JOSE RIZAL UNIVERSITY

16-Bit Arduino Visualizer Project

In Partial Fulfillment for the Subject

CPE C401–301G: Computer Architecture and Organization

Submitted to the Computer Engineering Department

Engr. Amor S. Ellares

Submitted By:

Bautista, Rizelle B.

Canas, Fiona Yvonne

Cuisia, Johann Maier

Hao, Joquin Alec P.

July 2025

JOSE RIZAL UNIVERSITY

Table of Contents

Title Page..... 1

Table of Contents..... 2

Introduction.....3

Objectives:..... 3

Materials:..... 3

Reflection:.....4

JOSE RIZAL UNIVERSITY

Google Drive Files:

https://drive.google.com/drive/folders/1PD69pjHk8jPfpGZv_bBLsC5dlAiZ-AtR?usp=drive_link

Introduction

This project simulates a complete 16-bit CPU system with visual feedback and interactive control. The project demonstrates fundamental computer architecture concepts including instructions execution, register operations, memory management, and program flow control through a hands-on, visual learning experience.

Objectives:

We'll create a system that visualizes how a simple program executes step by step, showing:

- Which instruction is currently being executed
- What the instruction does
- How long each instruction takes
- The program's data / variables changing in real-time

JOSE RIZAL UNIVERSITY

Materials:

Materials	Quantity
Arduino Mega 2560	1
Jumper Wires	several
4-Digit 7-Segment Display Common Cathode	1
LEDs	22
Quad Push Buttons	4
Resistor (220Ω)	16
Dual Axis Joystick	1
4x20 LCD I2C	1

‘

Full Execution Flow (Procedures)

1. **System Initialization**

   Power On -> Initialize Hardware -> Display the Welcome Screen -> Enter PROGRAM_SELECT state

2. **Program Selection Phase**
    - LCD shows: List of available programs
    - User action: Move joystick left/right to select program

3. **Program Loading Phase**
    - User action: Press LOAD button
    - System: Copies the selected program from flash memory to RAM
    - State change: PROGRAM_SELECT -> PROGRAM_LOADED
    - Display updates: Shows program name and size

4. **Execution Phase**

    Manual Execution (Step Mode)

    - User action: Press STEP button
    - System performs:
        1. Fetch: Read instruction at current Program Counter address
        2. Decode: Interpret the instruction opcode and operands
        3. Execute: Perform the operation
        4. Increment: Move Program Counter to next instruction
    - Displays update: Show current state

    Automatic Execution (Run Mode)

- User action: Press START / STOP button
- System: Executes steps automatically at set speed (default 800 ms per instruction)
- Can be paused: Press START / STOP again to pause

5. Fetch-Decode-Execute Increment Cycle Detail

- Fetch Phase
  - CPU reads the instruction at the current Program Counter address
  - Instruction is loaded into the current instruction register
- Decode Phase
  - CPU interprets the opcode to determine what operation to perform
  - Operands are identified and prepared
- Execute Phase
  - The actual operation is performed
  - Results are stored in registers or memory
  - Flags may be updated (for comparisons)
- Increment Phase
  - Program Counter is incremented to point to the next instruction
  - Exception: Jump instructions set PC to a specific address

6. Visual Feedback During Execution

- **LED Display (Program Counter)**

- Shows the current Program Counter value in binary
- Example: PC = 5 -> LEDs show 0000000 00000101

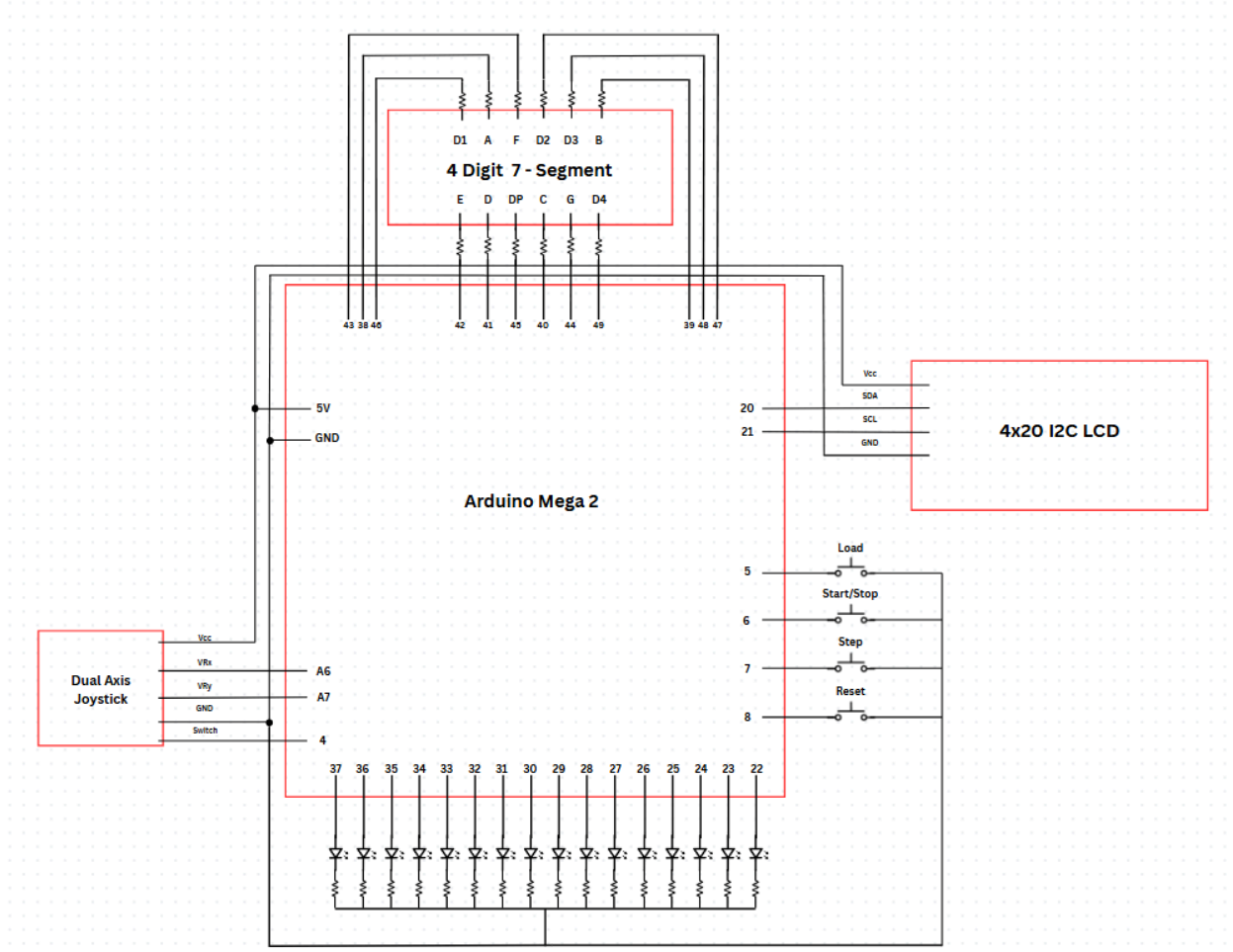
- **7-Segment Display**
  - Shows the current data value being processed
  - Updates when DISPLAY instruction are executed
- **LCD Display**
  - Line 1: Current program name and state
  - Line 2: Program Counter and current instruction
  - Line 3: Instruction count and cycle count
  - Line 4: Current register values (A, B, C, D)

Program Demonstrations:

1. Basic Math: Calculator (25+13) * 2 - 10 = 66
2. Two's Complement: Displays an equation that results in negative value which will display Two's Complement
3. 16-Bit Value: Displays a 16-Bit value
4. Binary Logic: Demonstrate bitwise operations such as: AND, OR, XOR

JOSE RIZAL UNIVERSITY

Schematic Diagram:



JOSE RIZAL UNIVERSITY

Pin Configurations:

LCD Pin	Arduino Pin	Description
VCC	5V	Power supply
GND	GND	Ground
SDA	Pin 20	I2C Data line
SCL	Pin 21	I2C Clock line

7-Segment Pin	Arduino Pin
Segment A	Pin 38
Segment B	Pin 39
Segment C	Pin 40
Segment D	Pin 41
Segment E	Pin 42

Segment F	Pin 43
Segment G	Pin 44
Segment DP	Pin 45
Digit 1 (Thousands)	Pin 46
Digit 2 (Hundreds)	Pin 47
Digit 3 (Tens)	Pin 48
Digit 4 (Ones)	Pin 49

