under-resourced or disaster-stricken environments, trained personnel may be unavailable, and even basic digital tools may be inaccessible. These limitations suggest a need for a compact, low-resource decision support system capable of performing basic triage functions.

Several software-based solutions and mobile apps exist to assist medical personnel in triage, but they rely on high-level computing platforms such as smartphones or tablets, which are power-hungry and often not viable in remote or emergency settings. Moreover, these systems are typically over-engineered for simple decision rules that could be handled by much simpler hardware. This gap presents an opportunity to explore the use of minimalistic embedded systems for implementing simplified medical logic, potentially improving access to life-saving tools in the field. This project aims to address this gap by developing a medical triage system that runs on a custom-designed 8-bit CPU implemented in Logisim, a digital circuit simulation tool.

1.2 Objectives

- To design and simulate a working 8-bit CPU architecture capable of executing a basic set of machine-level instructions.

- To implement medical triage logic based on binary-encoded inputs representing vital signs, symptom severity, and time since onset.

- To map inputs and outputs using a memory-mapped I/O approach to simulate real-world embedded system behavior.

- To demonstrate that the system can successfully classify patient urgency into Critical, Urgent, or Non-Urgent categories.

1.3 Specifications

The proposed system is designed with the following technical specifications:

- **CPU Architecture:** Custom 8-bit CPU with basic instruction set including arithmetic, logic, comparison, branching, and memory operations.
- **Instruction Set:** Custom ISA (Instruction Set Architecture) supporting LOAD, STORE, ADD, SUB, CMP, JMP, and conditional jump instructions (JZ, JNZ).
- **Memory Mapping:**
    - Program code stored in ROM (`0x00–0x3F`)
    - Input data stored in RAM (`0x40–0x7F`)
    - Output data written to RAM (`0x80–0xFF`)
- **I/O Handling:** Memory-mapped I/O for patient input (e.g., flags for vital signs, severity scores) and system output (triage category).
- **Triage Logic:**
    - Critical if vital signs are flagged
    - Urgent if severity exceeds threshold or prolonged symptoms

    - Non-Urgent otherwise

- **Simulation Environment:** Logisim (preferred for simulating digital logic circuits and CPUs)

The rationale behind these specifications is to ensure that the system remains simple and resource-efficient while still capable of mimicking real-world decision processes. This aligns with the objective of building deployable and robust logic suitable for embedded systems in critical care or remote medical applications.

II. Project Design

2.1 Theoretical Background

The design of this project is grounded in several key concepts from digital logic design, computer architecture, and embedded systems. The goal is to demonstrate how fundamental principles can be used to implement a medical triage system using an 8-bit CPU architecture.

1. Von Neumann Architecture

The CPU design follows the Von Neumann architecture model, where both data and instructions are stored in the same memory. This simplifies the design and emulates how traditional general-purpose processors function. The system uses a unified address and data bus to perform fetch-decode-execute cycles.

2. Instruction Cycle

The project uses a multi-step instruction cycle: Fetch, Decode, Execute, and Store (if applicable). These are synchronized using a **Step Counter**, which generates control signals at each stage.

3. Control Unit and Microprogramming

The control unit generates the control signals needed to manage data flow across the CPU. It is implemented using a combination of **control lines** and a **step counter**, simulating a microprogrammed control approach. Each instruction is broken down into smaller micro-operations distributed across steps.

4. Arithmetic Logic Unit (ALU)

The ALU is the heart of computation in the system. It is modularly constructed with:

- **Adder** and **Subtractor** for arithmetic operations.

- **Comparator** for decision-making.

- **Shift/Rotate Units** for bit manipulation.

- **Logic Engine** for bitwise operations like AND, OR, and XOR.

- **Zero Flag Unit** to determine result conditions (e.g., for jump logic).

5. Memory-Mapped I/O

Inputs (like patient vital flags) and outputs (triage decision) are managed via memory-mapped I/O. Specific RAM addresses are dedicated to simulating sensors and actuators, making the CPU interact with the simulated "environment."

6. Flags Register

A dedicated **Flags Register** stores CPU status flags (e.g., Zero, Carry). These are crucial for implementing conditional branching used in decision logic, like `Jump if Zero`.

2.2 Design Process

The design process was iterative, starting with a basic 8-bit CPU layout and gradually expanding it to support complex logic suitable for medical triage classification. Each design stage incorporated theoretical foundations and hardware logic principles to ensure reliable instruction execution and data handling.

1. Instruction Set Design

A minimal yet complete instruction set was defined to handle necessary operations for triage decision-making. The instruction categories include:

Operation Type	Instructions	Description
Memory Operations	`LIA`, `LIB`, `LDA`, `STA`	Load/Store data to/from RAM
Arithmetic	`ADD`, `SUB`	Perform addition and subtraction
Logical Operations	`AND`, `OR`, `XOR`, `NOT`	Perform bitwise logical operations
Comparison	`CMP`	Sets flags based on comparison
Control Flow	`JMP`, `JPZ`, `JNZ`, `RET`	Program flow control
Bit Manipulation	`RTL`, `RTR`	Shift bits left or right

Table 1. Summary of ISA Design

2. Truth Tables and Logic Design

The full adder circuit adds two 1-bit numbers (A and B) and an input carry (Cin) and outputs a sum (Sum) and a carry-out (Cout). This is foundational for 8-bit addition in the ALU.

A	**B**	**Cin**	**Sum**	**Cout**
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Table 2. Full Adder Logic

The comparator sets condition flags (`Zero`, `Greater`, `Less`) after subtracting register `B` from `A`. These flags control conditional branching.

A (8-bit)	B (8-bit)	A - B	Zero Flag	Greater Flag	Less Flag
0x10	0x10	0x00	1	0	0
0x15	0x10	0x05	0	1	0
0x08	0x10	-0x08	0	0	1

Table 3. Comparator Flag Logic

Each opcode activates a specific set of control signals to route data and trigger operations. This logic is vital in the control unit.

Opcode	Mnemonic	Load IR	ALU Op	Mem Read	Mem Write	Jump
01	LIA	1	0	1	0	0
30	ADD	1	1	0	0	0
70	JMP	1	0	0	0	1
20	STA	1	0	0	1	0

Table 4. Instruction Decoder Control Lines

These bitwise operations are performed directly in the ALU for logic processing, like decision-making and masking.

A	B	A AND B	A OR B	A XOR B
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Table 5. ALU Logic Gate Operations

This truth table maps control signal activation over clock cycles for one instruction execution. It’s the core of FSM sequencing.

Cycle	Control Signal	Function
1	IR ← M[PC]	Fetch instruction
2	MAR ← IR[addr]	Decode & prepare address
3	A ← M[MAR]	Load data (if load-type)
4	ALU ← A op B	Execute (if ALU-type)

5	FLAGS ← ALU result	Set status flags
6	PC ← PC+1 or jump	Branch or next instruction

Table 6. Control Unit Execution Steps

3. State Diagrams for Execution Steps

The CPU is based on a Finite State Machine (FSM) model with states representing stages of instruction execution:

State Diagram:

```
[FETCH] → [DECODE] → [EXECUTE] → [WRITEBACK / PC+1] → [FETCH]
```

Each instruction increments a **step counter** which enables specific control lines to:

- Load operands
- Trigger ALU operations
- Write results back
- Control the PC for branching

4. Register and Memory Layout

Register	Function
A, B	General-purpose registers
MAR2/3	Used for indirect addressing
IR	Instruction Register

PC	Program Counter
SP	Stack Pointer
FLAGS	Zero, Carry, Borrow, etc.

