CSC3050 Assignment 2 - Pipeline Simulator

The usage of simulator

This project report delves into the creation and functionality of a RISC-V Instruction Set Simulator, used to emulate the RISC-V architecture. This simulator, developed in C++, is designed to interpret and execute RISC-V assembly language instructions.

The environment, how to compile and run the project

The whole project is written on Mac with arm chips, and has been tested with released tests on both Ubuntu ans MacOS. For compiling and running, please first run the build the `build.sh` file and then run the `run-test-release.sh` for released test cases. If single run is needed, then please first change directory to build, `make` and then run with command like `./Simulator ../test-release/add.riscv -history` or without `-history` .

Implementation Overview

The simulator uses RISC-V processor architecture, including memory management, instruction decoding, execution, and pipeline management. Its implementation revolves around the simulation of a basic five-stage instruction pipeline characteristic of RISC-V processors: fetch, decode, execute, memory access, and write-back.

Memory Management and Initialization

First, we use a memory management module responsible for simulating the processor's memory space. It supports read and write operations across byte-addressable memory, accommodating different operation sizes with appropriate sign extension where necessary. The simulator initiates with a stack configuration, setting the stack base address and maximum size, ensuring an environment conducive to stack operations performed by the assembly programs being simulated.

```
Simulator::Simulator(MemoryManager* memory) {
    this ->pc = 0;
    this ->memory = memory;
    // Initialize all registers to 0.
}

Simulator::~Simulator() {}

void Simulator::init_stack(uint32_t base_addr, uint32_t max_size) {
    this->reg[2] = base_addr;
    this->max_stack_size = max_size;
    this->stack_base = base_addr;
    uint32_t current_addr = base_addr;
    uint32_t mem_hold = base_addr - max_size;
    while (current_addr > mem_hold) {
        if (memory->is_page_exit(current_addr)) {
            memory->set_byte(current_addr, 0);
        } else{
            memory->add_page(current_addr);
            memory->set_byte(current_addr, 0);
        }
        current_addr--;
    }
}
```

Pipeline Stages and Hazard Management

See more details in the corresponding stage with detialed comments.

- **Fetch Stage**: Instructions are fetched from memory based on the program counter (`pc`). This stage sets the stage for the instruction's journey through the pipeline.
- **Decode Stage**: Fetched instructions are decoded to determine their type, operands, and the specific operations to be performed. This stage is crucial for interpreting the instruction set and preparing for execution.
- **Execute Stage**: The core of the simulation where arithmetic operations, logical operations, and branch calculations are performed based on the decoded instruction.
- **Memory Access Stage**: Facilitates read and write operations to memory for load and store instructions, respectively. It also implements forwarding mechanisms to resolve data hazards.
- **Write-Back Stage**: Completes the instruction cycle by writing the results back to the register file. Forwarding from this stage ensures that subsequent instructions have immediate access to

updated data, addressing potential data hazards.

```
void Simulator::write_back() {
    if (!m_reg.write_reg || m_reg.dest_reg == 0) return;
    this->reg[this->m_reg.dest_reg] = this->m_reg.out;
}
```

Hazard Handling

The simulator incorporates detailed strategies for handling data, control, and memory hazards:

- **Data Hazards**: Data hazards arise when an instruction depends on the result of a previous instruction that has yet to complete its execution. Utilizes forwarding to resolve dependencies between instructions, allowing for the immediate use of execution results by subsequent instructions, thereby reducing stalls.
- **Data Hazards Realization**:
-   i. Determine the Necessity of Forwarding (isWriteNeeded): Checks if the current instruction writes to a register and that the register is not x0.
-   ii. Evaluate Forwarding Conditions (shouldForward): Determines if forwarding is applicable based on the instruction type, specifically excluding load instructions due to their memory dependency.
-   iii. Execute Forwarding: Forwards the execution result to the next instruction's source operands if they match the current instruction's destination register, subject to conditions that prevent redundant forwarding.

```
if (isWriteNeeded) {
        bool shouldForward = !(aluCal == LB || aluCal == LH || aluCal == LW || aluCal =
        // Forward data to the first source operand of the next instruction (rs1)
        if (shouldForward && this ->d_reg_new.rs1 == dest_reg) {
            // Ensure not repeatedly forward to the same destination registers
            if (!this ->execute_write_back || (this ->execute_wb_reg != dest_reg)) {
                updateExecuteWB(execute_write_back, execute_wb_reg, dest_reg);
                this ->d_reg_new.op1 = result;
                this ->history.data_hazard_count++;
            }
        }
```