LED	**Arduino Pin**
LED 1	Pin 22
LED 2	Pin 23
LED 3	Pin 24
LED 4	Pin 25
LED 5	Pin 26
LED 6	Pin 27

LED 7	Pin 28
LED 8	Pin 29
LED 9	Pin 30
LED 10	Pin 31
LED 11	Pin 32
LED 12	Pin 33
LED 13	Pin 34
LED 14	Pin 35
LED 15	Pin 36
LED 16	Pin 37

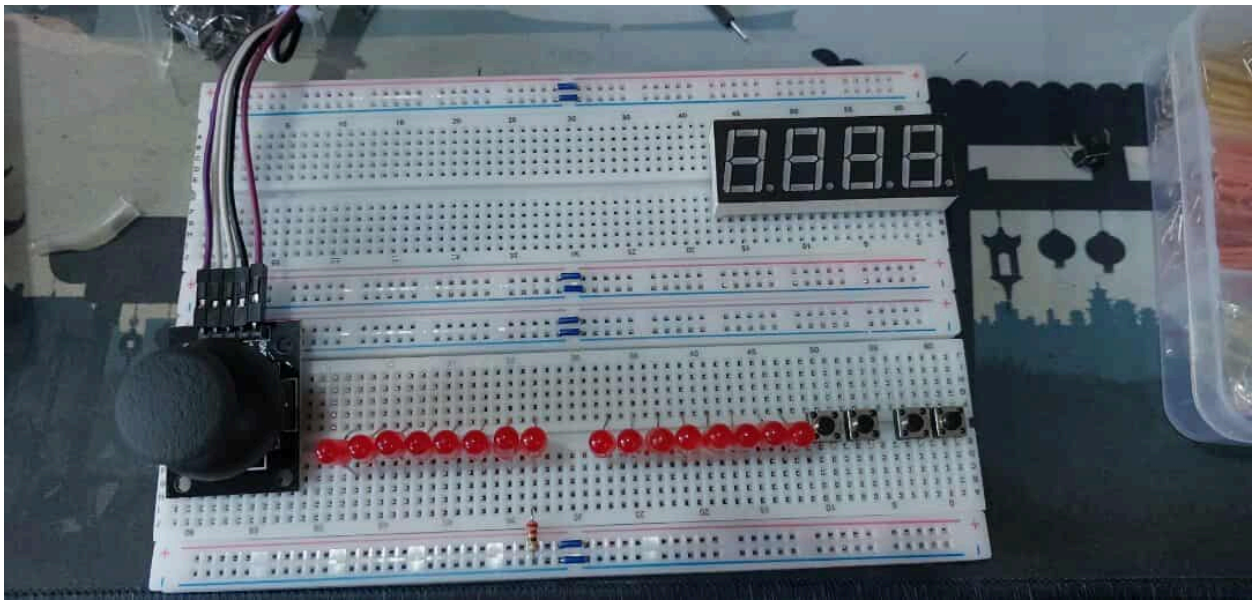
Joystick Pin	**Arduino Pin**
VCC	5V
GND	GND
VRX	Pin A0

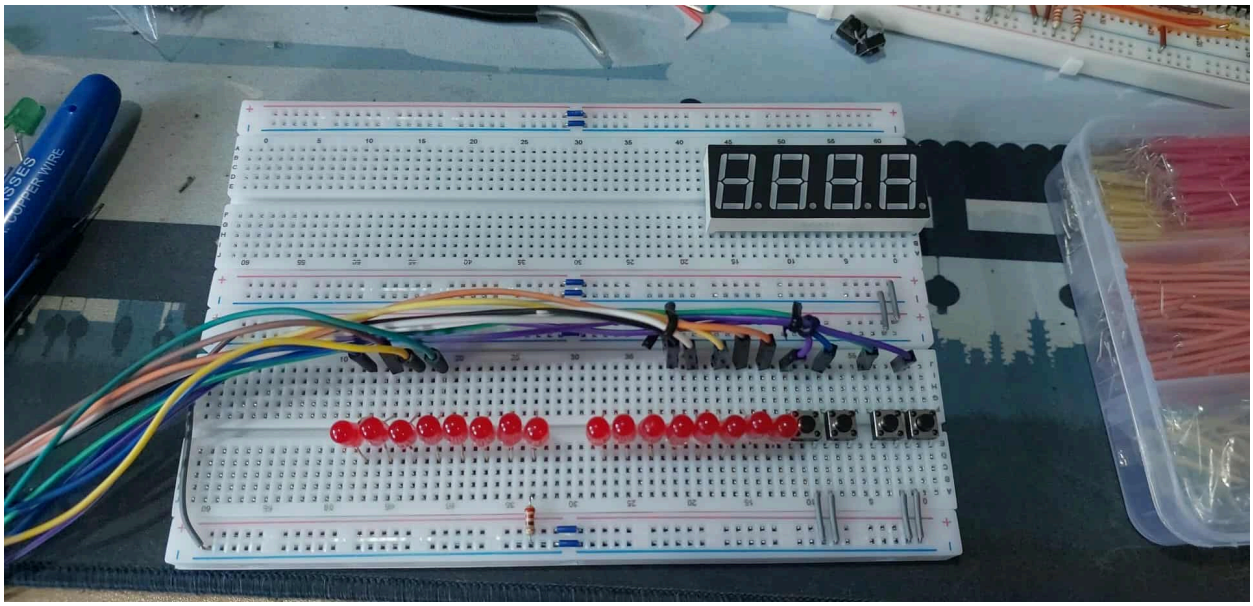
VRY	Pin A1
SW	Pin 4

Button Function	**Arduino Pin**
Load	Pin 5
Start/Stop	Pin 6
Step	Pin 7
Reset	Pin 8

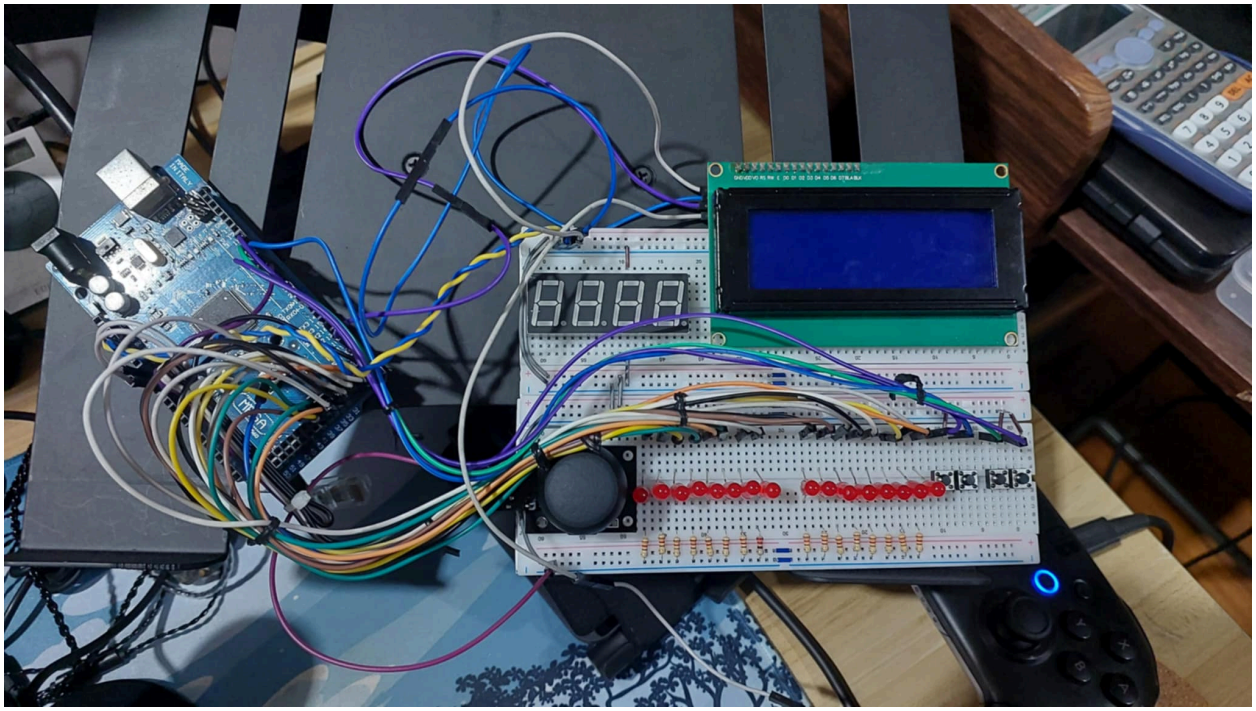
JOSE RIZAL UNIVERSITY

Pictures:





JOSE RIZAL UNIVERSITY



Final Output:

JOSE RIZAL UNIVERSITY

JOSE RIZAL UNIVERSITY

JOSE RIZAL UNIVERSITY

Reflection:

The system was able to visualize the processes that undergo inside the CPU, its status is visualized on the LCD display and as well as the program counter on the LEDs. The final output of the instruction is also properly displayed on the 7-Segment display. With its step by step visualization of the instruction steps, users will be able to observe and understand how the CPU fetches, decodes, executes, and increments each process.