Table 7. Register

Memory Map:

Address Range	**Usage**
0x0000–0x003F	ROM (Program Code)
0x0040–0x007F	RAM (Working Registers)
0x0080–0x00FF	Stack / Extended RAM
0xFFF8–0xFFFF	Pixel Display Framebuf

Table 8. Memory Map

The figures below are the screenshots for the circuit:

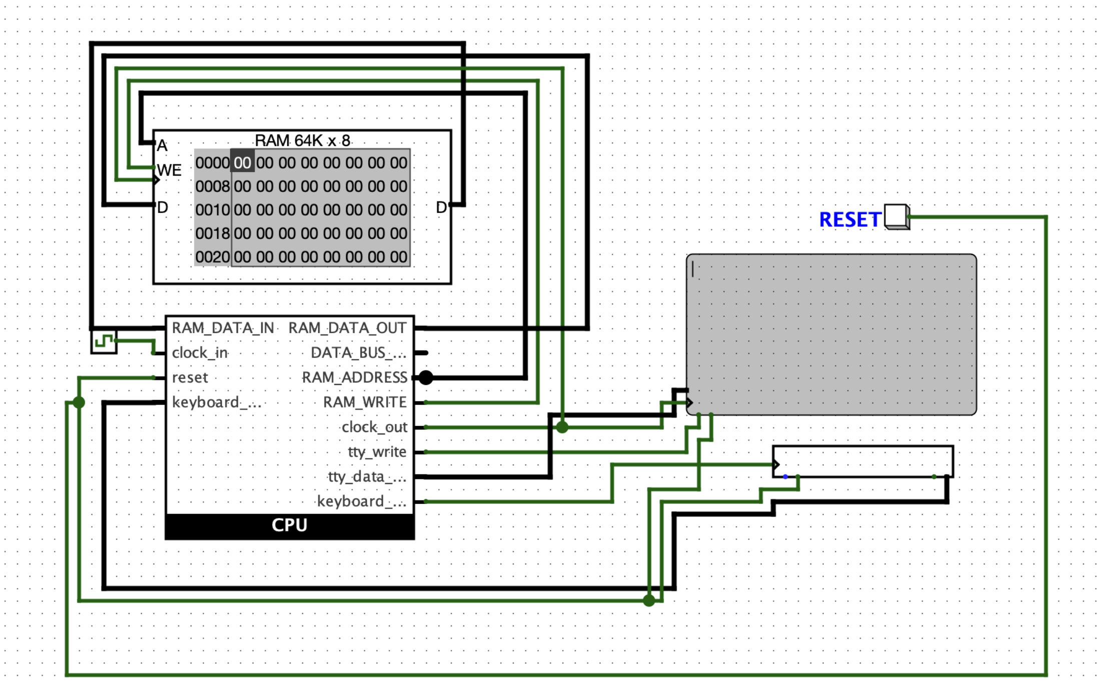


Figure 1. Main Circuit with CPU, RAM, TTY and Keyboard

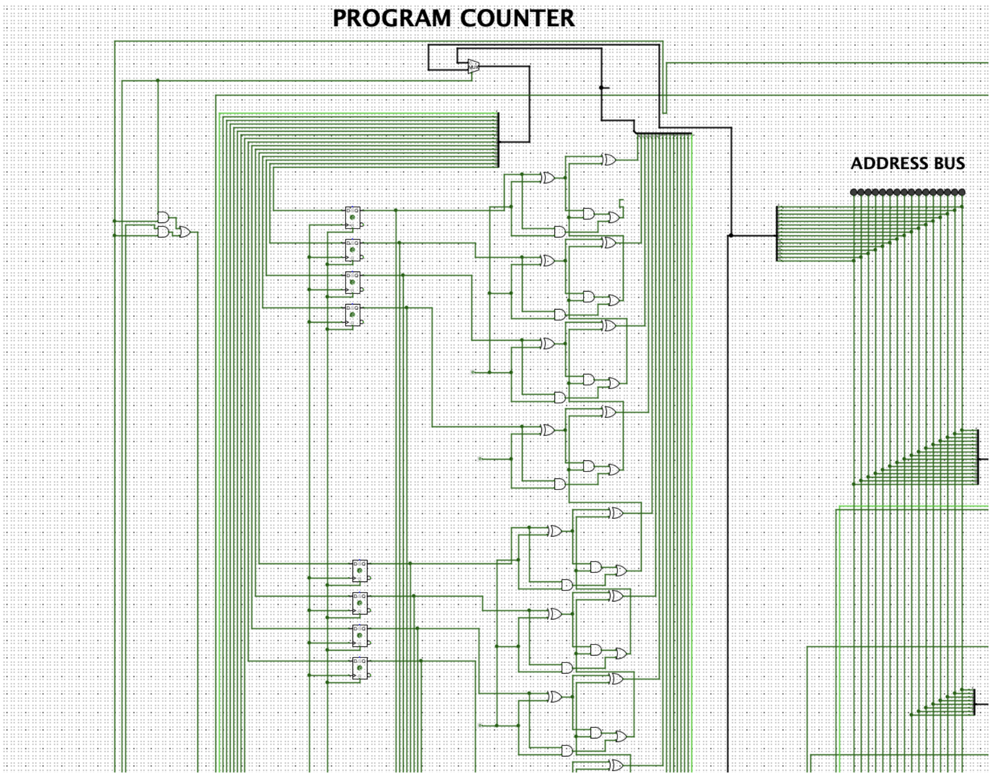


Figure 2. Program Counter and Address Bus

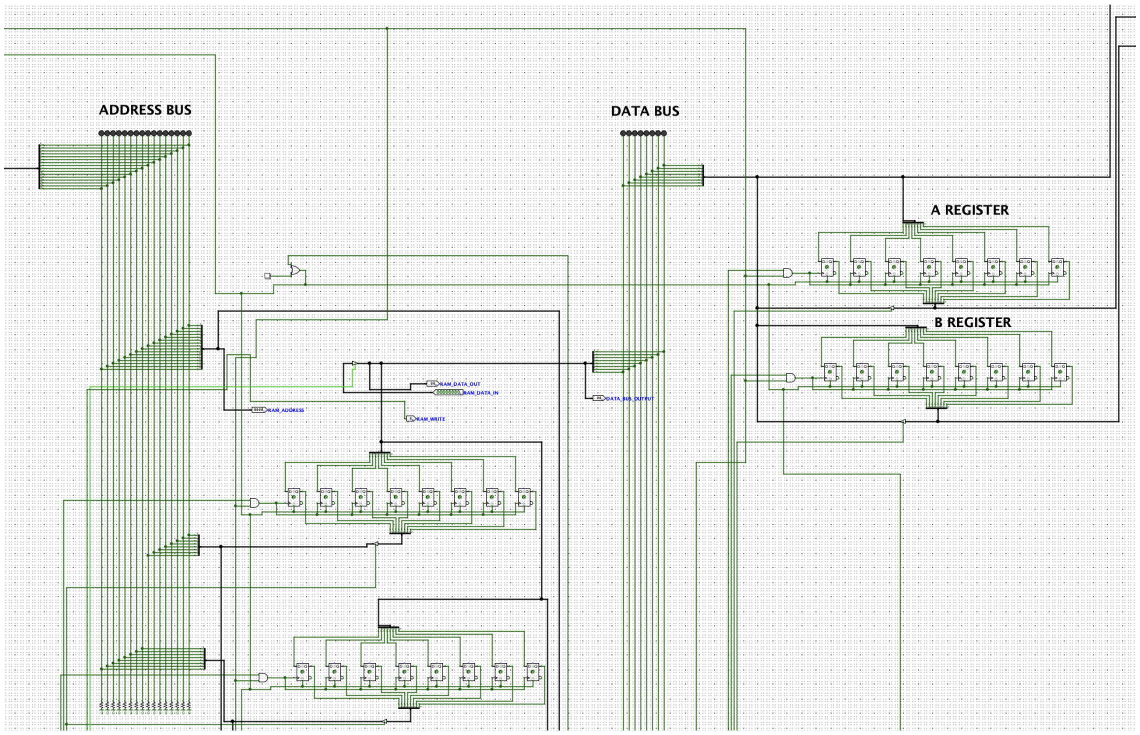


Figure 4. Data Bus and Registers

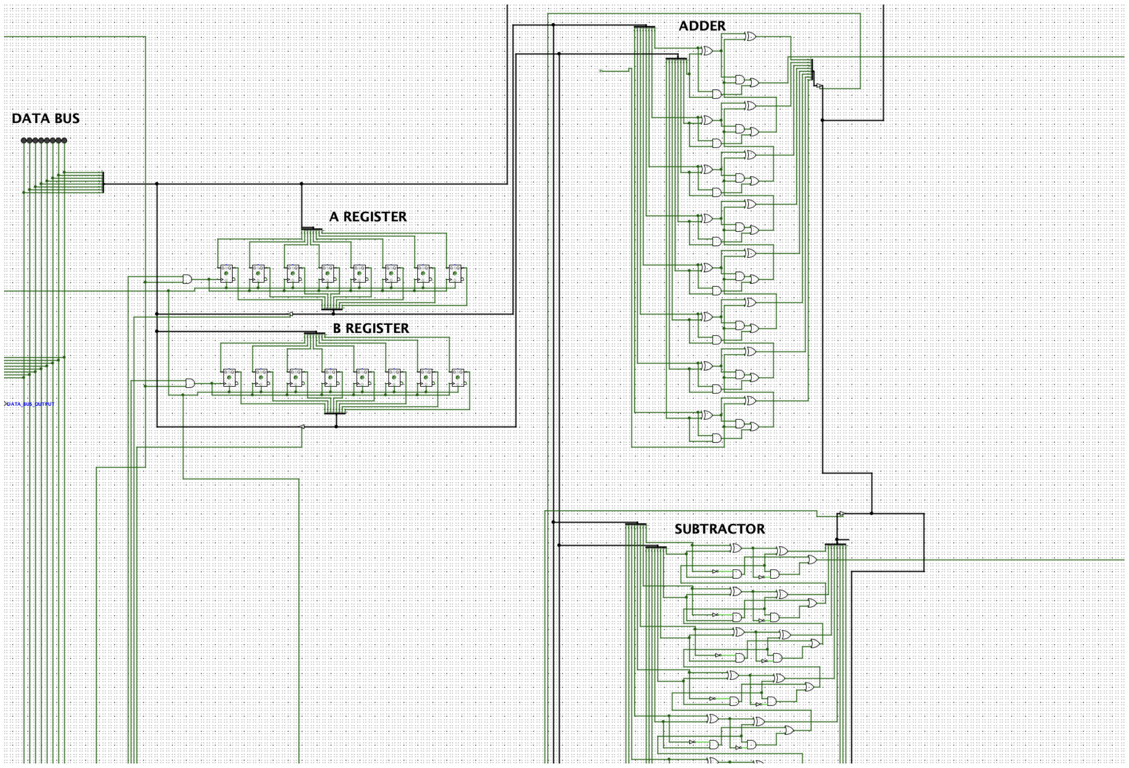


Figure 5. Adder and Subtractor

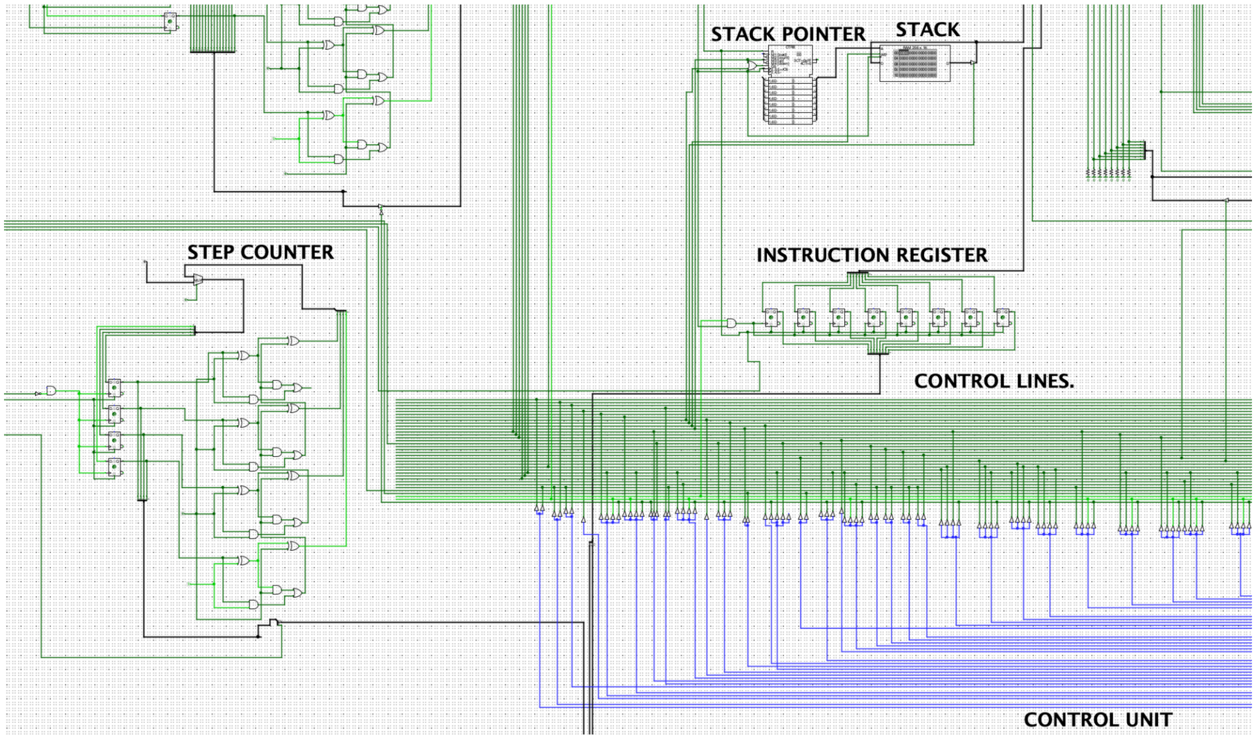


Figure 6. Stack, Instruction Register, Control Lines, Control Unit and Step Counter

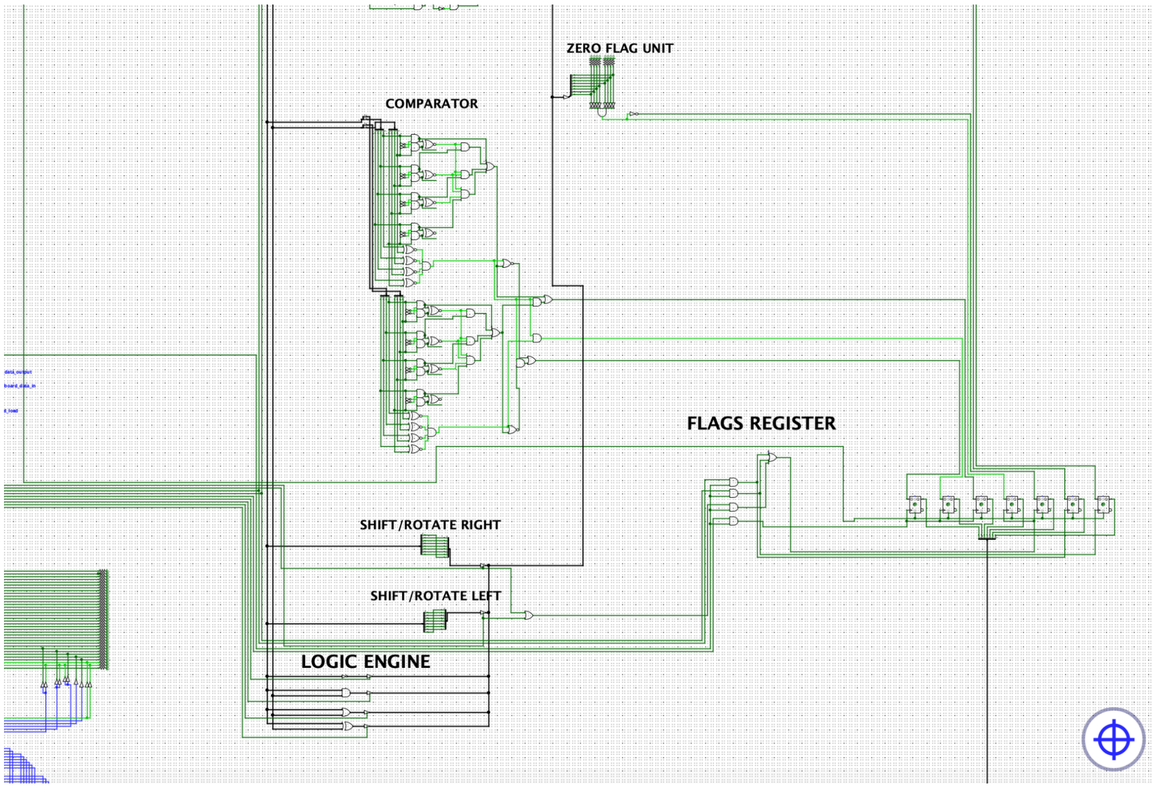


Figure 7. Comparator, Flags Register and Logic Engine

5. Triage Algorithm Implementation

The triage system follows a structured set of yes/no questions to determine the urgency of a patient's condition. Patients are categorized into four levels: **RED (Emergency)**, **ORANGE (Priority)**, **GREEN (Nonurgent)**, or **UNCATEGORIZED**.

RED Category – Emergency

If the patient answers **"Yes"** to **any** of the following:

- Severe chest pain or difficulty breathing
- Sudden weakness/numbness on one side
- Severe bleeding or major trauma
- Loss of consciousness or undiagnosed seizure
- Sudden severe headache (worst ever)
