Pipelines – Basic Control Hazards

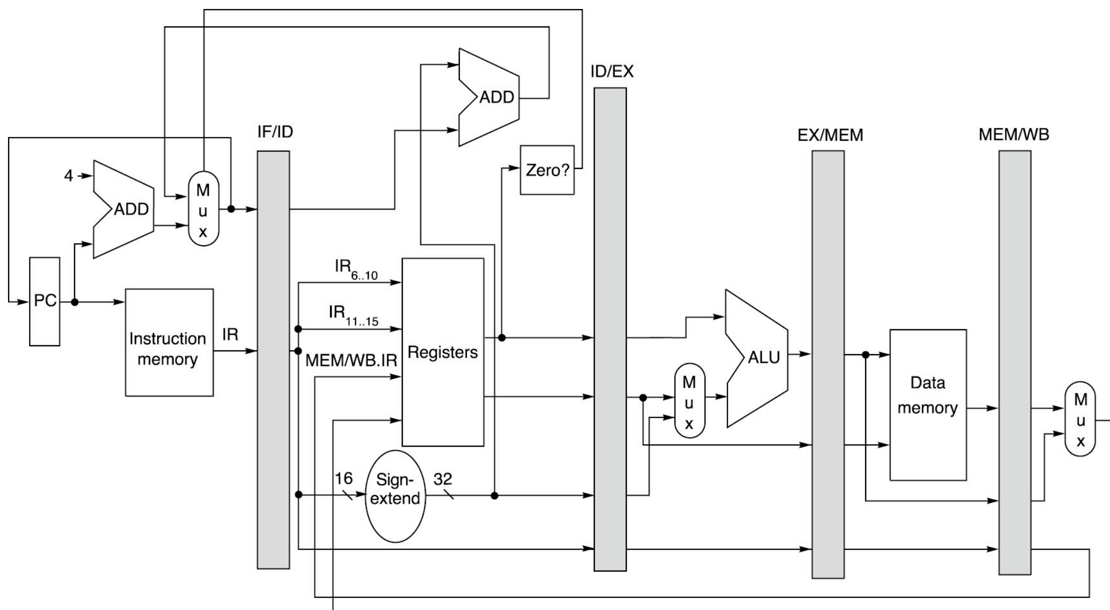
- Current architecture (previous lecture, figure 1) performs the branch test in the ALU cycle.
- PC for branches are resolved in the MEM cycle.
    - Uses the ALU to compose the branch taken target address.
    - Potential to cause 2 stall cycles, figure 1.

Instruction	Clock									
	1	2	3	4	5	6	7	8	9	10
Branch	IF	ID	EX	MEM	WB					
Branch successor		IF	ID	Flush						
Branch taken			Stall	Stall	IF	ID	EX	MEM	WB	
Branch taken + 1						IF	ID	EX	MEM	WB

Figure 1 : Branch Problem

Hardware Solutions

- By limiting ourselves to branch tests against zero and introducing an additional ALU, we are able to complete the branch test in the ID stage, figure 2.
    - Now incur a single stall cycle for branches taken.



© 2003 Elsevier Science (USA). All rights reserved.

Figure 2: Hardware Speedup for Branch test against zero

- Are there any drawbacks?
    - Has the sensitivity to data hazards changed?
    - Consider the case of a branch instruction based on a result calculated by a *preceding* ALU instruction.

'Software' Solutions

- Software approaches to minimizing the effect of branch hazards perform some form of *branch prediction*.
- Consider,

Predict-untaken (Predict-not-taken)	Assumes that the branch will not be taken. Required to flush the *branch successor* if wrong prediction, figure 3.
Predict always taken	Treat every branch as taken – no advantage for the simple pipe as the branch address is not known until the MEM stage.