JOSE RIZAL UNIVERSITY

Appendices

Source Code

- G-drive Link: https://drive.google.com/drive/folders/1orB-cYulPq6-g1cP0JWeY5NSkJdy6ehK?usp=sharing

Video Demo

- G-drive Link: https://drive.google.com/drive/folders/1T4-UrmAEzlvAfNgcdg3u1AnegezzepB1?usp=sharing

‘

Arduino Code:

```
 #include <Wire.h>
#include <LiquidCrystal_I2C.h>

// Hardware Configuration
LiquidCrystal_I2C lcd(0x27, 20, 4);  // 20x4 LCD with I2C address 0x27

// 7-Segment Display Pins for 5641AS (Common Cathode)
const int SEGMENT_PINS[] = {38, 39, 40, 41, 42, 43, 44, 45}; // A, B, C, D, E, F, G, DP
const int DIGIT_PINS[] = {46, 47, 48, 49}; // Digit 1, 2, 3, 4 (left to right)

// 7-Segment patterns for digits 0-9 and A-F (FIXED for common cathode 5641AS)
const byte DIGIT_PATTERNS[] = {
  0b00111111, // 0: A,B,C,D,E,F on
  0b00000110, // 1: B,C on
  0b01011011, // 2: A,B,G,E,D on
  0b01001111, // 3: A,B,G,C,D on
  0b01100110, // 4: F,G,B,C on
  0b01101101, // 5: A,F,G,C,D on
  0b01111101, // 6: A,F,G,E,D,C on
  0b00000111, // 7: A,B,C on
  0b01111111, // 8: All segments on
  0b01101111, // 9: A,B,C,D,F,G on
  0b01110111, // A: A,B,C,E,F,G on
```

```
  0b01111100, // B: C,D,E,F,G on
  0b00111001, // C: A,D,E,F on
  0b01011110, // D: B,C,D,E,G on
  0b01111001, // E: A,D,E,F,G on
  0b01110001  // F: A,E,F,G on
};

// Pin Definitions
const int LED_PINS[] = {22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37}; // 16 LEDs
const int JOYSTICK_X = A0;
const int JOYSTICK_Y = A1;
const int JOYSTICK_SW = 4;
const int BUTTON_LOAD = 5;
const int BUTTON_START_STOP = 6;
const int BUTTON_STEP = 7;
const int BUTTON_RESET = 8;

// System Constants
const int MAX_INSTRUCTIONS = 32;
const int MAX_PROGRAMS = 4;
const int NUM_REGISTERS = 4;

// System States
enum SystemState {
  PROGRAM_SELECT,
  PROGRAM_LOADED,
  RUNNING,
  PAUSED,
  HALTED
};

// Instruction Structure
struct Instruction {
  byte opcode;
  word operand1;
  word operand2;
  byte cycles;
  word address;
};

// Enhanced Opcode definitions
```


```
enum Opcodes {
  OP_NOP = 0,     // No operation
  OP_LOAD = 1,    // Load immediate value to register
  OP_ADD = 2,     // Add two registers
  OP_SUB = 3,     // Subtract two registers
  OP_DISPLAY = 4, // Display register value
  OP_HALT = 5,    // Halt execution
  OP_JUMP = 6,    // Unconditional jump
  OP_CMP = 7,     // Compare register with value
  OP_JUMP_LT = 8, // Jump if less than
  OP_JUMP_EQ = 9, // Jump if equal
  OP_DELAY = 10,  // Delay execution
  OP_MOVE = 11,   // Move register to register
  OP_MUL = 12,    // Multiply two registers
  OP_INC = 13,    // Increment register
  OP_DEC = 14,    // Decrement register
  OP_JUMP_GT = 15, // Jump if greater than
  OP_AND = 16,    // Bitwise AND
  OP_OR = 17,     // Bitwise OR
  OP_XOR = 18,    // Bitwise XOR
  OP_SHL = 19,    // Shift left
  OP_SHR = 20,
  OP_LEDOUT = 21
  };


enum Enhanced16BitOpcodes {
  OP_LOAD16 = 25,   // Load 16-bit immediate
  OP_ADD16 = 26,    // 16-bit addition
  OP_MUL16 = 27,    // 16-bit multiplication
  OP_DIV16 = 28,    // 16-bit division
  OP_MOD16 = 29,    // 16-bit modulo
  OP_LOADH = 30,    // Load high byte
  OP_LOADL = 31     // Load low byte
};

// Register definitions
enum RegisterType {
  REG_A = 0, REG_B = 1, REG_C = 2, REG_D = 3
};
```

```
// Opcode names for display
const char* opcodeNames[] = {
  "NOP", "LOAD", "ADD", "SUB", "DISP", "HALT", "JUMP", "CMP",
  "JLT", "JEQ", "DELAY", "MOVE", "MUL", "INC", "DEC", "JGT",
  "AND", "OR", "XOR", "SHL", "SHR"
};

// Register names for display
const char* registerNames[] = {"A", "B", "C", "D"};

// Global Variables
SystemState currentState = PROGRAM_SELECT;
word memory16[256];
word programCounter = 0;
word registers[NUM_REGISTERS] = {0};
byte selectedProgram = 0;
byte activeProgramSize = 0;
bool programRunning = false;
unsigned long lastCycleTime = 0;
unsigned long instructionStartTime = 0;
int executionSpeed = 800; // milliseconds per instruction
Instruction activeProgram[MAX_INSTRUCTIONS];
Instruction currentInstruction;
byte comparisonResult = 0;
byte statusFlags = 0; // Flags register for comparisons

// Button states
bool lastButtonStates[4] = {false, false, false, false};
bool currentButtonStates[4] = {false, false, false, false};

// Joystick states
int lastJoystickX = 512;
int lastJoystickY = 512;
bool lastJoystickSW = false;

// 7-Segment display variables
int displayValue = 0;
byte currentDigit = 0;
unsigned long lastDisplayRefresh = 0;
const unsigned long DISPLAY_REFRESH_INTERVAL = 3; // Faster refresh for smoother display
```