- Signs of anaphylaxis (swelling, rash)
- Suspected poisoning or overdose
- High fever with stiff neck or light sensitivity
  → **Then:** Patient is tagged as **RED**
  → **Action:** Seek **immediate emergency care**

ORANGE Category – Priority

If all RED conditions are **"No"**, proceed here.
If patient answers **"Yes"** to any of the following:

- Moderate to severe worsening pain
- Persistent vomiting or dehydration
- Fever with major discomfort not relieved at home
- New or worsening abdominal pain
- Persistent cough with mild shortness of breath

- Possible broken bone

- Signs of infection with fever (swelling/redness)
  → **Then:** Patient is tagged as **ORANGE**
  → **Action:** Seek **urgent care soon**

GREEN Category – Nonurgent

If all RED and ORANGE conditions are **"No"**, proceed here.
If patient answers **"Yes"** to any of the following:

- Common cold or mild flu symptoms

- Minor cuts or scrapes

- Mild aches or pains improving with rest

- Routine prescription refill

- Final fallback question (general yes)
  → **Then:** Patient is tagged as **GREEN**
  → **Action:** Rest and monitor, contact provider if needed

UNCATEGORIZED

If patient answers **"No"** to all known symptoms
→ **Then:** Patient is tagged as **UNCATEGORIZED**
→ **Action:** Advised to seek medical attention or call a nurse line if unsure

3. **Final Prompt**

    - Display condition result.

