HY425 Lecture 02: Pipelining

Dimitrios S. Nikolopoulos

University of Crete and FORTH-ICS

October 13, 2011

Review from last lecture

Important technological implications

- Latency lags bandwidth
- Shrinking transistors do not necessarily improve performance
- Power wall, deteriorating reliability
- Measuring and summarizing performance
  - Wall-clock time
  - Geometric mean of execution time ratios
  - No single averaging metric is perfect
- Quantitative principles of design
  - Parallelism, locality, common case fast, Amdahl's law

Update on assignments

Homework

- Homework 1 up today due in one week.

Processor review

Datapath

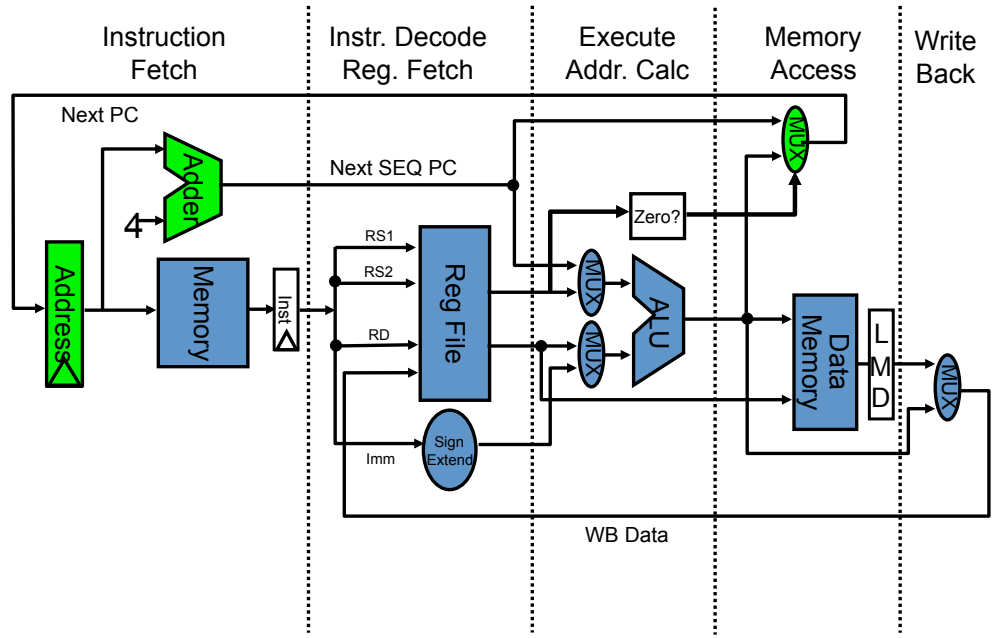
- Storage elements (registers, caches, memory)
- Functional (execution) units (ALU, adders)
- Operated by control signals

Control

- State machine producing control signals

Multi-cycle datapath

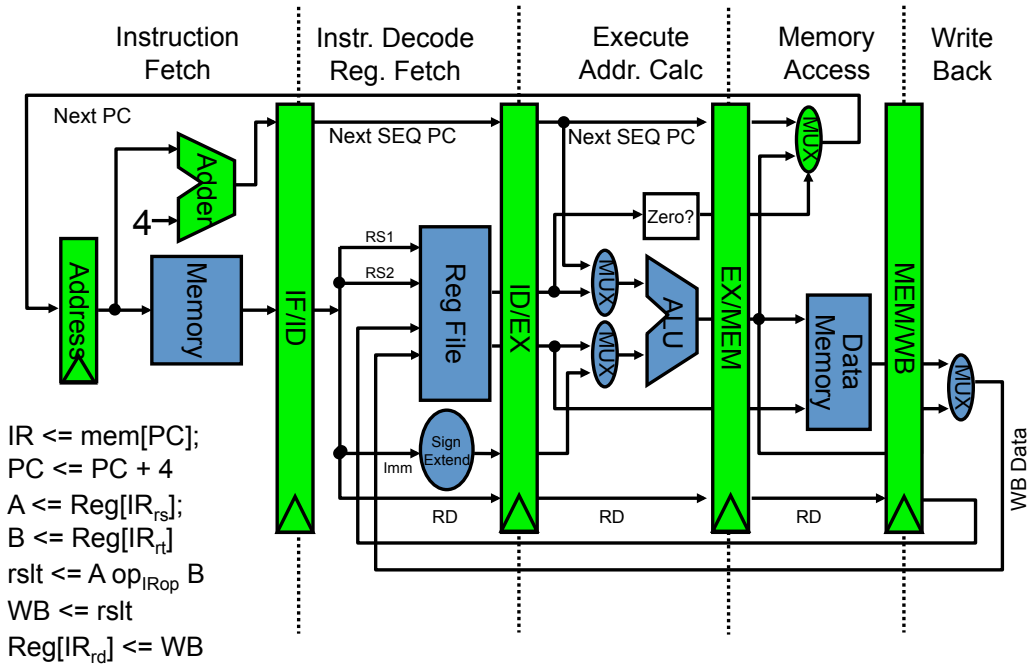
Five-stage instruction execution sequence



