Lab 4 Re-Order Buffer and Tomasulo's Algorithm

> Due: 2023/12/27 (Wed) 23:59:59

This lab takes 4 points. After previous labs, you should be very familiar with the simulator now. Being different from the implementation of Scoreboard in Lab 1, now you are required to achieve two goals:

1. Resolve data hazards by register renaming with Re-Order Buffer (ROB) and Tomasulo's Algorithm;
2. Issue instructions in out-of-order fashion.

> Lab 4 can be based on a fresh simulator. You can choose to base on floating point arithmetic operations. You can also choose to disable the branch prediction. If you want to keep branch prediction enabled, please use our patch on Piazza and follow the right routine in Tomasulo's algorithm for branch prediction handling.

After you finish this lab, you shall have an out-of-order issue, out-of-order completion yet in-order commit pipeline.

> Note: The design this lab refers to is different from what we have learnt in lectures since there are various implementations for Tomasulo's algorithm. Please refer to the textbook (Computer Architecture: A Quantitative Approach, especially Chapter 3.5 and 3.6) for more details.

1. Introduction to ROB and Tomasulo's Algorithm

The reorder buffer (ROB) holds the result of an instruction between the time when the operation associated with the instruction completes and the time when the instruction commits. The ROB therefore is a source of operands for instructions. However, the register file is not updated until the instruction commits (and we know definitively that the instruction should execute); thus, the ROB supplies operands in the interval between completion of instruction execution and instruction commit.

1.1. Re-Order Buffer

Each entry in the ROB contains at least four important fields: the instruction type, the destination field, the value field, and the ready field.

- The instruction type field indicates whether the instruction is a branch (and has no destination result), a store (which has a memory address destination), or a register operation (ALU operation or load, which has a register destination).
- The destination field supplies the register index (for loads and ALU operations) or the memory address (for stores) where the instruction result should be written.
- The value field is used to hold the value of the instruction result until the instruction commits.
- The ready field indicates that the instruction has completed execution, and the value is ready.

1.2. Reservation Station

Reservation stations buffer the operands of instructions waiting to be issued and **are associated with the functional units**.

ROB needs a place to buffer operations (and operands) between the time when the instructions issue and the time when they begin execution. This function is provided by the **Reservation Station**. Because every instruction has a position in the ROB until it commits, we tag a result using the ROB entry index. This tagging requires that the ROB entry assigned for an instruction must be tracked in the reservation station.

Each reservation station has seven fields:

1. `Op`

   The operation to perform on source operands `S1` and `S2`.

2. `Qj`, `Qk`

   The reservation stations that will produce the corresponding source operand; a value of zero indicates that the source operand is already available in `Vj` or `Vk`, or is unnecessary.

3. `Vj`, `Vk`

   The value of the source operands. Note that only one of the V fields or the Q field is valid for each operand. For loads, the Vk field is used to hold the offset field.

4. `A`

   Used to hold information for the memory address calculation for a load or store. Initially, the immediate field of the instruction is stored here; after the address calculation, the effective address is stored here.

5. `Busy`

   Indicates that this reservation station and its accompanying functional unit are occupied.

1.3. Other Enhancements

The register file is enhanced with a field, `Qi`. This is the index of the reservation station that contains the operation whose result should be stored into this register. If the value of `Qi` is blank (or 0), no currently active instruction is computing a result destined for this register, meaning that the value is simply the register contents.

The load and store operations require buffers to save the uncommitted load/store temporarily. In ROB-enhanced Tomasulo's algorithm, this has been merged to ROB. Therefore, each ROB entry has a field, A, which holds the result of the address once the address computing step of execution has been completed.

1.4. Instruction Execution

Here are the four steps involved in instruction execution:

1. Issue

   Get an instruction from the instruction queue. Issue the instruction if there is an empty reservation station entry and an empty slot in the ROB. Send the operands to the reservation station if they are available in either the registers or the ROB. Update the control entries to indicate the buffers are in use.

   The index of the ROB entry allocated for the result is also sent to the reservation station so that the index can be used to tag the result when it is used or written to ROB again.

   If either all reservations are full or the ROB is full, then the instruction issue is stalled until both have available entries.

2. Execute

   If one or more of the operands is not yet available, monitor the reservation station while waiting for the register to be computed. This step checks for RAW hazards.