JOSE RIZAL UNIVERSITY

```
// Display optimization variables
String lastDisplayLines[4] = {"", "", "", ""};
bool forceDisplayUpdate = false;
unsigned long lastDisplayUpdate = 0;
const unsigned long DISPLAY_UPDATE_INTERVAL = 150; // Faster updates

// Statistics
unsigned long totalInstructions = 0;
unsigned long totalCycles = 0;
unsigned long programStartTime = 0;

// Enhanced Programs
const Instruction program0[] PROGMEM = {
  // Basic Math: Calculate (25 + 13) * 2 - 10
  {OP_LOAD, REG_A, 25, 1, 0x0000},      // Load 25 into A
  {OP_LOAD, REG_B, 13, 1, 0x0001},      // Load 13 into B
  {OP_ADD, REG_A, REG_B, 2, 0x0002},    // A = A + B (25 + 13 = 38)
  {OP_DISPLAY, REG_A, 0, 1, 0x0003},    // Display A (38)
  {OP_LOAD, REG_C, 2, 1, 0x0004},       // Load 2 into C
  {OP_MUL, REG_A, REG_C, 3, 0x0005},    // A = A * C (38 * 2 = 76)
  {OP_DISPLAY, REG_A, 0, 1, 0x0006},    // Display A (76)
  {OP_LOAD, REG_D, 10, 1, 0x0007},      // Load 10 into D
  {OP_SUB, REG_A, REG_D, 2, 0x0008},    // A = A - D (76 - 10 = 66)
  {OP_DISPLAY, REG_A, 0, 1, 0x0009},    // Display final result (66)
  {OP_HALT, 0, 0, 1, 0x000A}            // Halt
};

const Instruction program1[] PROGMEM = {
  // Modified calculation: 35 - 15 = 20 - 50 = -30 * 2 = -60
  {OP_LOAD, REG_A, 35, 1, 0x0000},      // A = 35
  {OP_LOAD, REG_B, 15, 1, 0x0001},      // B = 15
  {OP_SUB, REG_A, REG_B, 2, 0x0002},    // A = A - B = 35 - 15 = 20
  {OP_DISPLAY, REG_A, 0, 1, 0x0003},    // Display intermediate result (20)
  {OP_DELAY, 50, 0, 50, 0x0004},        // Delay for visibility

  {OP_LOAD, REG_C, 50, 1, 0x0005},      // C = 50
  {OP_SUB, REG_A, REG_C, 2, 0x0006},    // A = A - C = 20 - 50 = -30
  {OP_DISPLAY, REG_A, 0, 1, 0x0007},    // Display intermediate result (-30)
  {OP_DELAY, 50, 0, 50, 0x0008},        // Delay for visibility

  {OP_LOAD, REG_D, 2, 1, 0x0009},       // D = 2
```


```
  {OP_MUL, REG_A, REG_D, 3, 0x000A},    // A = A * D = -30 * 2 = -60

  {OP_DISPLAY, REG_A, 0, 1, 0x000B},    // Display final result (-60)
  {OP_LEDOUT, REG_A, 0, 1, 0x000C},     // Show -60 in 2's complement on 16-bit LEDs
  {OP_HALT, 0, 0, 1, 0x000D}            // Halt
};



const Instruction program2[] PROGMEM = {
 // Work with larger 16-bit values
  {OP_LOAD, 0, 30000, 1, 0x0000},     // Load 30000 into A
  {OP_LOAD, 1, 25000, 1, 0x0001},     // Load 25000 into B
  {OP_ADD, 0, 1, 2, 0x0002},          // A = A + B = 55000
  {OP_DISPLAY, 0, 0, 1, 0x0003},      // Display 55000
  {OP_LOAD, 2, 1000, 1, 0x0004},      // Load 1000 into C
  {OP_MUL, 0, 2, 3, 0x0005},          // A = A * C (would overflow)
  {OP_DISPLAY, 0, 0, 1, 0x0006},      // Display result
  {OP_HALT, 0, 0, 1, 0x0007}
};

// NEW: Binary Logic Demonstration Program
const Instruction program3[] PROGMEM = {
  // Binary Logic: Demonstrate bitwise operations
  {OP_LOAD, REG_A, 85, 1, 0x0000},      // Load 85 (0x55) into A
  {OP_LOAD, REG_B, 170, 1, 0x0001},     // Load 170 (0xAA) into B
  {OP_DISPLAY, REG_A, 0, 1, 0x0002},    // Display A (85)
  {OP_DELAY, 50, 0, 50, 0x0003},        // Delay
  {OP_DISPLAY, REG_B, 0, 1, 0x0004},    // Display B (170)
  {OP_DELAY, 50, 0, 50, 0x0005},        // Delay
  {OP_AND, REG_A, REG_B, 2, 0x0006},    // A = A & B (AND operation)
  {OP_DISPLAY, REG_A, 0, 1, 0x0007},    // Display result (0)
  {OP_DELAY, 50, 0, 50, 0x0008},        // Delay
  {OP_LOAD, REG_A, 85, 1, 0x0009},      // Reload A
  {OP_OR, REG_A, REG_B, 2, 0x000A},     // A = A | B (OR operation)
  {OP_DISPLAY, REG_A, 0, 1, 0x000B},    // Display result (255)
  {OP_DELAY, 50, 0, 50, 0x000C},        // Delay
  {OP_LOAD, REG_A, 85, 1, 0x000D},      // Reload A
  {OP_XOR, REG_A, REG_B, 2, 0x000E},    // A = A ^ B (XOR operation)
  {OP_DISPLAY, REG_A, 0, 1, 0x000F},    // Display result (255)
  {OP_HALT, 0, 0, 1, 0x0010}            // Halt
```


```
};

const byte programSizes[] = {11, 12, 8, 17};  // Changed program1 size from 10 to 12
const char* programNames[] = {"BASIC MATH", "2's Complement", "LARGE NUMBER", "BINARY
LOGIC"};
const char* programDescs[] = {
  "(25+13)*2-10=66",
  "50-100-10=-60",  // Updated description
  "16-bit values",
  "Logic ops demo"
};

void setup() {
  Serial.begin(115200);

  // Initialize LCD
  lcd.init();
  lcd.backlight();
  lcd.clear();

  // Initialize LED pins
  for (int i = 0; i < 16; i++) {
    pinMode(LED_PINS[i], OUTPUT);
    digitalWrite(LED_PINS[i], LOW);
  }

  // Initialize 7-segment display pins
  for (int i = 0; i < 8; i++) {
    pinMode(SEGMENT_PINS[i], OUTPUT);
    digitalWrite(SEGMENT_PINS[i], LOW);
  }

  for (int i = 0; i < 4; i++) {
    pinMode(DIGIT_PINS[i], OUTPUT);
    digitalWrite(DIGIT_PINS[i], HIGH); // Common cathode - HIGH to turn off
  }

  // Initialize button pins
  pinMode(BUTTON_LOAD, INPUT_PULLUP);
  pinMode(BUTTON_START_STOP, INPUT_PULLUP);
  pinMode(BUTTON_STEP, INPUT_PULLUP);
```

```
  pinMode(BUTTON_RESET, INPUT_PULLUP);

  // Initialize joystick pins
  pinMode(JOYSTICK_SW, INPUT_PULLUP);

  // Initialize system
  currentState = PROGRAM_SELECT;
  selectedProgram = 0;

  Serial.println(F("16-Bit CPU Visualizer Initialized"));
  Serial.println(F("Use joystick to select program, buttons to control execution"));

  updateProgramSelectDisplay();
}



void readButtons() {
  currentButtonStates[0] = !digitalRead(BUTTON_LOAD);
  currentButtonStates[1] = !digitalRead(BUTTON_START_STOP);
  currentButtonStates[2] = !digitalRead(BUTTON_STEP);
  currentButtonStates[3] = !digitalRead(BUTTON_RESET);
}

void readJoystick() {
  int x = analogRead(JOYSTICK_X);
  int y = analogRead(JOYSTICK_Y);
  bool sw = !digitalRead(JOYSTICK_SW);

  // Detect joystick movement (with deadzone)
  if (abs(x - lastJoystickX) > 100 || abs(y - lastJoystickY) > 100) {
    if (currentState == PROGRAM_SELECT) {
      if (x < 300) { // Left
        selectedProgram = (selectedProgram + MAX_PROGRAMS - 1) % MAX_PROGRAMS;
        updateProgramSelectDisplay();
      } else if (x > 700) { // Right
        selectedProgram = (selectedProgram + 1) % MAX_PROGRAMS;
        updateProgramSelectDisplay();
      }
    }
  }
```

```
  lastJoystickX = x;
  lastJoystickY = y;
  lastJoystickSW = sw;
}

void handleProgramSelect() {
  if (buttonPressed(0)) { // Load button
    loadProgram(selectedProgram);
    currentState = PROGRAM_LOADED;
    updateProgramLoadedDisplay();
  }
}

void handleProgramLoaded() {
  if (buttonPressed(1)) { // Start/Stop button
    currentState = RUNNING;
    programRunning = true;
    programStartTime = millis();
    instructionStartTime = millis();
    Serial.println(F("\n=== PROGRAM EXECUTION STARTED ==="));
    updateExecutionDisplay();
  } else if (buttonPressed(2)) { // Step button
    executeStep();
    updateExecutionDisplay();
  } else if (buttonPressed(3)) { // Reset button
    resetExecution();
    updateProgramLoadedDisplay();
  }
}

void handleRunning() {
  if (buttonPressed(1)) { // Start/Stop button
    currentState = PAUSED;
    programRunning = false;
    updateExecutionDisplay();
  } else if (buttonPressed(3)) { // Reset button
    resetExecution();
    currentState = PROGRAM_LOADED;
    updateProgramLoadedDisplay();
  } else {
```

```
    // Automatic execution
    if (millis() - instructionStartTime >= executionSpeed) {
      executeStep();
      instructionStartTime = millis();
      updateExecutionDisplay();
    }
  }
}

void handlePaused() {
  if (buttonPressed(1)) { // Start/Stop button
    currentState = RUNNING;
    programRunning = true;
    instructionStartTime = millis();