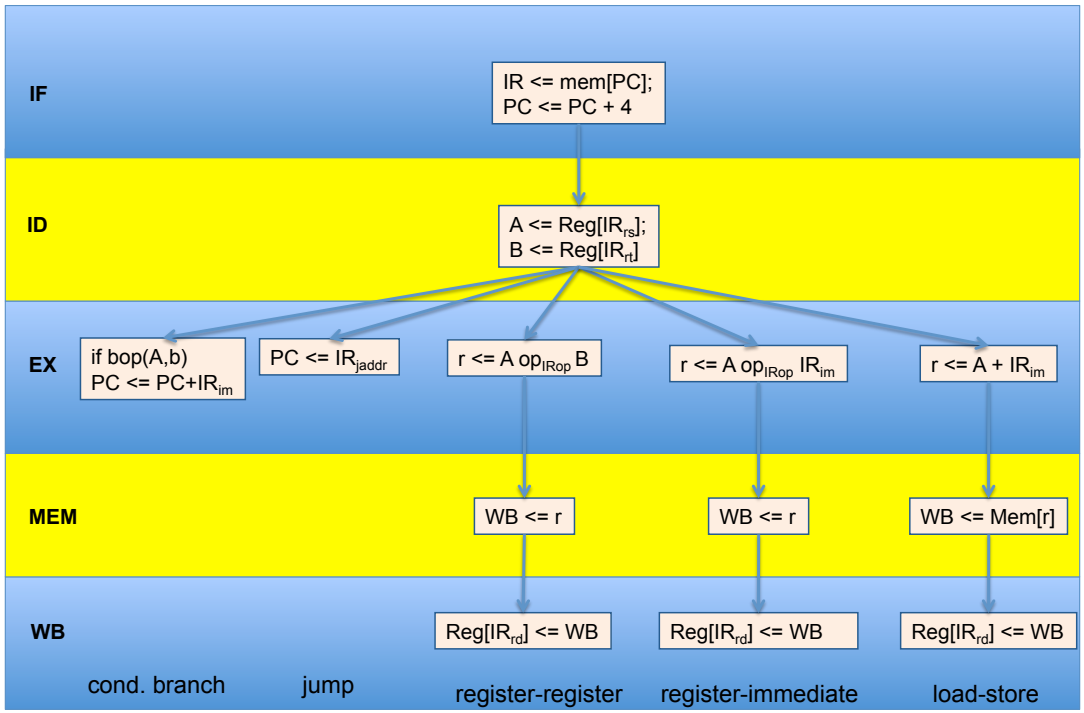
$Reg[IR_{rd}] <= Reg[IR_{rs}]\ op_{IRop}\ Reg[IR_{rt}]$

