ISA 2025 Project Documentation

Course: Computer Organization (0512.4400)

Academic Year: 2025, Semester B

Team Members:

- Shahaf Gino (ID: 207477639)
- Roi Yanai (ID: 322380114)

Date of Submission: July 10, 2025

Table of Contents

1. Project Overview

1.1 Project Requirements and Objectives

This project implements a complete computer system for the SIMP (Simplified MIPS) processor, a RISC-based architecture designed for educational purposes. The project consists of two main components: an assembler that converts assembly language programs into machine code, and a cycle-accurate simulator that executes the machine code while modeling all processor subsystems including interrupts, I/O devices, and memory.

The main deliverables include an assembler program written in C that handles all 22 SIMP instructions with proper label resolution and immediate value encoding, a comprehensive simulator that provides cycle-accurate execution with complete peripheral simulation, four test programs demonstrating various aspects of the processor capabilities, and complete documentation covering all implementation details.

1.2 SIMP Processor Specification

The SIMP processor is a 32-bit RISC architecture with 16 registers, 4096 words of unified instruction and data memory, and a comprehensive instruction set covering arithmetic, logical, memory, branch, and I/O operations. The processor includes special registers such as $zero (always zero) and $imm (automatically loaded with immediate values during instruction decode).

The instruction format supports both single-word and double-word encodings, with the bigimm flag determining whether an instruction uses an 8-bit or 32-bit immediate value. Instructions are executed in a cycle-accurate manner, with regular instructions taking one cycle and bigimm instructions taking two cycles.

1.3 I/O System and Peripherals

The SIMP processor includes a comprehensive I/O system with 23 hardware registers controlling various peripherals. The timer system provides programmable interrupts with 32-bit precision, while the disk system simulates a 128-sector hard drive with realistic DMA timing. The monitor system supports a 256×256 monochrome display with 8-bit grayscale values, and additional peripherals include 32 LEDs and a 7-segment display.

The interrupt system supports three interrupt sources: IRQ0 from the timer, IRQ1 from the disk controller, and IRQ2 from external events. Interrupts are handled with proper priority and timing, including nested interrupt prevention and precise cycle-accurate behavior.

2. System Architecture

2.1 Overall System Design

The project implements a two-stage processing pipeline where assembly source code is first processed by the assembler to generate machine code, which is then executed by the simulator. The assembler performs lexical analysis, symbol resolution, and code generation, while the simulator implements the complete processor including fetch-decode-execute cycles, interrupt handling, and peripheral simulation.

The system architecture follows a modular design with clear separation between assembler and simulator components. The assembler is implemented as a single-file solution focusing on simplicity and correctness, while the simulator uses a multi-file architecture with separate modules for different subsystems.

2.2 Instruction Set Architecture

The SIMP instruction set includes 22 instructions covering all essential operations. Arithmetic instructions (add, sub, mul) provide basic computation, while logical instructions (and, or, xor) and shift operations (sll, sra, srl) enable bit manipulation. Branch instructions (beq, bne, blt, bgt, ble, bge) provide conditional execution, and the jal instruction supports function calls.

Memory operations (lw, sw) enable data access, while I/O instructions (in, out) provide peripheral control. System instructions include reti for interrupt return and halt for program termination. Each instruction follows a consistent encoding format with fields for opcode, destination register, source registers, and immediate values.

2.3 Memory Organization

The SIMP processor uses a unified memory architecture where both instructions and data share the same 4096-word address space. Memory addresses are 12 bits wide, providing access to addresses 0x000 through 0xFFF. The processor uses word-addressed memory with 32-bit words, and all memory accesses are bounds-checked to prevent invalid operations.

The memory system supports both instruction fetches and data accesses, with proper handling of sequential instruction execution and dynamic address calculation for data operations. The assembler generates proper memory layouts with instructions placed sequentially and data placed at specified addresses using .word directives.

3. Assembler Implementation

3.1 Core Architecture and Data Structures

The assembler is implemented in a single C file (asm.c) using a three-phase approach. The main data structures include:

- **Label structure**: Stores label names and memory addresses for symbol resolution
- **WordEntry structure**: Handles .word directives for data placement
- **Lookup tables**: opcode_table[] maps instruction names to opcodes, register_table[] maps register names to numbers

The assembler uses global arrays to store word entries and labels, with counters tracking their quantities for processing.

3.2 Three-Phase Assembly Process

Phase 1 - Word Collection: The `collect_word_entries()` function scans the input file to identify all .word directives, parsing addresses and data values using the `parse_number()` function which handles both decimal and hexadecimal formats.

