Lab 1: Enabling Out-of-order Completion with Scoreboard

> Due: 2023/11/14 23:59:59

This lab takes 3 points. Do start as early as possible, as this lab may be non-trivial. This is NOT a teamwork and please finish it on your own.

In this lab, you will relieve the structure hazards happening in the execute stage. Some functional units (FU) may take several cycles to execute and thus stall the whole pipeline. However, those instructions that do nothing with the busy FU's should issue and execute.

1. Implications of Scoreboard

The out-of-order completion may introduce WAR or WAW hazards.

A solution for WAR can be:

1. Queue both the operation and copies of its operands
2. Read registers only during Read Operands stage

Having multiple instructions in execution means the execute stage will have multiple execution units ongoing. Therefore, scoreboard can keep tracks of dependencies, states and operations.

Scoreboard replaces the original 5-stage pipelines with a deeper one. Specifically, *Decode*, *Execute*, *Memory Access* and *Write Back* are replaced with five new stages:

1. *Issue (ID1)*

   This stage decodes instructions & checks for structural hazards.

   If a functional unit for the instruction is free and no other active instruction has the same destination register (WAW), scoreboard issues the instruction to the functional unit and updates its internal data structure (presented below in *Data Structures of Scoreboard*). If a structural or WAW hazard exists, then the instruction issue stalls, and no further instructions will issue until these hazards are cleared.

2. *Read Operands (ID2)*

   This stage waits until no data hazards, then read operands.

   A source operand is available if no earlier issued active instruction is going to write it, or if the register containing the operand is being written by a currently active functional unit. When the source operands are available, scoreboard tells the functional unit to proceed to read the operands from the registers and begin execution. Scoreboard resolves RAW hazards dynamically in this step, and instructions may be sent into execution out of order.

3. *Execute (EX) & Memory Access (MA)*

   The *Execute* stage operates on operands.

   The *Memory Access* stage reads/writes data from/to memory.

   The functional unit begins execution upon receiving operands. When the result is ready, it notifies scoreboard that it has completed execution.

4. *Write Back (WB)*

   This stage finishes the execution.

Once scoreboard is aware that the functional unit has completed execution, scoreboard checks for WAR hazards. If none, it writes results. If WAR, then it stalls the instruction. For the following example, scoreboard will stall `sub` until `add` reads operands.

```
div   x1,  x2,  x4   #  x1  <- x2 / x4
add   x10, x0,  x8   #  x10 <- x1 - x8
sub   x8,  x8,  x14  #  x8  <- x8 - x14
```

2. Data Structures of Scoreboard

Here is an illustration of the data structures used in scoreboard (from the reference at the end).

Instruction Status:

Instruction	dest	j	k	Issue	Read Operands	Execute Complete	Write Back
`lw`	`x6`	`34 +`	`x2`				
`lw`	`x2`	`45 +`	`x3`				
`mul`	`x1`	`x2`	`x4`				
`sub`	`x8`	`x6`	`x2`				
`div`	`x10`	`x1`	`x6`				
`add`	`x6`	`x8`	`x2`				

Functional Unit Status:

Time	FU Name	Busy	Op	Fi (Dest)	Fj (Src 1)	Fk (Src 2)	FU (Qj)	FU (Qk)	Fj? (Rj)	Fk? (Rk)
	IntAdd									
	Mul1									
	Mul2									
	Fadd									
	Fdiv									
	...									

Register Result Status:

Reg	x1	x2	x3	x4	x5	x6	x7	x8	x9	...
FU										

1. *Instruction status*

   Records which of 4 steps the instruction is in.

2. *Functional unit status*

   Indicates the state of the FU's. 9 fields for each functional unit:

Field	Meaning
Busy	Indicates whether the unit is busy or not
Op	Operation to perform in the unit (e.g., + or –)
Fi	Destination register
Fj, Fk	Source-register numbers
Qj, Qk	Functional units producing source registers Fj, Fk
Rj, Rk	Flags indicating when Fj, Fk are ready

3. *Register result status*

   Indicates which functional unit will write each register, if one exists. Blank when no ongoing instructions will write that register.

3. Detailed Scoreboard Pipeline Control

For the *Issue* stage, it waits until the FU is not busy and no pending instructions will write the destination register. When issuing an instruction, it marks the FU as busy, and leaves a record in the *functional unit status*.

For the *Read Operands* stage, it waits until Rj and Rk show the registers it reads have been ready. When reading operands, it marks the Rj and Rk as not ready.

For the *Execute* stage, it waits until the FU is done.

For the *Write Back* stage, it waits until Fj and Fk have been prepared by read operands (see the table below for details). When writing results back to the destination register, it marks the corresponding Rj and (or) Rk as ready, and clears the pending instructions in *register result status*.

In conclusion, the control of scoreboard in the pipeline can be summarized as the pseudo-code table below:

Instruction Status	Wait Until	Bookkeeping

Issue	`Busy[FU] == no` and `ResultDepUnit[DestReg] == blank`	`FUStatus[FU] = {`
		`Busy = yes`
		`Op = op`
		`Fi = DestReg`
		`Fj = S1`
		`Fk[FU] = S2`
		`Qj = ResultDepUnit[S1]`
		`Qk = ResultDepUnit[S2]`
		`Rj = Is Qj blank?`
		`Rk = Is Qk blank?`
		`};`
		`ResultDepUnit[DestReg]=FU`

Read Operands	`Rj and Rk == ready`	`Rj = not ready` `Rk = not ready`
Execution (EX & MA) complete	The FU done its execution	
Write Back	For any `FU` that `Fj` depends on, `Rj[FU] = not ready`, and, for any `FU` that `Fk` depends on, `Rk[FU] = not ready`	For any `FU` that produces `Fj`, `Rj = ready`; For any `FU` that produces `Fk`, `Rk = ready`; For current `FU`, `ResultDepUnit[Fi] = blank`, `Busy[FU] = no`

For the example of scoreboard, please refer to the reference at the end.

4. Tips on Implementation

You will implement the scoreboard specified as above in the simulator we used in Lab 0.

You may choose how to implement scoreboard by yourself. Here is a tip on the implementation steps when you are doing this lab:

1. Examine the handling of data bypass and hazard stall of the original simulator
2. Design your data structure for scoreboard
3. Split the original pipeline stages where necessary
4. Model the stall of structure hazards and unbypassable data hazards. Reflect the effects into scoreboard status
5. Implement detailed pipeline control

5. 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.

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

6. 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.

Handin your zip at Gradescope.

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

Reference:
Lecture 10: Case Study — CDC 6600 Scoreboard
By Professor Randy H. Katz
https://people.eecs.berkeley.edu/~randy/Courses/CS252.S96/Lecture10.pdf