    updateExecutionDisplay();
  } else if (buttonPressed(2)) { // Step button
    executeStep();
    updateExecutionDisplay();
  } else if (buttonPressed(3)) { // Reset button
    resetExecution();
    currentState = PROGRAM_LOADED;
    updateProgramLoadedDisplay();
  }
}

void handleHalted() {
  // Handle joystick for program selection when halted
  int x = analogRead(JOYSTICK_X);
  static unsigned long lastJoystickMove = 0;

  // Prevent rapid switching with debounce
  if (millis() - lastJoystickMove > 300) {
    if (x < 300) { // Left
      selectedProgram = (selectedProgram + MAX_PROGRAMS - 1) % MAX_PROGRAMS;
      updateHaltedDisplay();
      lastJoystickMove = millis();
    } else if (x > 700) { // Right
      selectedProgram = (selectedProgram + 1) % MAX_PROGRAMS;
      updateHaltedDisplay();
      lastJoystickMove = millis();
    }
```


```
  }

  if (buttonPressed(0)) { // Load button - load new program
    loadProgram(selectedProgram);
    currentState = PROGRAM_LOADED;
    updateProgramLoadedDisplay();
  } else if (buttonPressed(3)) { // Reset button - back to program select
    currentState = PROGRAM_SELECT;
    updateProgramSelectDisplay();
  }
}

bool buttonPressed(int buttonIndex) {
  bool pressed = currentButtonStates[buttonIndex] && !lastButtonStates[buttonIndex];
  lastButtonStates[buttonIndex] = currentButtonStates[buttonIndex];
  return pressed;
}

void loadProgram(byte programIndex) {
  const Instruction* sourceProgram;

  switch (programIndex) {
    case 0: sourceProgram = program0; break;
    case 1: sourceProgram = program1; break;
    case 2: sourceProgram = program2; break;
    case 3: sourceProgram = program3; break;
    default: return;
  }

  activeProgramSize = programSizes[programIndex];

  for (int i = 0; i < activeProgramSize; i++) {
    memcpy_P(&activeProgram[i], &sourceProgram[i], sizeof(Instruction));
  }

  resetExecution();
  Serial.print(F("Loaded program: "));
  Serial.println(programNames[programIndex]);
}

void resetExecution() {
```


```
  programCounter = 0;
  for (int i = 0; i < NUM_REGISTERS; i++) {
    registers[i] = 0;
  }
  totalInstructions = 0;
  totalCycles = 0;
  statusFlags = 0;
  programRunning = false;
  forceDisplayUpdate = true;
}

void executeStep() {
  if (programCounter >= activeProgramSize) {
    currentState = HALTED;
    updateHaltedDisplay();
    return;
  }

  // Fetch instruction
  currentInstruction = activeProgram[programCounter];

  // Check for halt
  if (currentInstruction.opcode == OP_HALT) {
    currentState = HALTED;
    Serial.println(F("\n=== PROGRAM HALTED ==="));
    updateHaltedDisplay();
    return;
  }

  // Serial output for debugging
  Serial.print(F("PC: "));
  Serial.print(programCounter);
  Serial.print(F(" | "));
  Serial.print(opcodeNames[currentInstruction.opcode]);

  // Execute instruction
  executeInstruction();

  // Update statistics
  totalInstructions++;
  totalCycles += currentInstruction.cycles;
```


```
  // Handle program counter increment (except for jumps)
  if (currentInstruction.opcode != OP_JUMP &&
      currentInstruction.opcode != OP_JUMP_LT &&
      currentInstruction.opcode != OP_JUMP_EQ &&
      currentInstruction.opcode != OP_JUMP_GT) {
    programCounter++;
  }

  Serial.println();
}

void executeInstruction() {
  switch (currentInstruction.opcode) {
    case OP_NOP:
      // No operation
      break;

    case OP_LOAD:
      registers[currentInstruction.operand1] = currentInstruction.operand2;
      Serial.print(F(" | "));
      Serial.print(registerNames[currentInstruction.operand1]);
      Serial.print(F(" = "));
      Serial.print(currentInstruction.operand2);  // Now handles 0-65535
      break;


    case OP_ADD:
      registers[currentInstruction.operand1] += registers[currentInstruction.operand2];
      Serial.print(F(" | "));
      Serial.print(registerNames[currentInstruction.operand1]);
      Serial.print(F(" += "));
      Serial.print(registerNames[currentInstruction.operand2]);
      Serial.print(F(" = "));
      Serial.print(registers[currentInstruction.operand1]);
      break;

    case OP_SUB:
      registers[currentInstruction.operand1] -= registers[currentInstruction.operand2];
      Serial.print(F(" | "));
      Serial.print(registerNames[currentInstruction.operand1]);
```

```
    Serial.print(F(" -= "));
    Serial.print(registerNames[currentInstruction.operand2]);
    Serial.print(F(" = "));
    Serial.print(registers[currentInstruction.operand1]);
    break;

  case OP_MUL:
    unsigned long result = (unsigned long)registers[currentInstruction.operand1] *
                           registers[currentInstruction.operand2];
    registers[currentInstruction.operand1] = result & 0xFFFF;  // Keep lower 16 bits      Serial.print(F(" |
"));
    Serial.print(registerNames[currentInstruction.operand1]);
    Serial.print(F(" *= "));
    Serial.print(registerNames[currentInstruction.operand2]);
    Serial.print(F(" = "));
    Serial.print(registers[currentInstruction.operand1]);
    break;

  case OP_DISPLAY:
    showDecimal(registers[currentInstruction.operand1]);
    Serial.print(F(" | Display: "));
    Serial.print(registers[currentInstruction.operand1]);
    break;

  case OP_JUMP:
    programCounter = currentInstruction.operand1;
    Serial.print(F(" | Jump to 0x"));
    if (programCounter < 0x10) Serial.print(F("0"));
    Serial.print(programCounter, HEX);
    break;

   case OP_LEDOUT:
    // Display register value on 16-bit LEDs in 2's complement form
    {
      word value = registers[currentInstruction.operand1];
      for (int i = 0; i < 16; i++) {
        digitalWrite(LED_PINS[i], (value & (1 << i)) ? HIGH : LOW);
      }
      Serial.print(F(" | LED Display: "));
      Serial.print(registerNames[currentInstruction.operand1]);
      Serial.print(F(" = "));
```


```
    Serial.print(value);
    Serial.print(F(" ("));
    // Show as signed 16-bit value
    int16_t signedValue = (int16_t)value;
    Serial.print(signedValue);
    Serial.print(F(", 0b"));
    for (int i = 15; i >= 0; i--) {
      Serial.print((value & (1 << i)) ? '1' : '0');
    }
    Serial.print(F(")"));
  }
  break;

case OP_CMP:
  uint16_t val1 = registers[currentInstruction.operand1];
  uint16_t val2 = registers[currentInstruction.operand2];  // <-- Fix: get value from register
  comparisonResult = val1 - val2;
  statusFlags = 0;
  if (val1 == val2) statusFlags |= 0x01; // Equal
  if (val1 < val2) statusFlags |= 0x02;  // Less
  if (val1 > val2) statusFlags |= 0x04;  // Greater

  Serial.print(F(" | Compare "));
  Serial.print(registerNames[currentInstruction.operand1]);
  Serial.print(F("("));
  Serial.print(val1);
  Serial.print(F(") with "));
  Serial.print(val2);
  if (statusFlags & 0x01) Serial.print(F(" [EQUAL]"));
  else if (statusFlags & 0x02) Serial.print(F(" [LESS]"));
  else if (statusFlags & 0x04) Serial.print(F(" [GREATER]"));
  break;


 case OP_JUMP_LT:
  if (statusFlags & 0x02) {
    programCounter = currentInstruction.operand1;
    Serial.print(F(" | Jump taken to 0x"));
  } else {
    programCounter++;
    Serial.print(F(" | Jump not taken"));
```


```
  }
  break;

case OP_JUMP_EQ:
  if (statusFlags & 0x01) {
    programCounter = currentInstruction.operand1;
    Serial.print(F(" | Jump taken to 0x"));
    if (programCounter < 0x10) Serial.print(F("0"));
    Serial.print(programCounter, HEX);
  } else {
    programCounter++;
    Serial.print(F(" | Jump not taken"));
  }