    - Wait for user input to reset the triage dialogue for the next patient.

III. Implementation and Simulation

3.1 Implementation

The 8-bit CPU was implemented in **Logisim** using a modular structure that includes a Program Counter (PC), Instruction Register (IR), Arithmetic Logic Unit (ALU), general-purpose registers (A and B), temporary registers (MAR2, MAR3), Stack Pointer (SP), flags, and a control unit. The CPU reads binary instructions from memory and executes them using a custom instruction set designed to support various logical, arithmetic, memory, and I/O operations, optimized for implementing triage decision-making.

Instruction Set Architecture (ISA)

The processor's instruction set consists of 1- to 3-byte instructions with fixed opcodes. Below is a summary of key instructions used:

1. No Operation

- `00`: **NOP** — No operation.

2. Load Immediate

- `01 XX`: **LIA** — Load immediate value XX into A register.
- `02 XX`: **LIB** — Load immediate value XX into B register.
- `03 XX`: **LIT** — Load immediate value to TTY display.

3. Load from RAM

- `10 XX XX`: **LDA** — Load from RAM address `XXXX` into A register.
- `11 XX XX`: **LDB** — Load from RAM into B register.
- `12 XX XX`: **LDT** — Load from RAM into TTY display.
- `13 XX XX`: **LD0** — Load from RAM into MAR2 (low byte).
- `14 XX XX`: **LD1** — Load from RAM into MAR3 (high byte).
- `15`: **LDI** — Load to A register from address formed by MAR2 and MAR3.

4. Store to RAM

- `20 XX XX`: **STA** — Store A register value to RAM.
- `21 XX XX`: **STB** — Store B register value to RAM.
- `22 XX XX`: **STK** — Store keyboard buffer to RAM and load next.
- `23`: **STI** — Store A to address from MAR2 and MAR3.

5. ALU Operations

- `30 XX XX`: **ADD** — Add A and B; store in A.
- `31 XX XX`: **SUB** — Subtract B from A.
- `32`: **RTR** — Rotate A register right.
- `33`: **RTL** — Rotate A register left.
- `34 XX XX`: **CMP** — Compare A and B; set flags.
- `35`: **NOT** — Bitwise NOT on A.
- `36 XX XX`: **AND** — AND A with RAM.
- `37 XX XX`: **OR** — OR A with RAM.
- `38 XX XX`: **XOR** — XOR A with RAM.

6. Pixel Display

- `52`: **UPD** — Update display from framebuffer (RAM FFF8–FFFF).

7. Jump/Branch/Call/Return

- `70 XX XX`: **JMP** — Jump to address.
- `71 XX XX`: **JPC** — Jump if carry flag set.
- `72 XX XX`: **JPB** — Jump if borrow flag set.
- `73 XX XX`: **JNZ** — Jump if not zero.

- `74 XX XX`: **JPZ** — Jump if zero.
- `75 XX XX`: **JPG** — Jump if greater-than.
- `76 XX XX`: **JPE** — Jump if equal.
- `77 XX XX`: **JPL** — Jump if less-than.
- `78 XX XX`: **JCAL** — Call subroutine (push PC).
- `79`: **RET** — Return (pop from stack to PC).

Addressing Modes

The CPU supports:

- **Immediate Addressing** — Used in instructions like `LIA XX`, where the operand is a literal.
- **Direct Addressing** — Used in most memory instructions like `LDA XX XX`.
- **Indirect Addressing via MAR2 and MAR3** — Used in `LDI` and `STI`, which form a 16-bit address from two 8-bit registers.

3.2 Simulation Results

Follow these steps to replicate the simulation:

1. **Run the Assembler**
   Open and run the `assembler.py` file using Python. This script converts your `.asm` program into machine code.

2. **Load the Assembly File**
   In the assembler UI, **load your `.asm` file** (the triage program file).

3. **Copy the Hex Code**
   After assembling, **copy the hex output** generated by the script.

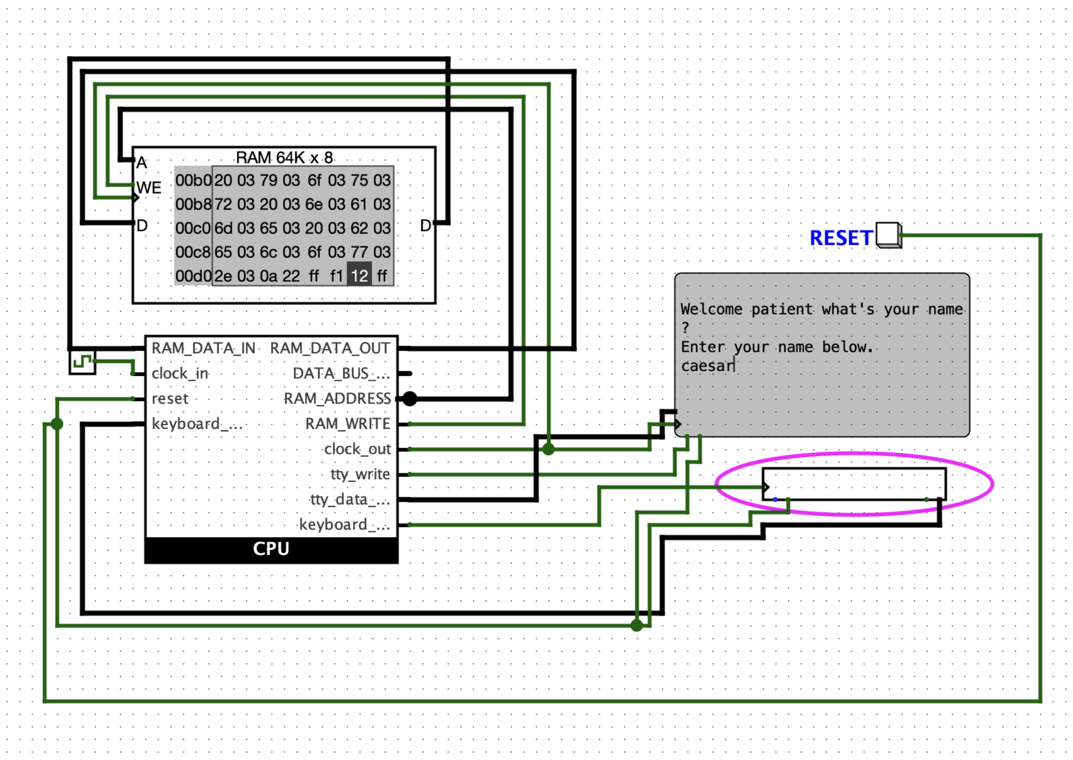
4. **Paste into RAM**
   Open your LogiSim circuit. Double-click the RAM module and **paste the hex code** into

memory.

5. **Enable Auto Tick**
   In LogiSim, enable the **Auto Tick** (simulate > tick frequency > enable) to run the CPU.

6. **Input Using the Keyboard**
   Use the on-screen **keyboard component** (highlighted in pink in the image) to enter patient responses.

