SIMP Project Documentation

This PDF explains the code structure of the simulator and the assembler programs, which are part of the SIMP project.
ID1: 209609783
ID2: 209344555
ID3: 209668425

Simulator

The simulator simulates the SIMP CPU fetch-decode-execute cycle, along with the peripherals (timer, disk, monitor) and interrupt handling. The simulator takes as input initial memory, disk, and irq2 state files, and produces output files for memory state, disk state, monitor display, and execution traces.

File Structure

- `main.c` : Main entry point. Parses arguments, initializes state, runs the main loop, and initiates logging to files and cleanup.
- `simulator.h` : Defines system constants (memory size, disk sectors, register indices) and core data structures (`SimState` , `Instruction`).
- `core.c` / `core.h` : Implements the CPU logic: instruction fetch, decode, execute (opcode switch), and interrupt handling.
- `memory.c` / `memory.h` : Manages memory access (read/write).
- `disk.c` / `disk.h` : Simulates hard disk operations with delay logic.
- `timer.c` / `timer.h` : Manages the hardware timer and interrupt generation.
- `monitor.c` / `monitor.h` : Handles writing to the monitor buffer.
- `files.c` / `files.h` : Handles file I/O for loading initial states (memory, disk, irq), logging trace as the simulator runs (register, hwreg, leds, 7-segment traces), and writing final states (memory, registers, disk, cycles, monitor).

Data Structures

- `SimState` : The central structure holding the entire simulator state:
  - `registers[16]` : General purpose registers.
  - `io_registers[23]` : IO registers.
  - `memory[4096]` : Memory (20-bit words).
  - `disk` : Disk storage.
  - `pc` : Program counter.
  - `monitor` : Monitor buffer.
  - `irq2_cycles` : Sorted array of scheduled external interrupts.

    - `word` : Uses a `uint32` as the storage type for all 20-bit values. Arithmetic is done in 32 bits, and results are masked back to 20 bits on writing back to memory.
- `Instruction` : Represents a decoded instruction with fields `opcode` , `rd` , `rs` , `rt` , and `is_imm` .
    - `opcode` : Numeric opcode value extracted from the 20-bit word.
    - `rd` , `rs` , `rt` : Register indices extracted from the 20-bit word. (including `$zero` and `$imm`).
    - `is_imm` : Boolean indicating whether the instruction takes the next word as an immediate (triggered when any operand is `$imm`).
    - Decode logic keeps the raw opcode and register indices only. Sign-extension and immediate fetching are handled in execution.

Implementation Details

- **Simulator Main Flow**: `main` initializes state, loads input files, and then runs a `while` loop, where each iteration executes `core_step` (fetch-decode-execute). After the loop ends (on halt), file outputs are written.
- **Instruction Fetching**: The `core_step` function fetches the instruction at the current `pc` from memory using `memory_read` .
- **Instruction Decoding**: The `decode_instruction` function extracts the opcode and register indices from the 20-bit instruction word. It sets `is_imm` if any operand is `$imm` .
- **Immediate Handling**: If `is_imm` is set, the simulator reads the next word (`pc + 1`) as the immediate value, sign-extends it to 20 bits, and saves the value to the `$imm` register. The `pc` then is increased by 2 to skip the immediate.
- **Instruction Execution**: A `switch` statement on `opcode` implements each instruction's behavior, writing to memory or registers as needed. Branch instructions update `pc` directly if needed (otherwise `pc` is incremented by default).
- **Interrupt Handling**: After the execution stage, pending interrupts are checked. If any are pending and enabled, the current `pc` is saved to `IOREG_IRQRETURN` , and `pc` is set to `IOREG_IRQHANDLER` .
- **Cycle Tracking**: At the end of each cycle, each peripheral is 'tick'ed (more info below), the `IOREG_CLKS` register is incremented, and `total_cycles` is incremented.
- **Memory Implementation**: Memory is a fixed array of 4096 20-bit `word` s (defined above). Reads and writes done using index access, and values are masked to 20 bits to emulate word width.
- **Disk Implementation**: Disk storage is a fixed array of 128 sectors, each sector containing 128 words. Read/write commands are initiated by writing to `IOREG_DISKCMD` with `IOREG_DISKSECTOR` and `IOREG_DISKBUFFER` already set. Then, the `disk_busy` flag is set, `disk_timer` is set to the disk delay, and the `disk_tick` function decreasese the timer at each simulation cycle. When the timer reaches zero, the operation is executed (data copied between memory and disk), the `disk_busy` flag is cleared, and the IRQ1 interrupt is raised.