  break;

case OP_JUMP_GT:
  if (statusFlags & 0x04) {
    programCounter = currentInstruction.operand1;
    Serial.print(F(" | Jump taken to 0x"));
    if (programCounter < 0x10) Serial.print(F("0"));
    Serial.print(programCounter, HEX);
  } else {
    programCounter++;
    Serial.print(F(" | Jump not taken"));
  }
  break;

case OP_DELAY:
  delay(currentInstruction.operand1 * 10);
  Serial.print(F(" | Delay "));
  Serial.print(currentInstruction.operand1 * 10);
  Serial.print(F("ms"));
  break;

case OP_MOVE:
  registers[currentInstruction.operand2] = registers[currentInstruction.operand1];
  Serial.print(F(" | "));
  Serial.print(registerNames[currentInstruction.operand2]);
  Serial.print(F(" = "));
  Serial.print(registerNames[currentInstruction.operand1]);
  Serial.print(F(" ("));
```

```
  Serial.print(registers[currentInstruction.operand1]);
  Serial.print(F(")"));
  break;

case OP_INC:
  registers[currentInstruction.operand1]++;
  Serial.print(F(" | "));
  Serial.print(registerNames[currentInstruction.operand1]);
  Serial.print(F("++ = "));
  Serial.print(registers[currentInstruction.operand1]);
  break;

case OP_DEC:
  registers[currentInstruction.operand1]--;
  Serial.print(F(" | "));
  Serial.print(registerNames[currentInstruction.operand1]);
  Serial.print(F("-- = "));
  Serial.print(registers[currentInstruction.operand1]);
  break;

case OP_AND:
  registers[currentInstruction.operand1] &= registers[currentInstruction.operand2];
  Serial.print(F(" | "));
  Serial.print(registerNames[currentInstruction.operand1]);
  Serial.print(F(" &= "));
  Serial.print(registerNames[currentInstruction.operand2]);
  Serial.print(F(" = "));
  Serial.print(registers[currentInstruction.operand1]);
  break;

case OP_OR:
  registers[currentInstruction.operand1] |= registers[currentInstruction.operand2];
  Serial.print(F(" | "));
  Serial.print(registerNames[currentInstruction.operand1]);
  Serial.print(F(" |= "));
  Serial.print(registerNames[currentInstruction.operand2]);
  Serial.print(F(" = "));
  Serial.print(registers[currentInstruction.operand1]);
  break;

case OP_XOR:
```

```
      registers[currentInstruction.operand1] ^= registers[currentInstruction.operand2];
      Serial.print(F(" | "));
      Serial.print(registerNames[currentInstruction.operand1]);
      Serial.print(F(" ^= "));
      Serial.print(registerNames[currentInstruction.operand2]);
      Serial.print(F(" = "));
      Serial.print(registers[currentInstruction.operand1]);
      break;

    case OP_SHL:
      registers[currentInstruction.operand1] <<= currentInstruction.operand2;
      Serial.print(F(" | "));
      Serial.print(registerNames[currentInstruction.operand1]);
      Serial.print(F(" <<= "));
      Serial.print(currentInstruction.operand2);
      Serial.print(F(" = "));
      Serial.print(registers[currentInstruction.operand1]);
      break;

    case OP_SHR:
      registers[currentInstruction.operand1] >>= currentInstruction.operand2;
      Serial.print(F(" | "));
      Serial.print(registerNames[currentInstruction.operand1]);
      Serial.print(F(" >>= "));
      Serial.print(currentInstruction.operand2);
      Serial.print(F(" = "));
      Serial.print(registers[currentInstruction.operand1]);
      break;

    case OP_HALT:
      Serial.print(F(" | HALT"));
      break;

    default:
      programCounter++;
      Serial.print(F(" | UNKNOWN OPCODE"));
      break;
  }
}

void updateProgramSelectDisplay() {
```


```
  String line0 = "SELECT PROGRAM:";
  String line1 = "< " + String(programNames[selectedProgram]) + " >";
  String line2 = programDescs[selectedProgram];
  String line3 = "Press LOAD to select";

  updateLCDIfChanged(0, line0);
  updateLCDIfChanged(1, line1);
  updateLCDIfChanged(2, line2);
  updateLCDIfChanged(3, line3);
}

void updateProgramLoadedDisplay() {
  String line0 = "LOADED: " + String(programNames[selectedProgram]);
  String line1 = "Size: " + String(activeProgramSize) + " instructions";
  String line2 = "START|STEP|RESET";
  String line3 = "A:" + String(registers[0]) + " B:" + String(registers[1]) + " C:" + String(registers[2]) + "
D:" + String(registers[3]);

  updateLCDIfChanged(0, line0);
  updateLCDIfChanged(1, line1);
  updateLCDIfChanged(2, line2);
  updateLCDIfChanged(3, line3);
}

void updateExecutionDisplay() {
  String stateStr = (currentState == RUNNING) ? "RUNNING" : "PAUSED";
  String line0 = stateStr + ": " + String(programNames[selectedProgram]);

  String line1 = "PC: 0x";
  if (programCounter < 0x10) line1 += "0";
  line1 += String(programCounter, HEX) + " ";

  if (programCounter < activeProgramSize) {
    line1 += opcodeNames[activeProgram[programCounter].opcode];
  } else {
    line1 += "END";
  }

  String line2 = "Inst: " + String(totalInstructions) + " Cycles: " + String(totalCycles);
  String line3 = "A:" + String(registers[0]) + " B:" + String(registers[1]) + " C:" + String(registers[2]) + "
D:" + String(registers[3]);
```


```
  updateLCDIfChanged(0, line0);
  updateLCDIfChanged(1, line1);
  updateLCDIfChanged(2, line2);
  updateLCDIfChanged(3, line3);
}

void updateHaltedDisplay() {
  String line0 = "HALTED - SELECT NEW:";
  String line1 = "< " + String(programNames[selectedProgram]) + " >";
  String line2 = "Inst:" + String(totalInstructions) + " Cyc:" + String(totalCycles);
  String line3 = "LOAD new | RESET menu";

  updateLCDIfChanged(0, line0);
  updateLCDIfChanged(1, line1);
  updateLCDIfChanged(2, line2);
  updateLCDIfChanged(3, line3);
}

void updateLCDIfChanged(int line, String newText) {
  if (newText != lastDisplayLines[line] || forceDisplayUpdate) {
    lcd.setCursor(0, line);
    lcd.print(newText);
    // Clear any remaining characters on the line
    for (int i = newText.length(); i < 20; i++) {
      lcd.print(" ");
    }
    lastDisplayLines[line] = newText;
  }
}

void updateLEDs() {
  // Display Register A value in binary on LEDs
  word value = registers[0]; // Register A
  for (int i = 0; i < 16; i++) {
    digitalWrite(LED_PINS[i], (value & (1 << i)) ? HIGH : LOW);
  }
}

void update7Segment() {
  // Refresh 7-segment display with multiplexing - now shows program counter
```


```
  if (millis() - lastDisplayRefresh >= DISPLAY_REFRESH_INTERVAL) {
    // Turn off all digits
    for (int i = 0; i < 4; i++) {
      digitalWrite(DIGIT_PINS[i], HIGH);
    }

    // Use program counter as display value
    int digit = 0;
    int tempValue = programCounter;

    // Extract the appropriate digit
    for (int i = 0; i < (3 - currentDigit); i++) {
      tempValue /= 10;
    }
    digit = tempValue % 10;

    // Set segments for the digit
    byte pattern = DIGIT_PATTERNS[digit];
    for (int i = 0; i < 8; i++) {
      digitalWrite(SEGMENT_PINS[i], (pattern & (1 << i)) ? HIGH : LOW);
    }

    // Turn on the current digit
    digitalWrite(DIGIT_PINS[currentDigit], LOW);

    // Move to next digit
    currentDigit = (currentDigit + 1) % 4;
    lastDisplayRefresh = millis();
  }
}


void showDecimal(word value) {
  displayValue = value;

  // Ensure value is within display range
  if (displayValue > 65535) displayValue = 65535;
  if (displayValue < 0) displayValue = 0;
}

void showHex(int value) {
```


```
  displayValue = value;

  // For hex display, we'll show the value as decimal for now
  // A more advanced implementation could show actual hex digits
  if (displayValue > 9999) displayValue = 9999;
  if (displayValue < 0) displayValue = 0;
}

void showBinary(int value) {
