Introduction

Only CSE 5302 and ECE 5402 students are required to complete this assignment.

In this assignment, you will implement a 4-stage pipelined riscv-uconn simulator with a scoreboard algorithm.

First, ensure that your git repository is up-to-date by executing git pull within the cse4302 directory. This will create a new pa4 directory in the repository root that contains the materials for this programming assignment. The directory and file structure of pa4 is the same as pa3. As before, you may only modify sim_stages.c.

The objective of this programming assignment is to modify sim_stages.c to implement a 4-stage pipelined riscv-uconn simulator with a scoreboard algorithm. The following sections provide a detailed description of the simulator implementation for this assignment. Unlike before, this assignment does not directly build upon previous programming assignments.

To receive full credit for this assignment, your simulator implementation must be fully functional and correct for the unit tests of PA3 (included in the unittests directory). Once you have completed this programming assignment, submit your sim_stages.c file via HuskyCT by the posted deadline.
1 Simulator Structure

The structure of this simulator is most similar to that of PA1 and PA2 with the following changes:

1. Scoreboard data structures and control variables have been added to sim_core.h. The following sections provide more details regarding these additions.

2. Changes to the pipeline stages are as follows:
   - There are now four (4) pipeline stages that correspond to those of a traditional scoreboard processor: Issue, Read Operands, Execute, and Write Result. As before, the stage implementations are in sim_stages.c.
   - The inputs and output of each stage are no longer chained together as in previous assignments. Instead, each stage directly operates on the global scoreboard state.
   - As was the case with the 5-stage pipelined simulator, the stages execute in reverse order.

3. The simulator core does not track committed instructions (like in PA2). Instead, committed instructions must be handled in the Write Result stage.

4. The pipe trace now shows scoreboard information.

2 4-Stage Pipelined Simulator with Scoreboard Algorithm

The objective of this programming assignment is to implement the scoreboard algorithm to dynamically exploit Instruction Level Parallelism (ILP) in a multi-cycle 4-stage pipeline comprising an Issue, Read Operands, Execute, and Write Result stage.

As this simulator implements a multi-cycle pipeline, instructions will require a different number of cycles in the execute stage according to their type. The LATENCY_MEMORY, LATENCY_BRANCH, and LATENCY_OTHER constants defined sim_core.c determine the instruction latency for memory, branch, and all other instructions respectively.

2.1 Scoreboard Data Structures and Control Variables

Data Structures

The scoreboard is accessed via the scoreboard[] array and consists of one hundred (100) ScoreboardEntry structures as indicated by the SCOREBOARD_ENTRIES macro. Both the ScoreboardEntry structure and SCOREBOARD_ENTRIES macro are defined in sim_core.h.

Each scoreboard entry holds a single instruction and stores information regarding the type, stage, and hazard of that instruction as well as its execution state throughout the pipeline. Each scoreboard entry has its own execution unit, so the number of execution units is equal to the number of scoreboard entries.

