Hazards in Pipelining:

By: Mahesh Neupane

Introduction:

Pipelining can efficiently increase the performance of a processor by overlapping execution of instructions. But the efficiency of the pipelining depends upon, how the problem encountered during the implementation of pipelining is handled. These problems are known as HAZARDS.

Types of Hazards:

a) Structural Hazards (Resource Bound)
b) Control Hazards (Pipelining Bubbles)
c) Data Hazards (Data dependencies)

Structural Hazards:

During the pipelining, the overlapped execution of instructions requires pipelining of functional units and duplication of resources to allow all possible combinations of instructions in the pipeline.
If some combination of instructions cannot be accommodated because of *a resource conflict,* the machine is said to have a structural hazard. This type of hazards occurs when two activities require the same resource simultaneously.
Common instances of structural hazards arise when:

1) Some functional unit is not fully pipelined then a sequence of instructions using that un-pipelined unit cannot proceed at the rate of one per clock cycle.
2) Some resource has not been duplicated enough to allow all combinations of instructions in the pipeline to execute.

Example:

If machine has shared a single-memory pipeline for data and instructions. As a result, when an instruction contains a data-memory reference (load-MEM), it will conflict with the instruction reference for a later instruction (instr 3-IF):

	Clock cycle number									
Instr	1	2	3	4	5	6	7	8		
Load	IF	ID	EX	MEM*	WB					
Instr 1		IF	ID	EX	MEM	WB				
Instr 2			IF	ID	EX	MEM	WB			
Instr 3				IF*	ID	EX	MEM	WB		

To resolve this, we stall the pipeline for one clock cycle when a data-memory access occurs. The effect of the stall is actually to occupy the resources for that instruction slot. The following table shows how the stalls are actually implemented.

Clock cycle number										
Instr	1	2	3	4	5	6	7	8	9	
Load	IF	ID	EX	MEM	WB					
Instr 1		IF	ID	EX	MEM	WB				
Instr 2			IF	ID	EX	MEM	WB			
Instr 3				stall	IF	ID	EX	MEM	WB	

Instruction 1 assumed not to be data-memory reference (load or store), otherwise Instruction 3 cannot start execution because of structural hazard.

We know that introduction of stall reduces the performance of the system but there are following reasons for allowing structural hazard while designing the system:

To reduce cost: For example, machines that support both an instruction and a cache access every cycle (to prevent the structural hazard of the above example) require at least twice as much total memory.
To reduce the latency of the unit: The shorter latency comes from the lack of pipeline registers that introduce overhead.

Control Hazard:

This type of hazard is caused by uncertainty of execution path, branch taken or not taken.

It is a hazard that arises when an attempt is made to make a decision before condition is evaluated.

It results when we branch to a new location in the program, invalidating everything we have loaded in our pipeline.

Control hazard can cause a greater performance loss for DLX pipeline than data hazards. When a branch is executed, it may or may not change the PC (program counter) to something other than its current value plus 4. If a branch changes the PC to its target address, it is a ***taken branch***; if it falls through, it is ***not taken***.

If instruction ***i*** is a ***taken branch***, then the PC is normally not changed until the end of MEM stage, after the completion of the address calculation and comparison.

Methods to Deal with Control Hazard:

- Pipeline stall until branch target known.
- Continue fetching instructions as if we won't take the branch but then invalidating the instruction if we do take the branch.
  - Always fetch the branch target After all, most branches are taken.
  - Precompute the target if architecture support (DLX doesn't support it).

- Delayed Branch: Perform instruction scheduling into branch delay slots (instruction after a branch)

  Always execute instructions following a branch regardless of whether branch taken or not taken.

The simplest method of dealing with branches is ***to stall*** the pipeline as soon as the branch is detected until we reach the **MEM** stage, which determines the new PC. The pipeline behavior looks like:

Branch	IF	**ID**	EX	**MEM**	WB					
Branch successor		**IF**(stall)	***stall***	***stall***	IF	ID	EX	MEM	WB	
Branch successor+1						IF	ID	EX	MEM	WB

The stall does not occur until after **ID** stage (where we know that the instruction is a branch).

This control hazard stall must be implemented differently from a data hazard, since the **IF** cycle of the instruction following the branch **must be repeated** as soon as we know the branch outcome. Thus, the first **IF** cycle is essentially a ***stall*** (because it never performs useful work), which comes to total 3 stalls.
Three clock cycles wasted for every branch is a significant loss.