  displayValue = value;

  // For binary display, we'll show the value as decimal for now
  // The LEDs already show the binary representation
  if (displayValue > 9999) displayValue = 9999;
  if (displayValue < 0) displayValue = 0;
}

// Additional utility functions for enhanced functionality

void printSystemStatus() {
  Serial.println(F("\n=== SYSTEM STATUS ==="));
  Serial.print(F("State: "));
  switch (currentState) {
    case PROGRAM_SELECT: Serial.println(F("PROGRAM_SELECT")); break;
    case PROGRAM_LOADED: Serial.println(F("PROGRAM_LOADED")); break;
    case RUNNING: Serial.println(F("RUNNING")); break;
    case PAUSED: Serial.println(F("PAUSED")); break;
    case HALTED: Serial.println(F("HALTED")); break;
  }

  Serial.print(F("Selected Program: "));
  Serial.println(programNames[selectedProgram]);

  Serial.print(F("Program Counter: 0x"));
  if (programCounter < 0x10) Serial.print(F("0"));
  Serial.println(programCounter, HEX);

  Serial.println(F("Registers:"));
  for (int i = 0; i < NUM_REGISTERS; i++) {
    Serial.print(F("  "));
    Serial.print(registerNames[i]);
```


```
    Serial.print(F(": "));
    Serial.print(registers[i]);
    Serial.print(F(" (0x"));
    if (registers[i] < 0x10) Serial.print(F("0"));
    Serial.print(registers[i], HEX);
    Serial.println(F(")"));
  }

  Serial.print(F("Instructions Executed: "));
  Serial.println(totalInstructions);

  Serial.print(F("Total Cycles: "));
  Serial.println(totalCycles);

  if (programRunning) {
    Serial.print(F("Runtime: "));
    Serial.print(millis() - programStartTime);
    Serial.println(F("ms"));
  }

  Serial.println(F("======================\n"));
}

void dumpProgram() {
  Serial.println(F("\n=== PROGRAM DUMP ==="));
  Serial.print(F("Program: "));
  Serial.println(programNames[selectedProgram]);
  Serial.print(F("Size: "));
  Serial.println(activeProgramSize);
  Serial.println(F("Address | Opcode | Op1 | Op2 | Cycles | Instruction"));
  Serial.println(F("--------|--------|-----|-----|--------|------------"));

  for (int i = 0; i < activeProgramSize; i++) {
    Serial.print(F("0x"));
    if (i < 0x10) Serial.print(F("0"));
    Serial.print(i, HEX);
    Serial.print(F("    | "));

    if (activeProgram[i].opcode < 10) Serial.print(F(" "));
    Serial.print(activeProgram[i].opcode);
    Serial.print(F("     | "));
```

```
if (activeProgram[i].operand1 < 10) Serial.print(F(" "));
Serial.print(activeProgram[i].operand1);
Serial.print(F("  | "));

if (activeProgram[i].operand2 < 10) Serial.print(F(" "));
Serial.print(activeProgram[i].operand2);
Serial.print(F("  | "));

if (activeProgram[i].cycles < 10) Serial.print(F(" "));
Serial.print(activeProgram[i].cycles);
Serial.print(F("      | "));

Serial.print(opcodeNames[activeProgram[i].opcode]);

// Add operand details based on instruction type
switch (activeProgram[i].opcode) {
  case OP_LOAD:
    Serial.print(F(" "));
    Serial.print(registerNames[activeProgram[i].operand1]);
    Serial.print(F(", "));
    Serial.print(activeProgram[i].operand2);
    break;
  case OP_ADD:
  case OP_SUB:
  case OP_MUL:
  case OP_AND:
  case OP_OR:
  case OP_XOR:
    Serial.print(F(" "));
    Serial.print(registerNames[activeProgram[i].operand1]);
    Serial.print(F(", "));
    Serial.print(registerNames[activeProgram[i].operand2]);
    break;
  case OP_DISPLAY:
    // Don't change 7-segment display, just log to serial
    Serial.print(F(" | Display: "));
    Serial.print(registers[currentInstruction.operand1]);
    break;
  case OP_JUMP:
  case OP_JUMP_LT:
```


```
    case OP_JUMP_EQ:
    case OP_JUMP_GT:
      Serial.print(F(" 0x"));
      if (activeProgram[i].operand1 < 0x10) Serial.print(F("0"));
      Serial.print(activeProgram[i].operand1, HEX);
      break;
    case OP_CMP:
      Serial.print(F(" "));
      Serial.print(registerNames[activeProgram[i].operand1]);
      Serial.print(F(", "));
      Serial.print(activeProgram[i].operand2);
      break;
    case OP_DELAY:
      Serial.print(F(" "));
      Serial.print(activeProgram[i].operand1 * 10);
      Serial.print(F("ms"));
      break;
    case OP_MOVE:
      Serial.print(F(" "));
      Serial.print(registerNames[activeProgram[i].operand2]);
      Serial.print(F(", "));
      Serial.print(registerNames[activeProgram[i].operand1]);
      break;
    case OP_INC:
    case OP_DEC:
      Serial.print(F(" "));
      Serial.print(registerNames[activeProgram[i].operand1]);
      break;
    case OP_SHL:
    case OP_SHR:
      Serial.print(F(" "));
      Serial.print(registerNames[activeProgram[i].operand1]);
      Serial.print(F(", "));
      Serial.print(activeProgram[i].operand2);
      break;
  }

  Serial.println();
}

Serial.println(F("====================\n"));
```

```
}

// Enhanced initialization function
void initializeSystem() {
  // Reset all display states
  forceDisplayUpdate = true;

  // Initialize all display lines
  for (int i = 0; i < 4; i++) {
    lastDisplayLines[i] = "";
  }

  // Clear all LEDs
  for (int i = 0; i < 16; i++) {
    digitalWrite(LED_PINS[i], LOW);
  }

  // Initialize 7-segment display
  displayValue = 0;
  currentDigit = 0;

  // Clear LCD
  lcd.clear();

  Serial.println(F("System initialized successfully!"));
  Serial.println(F("Available commands:"));
  Serial.println(F("- Use joystick to navigate"));
  Serial.println(F("- LOAD: Load selected program"));
  Serial.println(F("- START/STOP: Control execution"));
  Serial.println(F("- STEP: Execute one instruction"));
  Serial.println(F("- RESET: Reset execution state"));
  Serial.println(F("- Send 'status' for system status"));
  Serial.println(F("- Send 'dump' for program dump"));
}

// Serial command processing
void processSerialCommands() {
  if (Serial.available() > 0) {
    String command = Serial.readString();
    command.trim();
    command.toLowerCase();
```

```
if (command == "status") {
  printSystemStatus();
} else if (command == "dump") {
  dumpProgram();
} else if (command == "reset") {
  resetExecution();
  Serial.println(F("System reset complete."));
} else if (command == "help") {
  Serial.println(F("Available commands:"));
  Serial.println(F("- status: Show system status"));
  Serial.println(F("- dump: Show program listing"));
  Serial.println(F("- reset: Reset execution"));
  Serial.println(F("- speed <value>: Set execution speed (100-2000ms)"));
  Serial.println(F("- registers: Show register values"));
  Serial.println(F("- help: Show this help"));
} else if (command.startsWith("speed ")) {
  int newSpeed = command.substring(6).toInt();
  if (newSpeed >= 100 && newSpeed <= 2000) {
    executionSpeed = newSpeed;
    Serial.print(F("Execution speed set to "));
    Serial.print(executionSpeed);
    Serial.println(F("ms"));
  } else {
    Serial.println(F("Speed must be between 100 and 2000 ms"));
  }
} else if (command == "registers") {
  Serial.println(F("Register Values:"));
  for (int i = 0; i < NUM_REGISTERS; i++) {
    Serial.print(F("  "));
    Serial.print(registerNames[i]);
    Serial.print(F(": "));
    Serial.print(registers[i]);
    Serial.print(F(" (0x"));
    if (registers[i] < 0x10) Serial.print(F("0"));
    Serial.print(registers[i], HEX);
    Serial.print(F(", 0b"));
    Serial.print(registers[i], BIN);
    Serial.println(F(")"));
  }
} else if (command == "programs") {
```


```
      Serial.println(F("Available Programs:"));
      for (int i = 0; i < MAX_PROGRAMS; i++) {
        Serial.print(F("  "));
        Serial.print(i);