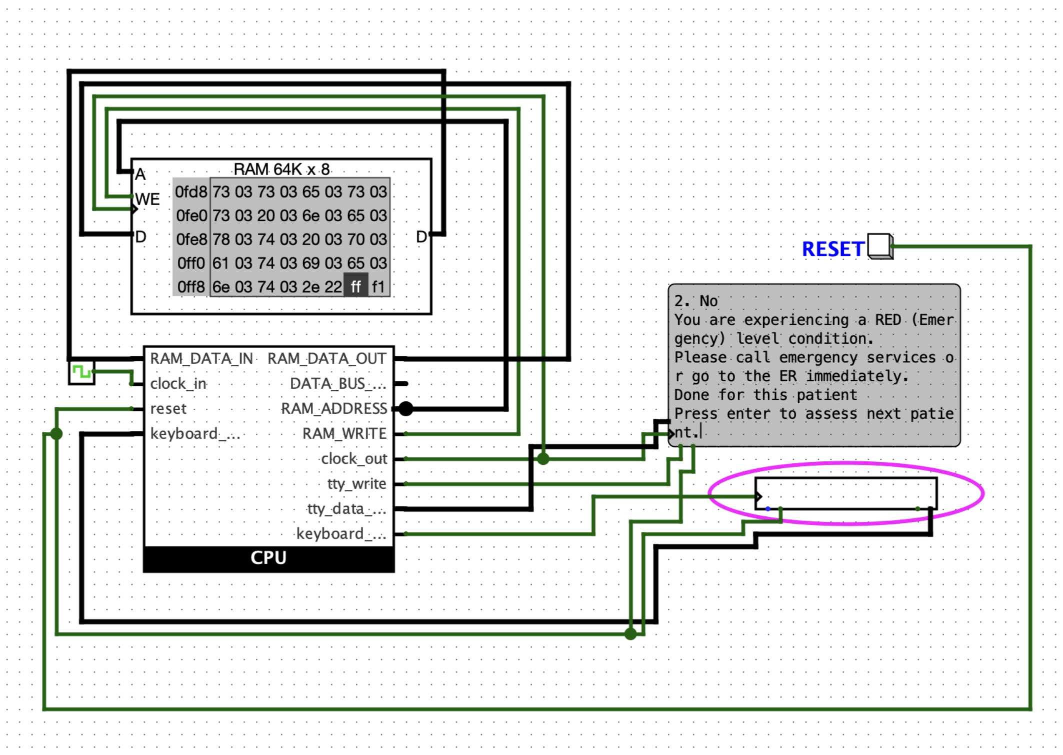
Below are the results of the simulation:



```
00b0 20 03 79 03 6f 03 75 03
00b8 72 03 20 03 6e 03 61 03
00c0 6d 03 65 03 20 03 62 03
00c8 65 03 6c 03 6f 03 77 03
00d0 2e 03 0a 22 ff f1 12 ff
```

```
Welcome patient what's your name
?
Enter your name below.
caesar
```

Figure 8. Simulation Result - Enter Patient Name



```
0fd8 73 03 73 03 65 03 73 03
0fe0 73 03 20 03 6e 03 65 03
0fe8 78 03 74 03 20 03 70 03
0ff0 61 03 74 03 69 03 65 03
0ff8 6e 03 74 03 2e 22 ff f1
```

```
2. No
You are experiencing a RED (Emer
gency) level condition.
Please call emergency services o
r go to the ER immediately.
Done for this patient
Press enter to assess next patie
nt.
```

Figure 9. Simulation Result - Triage Output

3.3 Analysis

The simulation results confirm that the triage system functions as intended, accurately processing user input and delivering appropriate medical response messages. Upon running the system, the CPU successfully reads the input provided through the on-screen keyboard and executes a preloaded program stored in RAM. This program interprets the responses and prints out triage instructions on the TTY display. For instance, when the user inputs "no" in response to a symptom question, the system correctly categorizes the condition as an emergency and displays the corresponding alert message. This demonstrates that the data flow between the CPU, RAM, and peripherals (keyboard and TTY) is functioning reliably.

The CPU's internal components including the program counter, instruction decoder, and memory control operate correctly, enabling sequential and conditional execution of instructions. The RAM module responds accurately to read and write requests, allowing the CPU to fetch instructions and display strings without errors. The proper response from the TTY indicates that the ASCII character data was transmitted and rendered correctly. These outcomes validate the logical integration and timing of the system components.

However, the simulation also reveals certain limitations. The input handling is very rigid, allowing only expected responses like "1" or "2." Furthermore, the interface is basic, limited to text output without interactivity beyond typing. Another practical limitation is the manual process of copying the assembled machine code into RAM before simulation can begin, which adds time and increases the risk of input errors.

Despite these constraints, the system provides a solid foundation for further development. Future improvements could include enhancing input validation, expanding the logic to handle more complex decision trees, and automating the loading of machine code into memory. These enhancements would improve usability, make the simulation more robust, and better mimic the behavior of real-world embedded healthcare devices.

IV. Conclusion and Recommendations

4.1 Project Thesis

This project aimed to implement a functioning 8-bit CPU within Logisim to simulate a basic medical triage system. The central problem addressed was the lack of a simple, customizable, and educationally accessible CPU design that could execute context-specific logic such as triage decision-making. In response to this, the CPU was designed to handle binary instructions that process patient data, evaluate conditions through arithmetic and logical operations, and assign medical priority levels accordingly.

Throughout the project, the integration of fundamental CPU components including the program counter, ALU, registers, stack pointer, and control unit enabled a functional execution pipeline. The processor successfully carried out basic instructions, responded to comparison conditions, and stored flags for branching decisions. This allowed the simulation of triage scenarios, such as determining whether a patient should be treated immediately, monitored, or deferred. The implementation not only met the stated objectives but also validated the use of a minimalist 8-bit system for real-world-inspired tasks, proving that simple processor designs can support domain-specific logic with sufficient efficiency and reliability.

4.2 Recommendations

While the project met its design and implementation goals, several areas were identified that lie beyond the scope of the current system but are worth exploring for future enhancements. One such area is interrupt handling. The current CPU lacks the ability to respond to asynchronous events, which limits its responsiveness in real-time systems. Incorporating hardware or software interrupts would significantly improve flexibility and enable dynamic response to emergency cases in the triage simulation.

Another limitation is the absence of real-time input/output integration. Presently, data is entered manually or simulated internally, restricting the system's interactivity. Future work could focus on designing external I/O controllers or interfaces to receive real patient data inputs or to output decisions in real-time, which would greatly enhance the realism and usability of the system. Additionally, the instruction set used is minimal and only supports basic operations. Expanding the instruction set to include functions like multiplication, division, and more complex conditional logic would increase the system's capacity to process nuanced medical criteria.

Moreover, the current flat memory model restricts the scale and modularity of the program. Implementing memory paging or segment-based addressing could allow for larger and more structured applications. Power and speed optimization were also not a focus of this simulation, but for real-world embedded systems, reducing power consumption and optimizing instruction cycles are critical, especially in portable or remote applications. Another valuable improvement would be to include a graphical output module or display interface that visualizes the triage decision, providing intuitive feedback to users, especially in training environments.

Lastly, the scalability of the design to 16-bit architecture or multicore configurations is another avenue worth considering. Although this would increase complexity, it would enable more sophisticated logic processing and multitasking, essential for handling concurrent medical evaluations. These recommendations point toward a promising roadmap for extending this 8-bit CPU into a more robust, real-time medical decision-support tool.