Each scoreboard entry contains the following:
### ScoreboardEntry Structure

<table>
<thead>
<tr>
<th>Struct Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>valid</td>
<td>Flag indicating whether entry is valid</td>
</tr>
<tr>
<td>instruction_number</td>
<td>Unique number identifying instruction</td>
</tr>
<tr>
<td>stage</td>
<td>Current pipeline stage of entry</td>
</tr>
<tr>
<td>operation</td>
<td>Opcode, or operation, of entry</td>
</tr>
<tr>
<td>f3</td>
<td>Function 3 bits of entry</td>
</tr>
<tr>
<td>f7</td>
<td>Function 7 bits of entry</td>
</tr>
<tr>
<td>hazard</td>
<td>Current hazard of entry, if applicable</td>
</tr>
<tr>
<td>src_reg_1</td>
<td>Source operand 1 (rs1) register index, if applicable</td>
</tr>
<tr>
<td>src_reg_2</td>
<td>Source operand 2 (rs2) register index, if applicable</td>
</tr>
<tr>
<td>dest_reg</td>
<td>Destination register (rd), if applicable</td>
</tr>
<tr>
<td>src_reg_1_data</td>
<td>Source operand 1 data, if applicable</td>
</tr>
<tr>
<td>src_reg_2_data</td>
<td>Source operand 2 data, if applicable</td>
</tr>
<tr>
<td>scb_1</td>
<td>Scoreboard entry corresponding to source operand 1, if applicable</td>
</tr>
<tr>
<td>scb_2</td>
<td>Scoreboard entry corresponding to source operand 2, if applicable</td>
</tr>
<tr>
<td>scb_1_ready</td>
<td>Flag indicating if source operand 1 is available</td>
</tr>
<tr>
<td>scb_2_ready</td>
<td>Flag indicating if source operand 2 is available</td>
</tr>
<tr>
<td>imm</td>
<td>Immediate value</td>
</tr>
<tr>
<td>target_address</td>
<td>PC target for branch and jump instructions</td>
</tr>
<tr>
<td>cycles</td>
<td>Number of cycles spent in execute stage</td>
</tr>
<tr>
<td>latency</td>
<td>Instruction latency</td>
</tr>
<tr>
<td>result</td>
<td>Operation result</td>
</tr>
</tbody>
</table>

**Figure 1: Fields of the ScoreboardEntry structure.**

The `stage` and `hazard` fields are represented respectively by the `Stage` and `Hazard` enumerations defined in `sim_core.h`.

A `register_result[]` array has also been added to store the index of the scoreboard entry that will eventually write to the corresponding register. The `register_result[]` array has thirty two (32) entries corresponding to the `riscv-uconn` registers. If a register is not pending a write from a scoreboard entry, it is set to -1.

#### 2.1.1 Control Variables

New in this assignment is the `br_taken_instruction_number` control variable. This variable holds the instruction number of the most recent branch or jump instruction that changed the control flow of the program. This is used to identify and flush mis-predicted instructions in the pipeline.

The `br_taken` control variable returns from previous assignments, except now it is used by both B-type and JAL/JALR instructions to indicate a change in control flow.

The `committed_instructions` global variable is incremented by one (1) after an instruction completes its write result stage.

### 3 Implementation

#### 3.1 Issue Stage

The issue stage is responsible for both fetching and issuing instructions.

If `br_taken` is not asserted, the next instruction is fetched from instruction memory at address `pc / 4`. If `br_taken` is asserted, the issue stage de-asserts `br_taken` and stalls by returning without any further processing.
After fetching, the instruction is decoded into a temporary State structure. The instruction is then checked for WAW hazards with any active instruction in the scoreboard. This is done by checking that the register_result[] array element corresponding to the current instruction’s destination register is -1, which indicates no active scoreboard entry will write to that register. If a hazard is detected, then the issue stage must stall and re-execute next cycle.

If no WAW hazard is detected, the scoreboard is searched for an available entry. A scoreboard entry is available if its valid flag is 0. If an available entry is found, the instruction is assigned to the entry. If no such entry is found, the issue stage stalls due to the structural hazard and re-executes next cycle. Every field of the scoreboard entry should be reset before it is populated.

To populate the scoreboard entry, the valid flag is set to 1 and the stage field is set to STAGE_ISSUE. The cycles field is set to 0 and the latency field is set according to the instruction type. The instruction_number is set to the global cycles variable to uniquely identify the instruction. The operation, f3, and f7 fields must be set according to the instruction’s opcode, funct3, and funct7 bits.

The src_reg_1 and src_reg_2 fields must be set to the register index of rs1 and rs2 respectively if those registers shall be read, or set to -1 otherwise. If the register_result[] array indicates that the current rs1 and/or rs2 registers are pending a write from a scoreboard entry (that is, the corresponding element is not -1), the scb_1 and scb_2 fields are set to the index of the scoreboard entry that will write to the rs1 and/or rs2 registers respectively. If the rs1 and/or rs2 registers are not read for an instruction, or if the register is not pending a write from another scoreboard entry, scb_1 and scb_2 are set to -1. If the rs1 and/or rs2 registers are already available, scb_1_ready and/or scb_2_ready are set to 1 respectively.