   When both operands are available at a reservation station, Execute stage executes the operation. Instructions may take multiple clock cycles in these stages.

3. Write Back

   When the result is available, write it into the ROB, with the ROB tag sent when the instruction issued, as well as to any reservation station waiting for this result. Mark the entry in the reservation station as available.

4. Commit

   This is the final stage of completing an instruction, after which only its result graduates.

   There are three different sequences of actions at commit depending on whether the committing instruction is a branch with an incorrect prediction, a store, or any other instruction (normal commit).

   - The normal commit case occurs when an instruction reaches the head of the ROB and its result is present in the buffer; at this point, the processor updates the register with the result and removes the instruction from the ROB.
   - Committing a store is similar except that memory is updated rather than a result register.
   - When a branch with incorrect prediction reaches the head of the ROB, it indicates that the speculation was wrong. The ROB is flushed and execution is restarted at the correct successor of the branch. If the branch was correctly predicted, the branch is finished.

Note that special actions are required for store instructions. If the value to be stored is available, it is written into the Value field of the ROB entry for the store. If the value to be stored is not available yet, ROB must be monitored until that value is broadcast, at which time the Value field of the ROB entry of the store is updated. For simplicity we assume that this occurs during the Memory Access stage of a store.

Once an instruction commits, its entry in the ROB is reclaimed, and the register or memory destination is updated, eliminating the need for the ROB entry. If the ROB fills, we simply stop issuing instructions until an entry is made free.

1.5. Load/Store Unit and Buffers

The Memory Access stage no longer acts as a stand-alone executing stage writing directly to the memory. All status change to the system must be completed at the commit stage. The results of store will be buffered to an ROB entry that will not be ready to memory unless the store commits.

2. Data Structures of ROB and Tomasulo's Algorithm

In ROB-enhanced Tomasulo's algorithm, we will have the following structures:

- Re-Order Buffer (`ROB`)
- Register status data structure (`RegisterStat`)
- Reservation station structure (`RS`).

2.1. Re-Order Buffer

Entry	Ready	Busy	Instruction	Destination	Value	Address
The index of the ROB entry	Is the result ready to use?	Is the instruction under execution?	Instruction type	The destination of the instruction (e.g. register index or ROB entry index)	The result value	Holds information for the memory address calculation

2.2. Register Status Data Structure

Field	Meaning	`x1`	`x2`	...	`x32`
Reorder	The ROB entry index				
Busy	Whether the register is busy				

2.3. Reservation Station Structure

Function Unit	**Busy**	`Op`	`Vj`	`Vk`	`Qj`	`Qk`	`Dest`	`A`
The FU this `RS` entry represents	Whether the FU is busy	Instruction type	Value of one source operand	Value of the other source operand	The index of the ROB entry produces `Vj`	The index of the ROB entry produces `Vk`	The destination of the instruction (e.g. register index or ROB entry index)	Holds information for the memory address calculation

3. Detailed ROB Pipeline Control

The tables in next few pages lists steps in the ROB-enhanced Tomasulo algorithm and what is required for each step.

For the issuing instruction, `rd` is the destination, `rs` and `rt` are the sources, `r` is the reservation station allocated, `b` is the assigned ROB entry, and `h` is the head entry of the ROB. `RS` is the reservation station data structure. The value returned by a reservation station is called the `result`. `RegisterStat` is the register data structure, `Regs` represents the actual registers, and `ROB` is the reorder buffer data structure.

3.1. Issue Control

A instruction will not only go through the action that all instructions will do, but also goes to the special handling. For example, an `addi` instruction, which is an arithmetic operation, will first go through the action for all general instructions, before going to the actions for "arithmetic operations and stores" and "arithmetic operations only". You are required to determine the correct workflow for specific instructions.