Multi-cycle datapath

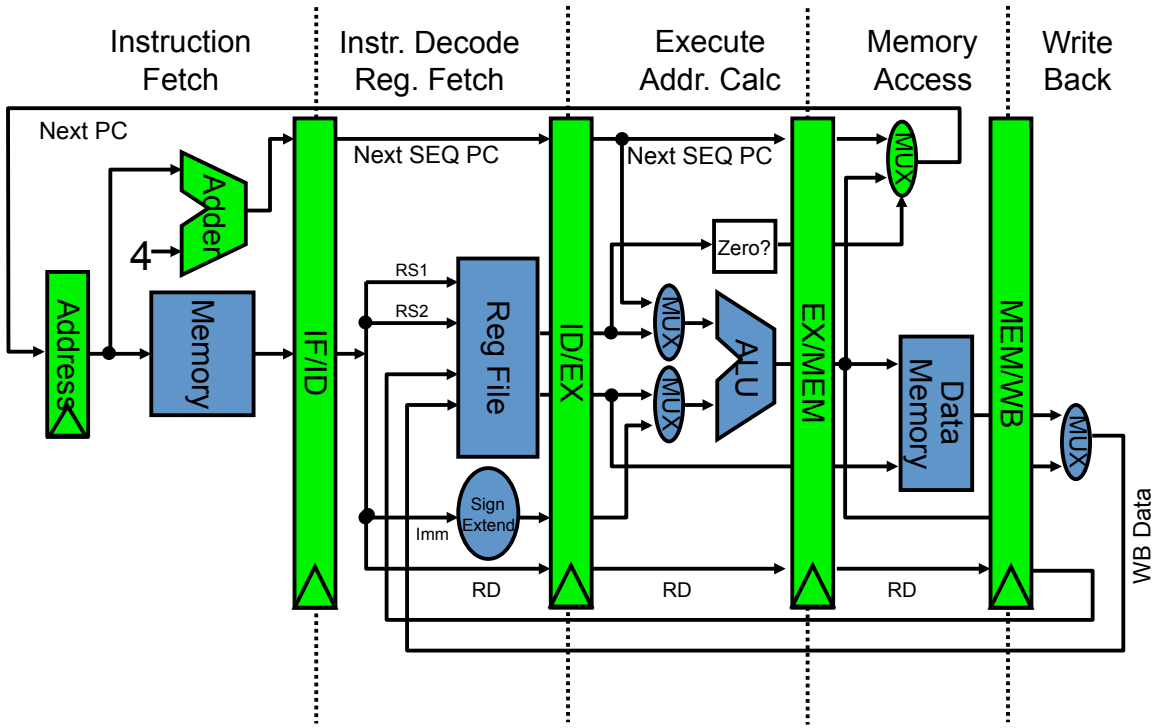
Five-stage instruction execution sequence