Appendix

A. Assembly Code

```
; START OF PROGRAM
START:

; VARIABLES
A = 0
B = 1
D = '1'
E = '2'
F = 100
G = '0'
H = '`'
X = 10
Z = 96

TEMP = 0
TEMP_1 = 0
TEMP_2 = 0
TEMP_3 = 0
TEMP_4 = 0

KEYBOARD = 0

; Player variables
PLAYER_NAME_PRINT_LOW = 0
PLAYER_NAME_PRINT_HIGH = 100

PLAYER_NAME_END_LOW = 0
PLAYER_NAME_END_HIGH = 100

; MAIN
MAIN:
LIT 0A
PRINT "Welcome patient what's your name?"
LIT 0A
PRINT "Enter your name below."
LIT 0A
```

```
; PLAYER NAME INPUT LOOP
NAME_INPUT_LOOP:
STK KEYBOARD
LDT KEYBOARD
LDA KEYBOARD
CMP A
JPE NAME_INPUT_LOOP
CMP Z
JPE MAIN
CMP X
JPE GAME_LOOP
LD0 PLAYER_NAME_END_LOW
LD1 PLAYER_NAME_END_HIGH
STI
LDA PLAYER_NAME_END_LOW
ADD B
STA PLAYER_NAME_END_LOW
LIA 00
STA KEYBOARD
JMP NAME_INPUT_LOOP

; MAIN GAME LOOP
GAME_LOOP:
LDA PLAYER_NAME_END_LOW
SUB B
STA PLAYER_NAME_END_LOW

START_GAME:
LIT 0A
PRINT "Welcome Patient"

CONTINUE_GAME:
LIT 0A
PRINT "Medical Triage Station"
LIT 0A
PRINT "Let's determine how urgent your symptoms are."

; START TRIAGE TREE
QUESTION_C1:
LIT 0A
PRINT "Are you experiencing any of the following?"
LIT 0A
PRINT "1. Severe chest pain or difficulty breathing"
LIT 0A
```

```
PRINT "2. No"

QUESTION_C1_SELECTION_LOOP:
STK KEYBOARD
LDA KEYBOARD
CMP A
JPE QUESTION_C1_SELECTION_LOOP
CMP D
JPE RED_TAG
CMP E
JPE QUESTION_C2
JMP QUESTION_C1_SELECTION_LOOP

QUESTION_C2:
LIT 0A
PRINT "Sudden weakness or numbness on one side?"
LIT 0A
PRINT "1. Yes"
LIT 0A
PRINT "2. No"

QUESTION_C2_SELECTION_LOOP:
STK KEYBOARD
LDA KEYBOARD
CMP A
JPE QUESTION_C2_SELECTION_LOOP
CMP D
JPE RED_TAG
CMP E
JPE QUESTION_C3
JMP QUESTION_C2_SELECTION_LOOP

QUESTION_C3:
LIT 0A
PRINT "Severe bleeding or major trauma?"
LIT 0A
PRINT "1. Yes"
LIT 0A
PRINT "2. No"

QUESTION_C3_SELECTION_LOOP:
STK KEYBOARD
LDA KEYBOARD
CMP A
```

```
JPE QUESTION_C3_SELECTION_LOOP
CMP D
JPE RED_TAG
CMP E
JPE QUESTION_C4
JMP QUESTION_C3_SELECTION_LOOP

QUESTION_C4:
LIT 0A
PRINT "Loss of consciousness or seizure (not diagnosed)?"
LIT 0A
PRINT "1. Yes"
LIT 0A
PRINT "2. No"

QUESTION_C4_SELECTION_LOOP:
STK KEYBOARD
LDA KEYBOARD
CMP A
JPE QUESTION_C4_SELECTION_LOOP
CMP D
JPE RED_TAG
CMP E
JPE QUESTION_C5
JMP QUESTION_C4_SELECTION_LOOP

QUESTION_C5:
LIT 0A
PRINT "Sudden severe headache (worst ever)?"
LIT 0A
PRINT "1. Yes"
LIT 0A
PRINT "2. No"

QUESTION_C5_SELECTION_LOOP:
STK KEYBOARD
LDA KEYBOARD
CMP A
JPE QUESTION_C5_SELECTION_LOOP
CMP D
JPE RED_TAG
CMP E
JPE QUESTION_C6
JMP QUESTION_C5_SELECTION_LOOP
```

```
QUESTION_C6:
LIT 0A
PRINT "Signs of anaphylaxis (swelling, rash, etc)?"
LIT 0A
PRINT "1. Yes"
LIT 0A
PRINT "2. No"

QUESTION_C6_SELECTION_LOOP:
STK KEYBOARD
LDA KEYBOARD
CMP A
JPE QUESTION_C6_SELECTION_LOOP
CMP D
JPE RED_TAG
CMP E
JPE QUESTION_C7
JMP QUESTION_C6_SELECTION_LOOP

QUESTION_C7:
LIT 0A
PRINT "Suspected poisoning or overdose?"
LIT 0A
PRINT "1. Yes"
LIT 0A
PRINT "2. No"

QUESTION_C7_SELECTION_LOOP:
STK KEYBOARD
LDA KEYBOARD
CMP A
JPE QUESTION_C7_SELECTION_LOOP
CMP D
JPE RED_TAG
CMP E
JPE QUESTION_C8
JMP QUESTION_C7_SELECTION_LOOP

QUESTION_C8:
LIT 0A
PRINT "High fever with stiff neck or light sensitivity?"
LIT 0A
PRINT "1. Yes"
```

```
LIT 0A
PRINT "2. No"

QUESTION_C8_SELECTION_LOOP:
STK KEYBOARD
LDA KEYBOARD
CMP A
JPE QUESTION_C8_SELECTION_LOOP
CMP D
JPE RED_TAG
CMP E
JPE QUESTION_D1
JMP QUESTION_C8_SELECTION_LOOP

; ORANGE SYMPTOMS
QUESTION_D1:
LIT 0A
PRINT "Moderate to severe worsening pain?"
LIT 0A
PRINT "1. Yes"
LIT 0A
PRINT "2. No"

QUESTION_D1_SELECTION_LOOP:
STK KEYBOARD
LDA KEYBOARD
CMP A
JPE QUESTION_D1_SELECTION_LOOP
CMP D
JPE ORANGE_TAG
CMP E
JPE QUESTION_D2
JMP QUESTION_D1_SELECTION_LOOP

QUESTION_D2:
LIT 0A
PRINT "Persistent vomiting or dehydration?"
LIT 0A
PRINT "1. Yes"
LIT 0A
PRINT "2. No"

QUESTION_D2_SELECTION_LOOP:
STK KEYBOARD
```

```
LDA KEYBOARD
CMP A
JPE QUESTION_D2_SELECTION_LOOP
CMP D
JPE ORANGE_TAG
CMP E
JPE QUESTION_D3
JMP QUESTION_D2_SELECTION_LOOP

QUESTION_D3:
LIT 0A
PRINT "Fever with major discomfort not relieved at home?"
LIT 0A
PRINT "1. Yes"
LIT 0A
PRINT "2. No"

QUESTION_D3_SELECTION_LOOP:
STK KEYBOARD
LDA KEYBOARD
CMP A
JPE QUESTION_D3_SELECTION_LOOP
CMP D
JPE ORANGE_TAG
CMP E
JPE QUESTION_D4
JMP QUESTION_D3_SELECTION_LOOP

QUESTION_D4:
LIT 0A
PRINT "New or worsening abdominal pain?"
LIT 0A
PRINT "1. Yes"
LIT 0A
PRINT "2. No"

QUESTION_D4_SELECTION_LOOP:
STK KEYBOARD
LDA KEYBOARD
CMP A
JPE QUESTION_D4_SELECTION_LOOP
CMP D
JPE ORANGE_TAG
CMP E
```

```
JPE QUESTION_D5
JMP QUESTION_D4_SELECTION_LOOP

QUESTION_D5:
LIT 0A
PRINT "Persistent cough with mild shortness of breath?"
LIT 0A
PRINT "1. Yes"
LIT 0A
PRINT "2. No"

QUESTION_D5_SELECTION_LOOP:
STK KEYBOARD
LDA KEYBOARD
CMP A
JPE QUESTION_D5_SELECTION_LOOP
CMP D
JPE ORANGE_TAG
CMP E
JPE QUESTION_D6
JMP QUESTION_D5_SELECTION_LOOP

QUESTION_D6:
LIT 0A
PRINT "Possible broken bone?"
LIT 0A
PRINT "1. Yes"
LIT 0A
PRINT "2. No"

QUESTION_D6_SELECTION_LOOP:
STK KEYBOARD
LDA KEYBOARD
CMP A
JPE QUESTION_D6_SELECTION_LOOP
CMP D
JPE ORANGE_TAG
CMP E
JPE QUESTION_D7
JMP QUESTION_D6_SELECTION_LOOP

QUESTION_D7:
LIT 0A
PRINT "Signs of infection with fever (swelling/redness)?"
```

```
LIT 0A
PRINT "1. Yes"
LIT 0A
PRINT "2. No"

QUESTION_D7_SELECTION_LOOP:
STK KEYBOARD
LDA KEYBOARD
CMP A
JPE QUESTION_D7_SELECTION_LOOP
CMP D
JPE ORANGE_TAG
CMP E
JPE QUESTION_E1
JMP QUESTION_D7_SELECTION_LOOP

; GREEN SYMPTOMS
QUESTION_E1:
LIT 0A
PRINT "Common cold or mild flu symptoms?"
LIT 0A
PRINT "1. Yes"
LIT 0A
PRINT "2. No"

QUESTION_E1_SELECTION_LOOP:
STK KEYBOARD
LDA KEYBOARD
CMP A
JPE QUESTION_E1_SELECTION_LOOP
CMP D
JPE GREEN_TAG
CMP E
JPE QUESTION_E2
JMP QUESTION_E1_SELECTION_LOOP

QUESTION_E2:
LIT 0A
PRINT "Minor cuts or scrapes?"
LIT 0A
PRINT "1. Yes"
LIT 0A
PRINT "2. No"
```

```
QUESTION_E2_SELECTION_LOOP:
STK KEYBOARD
LDA KEYBOARD
CMP A
JPE QUESTION_E2_SELECTION_LOOP
CMP D
JPE GREEN_TAG
CMP E
JPE QUESTION_E3
JMP QUESTION_E2_SELECTION_LOOP

QUESTION_E3:
LIT 0A
PRINT "Mild aches or pains improving with rest?"
LIT 0A
PRINT "1. Yes"
LIT 0A
PRINT "2. No"

QUESTION_E3_SELECTION_LOOP:
STK KEYBOARD
LDA KEYBOARD
CMP A
JPE QUESTION_E3_SELECTION_LOOP
CMP D
JPE GREEN_TAG
CMP E
JPE QUESTION_E4
JMP QUESTION_E3_SELECTION_LOOP

QUESTION_E4:
LIT 0A
PRINT "Routine prescription refill?"
LIT 0A
PRINT "1. Yes"
LIT 0A
PRINT "2. No"

QUESTION_E4_SELECTION_LOOP:
STK KEYBOARD
LDA KEYBOARD
CMP A
JPE QUESTION_E4_SELECTION_LOOP
CMP D
```

```
JPE GREEN_TAG
CMP E
JPE QUESTION_E5
JMP QUESTION_E4_SELECTION_LOOP

QUESTION_E5:
LIT 0A
PRINT "1. Yes"
LIT 0A
PRINT "2. No"

QUESTION_E5_SELECTION_LOOP:
STK KEYBOARD
LDA KEYBOARD
CMP A
JPE QUESTION_E5_SELECTION_LOOP
CMP D
JPE GREEN_TAG
CMP E
JPE UNCATEGORIZED
JMP QUESTION_E5_SELECTION_LOOP

; RESULTS
RED_TAG:
LIT 0A
PRINT "You are experiencing a RED (Emergency) level condition."
LIT 0A
PRINT "Please call emergency services or go to the ER immediately."
JMP GAME_OVER

ORANGE_TAG:
LIT 0A
PRINT "You are experiencing an ORANGE (Priority) condition."
LIT 0A
PRINT "Please seek urgent medical attention as soon as possible."
JMP GAME_OVER

GREEN_TAG:
LIT 0A
PRINT "You are experiencing a GREEN (NONURGENT) condition."
LIT 0A
PRINT "Please rest and monitor your symptoms. Follow up with a provider if needed."
JMP GAME_OVER
```

```
UNCATEGORIZED:
LIT 0A
PRINT "We couldn't categorize your symptoms."
LIT 0A
PRINT "If you're unsure, please seek medical attention or call a nurse line."
JMP GAME_OVER

; GAME OVER
GAME_OVER:
LIT 0A
PRINT "Done for this patient"
LIT 0A
PRINT "Press enter to assess next patient."

GAME_OVER_SELECTION_LOOP:
STK KEYBOARD
LDA KEYBOARD
CMP A
JPE GAME_OVER_SELECTION_LOOP
CMP X
JPE START
JMP GAME_OVER_SELECTION_LOOP

CLEAR_VARIABLES:
LIA 00
STA TEMP_4
STA TEMP_3
STA TEMP_2
STA TEMP_1
RET

; Split number into digits for printing
SPLIT_NUMBER:
MINUS_100:
LDA TEMP_4
CMP F
JPL MINUS_10
SUB F
STA TEMP_4
LDA TEMP_3
ADD B
STA TEMP_3
CMP TEMP_4
JPL MINUS_100
```

```
JPE MINUS_100

MINUS_10:
LDA TEMP_4
CMP X
JPL RETURN
SUB X
STA TEMP_4
LDA TEMP_2
ADD B
STA TEMP_2
CMP TEMP_4
JPL MINUS_10
JPE MINUS_10
RET

PRINT_SPLIT_NUMBER:
LDA TEMP_4
ADD G
STA TEMP_1
LDA TEMP_2
ADD G
STA TEMP_2
LDA TEMP_3
ADD G
STA TEMP_3
LDT TEMP_3
LDT TEMP_2
LDT TEMP_1
JCAL CLEAR_VARIABLES
RET

RETURN:
RET
```

