Pipeline Hazards

(Notes based on: *Computer Architecture: A Quantitative Approach, 5th Edition*, John L. Hennessy and David A. Patterson, Morgan Kaufmann, 2012)

3 categories of hazards:

(i) Structural hazards: hardware resource conflicts, i.e. two different instructions in the pipeline need the same resource during the same clock cycle. Example: If a unified cache, then IF (always) needs an instruction and MEM may need a data word.

(ii) Data hazards: an instruction depends on the results of a previous instruction that has not yet completed passing through the pipeline.

(iii) Control hazards: pipelining of branches and other instructions that change the PC.

If a hazard occurs, then the pipeline is stalled. Instructions issued later than the stalled instruction are also stalled, but instructions issued before that continue their execution in the pipeline. Thus, no new instructions are issued during the stall. A stall is also called a pipeline bubble or just a bubble, since it "floats" through the pipeline taking up space but performing no useful work.

Data hazard example: Consider the following sequence of instructions:

```
DADD    R1, R2, R3
DSUB    R4, R1, R5
AND     R6, R1, R7
OR      R8, R1, R9
XOR     R10, R1, R11
```

The result of the DADD instruction goes into R1, but not until the WB stage. The next 4 instructions all use R1 as an operand. We need to make sure that they are using the value computed by the DADD instruction, which is what the programmer intended.

Assume that the DADD instruction IF stage occurs in cycle 1. The DSUB and AND instructions have a data hazard, because R1 is written in (the first half of) cycle 5, but is read by DSUB in cycle 3 and by AND in cycle 4. The OR instruction does not have a data hazard because it reads R1 during the second half of cycle 5, whereas the write to R1 occurs in the first half of cycle 5. There is also no hazard for the XOR instruction, which reads R1 in cycle 6.

The solution to the above data hazards is to use forwarding. The correct result for R1 is in a pipeline register even if it has not yet been written back to the register file. So, just make that pipeline register value be a possible ALU input by feeding it back to MUXs present at the ALU inputs. (It may be needed as a "left input" or a "right input"). Specifically:

1. The ALU result from both the EX/MEM and MEM/WB pipeline registers is fed back to each of the ALU inputs.
2. If the forwarding logic detects a data hazard, the control logic selects the forwarded value rather than the value read from the register file (which is the old or "stale" value).

In the previous example, the value in EX/MEM is sent to an ALU input for DSUB. The value in MEM/WB is sent to an ALU input for AND.

Data hazards requiring stalls: Not all data hazards can be eliminated by forwarding. For example, in a load instruction, the value from data memory is available at the end of MEM. If the next instruction uses that as an operand (which would be in its EX stage, which would be at the same time as MEM of the load instruction), then this cannot be accommodated. The only possibility is to stall the pipeline for one cycle. For example, the following timing will not work:

LD	R1, 0(R2)	IF	ID	EX	MEM	WB		
DSUB	R4, R1, R5		IF	ID	EX	MEM	WB	
AND	R6, R1, R7			IF	ID	EX	MEM	WB
OR	R8, R1, R9				IF	ID	EX	MEM WB

So, we have to stall the pipeline as follows:

LD	R1, 0(R2)	IF	ID	EX	MEM	WB		
DSUB	R4, R1, R5		IF	ID	stall	EX	MEM	WB
AND	R6, R1, R7			IF	stall	ID	EX	MEM WB
OR	R8, R1, R9				stall	IF	ID	EX MEM WB

Branch hazards: (a type of control hazard) If a branch is not taken, the PC is incremented by 4. If the branch is taken, then the PC value is changed to the branch target address. A condition is checked and the branch target address is computed in ID. If the branch is taken, the PC is changed at the end of ID.

How do deal with branch hazards? Simplest method is to redo the fetch of the instruction following the branch once we detect the branch in ID, which is when the instructions are decoded.

-> one stall cycle for every branch (since IF is repeated once the branch outcome is known), so not the most efficient procedure

Four methods to reduce branch penalties:

(i) Freeze or flush the pipeline – hold or delete any instructions after the branch until the branch destination is known. Simple to implement, but not the best performance. Gives a one-cycle penalty. The first IF in the branch successor is essentially a stall since it is not used. Note that in the case of the branch not taken, the second IF will fetch the same instruction again.

Branch instr.	IF	ID	EX	MEM	WB		
Branch successor		IF	IF	ID	EX	MEM	WB (note the 2nd IF)
Branch successor + 1				IF	ID	EX	MEM WB

(ii) Predicted-not-taken – Treat every branch as not taken, in other words, just continue fetching instructions normally. However, we don't want to change the state of the processor until the branch outcome is known. In the case the branch is taken, any changes caused by the misprediction must be undone. We need to turn the fetched wrong instruction into a no-op and restart the fetch at the target address.

(iii) Predicted-taken

(iv) Delayed branch – there is a "branch delay slot" following a branch instruction. The instruction following the branch (in this delay slot) is always executed. Then, the instruction after that is the one at the branch target if the branch is taken. The compiler tries to put a useful instruction into the branch delay slot, so that that cycle is not wasted.

Static branch prediction: Uses information available at *compile time*.

Branches are often highly biased towards either taken or untaken. In other words, for a given branch instruction, it will mostly be either taken or untaken, rather than roughly equal taken/untaken. (An example would be a branch used in a for-loop: it is always taken, except for the last iteration.)

Use *profile information* from *earlier runs* of the program to statically predict the behavior of each branch instruction during compilation. This works particularly well for floating-point benchmarks, and less well for integer benchmark programs.

Dynamic branch prediction: Uses information based on the run-time execution of the program.

Can use a branch-prediction buffer (BTB) or a branch history table.

A BTB is a small memory indexed by the low-order bits of the address of the branch instructions in a program. For a 1-bit predictor, the data stored at each memory location is one bit that indicates if the branch was taken last time or not.

- Since we use only the low-order address bits (to keep the buffer size small), we may be mixing up the recent history of different branch instructions – but it doesn't matter! We treat the bit as a hint that is assumed to be correct and fetch the corresponding instruction. (So, if the hint is untaken, then we fetch the next sequential instruction; if the hint is taken, then we fetch the instruction at the branch target address.)
- If the prediction turns out to be wrong, the prediction bit is inverted and stored back in the BTB. (i.e., we update our prediction based on this most recent experience)

2-bit predictor: Usually has better performance than a 1-bit predictor, because it takes two wrong predictions in a row before the prediction is changed. Better performance for branches that strongly favor either taken or untaken, as in loops.