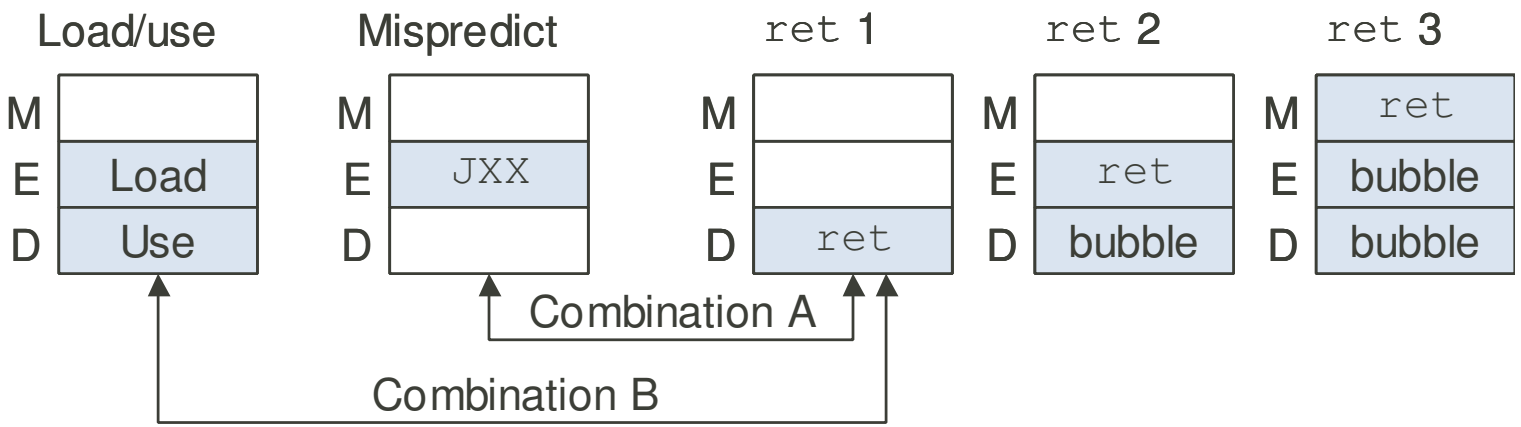
CS429: Computer Organization and Architecture

Pipeline IV

Warren Hunt, Jr. and Bill Young
Department of Computer Sciences
University of Texas at Austin

Last updated: November 5, 2014 at 11:25

Control Combinations



Two special cases can arise on the same clock cycle.

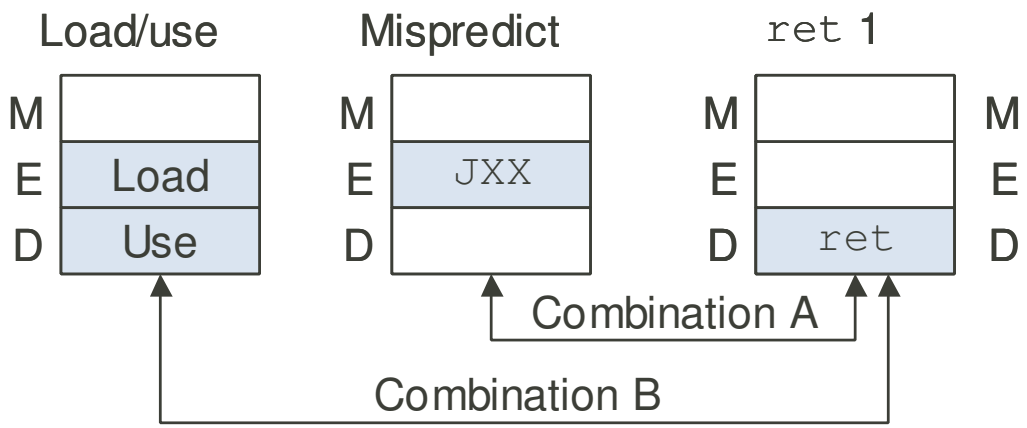
Combination A:

- Not-taken branch
- `ret` instruction at branch target

Combination B:

- Instruction that reads from memory to %esp
- Followed by `ret` instruction

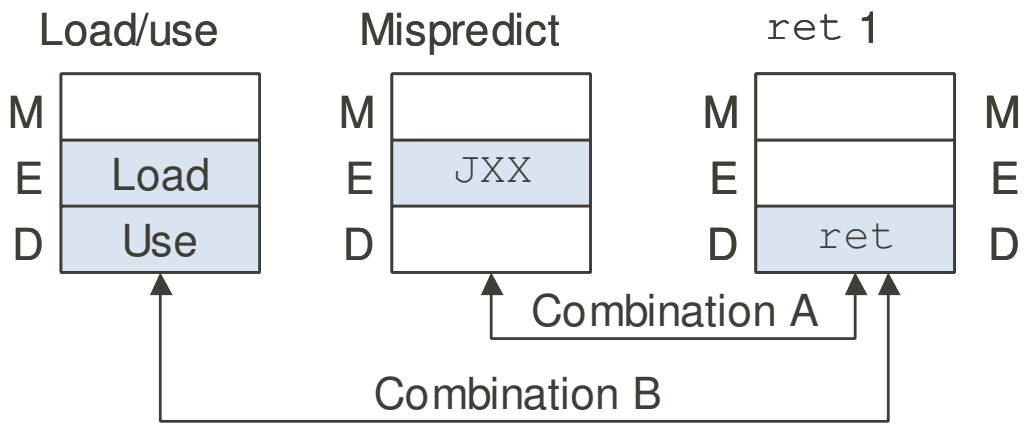
Control Combination A



Condition	F	D	E	M	W
Processing ret	stall	bubble	normal	normal	normal
Mispredicted Branch	normal	bubble	bubble	normal	normal
Combination	stall	bubble	bubble	normal	normal

- Should handle as mispredicted branch.
- Stalls F pipeline register.
- But PC selection logic will be using `M_valM` anyway.