Phase 2 - Label Resolution: The `Getlabels()` function builds the symbol table by scanning all lines, identifying labels (marked with colons), and calculating their memory addresses. The function accounts for instruction sizes using `estimate_if_bigimm()` to determine if instructions require two memory words.

Phase 3 - Code Generation: The `assembler_second_run()` function processes each instruction line, calling `line_to_hexa()` to convert assembly instructions to machine code. This function tokenizes input lines, looks up opcodes and registers, resolves labels using `lookup_label()`, and generates proper instruction encodings.

3.3 Instruction Encoding Logic

The `line_to_hexa()` function implements the core encoding logic:

1. **Tokenization**: Parses instruction components (opcode, rd, rs, rt, immediate)
2. **Lookup**: Uses `lookup_word()` to convert mnemonics to numeric values
3. **Bigimm Detection**: Automatically determines if bigimm encoding is needed based on immediate value size or label presence
4. **Encoding**: Packs instruction fields into 32-bit words according to SIMP format
5. **Output**: Writes encoded instructions and data values to the output file

The `parse_number()` function handles immediate values, supporting decimal numbers, hexadecimal (0x prefix), and automatic label resolution through the symbol table.

4. Simulator Implementation

4.1 Core Architecture and Main Components

The simulator uses a modular design with three main files:

- **main.c**: Contains the main simulation loop `fetch_decode_execute()` and print-to-output-files functions
- **data.c**: Implements all subsystem functionality (memory, registers, I/O, disk, interrupts, monitor)
- **fe_de_ex.c**: Handles instruction fetch, decode, and execute operations

The main simulation loop coordinates all components, processing one instruction per cycle while maintaining precise timing for all subsystems.

4.2 Instruction Processing Pipeline

Fetch Stage: The `instruction_fetch()` function in fe_de_ex.c reads instructions from memory using `read_instruction_from_memory()`. The `increase_pc()` function advances the program counter by 1 for regular instructions or 2 for bigimm instructions.

Decode Stage: The `instruction_decode()` function parses instruction fields (opcode, registers, immediate values) and handles bigimm instructions by reading the second word for 32-bit immediates. The $imm register is automatically updated during this stage.

Execute Stage: The `instruction_execute()` function implements all 22 SIMP instructions using a switch statement. Each instruction performs its operation (arithmetic, logical, memory access, or control flow) and updates the program counter appropriately.

4.3 Memory and Register Management

The memory system in data.c provides unified storage through key functions:

- `memory_init()`: Initializes all memory to zero
- `load_instruction()`: Loads machine code from input file
- `read_instruction_from_memory()`: Fetches instructions for execution
- `read_data_from_memory()` and `write_data_to_memory()`: Handle data accesses

The register file implements special behavior for $zero (always returns 0) and $imm (updated automatically during decode). The `get_register()` and `set_register()` functions provide controlled access with bounds checking.

4.4 I/O System and Hardware Registers

The I/O system manages 23 hardware registers with specific bit-width limits defined in IO_REGISTER_SIZES[]. Key functions include:

- `read_from_io()` and `write_to_io()`: Handle register access with automatic tracing
- `write_hwregtrace()`: Logs all I/O operations with cycle timestamps
- `io_names_for_output()`: Maps register numbers to readable names

Each register enforces its bit-width using masking operations to ensure realistic hardware behavior.

4.5 Interrupt System Implementation

The interrupt system uses several coordinated functions:

- `load_irq2()`: Loads external interrupt events from input file
- `check_irq2()`: Triggers IRQ2 at specified cycle numbers
- `update_timer()`: Manages timer interrupts (IRQ0) when timer reaches maximum value
- `handle_all_interrupts()`: Processes all interrupt sources with proper priority

Interrupts are only processed when not already in an interrupt service routine, preventing nested interrupts. The system saves the current PC and jumps to the interrupt handler address.

4.6 Timer System Implementation

The timer system provides programmable interrupt generation with cycle-accurate timing. The `update_timer()` function increments the timer counter each cycle when enabled through the `timerenable` register. When the counter reaches the value stored in `timermax`, it triggers IRQ0 by setting `irq0status` to 1 and resets `timercurrent` to zero.

The timer operates independently of other system components and is updated every cycle regardless of the current instruction being executed. This ensures precise timing for interrupt generation and realistic system behavior.

4.7 Disk System Implementation

The disk system simulates a realistic hard drive with 128 sectors and DMA operations. The `Process_disk_command()` function manages disk operations with authentic 1024-cycle timing delays. When a disk command is issued, the system sets `diskstatus` to busy and begins the timing countdown.

The `read_data_sector()` function transfers data from the specified disk sector to memory, while `write_data_sector()` performs the reverse operation. Both functions handle 128-word transfers and include proper error checking. When operations complete, the system triggers IRQ1 and resets the disk status to ready.