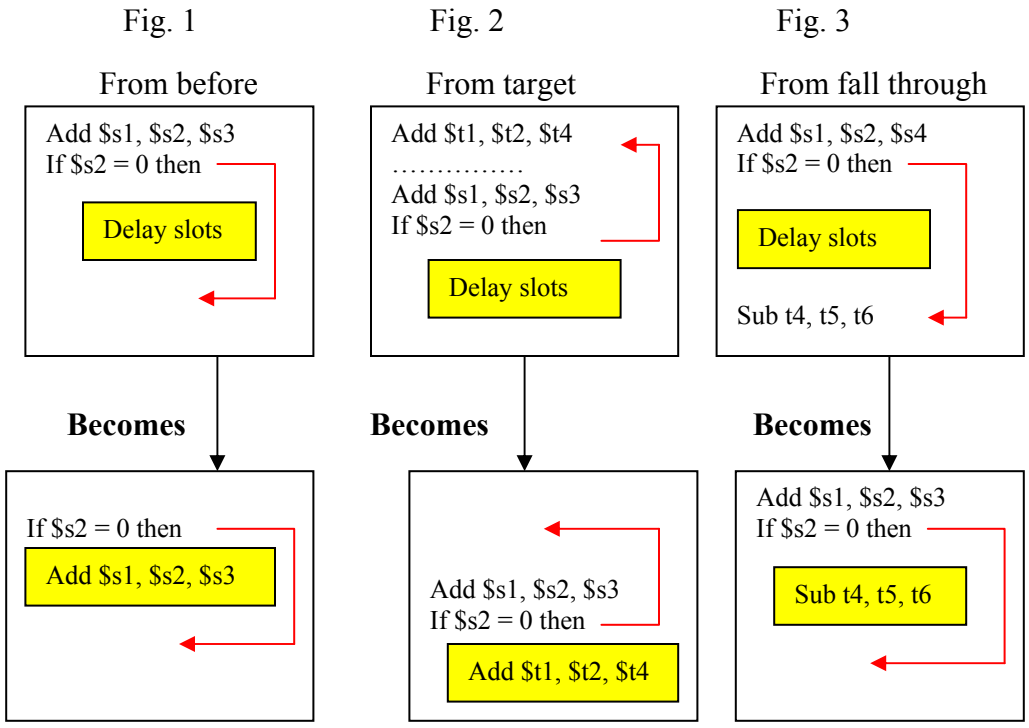
With a 30% branch frequency and an ideal CPI of 1, the machine with branch stalls achieves only ***half*** the ideal speedup from pipelining.
The number of clock cycles can be reduced by two steps:

- Find out whether the branch is taken or not taken ***earlier*** in the pipeline;
- Compute the taken PC (i.e., the address of the branch target) ***earlier***.

The Branch delay slot scheduling method helps reducing the branch penalty.

There are three modes of scheduling the delay slot in the branching instructions:

- Scheduling before the branch target instruction.(fig. 1)
- Scheduling from the target instruction.(fig. 2)
- Scheduling from fall through instruction.(fig. 3)



Data Hazards:

- **Data hazards** occur when the pipeline changes the order of read/write accesses to operands so that the order differs from the order seen by sequentially executing instructions on the unpipelined machine.
- Data hazards are also known as data dependency. Data dependency is the condition in which the outcome of the current operation is dependent on the outcome of a previous instruction that has not yet been executed to completion because of the effect of the pipeline.
- Data hazards arise because of the need to preserve the order of the execution of instructions.

The following example shows the data hazards:

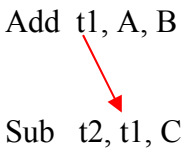
Add R2, R1, R0 [R2]←[R1] + [R0]

Sub R1, R5, R6 [R4]←[R5] – [R6]

Data hazards are classified into three types:

1. RAW- Read After Write (also known as True Data Dependency)
2. WAW- Write After Write (also known as Output Dependency)
3. WAR – Write After Read (also known as Anti Data Dependency)

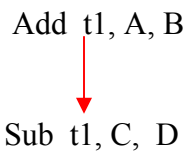
- RAW hazard is most common hazard that occurs when a read operation takes place after a write. The following example shows the possible RAW hazards:



Solution: Internal forwarding (this can be used for all types of Data hazards)

WAW hazard is the hazard that occurs when a write operation follows another write operation. This hazard is present only in pipelines that write in more than one pipe stage or allow an instruction to proceed even when a previous instruction is stalled.

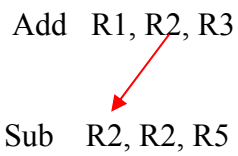
- This following example shows the possible WAW hazard:



Solution: The WAW hazards can be avoided by doing following changes in the pipelining:

- Move write back for an ALU operation into the MEM stage, since the data value is available be then.
- By assuming that the data memory access takes place in 2 pipelining stages.

- WAR hazard is the hazard that occurs when write operation follows read operation. This following example shows the possible WAR hazard:



Solution: The WAR hazard can be avoided by internal forwarding and by modifying pipeline architecture such that consecutive Write and Read occur after few clock cycles.

References:

http://websrv.cs.fsu.edu/~tyson/CDA5155/refs.html
http://www.cs.ucsd.edu/classes/wi99/cse141_B/lectures.html
http://www.cs.caltech.edu/courses/cs184/ winter2001/slides/day5_2up.pdf
http://www.cs.iastate.edu/~prabhu/Tutorial/PIPELINE/compSched.html
http://www.mmdb.ece.ucsb.edu/~ece154/lecture5.pdf