Instruction operation control

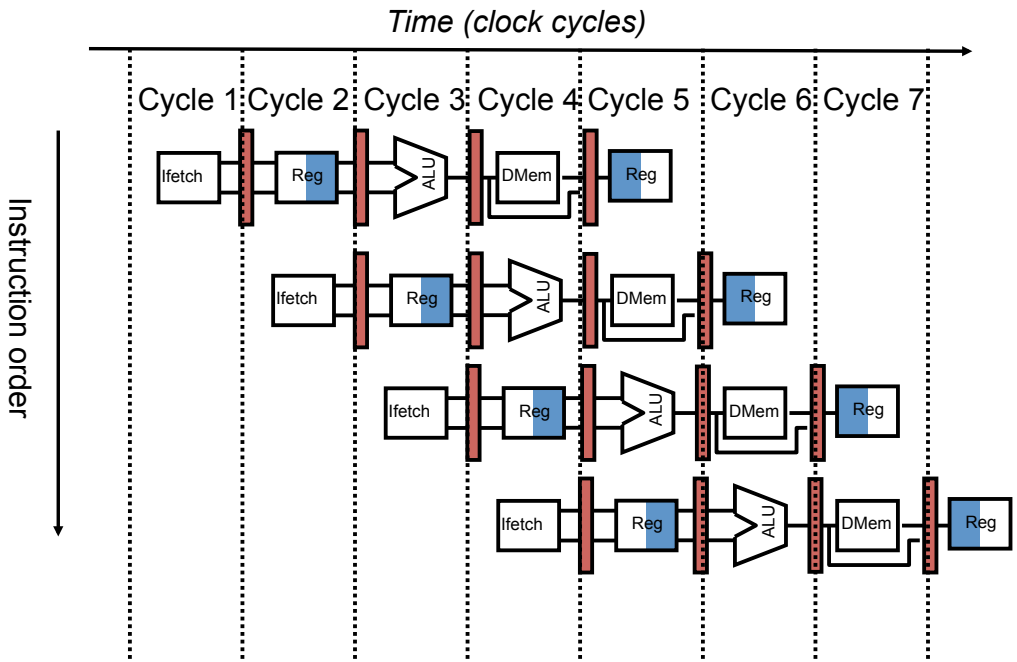


Data stationary control



Local decoding logic for each pipeline stage

Simplified visualization of pipelines



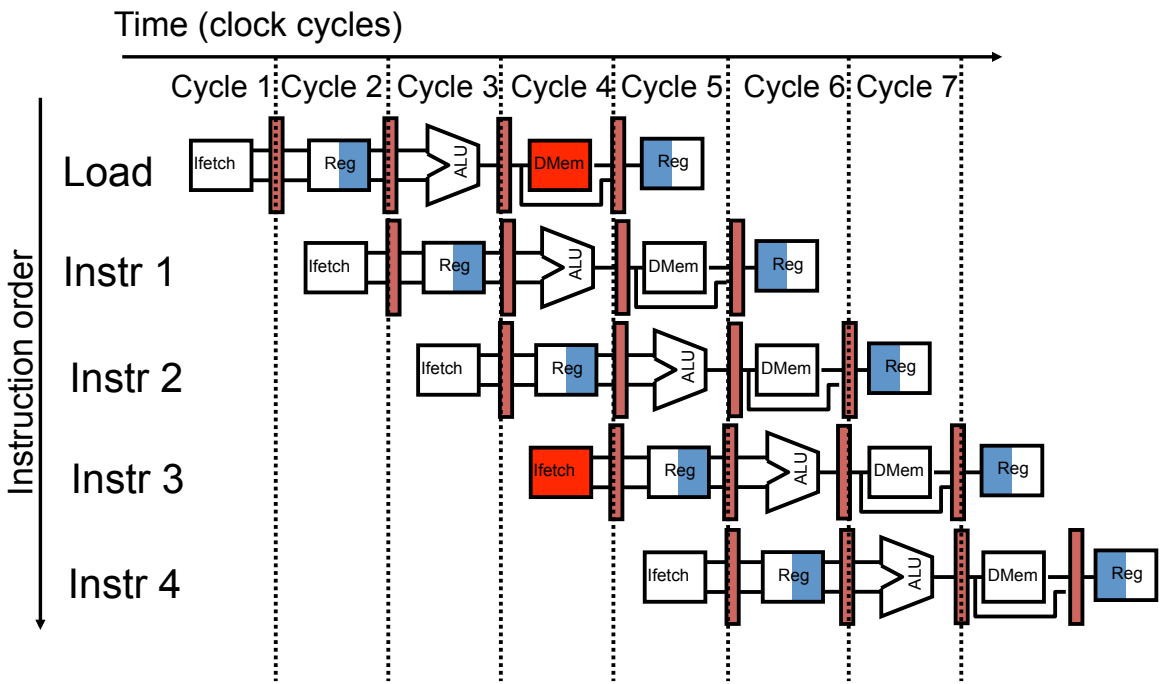
Limits of pipelining

Hazards

- Hazard is a condition which prevents an instruction from executing during a pipeline stage
- Structural hazards occur when the hardware does not have enough resources in a pipeline stage to accommodate an instruction
  - Older instructions occupy resources in same stage
- Data hazards occur when an instruction needs input from a prior instruction and the input is not ready
- Control hazards occur when execution of an instruction depends on a branch and branch outcome is not known yet
  - Missing taken/not taken or destination address information