- **Control Hazards**: Control hazards occur when the pipeline executes instructions that follow a branch/jump instruction before the branch/jump decision is resolved. This resolving method addresses such hazards by updating the PC and clearing the pipeline of instructions that should not be executed. Employs bubbles and a basic branch prediction mechanism to manage the flow of instructions through the pipeline, ensuring accurate execution PCs.

- **Control Hazards Realization**:
- i. The function `control_hazard_update(uint32_t decodePC)` updates the simulator state to handle a control hazard, typically arising from branch or jump instructions.
- ii. The function sets the main program counter (pc) to decodePC, changing the execution flow to the new address. It inserts bubbles into the fetch (f_reg_new) and decode (d_reg_new) pipeline stages to prevent the execution of instructions that were fetched or decoded before the branch/jump decision was made. This is essential to maintain correct program executions, as these instructions should be "cancelled" and not executed.

```
if (branch){
        control_hazard_update(decodePC);
        this ->history.control_hazard_count++;
    }
```

```
void Simulator::control_hazard_update(uint32_t decodePC) {
    this->pc = decodePC;
    this->f_reg_new.bubble = this->d_reg_new.bubble = true;
}
```

- **Memory Hazards**: Initiates pipeline control actions to handle memory hazards. Memory hazards occur when an instruction depends on the result of a load operation from a preceding instruction. Since load operations may take more than one cycle to complete, subsequent instructions that rely on loaded data must wait.
- **Memory Hazards Realization**:
- i. Introduces stalls in the fetch (f_reg_new) and decode (d_reg_new) stages by setting their stall counters. This effectively pauses these stages for two cycles, allowing time for the load operation to complete and
  for the loaded data to be available for subsequent instructions.
- ii. Inserts a bubble into the execute (e_reg_new) stage, indicating that no instruction should be executed in the next cycle. This helps in aligning the pipeline stages correctly after the stall is introduced.

Understanding of memory management, ELF file loader

Memory management

1. First, the memory manager initializes a two-dimensional array to represent memory, with each cell potentially holding a page of memory. This design mimics a simplified version of virtual memory, where the first level of the array could represent a page table pointing to actual memory pages in the second level. The `access_latency` variable is like the real-world delays in memory access.
2. The constructor and destructor ensure that memory is allocated and deallocated responsibly, preventing memory leaks, which is a crucial aspect when simulating complex programs that might run for thousands or even millions of cycles. The `add_page` function dynamically allocates memory only when needed, which is efficient and reflects the lazy allocation strategies seen in operating systems.
3. The memory manager also provides functions for translating an address into its constituent parts: the first and second level indices in the memory array, and the offset within a page. This scheme allows for a flexible and efficient way to simulate a large memory space without the need for physical memory.
4. Reading and writing operations, such as `get_byte` and `set_byte` , incorporate access latency, simulating the time it takes to retrieve or store information in memory.

ELF file loader

An ELF file is a structured way to combine the code and data that make up a program so that a system knows how to run it.
The ELF file loader's job is pretty straightforward but critical: it reads through the ELF file, grabs the necessary bits of code and data, and places them into the simulator's memory where they belong.
The files like `elfio.hpp` , `elfio_dump.hpp` , and others are instrumental in this process. They contain definitions and implementations for parsing ELF files, reading sections and segments, and potentially other handling dynamic linking or relocation actions, depending on the program's complexity.

The history information of simulator

The simulator maintains a history of its operations, tracking executed instructions, cycle counts, and instances of various hazards. If you want to print out the history, you should input `-history` in the end and run it. (E.g. `./Simulator ../test-release/add.riscv -history` For this insturction, you need to run it under the build directory.)
Example of history:

```
---------History---------
Cycle count = 1422
Inst count = 1006
Stall cycle count = 284
Data hazard count = 671
Control hazard count = 196
Memory hazard count = 71
----------Exit-----------
```