Figure 3: Operation of Simple Branch-Untaken prediction

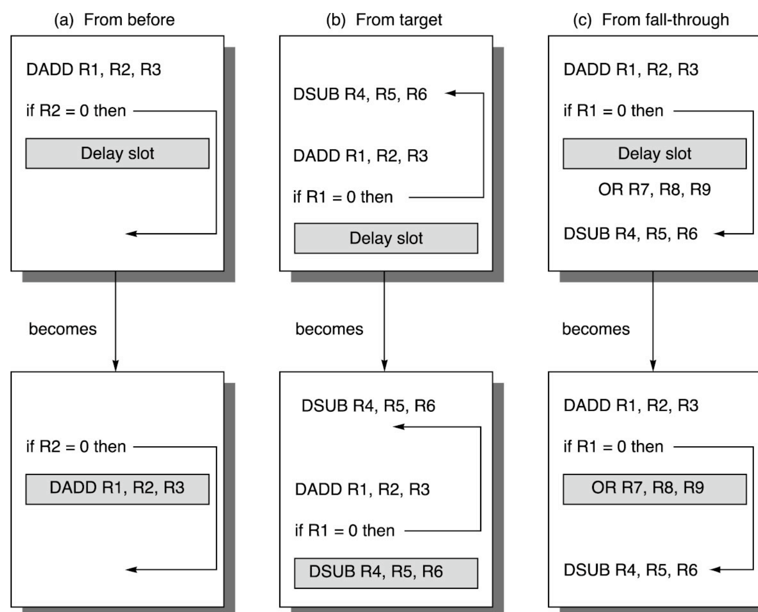
Case: branch not taken									
Instruction	Clock								
	1	2	3	4	5	6	7	8	9
Branch not taken	IF	ID	EX	MEM	WB				
Instruction (i + 1)		IF	ID	EX	MEM	WB			
Instruction (i + 1)			IF	ID	EX	MEM	WB		
Instruction (i + 3)				IF	ID	EX	MEM	WB	
Case: branch taken									
Branch taken	IF	ID	EX	MEM	WB				
Instruction (i + 1)		IF	idle	idle	idle	idle			
Branch target			IF	ID	EX	MEM	WB		
Branch target + 1				IF	ID	EX	MEM	WB	

- Suitability of a predict-not-taken scheme is strongly influenced by how closely code conforms to the prediction. This is where the compiler may also help.
- Consider a pipe with a branch taken delay of ONE (c.f. hardware modification).

- Irrespective of whether the branch is taken or not, the instruction representing the *branch successor* is always taken.

Case of Branch taken	Case of no Branch taken
Branch instruction; Sequential successor(1); Branch target if taken;	Branch instruction; Sequential successor(1); Sequential successor(2);

  - Implication – use the branch successor as a generic slot for instructions that are always executed (becomes a *branch delay slot*).
  - Figure 3 summarizes some of the code patterns used by the compiler to usefully employ the 'branch delay slot'.



© 2003 Elsevier Science (USA). All rights reserved.

Figure 3: Branch Delay Slot Scheduling.

(a) Instruction scheduled is completely independent of branch or no branch decision;

(b) Dependency on R1 – need to insert a duplicate instruction for the delay slot;

(c) Dependency on R1 – wasted delay slot when branch taken AND R7 content cannot effect branch taken path.

- Limitations,
    - Restrictions regarding the types of instruction sequences that can be scheduled.
    - Ability to predict which direction the branch will go at compile time.

Analyzing Basic Pipeline Performance

- Assumptions on encountering any of the generic pipe hazards our pipe will,

a. Stall all instructions STARTING at a later point than the instruction currently causing the hazard;

b. Complete execution of instructions STARTING at an earlier point than the instruction currently causing the hazard;

c. No new instructions are fetched during a stall.

Analytical model

- Speedup from pipelining is,

$$\frac{\text{Average Instruction Time UNPIPELINED}}{\text{Average Instruction Time PIPELINED}}$$

$$= \frac{\text{CPI(unpiped)} \times \text{Clock period(unpiped)}}{\text{CPI(piped)} \times \text{Clock period(piped)}}$$

- Pipelining should therefore decrease either the CPI of the clock cycle time to be successful.

Case of generic pipeline Stalls

- Divide CPI between IDEAL (typically 1) and that due to stalls, or

CPI(piped) = CPI(Ideal) + Pipeline stall cycles per instruction

CPI(piped) = CPI(Ideal) + Pipeline stall cycles per instruction
= 1 + Pipeline stall cycles per instruction

- If the clock period of both piped and unpiped machines are the same, then

Speedup =

$$\frac{\text{CPI(unpiped)}}{1 + \text{Pipeline stall cycles per instruction}}$$

- Additionally,
    - IF stages are perfectly balanced (same clock cycles per stage)
    - THEN CPI(unpiped) = Pipeline depth
    - THEREFORE in this special case pipelining improves performance by the pipe depth.
    - Represents the upper bound on pipeline speedup – Does this imply that we should attempt to maximize pipe length?

Special Case of Branch Penalties

- What factors effect a stall under a control hazard?
    - Frequency of the hazard, and
    - Penalty associated with the hazard (number of cycles lost).
- Pipeline stall cycles per instruction = Branch Frequency × Branch Penalty.

Example

Let the frequency for various branch instructions be as follows,

- Conditional Branches: 15% of which 60% are taken
- Jumps and Calls: 1%

The architecture consists of a 4 stage pipe in which the branch is resolved,

- at the end of the 2nd cycle for unconditional branches, and;
- at the end of the 3rd cycle for conditional branches.

Let the 4 pipe stages correspond to IF, ID, ES and WB and assume that only the first stage may execute independently of branch resolution. Ignoring other sources of pipe stalls, how much faster would the CPU be if no branch hazards exist?