        Serial.print(F(": "));
        Serial.print(programNames[i]);
        Serial.print(F(" - "));
        Serial.println(programDescs[i]);
      }
    } else {
      Serial.println(F("Unknown command. Send 'help' for available commands."));
    }
  }
}

// Enhanced debugging functions
void debugInstruction(const Instruction& inst) {
  Serial.print(F("DEBUG: Opcode="));
  Serial.print(inst.opcode);
  Serial.print(F(" ("));
  Serial.print(opcodeNames[inst.opcode]);
  Serial.print(F("), Op1="));
  Serial.print(inst.operand1);
  Serial.print(F(", Op2="));
  Serial.print(inst.operand2);
  Serial.print(F(", Cycles="));
  Serial.print(inst.cycles);
  Serial.print(F(", Addr=0x"));
  Serial.println(inst.address, HEX);
}

void debugRegisters() {
  Serial.print(F("REGS: "));
  for (int i = 0; i < NUM_REGISTERS; i++) {
    Serial.print(registerNames[i]);
    Serial.print(F("="));
    Serial.print(registers[i]);
    if (i < NUM_REGISTERS - 1) Serial.print(F(", "));
  }
  Serial.println();
}
```


```
// Performance monitoring functions
void calculatePerformanceMetrics() {
  if (totalCycles > 0) {
    unsigned long runtime = millis() - programStartTime;
    float instructionsPerSecond = (float)totalInstructions / (runtime / 1000.0);
    float cyclesPerSecond = (float)totalCycles / (runtime / 1000.0);

    Serial.println(F("\n=== PERFORMANCE METRICS ==="));
    Serial.print(F("Runtime: "));
    Serial.print(runtime);
    Serial.println(F(" ms"));

    Serial.print(F("Instructions/sec: "));
    Serial.println(instructionsPerSecond, 2);

    Serial.print(F("Cycles/sec: "));
    Serial.println(cyclesPerSecond, 2);

    Serial.print(F("Average cycles per instruction: "));
    Serial.println((float)totalCycles / totalInstructions, 2);

    Serial.println(F("============================\n"));
  }
}

// Enhanced display functions
void displayProgramCounter() {
  // Show program counter on 7-segment display
  showHex(programCounter);
}

void displayRegisterOnLEDs(int regIndex) {
  // Display a specific register value on LEDs
  if (regIndex >= 0 && regIndex < NUM_REGISTERS) {
    byte value = registers[regIndex];
    for (int i = 0; i < 8; i++) {
      digitalWrite(LED_PINS[i], (value & (1 << i)) ? HIGH : LOW);
    }
    // Clear upper LEDs
    for (int i = 8; i < 16; i++) {
```

```
      digitalWrite(LED_PINS[i], LOW);
    }
  }
}

void displayStatusFlags() {
  // Display status flags on upper LEDs
  for (int i = 8; i < 16; i++) {
    digitalWrite(LED_PINS[i], LOW);
  }

  // Show comparison flags
  if (statusFlags & 0x01) digitalWrite(LED_PINS[8], HIGH);  // Equal
  if (statusFlags & 0x02) digitalWrite(LED_PINS[9], HIGH);  // Less than
  if (statusFlags & 0x04) digitalWrite(LED_PINS[10], HIGH); // Greater than
  if (programRunning) digitalWrite(LED_PINS[11], HIGH);     // Running indicator
  if (currentState == HALTED) digitalWrite(LED_PINS[12], HIGH); // Halted indicator
}

// Enhanced LED patterns
void animateLEDs() {
  static unsigned long lastAnimation = 0;
  static int animationStep = 0;

  if (millis() - lastAnimation > 100) {
    if (currentState == PROGRAM_SELECT) {
      // Spinning animation during program select
      for (int i = 0; i < 16; i++) {
        digitalWrite(LED_PINS[i], LOW);
      }
      digitalWrite(LED_PINS[animationStep % 16], HIGH);
      animationStep++;
    } else if (currentState == RUNNING) {
      // Pulse animation during execution
      int brightness = (sin(animationStep * 0.1) + 1) * 127;
      animationStep++;
    }
    lastAnimation = millis();
  }
}
```


```
// Memory and stack simulation
byte memory[256]; // Simulated memory
byte stackPointer = 0xFF; // Stack pointer starts at top

void pushStack(byte value) {
  if (stackPointer > 0) {
    memory[stackPointer] = value;
    stackPointer--;
  }
}

byte popStack() {
  if (stackPointer < 0xFF) {
    stackPointer++;
    return memory[stackPointer];
  }
  return 0;
}

// Enhanced instruction set (future expansion)
void executeEnhancedInstruction() {
  // Placeholder for future enhanced instructions
  switch (currentInstruction.opcode) {
    case 21: // OP_PUSH
      pushStack(registers[currentInstruction.operand1]);
      break;
    case 22: // OP_POP
      registers[currentInstruction.operand1] = popStack();
      break;
    case 23: // OP_CALL
      pushStack(programCounter + 1);
      programCounter = currentInstruction.operand1;
      break;
    case 24: // OP_RET
      programCounter = popStack();
      break;
  }
}

// Error handling and validation
bool validateInstruction(const Instruction& inst) {
```


```
  // Check if opcode is valid
  if (inst.opcode > OP_SHR) {
    Serial.print(F("ERROR: Invalid opcode "));
    Serial.println(inst.opcode);
    return false;
  }

  // Check if register operands are valid
  if (inst.operand1 >= NUM_REGISTERS &&
      (inst.opcode == OP_LOAD || inst.opcode == OP_ADD ||
       inst.opcode == OP_SUB || inst.opcode == OP_DISPLAY)) {
    Serial.print(F("ERROR: Invalid register "));
    Serial.println(inst.operand1);
    return false;
  }

  return true;
}

// System diagnostics
void runDiagnostics() {
  Serial.println(F("\n=== SYSTEM DIAGNOSTICS ==="));

  // Test LEDs
  Serial.println(F("Testing LEDs..."));
  for (int i = 0; i < 16; i++) {
    digitalWrite(LED_PINS[i], HIGH);
    delay(50);
    digitalWrite(LED_PINS[i], LOW);
  }

  // Test 7-segment display
  Serial.println(F("Testing 7-segment display..."));
  for (int i = 0; i < 10; i++) {
    showDecimal(i);
    delay(200);
  }

  // Test LCD
  Serial.println(F("Testing LCD..."));
  lcd.clear();
```

```
  for (int i = 0; i < 4; i++) {
    lcd.setCursor(0, i);
    lcd.print("Line " + String(i + 1) + " Test");
    delay(500);
  }

  Serial.println(F("Diagnostics complete."));
  Serial.println(F("==========================\n"));
}

// Configuration management
void saveConfig() {
  // Future: Save configuration to EEPROM
  Serial.println(F("Configuration saved (simulated)"));
}

void loadConfig() {
  // Future: Load configuration from EEPROM
  Serial.println(F("Configuration loaded (simulated)"));
}

// Enhanced setup function
void enhancedSetup() {
  Serial.begin(115200);

  // Initialize all hardware
  initializeSystem();

  // Run diagnostics if requested
  Serial.println(F("Send 'diag' to run diagnostics"));

  // Load configuration
  loadConfig();

  // Display welcome message
  Serial.println(F("\n=== 16-BIT CPU VISUALIZER ==="));
  Serial.println(F("Version 2.0 - Enhanced Edition"));
  Serial.println(F("Ready for operation!"));
  Serial.println(F("============================\n"));
}
```

```
// Main loop with enhanced features
void enhancedMainLoop() {
  // Process serial commands
  processSerialCommands();

  // Read inputs
  readButtons();
  readJoystick();

  // Handle state machine
  switch (currentState) {
    case PROGRAM_SELECT:
      handleProgramSelect();
      animateLEDs();
      break;
    case PROGRAM_LOADED:
      handleProgramLoaded();
      break;
    case RUNNING:
      handleRunning();
      animateLEDs();
      break;
    case PAUSED:
      handlePaused();
      break;
    case HALTED:
      handleHalted();
      break;
  }

  // Update displays
  updateLEDs();
  update7Segment();

  // Reset force update flag
  if (forceDisplayUpdate) {
    forceDisplayUpdate = false;
  }

  delay(10); // Small delay for stability
}
```

```
// Replace the original loop() function with this enhanced version
void loop() {
  enhancedMainLoop();
}
```