If applicable, the dest_reg field is set to the register index that the instruction will write to, or set to -1 otherwise (Note: remember that JAL/JALR instructions set their rd field to zero when they are not going to write back to a register). Moreover, instructions that write to a register must also set the corresponding register_result[] array element to the scoreboard entry (the current entry) index that will write to that register.

The imm and target_address fields must also be populated for the relevant instructions.

Finally, the simulator implements a static not-taken branch predictor that assumes the program counter increments sequentially until a branch or jump modifies the control flow. Consequently, the issue stage must increment the program counter by 4 after an instruction has been assigned to a scoreboard entry and before returning.

### 3.2 Read Operands Stage

The read operands stage loops through all the entries in the scoreboard and processes the entries that have completed the issue stage. This is done by checking the valid and stage fields of each scoreboard entry.

If br_taken is asserted, mis-predicted instructions must be flushed. In this case, every scoreboard entry with an instruction number greater than the br_taken_instruction_number global control variable must be flushed by clearing its valid flag and corresponding register_result[] array entry if applicable.

If there is no mis-prediction, and if applicable, the scb_1_ready and/or scb_2_ready fields of each active scoreboard entry are checked to determine if the instruction’s operands are ready to be read in the current cycle. If the operands are ready, the rs1 and/or rs2 registers are read into src_reg_1_data and/or src_reg_2_data respectively. If the scoreboard entry is waiting for one or more operands, it is stalled in this stage and its hazard field is set to HAZARD_RAW (Note: Remember to reset the hazard field when the RAW hazard is cleared. An instruction is allowed to read registers when RAW dependency is cleared).

After an entry has read its operands, its stage field is set to STAGE_READ_OPERANDS.
Memory Dependency Handling

Memory addresses for LW and SW are not computed until the execute stage. Consequently, a younger LW may have its address available before an older SW is still waiting for its address. If the LW and SW were to be done at the same address, then the LW will have incorrect value. To mitigate this issue, we take a conservative approach for this assignment: LW instructions will stall in the read operands stage until all previous SW instructions have completed.

3.3 Execute Stage

The execute stage loops through all the entries in the scoreboard and processes the entries that have completed the read operands stage. Mis-predicted instructions are also flushed like in the previous stage.

Each active scoreboard entry tracks its execution cycles for its instruction in this stage by incrementing the cycles field. The LATENCY_MEMORY, LATENCY_BRANCH, and LATENCY_OTHER constants determine the number of cycles memory, branch/jump, and all other instructions take in the execute stage. If an instruction requires multiple cycles to execute, it is stalled in this stage accordingly. In the last cycle of execution for an instruction, the relevant operation is performed to compute the result and stored in the result field, and stage field is set to STAGE_EXECUTE.

3.4 Write Result Stage

The write back stage loops through all the entries in the scoreboard and processes the entries that have completed the execute stage.

Each active scoreboard entry is checked for a WAR hazard with an older instruction in an active scoreboard entry. If a WAR hazard is detected, the scoreboard entry stalls in the write result stage. Otherwise, the scoreboard entry’s instruction is allowed to write its result by either writing to the destination register or writing to memory. Moreover, the scoreboard entry also updates scb_1_ready and/or scb_2_ready of any instruction that is waiting for the current instruction to complete. Do note that multiple instructions may be ready to write their result in a given cycle, and the machine imposes no restrictions on the number of instructions that may write-back in a single cycle. In other words, the write result stage attempts to commit as many instruction as possible in a given cycle.

After an instruction has completed, its scoreboard entry is de-allocated by de-asserting the valid flag of the entry and the corresponding register_result[] array element is set to -1 if applicable. Finally, the committed_instructions global variable is also incremented by 1 to keep count of committed instructions.

Control Flow

When a scoreboard entry is ready to write-back an instruction that changes control flow (JAL, JALR, and B-types), the write result stage (1) asserts the br_taken signal, (2) updates the br_taken_instruction_-number variable, and (3) updates the program counter.