ISA Project Documentation

Project Overview

This project focuses on the implementation of an assembler and simulator for the SIMP processor, a simplified version of the MIPS processor. The project aims to enhance our understanding of Instruction Set Architecture (ISA), input/output operations, and low-level programming using the C programming language.

The project consists of two separate components:

1. Assembler – Converts SIMP Assembly code into machine-readable format.
2. Simulator – Executes the binary instructions generated by the assembler, simulating the processor's behavior.

Additionally, we implemented four test programs in Assembly to validate the correctness of our assembler and simulator.

SIMP Instruction Format

The SIMP processor follows a fixed 48-bit instruction format. Each instruction is structured as follows:

47:40	39:36	35:32	31:28	27:24	23:12	11:0
Opcode	rd	rs	rt	rm	Immediate1	Immediate2

- Opcode (8 bits): Specifies the operation to be executed.
- rd, rs, rt, rm (4 bits each): Registers used in the instruction.
- Immediate1 & Immediate2 (12 bits each): Used for immediate values or memory addresses.

Example Instruction Encoding

Assembly:

add $t1, $t2, $imm1, $zero, 5, 0

Machine Code (Hexadecimal):

0x008910005000

Where:

- 00 = Opcode for add
- 8 = Register $t1
- 9 = Register $t2
- 1 = Register $imm1
- 0 = Register $zero
- 005 = Immediate1 value
- 000 = Immediate2 value

This format is used to ensure efficient execution and easy parsing in the assembler and simulator.

Assembler Overview

The assembler is responsible for translating SIMP assembly instructions to a machine code that can be executed by the simulator. The process is divided into two main passes:

Pass 1: Label Collection

1. Reads the input assembly file line by line.
2. Identifies labels and stores them in a linked list, where each node contains a label's address.
3. Skips processing actual instructions during this phase but keeps track of the program counter (PC).

Pass 2: Instruction Encoding

1. Reads the assembly file again, now using the label list for address resolution.
2. Extracts each instruction's parameters and stores their integer values in a Command structure.
3. Analyze the Command structure's arguments and convert them to hexadecimal representations.
4. Concatenate each parameter's hexadecimal representation to complete a 12-digit hexadecimal string.
5. Write the hexadecimal string to a new line in imemin.txt.
6. When encountering a pseudo instruction analyze it's parameters and directly write in the right row in dmemin.txt.

Assembler Components

- Parsing & Tokenization: Splits each line into meaningful tokens (opcode, registers, immediates, and labels).
- Linked list Management: Stores labels and their corresponding memory addresses.
- Instruction Encoding: Converts assembly instructions into hexadecimal representations.
- File Writing: Outputs machine code to imemin.txt and dmemin.txt.

Assembler Key Functions

- buildCommand() – Receives a SIMP full command line string, extracts the parameters of the operation, registers and values it contains and builds a Command struct.
- find_opcode() – Recieves a string of a SIMP command and returns it's int value.
- find_register() – Recieves a string of a SIMP register and returns it's int value.
- find_immediate_type() – Determines if a value in the immediate field is decimal, hexadecimal, or a label.
- string_to_int() – receives a string of decimal, hexadecimal value or a label and returns it's int value. Mostly (but not only) used to convert immediate field values.
- cmd_to_hex_line() – Converts a parsed instruction into its 48-bit hexadecimal representation.
- get_pseudo_instruction() – receives a SIMP full command line of a pseudo instruction and extracts the address and value it contains.
- print_2D_array_to_file() – Writes data memory image to output files.

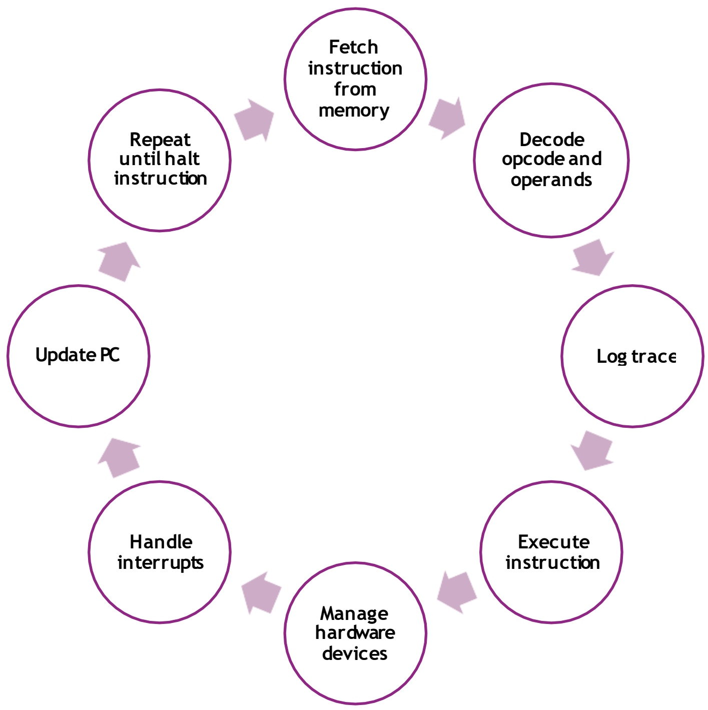
Additional helper functions and definitions

- remove_comments() – receives a SIMP full command line and removes comments.
- typdef struct Command – a structure to store a SIMP command's parameters as integers.
- typedef struct Label – a structure to store a label's address. All labels are stored in a linked list. Therefore, it stores contains a pointer to the next label. Additionally, there are helper functions to manage the list, namely add_label() and find_label().
- concat_hex_str() – A helper function that receives a parameter's int value and concatenates it's hexadecimal representation to a hexadecimal string.
- int_to_hex() – A helper function that receives a parameter's int value and returns a string of it's hexadecimal representation.
- is_label() – receives a string line and checks if it is a label.
- dmem_initialize() – initializes a 2D array with 4096 lines that will hold the data in dmemin.txt.