B. Assembler Code

```
import tkinter as tk
from tkinter import filedialog, messagebox


# Assembler dictionary for opcodes and their values
opcodes = {
```

```
           "NOP": "00", "LIA": "01", "LIB": "02", "LIT": "03", "LDA": "10", "LDB":
"11", "LDT": "12", "LD0": "13", "LD1": "14", "LDI": "15",
           "STA": "20", "STB": "21", "STK": "22", "STI": "23",
           "ADD": "30", "SUB": "31", "RTR": "32", "RTL": "33", "CMP": "34",  "NOT":
"35" , "AND": "36" , "OR": "37", "XOR": "38" , "RVA": "50", "RVD": "51", "UPD": "52",
"JMP": "70",
           "JPC": "71", "JPB": "72", "JNZ": "73", "JPZ": "74", "JPG": "75",
           "JPE": "76", "JPL": "77", "JCAL": "78", "RET": "79","JCL": "7A", "JCE":
"7B", "JCG": "7C" , "JPI": "7D",      }



# Helper function to convert a string into LIT instructions
def convert_string_to_lit(string):
   lit_instructions = []
   for char in string:
       ascii_value = f"{ord(char):02X}"  # Convert char to 2-digit hex ASCII
       lit_instructions.append('03')      # LIT opcode for loading immediate into TTY
       lit_instructions.append(ascii_value)
   return lit_instructions
# Dictionary to store variables and their memory addresses
variables = {}
next_available_address = 0xFFFF  # Start variable storage at address 0xFFFF

# Helper function to allocate memory for a variable
def allocate_variable(name, value):

   global next_available_address
   if next_available_address < 0x0000:
       raise Exception("Out of memory for variables!")

   # Check if value is a character in quotes
   if value.startswith("'") and value.endswith("'") and len(value) == 3:
       ascii_value = ord(value[1])  # Extract ASCII value of the character
   else:
       ascii_value = int(value)  # Treat it as a base-10 integer

   # Store the variable in the variables dictionary with its memory address
   variables[name] = f"{next_available_address:04X}"
   machine_code = ["01", f"{ascii_value:02X}", "20",
f"{next_available_address:04X}"[:2], f"{next_available_address:04X}"[2:]]  # LIA
value, STA address
```

```python
    next_available_address -= 1  # Decrease available memory address
    return machine_code


def process_operand(operand):
    # Check if operand is a character (e.g., 'A')
    if operand.startswith("'") and operand.endswith("'") and len(operand) == 3:
        return f"{ord(operand[1]):02X}"  # Convert char to its ASCII hex value
    else:
        # Assume the operand is a hex value and return it as-is
        return operand


# Updated Assembler function to handle LIA and LIT with ASCII values or variables
def assemble_code(assembly_code):
    global variables, next_available_address
    machine_code = []
    labels = {}  # Store label positions
    unresolved_jumps = []  # Track unresolved jump addresses
    lines = assembly_code.strip().split("\n")
    current_address = 0  # Track the current address for labels

    # First pass: Identify labels, variables, and store their addresses
    for i, line in enumerate(lines):
        line = line.strip()
        if not line or line.startswith(";"):  # Skip empty lines and comments
            continue

        # Handle variable assignment (e.g., x = 3)
        if '=' in line:
            parts = line.split('=')
            var_name = parts[0].strip()
            var_value = parts[1].strip()
            # Allocate memory for the variable and add LIA/STA instructions to machine
code
            machine_code.extend(allocate_variable(var_name, var_value))
            current_address += 5  # LIA and STA both take 5 bytes
            continue

        if '+' in line:
            parts = line.split('+')
            var1 = parts[0].strip()  # The first variable (e.g., rect_0)
            var2 = parts[1].strip()  # The second variable (e.g., square_0)

            if var1 in variables and var2 in variables:
                # Generate the LDA, ADD, STA instructions
                machine_code.append("10")  # LDA opcode
                machine_code.append(variables[var1][:2])  # First byte of rect_0
address

                machine_code.append(variables[var1][2:])  # Second byte of rect_0
address

                current_address += 3

                machine_code.append("30")  # ADD opcode
                machine_code.append(variables[var2][:2])  # First byte of square_0
address

                machine_code.append(variables[var2][2:])  # Second byte of square_0
address

                current_address += 3

                machine_code.append("20")  # STA opcode
                machine_code.append(variables[var1][:2])  # First byte of rect_0
address

                machine_code.append(variables[var1][2:])  # Second byte of rect_0
address

                current_address += 3
            else:
                raise Exception(f"Undefined variable(s): {var1}, {var2}")
            continue


        if '-' in line:
            parts = line.split('-')
            var1 = parts[0].strip()  # The first variable (e.g., rect_0)
            var2 = parts[1].strip()  # The second variable (e.g., square_0)

            if var1 in variables and var2 in variables:
                # Generate the LDA, ADD, STA instructions
                machine_code.append("10")  # LDA opcode
                machine_code.append(variables[var1][:2])  # First byte of rect_0
address

                machine_code.append(variables[var1][2:])  # Second byte of rect_0
address

                current_address += 3
                machine_code.append("31")  # SUB opcode
                machine_code.append(variables[var2][:2])  # First byte of square_0
address
                machine_code.append(variables[var2][2:])  # Second byte of square_0
address
                current_address += 3

                machine_code.append("20")  # STA opcode
                machine_code.append(variables[var1][:2])  # First byte of rect_0
address
                machine_code.append(variables[var1][2:])  # Second byte of rect_0
address
                current_address += 3
            else:
                raise Exception(f"Undefined variable(s): {var1}, {var2}")
            continue


        # Check if the line is a label
        if line.endswith(":"):
            label = line[:-1]
            labels[label] = f"{current_address:04X}"  # Store the label address
            continue

        # Detect PRINT statement
        if line.startswith("PRINT"):
            # Extract the string within quotes
            start_index = line.find('"') + 1
            end_index = line.rfind('"')
            if start_index != -1 and end_index != -1:
                string_to_print = line[start_index:end_index]  # Keep original case for
string
                # Convert the string to LIT instructions
                lit_instructions = convert_string_to_lit(string_to_print)
                machine_code.extend(lit_instructions)  # Add LIT instructions to
machine code
                current_address += len(lit_instructions)  # Each LIT instruction is 2
bytes
            else:
                raise Exception(f"Invalid PRINT syntax: {line}")
            continue
```

```python
        # Convert line to uppercase (for opcodes, labels, etc.)
        line = line.upper()

        parts = line.split()
        instruction = parts[0]

        # Check if the instruction is valid
        if instruction in opcodes:
            opcode = opcodes[instruction]
            machine_code.append(opcode)
            current_address += 1  # Each opcode is 1 byte

            # Handle immediate and memory address operands
            if len(parts) == 2:
                operand = parts[1]

                # Process operand for LIA and LIT (handle ASCII, variable, or integer)
                if instruction in ['LIA', 'LIB', 'LIT',]:
                    processed_operand = process_operand(operand)
                    machine_code.append(processed_operand)
                    current_address += 1
                # Handle variable reference (e.g., LDA x)
                elif operand in variables:
                    address = variables[operand]
                    machine_code.append(address[:2])  # First byte of the address
                    machine_code.append(address[2:])  # Second byte of the address
                    current_address += 2

                # Handle jump commands (JPC, JMP, etc.)
                elif instruction.startswith('J'):
                    # Handle labels for jumps

                    if operand in labels:
                        address = labels[operand]
                        machine_code.append(address[:2])  # First byte of label address
                        machine_code.append(address[2:])  # Second byte of label
address
                    else:
                        # If label is not yet defined, record its position for
resolution later
                        unresolved_jumps.append((len(machine_code), operand))
                        machine_code.append('00')  # Placeholder bytes
                        machine_code.append('00')
                    current_address += 2

                else:  # Memory address
                    if len(operand) == 4:
                        machine_code.append(operand[:2])  # First byte
                        machine_code.append(operand[2:])  # Second byte
                        current_address += 2
                    else:
                        raise Exception(f"Memory address must be 4 hex digits:
{operand}")
            elif len(parts) == 1 and instruction in ['NOP', 'RTR',
'RTL','STI','LDI','RET','JPI','UPD', 'NOT',]:
                continue  # These instructions are single-byte opcodes
            else:
                raise Exception(f"Invalid syntax: {line}")
        else:
            raise Exception(f"Unknown instruction: {instruction}")

    # Second pass: Resolve unresolved jumps
    for pos, label in unresolved_jumps:
        if label in labels:
            address = labels[label]
            machine_code[pos] = address[:2]  # First byte of label address (high byte)
            machine_code[pos + 1] = address[2:]  # Second byte of label address (low
byte)

        else:
            raise Exception(f"Undefined label: {label}")
    variables.clear()
    next_available_address = 0xFFFF
    return machine_code  # Important! Return the generated machine code


# Function to compile the code
def compile_code():
    assembly_code = text_editor.get("1.0", tk.END)
    try:
        machine_code = assemble_code(assembly_code)
        if machine_code:
            output_text.delete("1.0", tk.END)
```

```
           output_text.insert(tk.END, " ".join(machine_code))  # Display machine code
in text area
   except Exception as e:
       messagebox.showerror("Compilation Error", str(e))  # Show error message




# Function to load a file
def load_file():
   file_path = filedialog.askopenfilename(filetypes=[("Assembly files", "*.asm"),
("All files", "*.*")])
   if file_path:
       with open(file_path, "r") as file:
           text_editor.delete("1.0", tk.END)
           text_editor.insert(tk.END, file.read())

# Function to save the file
def save_file():
   file_path = filedialog.asksaveasfilename(defaultextension=".asm",
filetypes=[("Assembly files", "*.asm"), ("All files", "*.*")])
   if file_path:
       with open(file_path, "w") as file:
           file.write(text_editor.get("1.0", tk.END))

# Create the main window
root = tk.Tk()
root.title("Assembly Text Editor & Assembler")

# Create the text editor
text_editor = tk.Text(root, height=5, width=80)
text_editor.pack(padx=10, pady=10)

# Create the output display for machine code
output_text = tk.Text(root, height=40, width=60, bg="lightgray")
output_text.pack(padx=10, pady=10)

# Create the button frame
button_frame = tk.Frame(root)
button_frame.pack()

# Load, Save, and Compile buttons
```

```
load_button = tk.Button(button_frame, text="Load", command=load_file)
save_button = tk.Button(button_frame, text="Save", command=save_file)
compile_button = tk.Button(button_frame, text="Compile", command=compile_code)

load_button.grid(row=0, column=0, padx=5, pady=5)
save_button.grid(row=0, column=1, padx=5, pady=5)
compile_button.grid(row=0, column=2, padx=5, pady=5)

# Start the GUI loop
root.mainloop()
```