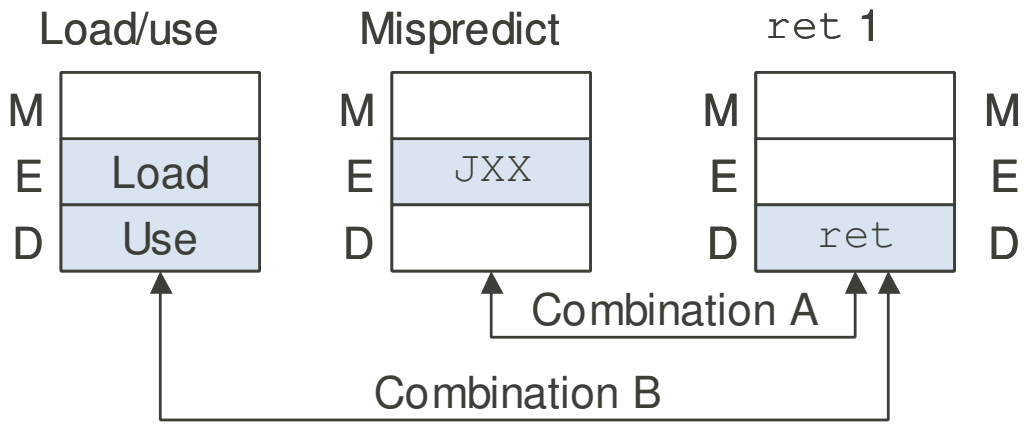
Control Combination B: First Attempt



Condition	F	D	E	M	W
Processing ret	stall	bubble	normal	normal	normal
Load/Use Hazard	stall	stall	bubble	normal	normal
Combination	stall	bubble + stall	bubble	normal	normal

- Would attempt to bubble and stall pipeline register D.
- Signalled by processor as pipeline error.

Control Combination B: Correct Approach



Condition	F	D	E	M	W
Processing ret	stall	bubble	normal	normal	normal
Load/Use Hazard	stall	stall	bubble	normal	normal
Combination	stall	stall	bubble	normal	normal

- Load / use hazard should get priority.
- `ret` instruction should be held in decode stage for additional cycle.

Corrected Pipeline Control Logic

Condition	F	D	E	M	W
Processing ret	stall	bubble	normal	normal	normal
Load/Use Hazard	stall	stall	bubble	normal	normal
Combination	stall	stall	bubble	normal	normal

- Load / use hazard should get priority.
- `ret` instruction should be held in decode stage for additional cycle.

Pipeline Summary

Data Hazards

- Most handled by forwarding with no performance penalty
- Load / use hazard requires one cycle stall

Control Hazards

- Cancel instructions when detect mispredicted branch; two cycles wasted
- Stall fetch stage while `ret` pass through pipeline; three cycles wasted.

Control Combinations

- Must analyze carefully
- First version had a subtle bug
- Only arises with unusual instruction combination

Performance Analysis with Pipelining

$$\text{CPU time} = \frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} * \frac{\text{Cycles}}{\text{Instruction}} * \frac{\text{Seconds}}{\text{Cycle}}$$

- Ideal pipelined machine: CPI $= 1$
  - One instruction completed per cycle.
  - But much faster cycle time than unpipelined machine.
- However, hazards work against the ideal
  - Hazards resolved using forwarding are fine.
  - Stalling degrades performance and instruction completion rate is interrupted.
- CPI is a measure of the "architectural efficiency" of the design.

Computing CPI

CPI is a function of useful instructions and bubbles:

$$CPI = \frac{C_i + C_b}{C_i} = 1.0 + \frac{C_b}{C_i}$$

You can reformulate this to account for:

- load penalties (lp)
- branch misprediction penalties (mp)
- return penalties (rp)

$$CPI = 1.0 + lp + mp + rp$$

Computing CPI (2)

- So, how do we determine the penalties?
  - Depends on how often each situation occurs on average.
  - How often does a load occur and how often does that load cause a stall?
  - How often does a branch occur and how often is it mispredicted?
  - How often does a return occur?
- We can measure these using:
  - a simulator, or
  - hardware performance counters.
- We can also estimate them through historical averages.
  - Then use estimates to make early design tradeoffs for the architecture.

Computing CPI (3)

Cause	Name	Instruction Frequency	Condition Frequency	Stalls	Product
Load/use	lp	0.30	0.3	1	0.09
Mispredict	mp	0.20	0.4	2	0.16
Return	rp	0.02	1.0	3	0.06
Total penalty					0.31

$$\text{CPI} = 1 + 0.31 = 1.31 == 31\%$$

This is not ideal.

This gets worse when:

- you also account for non-ideal memory access latency;
- deeper pipeline (where stalls per hazard increase).