Simulator Structure and Operation

The SIMP simulator is designed to emulate the behavior of the SIMP processor by following the fetch-decode-execute cycle. The simulator is modular, with different components managing instruction execution, hardware interactions, and data processing.

Simulator Execution Flowchart



1. Simulator Workflow

1. **Initialization:**
   - Loads instruction memory (imemin.txt), data memory (dmemin.txt), disk content (diskin.txt), and external interrupt events (irq2in.txt).
   - Initializes registers, hardware components (LEDs, timer, disk, monitor), and interrupt handlers.
2. **Fetch-Decode-Execute Cycle:**
   - **Fetch:** Retrieves instructions from instruction memory based on the Program Counter (PC).
   - **Decode:** Parses the instruction into opcode, registers, and immediate values.
   - **Log Trace:** Records the current state before execution for debugging purposes.
   - **Execute:** Executes operations (arithmetic, memory, I/O, control flow).
   - **Manage Hardware:** Handles disk, timer, monitor, and I/O operations.
   - **Interrupt Handling:** Checks and responds to pending interrupts (IRQ0, IRQ1, IRQ2).
   - **Update PC:** Increments or modifies the Program Counter based on control flow.

3. **Completion:**
    - Stops execution upon encountering a halt instruction.
    - Writes final outputs to 10 different files (e.g., dmemout.txt, regout.txt, trace.txt, diskout.txt).

2. Structure

The simulator is divided into multiple C source files to ensure modularity, readability, and maintainability:

A. Core Simulation Logic

- **main.c**:
    - Entry point for the simulator.
    - Handles file input/output operations and initializes the simulation environment.
    - Calls the simulate() function to start execution.
- **simulation.c**:
    - Implements the main **fetch-decode-execute** cycle.
    - Coordinates between fetching, decoding, executing instructions, and managing hardware components.

B. Instruction Execution

- **Oparations.c**:
    - Handles execution of arithmetic (add, sub, mac), comparison, branching (beq, bne), and memory instructions (lw, sw).

C. I/O Operations and Interrupt Handling

- **io_operations.c**:
    - Manages in, out, and reti instructions.
    - Handles hardware register interactions and interrupt service routines (ISR).

D. Hardware Device Management

- **device_oparations.c**:
    - Manages operations for the disk, monitor, timer, and LEDs.
    - Implements Direct Memory Access (DMA) for disk operations.

E. Input and Output Management

- **input.c**:
    - Loads data from input files (imemin.txt, dmemin.txt, diskin.txt, irq2in.txt).
- **output.c**:

- Writes simulation results to output files (dmemout.txt, regout.txt, trace.txt, diskout.txt, etc.).
- Logs LED status, monitor display changes, and hardware register activities.

F. Utility Functions

- **utils.c**:
  - Provides helper functions for data conversion, validation, and memory management.

G. Header Files

- **simulator_functions.h**:
  - Declares all functions, constants, macros, and data structures used across the simulator's modules.
  - Ensures modular communication between different components.

3. Input and Output Files

Input Files (4):

1. imemin.txt – Initial instruction memory.
2. dmemin.txt – Initial data memory.
3. diskin.txt – Initial disk content.
4. irq2in.txt – External interrupt event timings.

Output Files (10):

1. dmemout.txt – Final state of data memory.
2. regout.txt – Final register values (R3–R15).
3. trace.txt – Execution trace for debugging.
4. hwregtrace.txt – Hardware register activity log.
5. cycles.txt – Total executed clock cycles.
6. leds.txt – LED status log.
7. display7seg.txt – 7-segment display output.
8. diskout.txt – Final disk content.
9. monitor.txt – Final monitor display in text format.
10. monitor.yuv – Binary monitor output for graphical display.

4. Simulator Key Functions

- **simulate()** – Manages the main simulation loop.
- **fetch_instruction()** – Retrieves the next instruction based on the PC.
- **decode_instruction()** – Decodes the fetched instruction into meaningful parts.
- **execute_instruction()** – Executes the decoded instruction.
- **handle_interrupts()** – Handles hardware interrupts.
- **manage_disk()** – Manages disk read/write operations.
- **handle_timer()** – Manages timer events and triggers IRQ0.
- **log_trace()** – Logs instruction execution details for debugging.
- **write_data_memory()** – Writes the final data memory state to output.
- **write_registers()** – Writes final register values to output.

Test Programs

Each test program verifies specific functionality of the assembler and simulator:

1. Matrix Multiplication (mulmat.asm)

- Computes: result[i][j] = sum(matrix1[i][k] * matrix2[k][j]).
- Uses: Load (lw), Store (sw), Arithmetic (add, mac).

2. Binomial Coefficient (binom.asm)

- Recursive formula:
- C(n, k) = C(n-1, k-1) + C(n-1, k)
- Registers: Stack-based recursion for depth tracking.

3. Circle Drawing (circle.asm)

- Algorithm: Midpoint Circle Algorithm for pixel rendering.
- Uses: out for pixel updates, iterative calculations.

4. Disk Sector Shifting (disktest.asm)

- Shifts first 8 disk sectors forward.
- Ensures correct order of read/write to prevent overwriting data.