- **Timer Implementation**: When enabled, `timer_tick` increments `TIMERCURRENT` at each clock cycle and compares against `TIMERMAX`. If they match, the timer is reset to zero, and the IRQ0 interrupt is raised.
- **IRQ2 Implementation**: Before starting the simulation, irq2 input cycles are loaded into a sorted array `irq2_cycles` and the count `irq2_count`, and `irq2_index` is initialized to zero. At each cycle, the simulator checks if the current cycle count matches the next scheduled irq2 cycle (`irq2_cycles[irq2_index]`). If so, it raises the IRQ2 interrupt, and increments `irq2_index`. `irq2_count` is used to check if all irq2 events were handled.
- **Monitor**: Writes to the monitor data/addr registers update the monitor buffer in memory. When the simulator halts, the monitor buffer is written to the `txt/yuv` output files.
- **IO Register Operations**: The `in` / `out` instructions copies the relevant data from/to the IO register, and access is logged. In the `out` command, a `switch` statement handles specific behavior for the LEDs, 7-segment display, and disk, and monitor.
- **File I/O**: The `files.c` module handles all file reading/writing. Input files are read at initialization to set up memory, disk, and irq2 state in the `SimState` struct. During simulation, trace files are written at each cycle for registers, hwregs, leds, and 7-segment display. At the end of simulation, final state files are written for memory, registers, disk, total cycles, and monitor output.

Assembler

The assembler's code is all contained within a single `main.c` file.

File Structure

- `main.c`: Entry point to the assembler.
  - Parses command-line arguments (assembly input file path and output `memin.txt` path).
  - Manages file opening, execution of the assembly passes, output generation, and cleanup.

Data Structures

- **Symbol Table**
  - Stores all labels defined in the program along with their memory addresses.
  - Supports adding new labels, detecting duplicate label definitions, and resolving label addresses during the second pass.
- **Memory Image**
  - Uses a fixed size memory array representing the simulator memory.
  - Ensures the output contains exactly one word per memory address.
- **Lookup Tables**
  - Maps opcode text to numeric opcode values.

- Maps register names to register indices.

Implementation Details

Two-pass assembly process

- **Pass 1 – Symbol collection**
  - Reads the assembly source line by line.
  - Removes and validates comments.
  - Identifies label definitions and stores them in a symbol table.
  - Tracks the program counter (PC) to compute label addresses.
  - Determines instruction size, including extra memory words for immediate values.
- **Pass 2 – Code generation**
  - Re-reads the assembly source.
  - Assembles instructions into a memory array of fixed size.
  - Resolves label references used as immediates.
  - Applies `.word` lines to write values into memory.
  - Produces the final memory file.

Parsing and Validation

- **Tokenization**: Input lines are split into tokens, treating commas as separators.
- **Label validation**: Labels are validated to make sure they follow the required syntax (case sensitive).
- **Numeric parsing**: Supports decimal and hexadecimal values, masks values to the 20-bit word width.
- **Comment handling**: Ensures comments (`#`) can only appear after the required operands for each line type.

Instruction Handling

- Each opcode is mapped to numeric opcode values using the lookup table.
- Each register name is mapped to its index using the lookup table.
- Instructions are encoded into 20-bit words.
- Instructions that have an immediate value are encoded into 2 20-bit words (the instruction, and the immediate value).

Output

Produces a `memin.txt` file containing:

- Exactly `MEM_SIZE` lines
- One 20-bit word per line
- Each word printed as a zero-padded hexadecimal value

Test Programs

The 4 test programs `binom`, `disktest`, `sort`, `triangle` are contained in the respective folder, each containing all the input files (`asm` files) and the output files.