4.8 Monitor System Implementation

The monitor system supports a 256×256 pixel monochrome display with 8-bit grayscale values. The `write_pixel()` function updates the framebuffer when the `monitorcmd` register is set to 1, using the address specified in `monitoraddr` and the pixel value in `monitordata`.

The system maintains an internal framebuffer array that represents the display state. Output generation functions `write_monitor_text()` and `write_yuv()` create both text and binary format files for display verification and external viewing using YUV players.

4.9 Cycle-Accurate Execution

The main simulation loop in `fetch_decode_execute()` maintains precise timing:

1. **Instruction Processing**: Handles both single-cycle and two-cycle (bigimm) instructions
2. **Clock Management**: `increase_clock()` increments the cycle counter precisely
3. **Subsystem Updates**: Calls `update_timer()`, `Process_disk_command()`, and interrupt handlers each cycle
4. **Output Generation**: Updates trace files and peripheral outputs with accurate timestamps

The simulator ensures that all operations occur in the correct cycle order, providing realistic timing behavior for educational purposes.

4.10 Main Function Implementation

The `main()` function in main.c serves as the central coordinator for the entire simulation process. It handles command-line argument validation, file management, system initialization, and the main simulation loop execution.

Command-Line Processing: The function first validates that exactly 13 command-line arguments are provided, corresponding to the input files (memin.txt, diskin.txt, irq2in.txt) and 10 output files. If the argument count is incorrect, the program terminates gracefully.

System Initialization: The main function initializes all system components in the correct order:

- `registers_init()`: Initializes the 16-register file with proper $zero and $imm handling
- `memory_init()` and `load_instruction()`: Sets up the 4096-word memory and loads the machine code
- `io_init()`: Initializes all 23 I/O registers with proper bit-width limits
- `disk_init()`: Sets up the disk system and copies input disk contents to working disk
- `init_monitor()`: Initializes the 256×256 display framebuffer
- `load_irq2()`: Loads external interrupt events from the input file

Main Simulation Loop: The function calls `fetch_decode_execute()` which implements the core simulation loop. This function continues execution until the halt instruction is encountered, processing each instruction with cycle-accurate timing while managing all subsystems.

Output File Generation: After simulation completion, the main function orchestrates the generation of all required output files:

- `write_memory_out()`: Outputs final memory state
- `write_registers_to_file()`: Outputs registers R2-R15 in hexadecimal format
- `write_monitor_text()` and `write_yuv()`: Generate monitor output in both text and binary formats
- `write_total_cycles()`: Outputs the final cycle count

The main function ensures proper resource management by handling file operations safely and providing a clean interface between the command-line environment and the simulation engine. It demonstrates good software engineering practices with clear separation of concerns and proper error handling throughout the initialization and execution phases.

5. Test Programs

5.1 Sort Program (sort.asm) - Bubble Sort Implementation

The sort program implements bubble sort for 16 integers stored at memory addresses 0x100-0x10F. The algorithm uses nested loops with careful register management:

```
outer_loop:
    add $v0, $zero, $imm, 14        # $v0 = 14 (array size - 1)
    bgt $imm, $t0, $v0, sort_complete  # if i > 14, done
    add $t1, $zero, $zero, 0        # $t1 = 0 (inner loop counter)
    sub $s2, $v0, $t0, 0            # $s2 = 14 - i (max j value)

inner_loop:
    bgt $imm, $t1, $s2, inner_complete  # if j > (14-i), exit inner loop
    add $a0, $t2, $t1, 0            # $a0 = base + j
    add $a1, $a0, $imm, 1           # $a1 = base + j + 1
    lw $s0, $a0, $zero, 0           # $s0 = array[j]
    lw $s1, $a1, $zero, 0           # $s1 = array[j+1]
    ble $imm, $s0, $s1, no_swap     # if array[j] <= array[j+1], skip
    sw $s1, $a0, $zero, 0           # array[j] = array[j+1]
    sw $s0, $a1, $zero, 0           # array[j+1] = original array[j]
```

The program uses $t0 and $t1 as loop counters, $t2 as the base address (0x100), and $s0/$s1 for temporary storage during swaps. The algorithm performs bubble sort's standard compare-and-swap operation with proper bounds checking.

