Detailed Code Documentation

4-bit Microcontroller Simulator

March 5, 2025

Contents

1 Introduction

This document provides an in-depth technical explanation of the source code for the **4-bit Microcontroller Simulator**. The simulator is implemented in a single HTML file utilizing HTML, CSS, and JavaScript. Its purpose is to emulate a basic 4-bit microcontroller and demonstrate key concepts in microcontroller programming. This documentation is intended for engineers and developers seeking a thorough understanding of the inner workings of the code.

2 Overall Code Structure

The code is organized into three main components:

1. **HTML**: Defines the user interface, including the code editor, control buttons, register/flag displays, memory map, and instruction set summary.
2. **CSS**: Provides styling and layout using CSS Grid, custom properties, and responsive design techniques to ensure a clean and intuitive interface.
3. **JavaScript**: Contains the logic for compiling the assembly-like code, executing instructions, updating the internal state (registers, flags, memory, PC), and refreshing the display.

3 HTML Structure

3.1 Interface Layout

The HTML portion creates several key interface elements:

- **Code Editor**: A `<textarea>` is provided for users to input or modify assembly code.
- **Toolbar**: Contains buttons such as `Compile`, `Step`, `Run`, `Stop`, and `Reset`; dropdown selectors allow configuration of execution speed and loading of example code snippets.
- **Register and Flag Panel**: Displays the values of four 4-bit registers (R0–R3) and status flags (Zero, Carry, Overflow).
- **Memory and PC Panel**: Shows the current Program Counter (PC) value and a visual map of 16 memory cells, simulating a small RAM.
- **Instruction Set Panel**: Lists and briefly explains the supported instructions.

4 CSS Styling

4.1 Layout and Visual Design

The CSS styling is defined in the `<style>` section of the HTML document:

- **Custom Variables**: Color and font variables ensure a consistent look and feel.
- **Grid Layout**: CSS Grid is used to divide the interface into two columns, separating the code editor from the simulation output panels.

- **Component Styles**: Specific styling is provided for buttons, the text area, and the memory cells. The cell corresponding to the current PC is highlighted for clarity.
- **Responsive Design**: The layout is designed to adjust to various screen sizes.

5 JavaScript Code Structure

The JavaScript section is the core of the simulator. It is responsible for reading the code, simulating the microcontroller, and updating the user interface.

5.1 Global Object: `uC`

The simulator state is encapsulated in the global object `uC`, which includes:

- **registers**: An array of four 4-bit registers (R0, R1, R2, R3).
- **memory**: An array of 16 cells, each storing a 4-bit value (simulating RAM).
- **pc**: The Program Counter (PC), a 4-bit value that tracks the current instruction address. It wraps around modulo 16.
- **flags**: An object containing the status flags: `zero`, `carry`, and `overflow`.
- **program**: An array that holds the compiled instructions.
- **labels**: A mapping of labels (from the assembly code) to their corresponding instruction addresses.
- **running**: A boolean flag indicating if continuous execution is active.
- **speed**: A numeric value representing the delay between instruction executions.

5.2 Compilation Process: `compile()`

The `compile()` function translates the assembly-like code into an executable program:

a. **Input Reading**: Retrieves the code from the `<textarea>`.

b. **Preprocessing**: Removes comments (anything following a `;`), trims whitespace, and converts the text to uppercase.

c. **Label Mapping**: Lines ending with a colon (`:`) are identified as labels. These are stored in the `uC.labels` mapping with the current instruction address.

d. **Tokenization and Parsing**: Each non-label line is tokenized and parsed into an instruction object containing:
   - The instruction **type** (e.g., `MOV`, `ADD`, `CMP`, `LOAD`, `STORE`, etc.).
   - The **operands** (registers, immediate values, or labels).
   - The original code line for reference.

e. **Program Array Population**: Each instruction object is appended to `uC.program` in sequence.

5.3 Instruction Execution: `step()`

The `step()` function is responsible for executing a single instruction:

- **Instruction Fetch**: Retrieves the current instruction from `uC.program` using the PC.
- **Instruction Decoding and Execution**: A `switch` statement is used to:
  - Execute `MOV` for data transfer or immediate value loading.
  - Perform ALU operations such as `ADD`, `SUB`, `AND`, and `OR` while updating flags (carry and zero).
  - Execute `CMP` to compare a register with zero.
  - Handle branching instructions (`JMP`, `JZ`, `JNZ`) by modifying the PC.
  - Execute memory operations (`LOAD` and `STORE`) to interact with the memory array.
- **Program Counter Update**: The PC is incremented and wrapped around modulo 16.
- **UI Update**: Calls to `updateDisplay()` and `updateMemoryView()` refresh the interface to reflect the new state.

5.4 Continuous Execution: `run()`

The `run()` function enables continuous execution of the program:

- Sets `uC.running` to `true`.
- Uses a recursive call via `setTimeout()` to repeatedly invoke `step()` at intervals defined by `uC.speed`.
- Execution halts when the program completes or when `stop()` is called.

5.5 Stop and Reset Functions

- `stop()`: Sets `uC.running` to `false`, halting continuous execution.
- `reset()`: Resets the microcontroller state by:
  - Zeroing all registers.
  - Resetting the PC to 0.
  - Clearing the flags.
  - Updating the UI to reflect the initial state.

5.6 User Interface Update Functions

- `updateDisplay()`: Iterates over the registers and flags, updating the corresponding HTML elements. Registers are displayed in both binary (4-bit) and decimal formats.
- `updateMemoryView()`: Updates the visual memory map by iterating through the `uC.memory` array and highlighting the cell corresponding to the current PC.
- `log()`: Appends messages to a console area in the UI for debugging and execution feedback.

5.7 Example Code Loader: `loadExample(example)`

This function loads pre-defined code examples into the editor:

- It accepts a parameter that identifies the example (e.g., `fibonacci`, `arithmetic`, etc.).
- It replaces the content of the code editor (`<textarea>`) with the selected example code.

5.8 Event Handling

- An event listener is attached to capture `keydown` events, allowing users to step through instructions by pressing the Space key.
- Button clicks in the toolbar trigger functions such as `compile()`, `step()`, `run()`, `stop()`, and `reset()`.

6 Instruction Set Implementation Details

6.1 Data Movement Instructions

- **MOV Rd, Rs**: Copies the content of source register `Rs` to destination register `Rd`.
- **MOV Rd, #n**: Loads an immediate 4-bit value `n` (where $0 \leq n \leq 15$) into register `Rd`.

6.2 Arithmetic and Logical Instructions

- **ADD Rd, Rs**: Adds the value in register `Rs` to `Rd` using modulo-16 arithmetic. The Carry flag is set if the sum exceeds 15.
- **SUB Rd, Rs**: Subtracts the value in `Rs` from `Rd` (modulo 16). The Carry flag is set if `Rd` is less than `Rs`.
- **AND Rd, Rs** and **OR Rd, Rs**: Perform bitwise AND and OR operations respectively between registers `Rd` and `Rs`.

6.3 Comparison and Branching Instructions

- **CMP Rs**: Compares the value in register `Rs` with zero and sets the ZERO flag if the value is 0.
- **JMP label**: Performs an unconditional jump to the instruction at the specified label.
- **JZ label** and **JNZ label**: Execute conditional jumps based on the state of the ZERO flag.

6.4 Memory Operations

- **LOAD Rd, #n**: Loads the value from memory cell `n` into register `Rd`.
- **STORE Rs, #n**: Stores the value from register `Rs` into memory cell `n`.

For further details, modifications, or contributions, please refer to the GitHub repository:

`https://github.com/brunovmcastanho/uC-Simulator`