Status	Wait Until	Action / Bookkeeping

All instructions	Reservation station (`r`) and ROB (`b`) both available	`if (RegisterStat[rs].busy):`
		`  h = RegisterStat[rs].Reorder`
		`  if (ROB[h].Ready):`
		`  RS[r].Vj = ROB[h].Value`
		`  RS[r].Qj = 0`
		`  else:`
		`  RS[r].Qj = h`
		`else:`
		`  RS[r].Vj = Regs[rs]`
		`  RS[r].Qj = 0`
		`RS[r].Busy = True`
		`RS[r].Dest = b`
		`ROB[b].Instruction = opcode`
		`ROB[b].Dest = rd`
		`ROB[b].Ready = False`

Arithmetic operations and stores		`if (RegisterStat[rt].Busy):`
		`  h = RegisterStat[rt].Reorder`
		`  if (ROB[h].Ready):`
		`  RS[r].Vk = ROB[h].Value`
		`  RS[r].Qk = 0`
		`  else:`
		`  RS[r].Qk = h`
		`else:`
		`  RS[r].Vk = Regs[rt]`
		`  RS[r].Qk = 0`

Arithmetic operations		`RegisterStat[rd].Reorder = b`
		`RegisterStat[rd].Busy = True`
		`ROB[b].Dest = rd`

Loads		`RS[r].A = imm`
		`RegisterStat[rt].Reorder = b`
		`RegisterStat[rt].Busy = True`
		`ROB[b].Dest = rt`

Stores		`RS[r].A = imm`

3.2. Execute Control

Status	Wait Until	Action / Bookkeeping
Execute arithmetic operations	`RS[r].Qj == 0` and `RS[r].Qk == 0`	Compute results with operands in `Vj` and `Vk`
Load (step 1)	`RS[r].Qj == 0` and there are no stores earlier in the queue	`RS[r].A = RS[r].Vj + RS[r].A`
Load (step 2)	Load step 1 is done and all stores earlier in ROB have different address	Read from `Mem[RS[r].A]`
Store	`RS[r].Qj == 0` and store at queue head	`ROB[h].Address = RS[r].Vj + RS[r].A`

3.3. Write Back Control

Status	Wait Until	Action / Bookkeeping

All (store excluded)	Execution done at `r`	`b = RS[r].Dest`
		`RS[r].Busy = False`
		`For all x where RS[x].Qj == b:`
		`  RS[x].Vj = result`
		`  RS[x].Qj = 0`
		`For all x where RS[x].Qk == b:`
		`  RS[x].Vk = result`
		`  RS[x].Qk = 0`
		`ROB[b].Value = result`
		`ROB[b].Ready = True`

Store	Execution done at `r` and `RS[r].Qk == 0`	`ROB[h].Value = RS[r].Vk`

3.4. Commit Control

> Note: You may omit the misprediction handling if you do not enable branch prediction.

Status	Wait Until	Action / Bookkeeping

Commit	Instruction is at the head of the ROB (i.e. entry `h` h) and `ROB[h].Ready` `True`	`d = ROB[h].Dest`
		`if ROB[h].Instruction == Branch:`
		`  if branch_mispredicted:`
		`  clear ROB[h], RegisterStat`
		`  fetch branch dest into pipeline`
		`elif ROB[h].Instruction == Store:`
		`  Mem[ROB[h].Address] = ROB[h].Value`
		`else:`
		`  Regs[d] = ROB[h].Value`
		`ROB[b].Busy = False`
		`if RegisterStat[d].Reorder == h:`
		`  RegisterStat[d].Busy = False`

4. Report

You need to prepare a report for this lab. The lab report must include at least the following sections:

1. A brief description of your implementation
2. Quantitative evaluation and analysis on your results, test case(s), etc.
3. Your answers to the following questions:
    - Is this implementation optimal?
    - How can you further optimize this design?

   Do make citations if you want to quote from existing literatures.

In your evaluation, you shall give a comparison between the performance *before* and *after* enabling ROB and Tomasulo's algorithm. (e.g. the count of structure/data hazards, the stalled cycles caused by structure/data hazards, etc.)

5. Hand-in

The submission shall include source codes, test case(s) and report. You shall pack the source code, test cases and lab report into a zip. Hand in your zip at Gradescope.

> **REMINDER**: Gradescope has a hard limit of 256 files per submission, please keep your zip according to this limitation.

For each student, we will evaluate only the latest submission before the deadline. We will not evaluate any submission that passes the deadline.

Any file that cannot be unpacked, read, or executed may lead to zero point. Please double check all your file(s) before submission. No second chance would be given after the deadline.

All rules enforced to previous labs are applicable to this lab.