5.2 Factorial Program (factorial.asm) - Recursive Implementation

The factorial program demonstrates recursive function calls with stack management:

```
factorial:
    sub $sp, $sp, $imm, 2         # Allocate 2 words on stack
    sw $ra, $sp, $imm, 1          # Save return address
    sw $a0, $sp, $imm, 0          # Save parameter n
    beq $imm, $a0, $zero, base_case # if n == 0, goto base case

    sub $a0, $a0, $imm, 1         # n = n - 1
    jal $ra, $imm, $zero, factorial # Recursive call

    lw $a0, $sp, $imm, 0          # Restore original n
    mul $v0, $v0, $a0, 0          # result = result * n

base_case:
    add $v0, $zero, $imm, 1       # return 1
```

The program uses proper MIPS calling conventions with $a0 for parameters, $v0 for return values, and $ra for return addresses. Each recursive call saves its context on the stack and properly restores it, demonstrating advanced stack management techniques.

5.3 Rectangle Program (rectangle.asm) - Graphics Implementation

The rectangle program draws a filled rectangle on the 256×256 monitor by reading coordinates from memory and using nested loops:

```
# Extract coordinates from packed format
and $a0, $s0, $imm, 0xFF            # A_x = bits 7:0
srl $a1, $s0, $imm, 8               # Shift right by 8
and $a1, $a1, $imm, 0xFF            # A_y = bits 15:8

# Calculate rectangle dimensions
sub $t0, $a2, $a0, 0                # width = C_x - A_x
add $t0, $t0, $imm, 1               # width = width + 1

# Calculate starting offset in framebuffer
add $t2, $zero, $imm, 256           # screen width
mul $v0, $a1, $t2, 0                # offset = A_y * 256
add $v0, $v0, $a0, 0                # offset = offset + A_x

row_loop:
    bge $imm, $a1, $t1, draw_complete
    col_loop:
        bge $imm, $a0, $t0, end_row
        # Calculate pixel offset and draw
        out $a2, $zero, $imm, 20        # Set monitor address
        out $s0, $zero, $imm, 21        # Set pixel color (white)
        out $s1, $zero, $imm, 22        # Execute write command
```

The program demonstrates bit manipulation for coordinate extraction, 2D-to-1D address conversion, and I/O register usage for graphics operations.

5.4 Disk Test Program (disktest.asm) - I/O Operations

The disk test program performs complex I/O operations, reading multiple sectors and processing data:

```
# Read sector example
read_sector_0:
    out $t0, $zero, $imm, 15       # Set DISKSECTOR = 0
    out $t1, $zero, $imm, 16       # Set DISKBUFFER = 0x200
    add $t2, $zero, $imm, 1        # Read command = 1
    out $t2, $zero, $imm, 14       # Execute DISKCMD = 1

wait_sector_0:
    in $t2, $zero, $imm, 17        # Check DISKSTATUS
    bne $imm, $t2, $zero, wait_sector_0  # Wait until ready

# Data processing loop
sum_loop:
    bge $imm, $s0, $s1, write_results   # Check loop condition
    # Calculate addresses for each sector
    add $t0, $zero, $imm, 0x200    # Sector 0 base
    add $a0, $t0, $s0, 0           # sector0[i] address
```

```
# Load and sum values from all sectors
lw $t0, $a0, $zero, 0          # Load sector0[i]
lw $t1, $a1, $zero, 0          # Load sector1[i]
add $t0, $t0, $t1, 0           # Sum values

# Store result
sw $t0, $t1, $zero, 0          # Store sum
```

The program demonstrates asynchronous I/O with proper status checking, memory buffer management, and data processing across multiple sectors. It showcases realistic system programming with DMA operations and interrupt handling.

5.5 Testing and Validation Strategy

In addition to writing these programs and verifying the correctness of the assembler through proper machine code generation, we used them to thoroughly validate the simulator's functionality. By inserting various inputs into these programs and observing the outputs, we confirmed that the simulator does not crash under different conditions and produces correct results across all test scenarios.

Each program tests different aspects: sort validates arithmetic and memory operations, factorial tests recursive calls and stack management, rectangle validates graphics I/O, and disktest confirms complex peripheral operations. This comprehensive testing approach ensured that both the assembler and simulator components work together seamlessly to execute complex programs correctly.

6. Final Review

This project successfully delivered a comprehensive implementation of the SIMP processor including a complete assembler and cycle-accurate simulator. The assembler provides robust translation from assembly language to machine code with automatic bigimm detection and proper symbol resolution using a three-phase approach. The simulator implements cycle-accurate execution with comprehensive interrupt handling, realistic peripheral simulation, and detailed execution tracing.

Key technical achievements include a complete interrupt system supporting three interrupt sources with proper priority handling, full peripheral simulation including programmable timer, realistic disk operations with DMA timing, and a pixel-addressable monitor system. The implementation demonstrates deep understanding of computer architecture principles while providing an educational platform for exploring processor design and assembly language programming.

The complete system successfully executes complex programs and handles all specified features according to the ISA specification, serving as both a testament to technical capabilities and a foundation for continued exploration in computer architecture and systems programming.