

ISA Project Documentation

Ahmad Foqara
id:322389982

Mohamed Foqara
id:212193916

Project Overview

This project simulates a simplified RISC processor named SIMP. The system includes an assembler written in C (ASM1.c), a cycle-accurate simulator (SIM1.c), and multiple assembly programs. These programs demonstrate core processor functionality such as recursion, memory manipulation, disk I/O, interrupt handling, and graphical operations on a simulated monitor.

1. Assembler (ASM1.c)

The assembler reads assembly source files, identifies labels, instructions, and data directives, and generates a hexadecimal instruction memory file (memin.txt). It handles the parsing and encoding of each instruction, including differentiating between short (imm8) and extended (imm32) immediates.

Features include:

- Parsing word directives to hex data
- Label resolution and back-patching
- Instruction format encoding based on opcode and operands
- Supporting branching with symbolic labels
- Distinguishing between immediate sizes (8-bit - 32-bit)

Example from ASM1.c :

```
char opcode[8];
get_opcode(line, opcode);
char rd[4];
get_rd(line, rd);
... // Parsing registers and determining immediate type
if (big || is_label) {
    IMEM[pc][0] = opcode[0]; // Instruction memory
formatting
    ...
}
```

2. Simulator (SIM1.c)

The simulator emulates the SIMP processor using instruction and hardware logic defined in C. It performs fetch-decode-execute cycles, handles interrupts,

manages I/O devices (disk, timer, LEDs, 7-seg, monitor), and writes the output to several trace and state files.

Key Features:

- Instruction decoding and execution using dispatch tables
- Full interrupt support (irq0, irq1, irq2) and reti
- Hardware register I/O through in/out instructions
- Cycle counter and PC management with support for branching
- Output file generation (trace.txt, regout.txt, monitor.txt, etc.)

Example from SIM1.c:

```
if (((IORegister[0] && IORegister[3]) || ... ) && interaptMode)  
{  
    IORegister[7] = pc; // Save current PC  
    pc = IORegister[6]; // Jump to interrupt handler  
    interaptMode = 0;  
}
```

3. Assembly Programs

factorial.asm

Implements recursive factorial. Uses jal for recursion, \$sp for stack storage, and mul for multiplication. The base case checks \$a0 == 0, then unwinds the stack and stores the result.

rectangle.asm

Draws a white rectangle on a 256x256 pixel monitor. Coordinates for the top-left and bottom-right corners are loaded from memory. The loop calculates x and y positions from pixel address and compares them with the bounds. Pixels within the bounds are written with luminance 255.

sort.asm

Implements bubble sort. Iterates through memory array starting at 0x100 and repeatedly swaps adjacent values if they are out of order. Demonstrates conditional logic and loop construction in SIMP assembly.

disktest.asm

This program reads 4 sectors from the simulated disk into buffer memory, sums their contents into a specific address, then writes the result back to disk. It

handles diskstatus, sector commands, and uses a recursive subroutine (accumulate_data) to calculate the total. Interrupt handling and polling are used to ensure disk readiness.

4. Key Functions Used in Implementation

This section explains the main C functions implemented in the assembler and simulator, describing their roles and contribution to the SIMP project.

- createMonitor(), createLeds(), createDisplay7Seg(): Write the final states of I/O devices to output files.
- createRegout(), createCycles(), createHwregtrace(): Generate additional logs and status files.

5. Functions from ASM1.c (Assembler)

- main():
Reads an input asm file, determines the line type, and processes labels, instructions, and data accordingly.
Outputs the final memin.txt.
- get_inst():
Parses an instruction line, extracts opcode, registers, and immediate values, and converts it to encoded memory format. Handles both 8-bit and 32-bit immediates.
- DecToHex():
Converts a signed/unsigned decimal number into its hexadecimal string representation, with zero-padding.
- imm_is_a_label():
Determines if the last operand in an instruction is a label, which affects whether to use 32-bit encoding.
- get_opcode(), get_rd(), get_rs(), get_rt():
These extract and encode respective fields from an instruction line into the instruction memory.

- `put_data_in_memory()`:

Processes word directives and stores initial data values into memory.

Functions from SIM1.c (Simulator)

- `main()`:

Loads memory and disk contents, reads IRQ configuration, and simulates instruction execution cycle by cycle until HALT. Handles interrupts and tracks hardware state.