C. **ISA Instructions**

Instruction Set Architecture
00 - No Operation
00: NOP — No operation (does nothing).

Load Immediate Values
Requires 2-digit hex value (XX)

01 XX: LIA — Load immediate value to A register

02 XX: LIB — Load immediate value to B register

03 XX: LIT — Load immediate value to TTY display

Load from RAM
Requires 4-digit hex address (XXXX)

10 XX XX: LDA — Load value from RAM to A register

11 XX XX: LDB — Load value from RAM to B register

12 XX XX: LDT — Load value from RAM to TTY display

13 XX XX: LD0 — Load value from RAM to MAR2

14 XX XX: LD1 — Load value from RAM to MAR3

Does not require hex value

15: LDI — Load value to A register from address formed by MAR2 (low byte) and MAR3 (high byte)

Store to RAM
Requires 4-digit hex address (XXXX)

20 XX XX: STA — Store value from A register to RAM

21 XX XX: STB — Store value from B register to RAM

22 XX XX: STK — Store value from keyboard buffer to RAM and load next value from buffer

Does not require hex value

23: STI — Store value from A register to RAM using MAR2 + MAR3 as address

ALU Operations
Requires 4-digit hex address (XXXX)

30 XX XX: ADD — Add A and B registers; store result in A

31 XX XX: SUB — Subtract B from A; store result in A

Does not require hex value

32: RTR — Rotate A register right by 1 bit

33: RTL — Rotate A register left by 1 bit

Requires 4-digit hex address

34 XX XX: CMP — Compare A and B registers; update flags

Does not require hex value

35: NOT — Bitwise NOT on A register

36 XX XX: AND — AND A register with RAM value; store result in A

37 XX XX: OR — OR A register with RAM value; store result in A

38 XX XX: XOR — XOR A register with RAM value; store result in A

Pixel Display Opcodes
The pixel display uses the last 8 RAM addresses (FFF8–FFFF). Each address represents one line of the display:

FFFF: Top line

FFF8: Bottom line

Does not require hex value

52: UPD — Update display with values from frame buffer

Jump / Branch / Call / Return
Requires 4-digit hex address (XXXX)

70 XX XX: JMP — Unconditional jump to address

71 XX XX: JPC — Jump if carry flag is set

72 XX XX: JPB — Jump if borrow flag is set

73 XX XX: JNZ — Jump if not-zero flag is set

74 XX XX: JPZ — Jump if zero flag is set

75 XX XX: JPG — Jump if greater-than flag is set

76 XX XX: JPE — Jump if equals flag is set

77 XX XX: JPL — Jump if less-than flag is set

78 XX XX: JCAL — Push PC to stack, increment SP, jump to address

Does not require hex value

79: RET — Pop address from stack into PC; decrement SP

Acknowledgement

I would like to express my gratitude to you sir for your dedication and guidance throughout the course. Thank you for the valuable lessons and support, I've learned a lot under your instruction. I wish you all the best in your future endeavors as you conclude your final semester of teaching in UP.

Additionally, I would like to acknowledge the help of a tutorial from AJAX, which I referred to in order to better understand Logisim circuit design concepts..