- `regAssign()`:

Parses an instruction from memory and decodes the opcode, registers, and immediate fields into usable variables for execution.

- `instructionOp()`:

Dispatches instruction execution based on the decoded opcode. Calls functions like `addF()`, `mulF()`, `lwF()`, etc.

- `trace()`:

Writes the current state of all registers and the PC to ``trace.txt`` for each executed instruction.

- `addF()`, `subF()`, `mulF()`, etc...:

These implement the core logic for each instruction. They perform the corresponding operation and update the destination register and PC.

- `openMEMIN()`, `openDISKIN()`, `openIrq2n()`:

Load memory, disk, and IRQ configuration from input files.

6. Detailed Explanation of All Functions

Functions from ASM1.c

- `main()`:

Entry point. Opens the input `.asm` file, reads it line by line, determines the line type, and processes it. Produces the `memin.txt` file.

- `removeComment()`:

Removes any comments (marked by `#`) from a line of assembly code.

- `lineType()`:

Classifies a line into label, instruction, data (word), or empty.

- `get_label()`: Extracts a label from a label-only line and maps it to its PC.

- `get_label_and_inst()`: Handles lines with both a label and an instruction.
- `get_inst()`: Parses instruction components (opcode, rd, rs, rt, imm) and builds the corresponding memory format.
- `imm_is_a_label()`: Checks if the immediate value is a label.
- `get_imm()`: Extracts and formats an immediate operand to hexadecimal.
- `DecToHex()`: Converts a decimal number (signed or unsigned) to hexadecimal string.
- `get_opcode()`: Detects the opcode mnemonic and converts it to its hexadecimal code.
- `get_rd()`, `get_rs()`, `get_rt()`: Extract register numbers from a line and encode them.
- `put_data_in_memory()`: Processes .word lines to store data at specific memory addresses.
- `fill_pc_of_labels()`: Assigns memory addresses (PC values) to labels for later replacement.
- `write_memin()`: Writes the final instruction memory (IMEM) contents to `memin.txt`.
- `put_imm_in_arr()`: Stores labels used as immediates for future resolution.
- `from_char_hex_to_deci()`: Converts a single hex char to decimal.
- `from_char_to_deci()`: Converts a single digit character to its numeric value.
- `from_char_hex_to_int()` / `from_char_deci_to_int()`: Convert string representations of numbers to integers.
- `get_bigimm()`: Determines if an instruction uses 32-bit immediate (based on size or label).
- `get_imm8()` / `get_imm32()`: Parse and validate imm8/imm32 values from instruction lines.

Functions from SIM1.c

- `main()`:
Initializes memory and hardware, loads input files, and runs the simulation loop until HALT.
- `openMEMIN()`, `openDISKIN()`, `openIrq2n()`:
Read initial memory, disk, and interrupt input configurations.
- `openFile()`:
Creates/empties a file to prepare for output.
- `regAssign()`:
Decodes a raw instruction into opcode, register numbers, and immediate values.
- `trace()`: Appends a line to `trace.txt` showing the current cycle, instruction, and register values.
- `instructionOp()`: Dispatches instruction execution to the appropriate operation handler based on opcode.
- `createHwregtrace()`, `createLeds()`, `createDisplay7Seg()`, `createMonitor()`, `createCycles()`, `createRegout()`, `createDISKOUT()`, `createMemout()`:
Each creates a specific output trace or status file.
- `addF()`, `subF()`, `mulF()`, `andF()`, `orF()`, `xorF()`, `sllF()`, `sraF()`, `srlF()`:
Implement ALU instructions (arithmetic/logic).
- `beqF()`, `bneF()`, `bltF()`, `bgtF()`, `bleF()`, `bgeF()`:
Implement conditional branching instructions.
- `jalF()`:
Implements jump-and-link: stores return address and updates PC.
- `lwF()`, `swF()`:
Implement load/store memory instructions.
- `retiF()`:
Implements return from interrupt, restoring PC from IRQ return register.

- `inF()`, `outF()`:
Read from/write to IO registers.

7. Summary

This project provides a full-stack simulation of a custom RISC processor. From writing low-level assembly to implementing a compiler and simulator, each part demonstrates deep understanding of computer architecture concepts. The four programs test memory operations, recursion, graphics, sorting algorithms, and disk I/O.