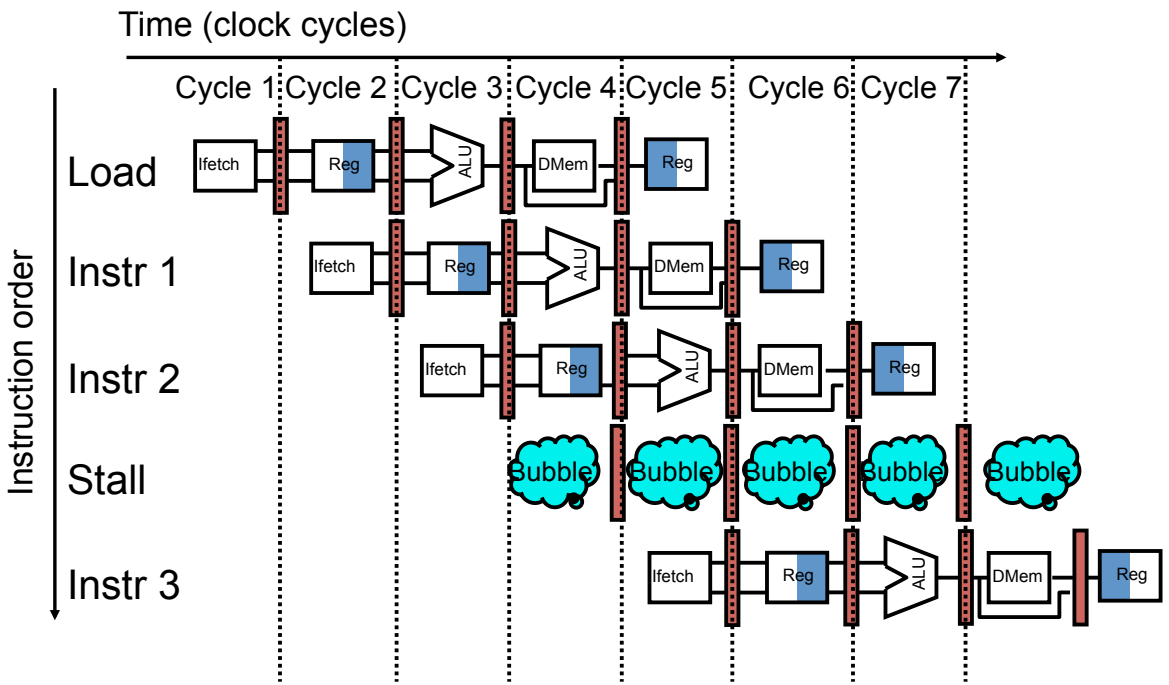
Example of structural hazard

Single memory port for instructions and data



Resolving structural hazards

Bubbles



Bubbles

Software vs. hardware bubbles

- Software (compiler) inserts bubbles by inserting `nop` instructions in the pipeline
- Hardware uses hazard detection unit in the control logic
  - Detection unit evaluates conditions for hazards
  - Stalls the pipeline briefly (one cycle) to resolve the hazard
  - Stalling the pipeline at any stage amounts to zeroing output control signals

Pipeline control

Recap

- Control signals in EX stage (ALUOp, RegDst, ALUSrc)
- Control signals in MEM stage (Branch, MemRead, MemWrite)
- Control signals in WB stage (MemtoReg, RegWrite)

Impact of pipeline stalls on performance

Speedup of pipelining

$$Speedup_{pipelined} = \frac{avg\ instruction\ time\ unpipelined}{avg\ instruction\ time\ pipelined}$$

$$= \frac{CPI_{unpipelined} \times Clock\ cycle_{unpipelined}}{CPI_{pipelined} \times Clock\ cycle_{pipelined}}$$

$$= \frac{CPI_{unpipelined}}{CPI_{pipelined}} \times \frac{Clock\ cycle_{unpipelined}}{Clock\ cycle_{pipelined}}$$

$$CPI_{pipelined} = IdealCPI + Pipeline\ stall\ cycles\ per\ instruction$$

$$= 1 + Pipeline\ stall\ cycles\ per\ instruction$$

$$Speedup_{pipelined} = \frac{CPI_{unpipelined}}{1 + Pipeline\ stall\ cycles\ per\ instruction}$$

Impact of pipeline stalls on performance

Speedup of pipelining

$$Speedup_{pipelined} = \frac{1}{1 + Pipeline\ stall\ cycles\ per\ instruction} \times \frac{Clock\ cycle_{unpipelined}}{Clock\ cycle_{pipelined}}$$

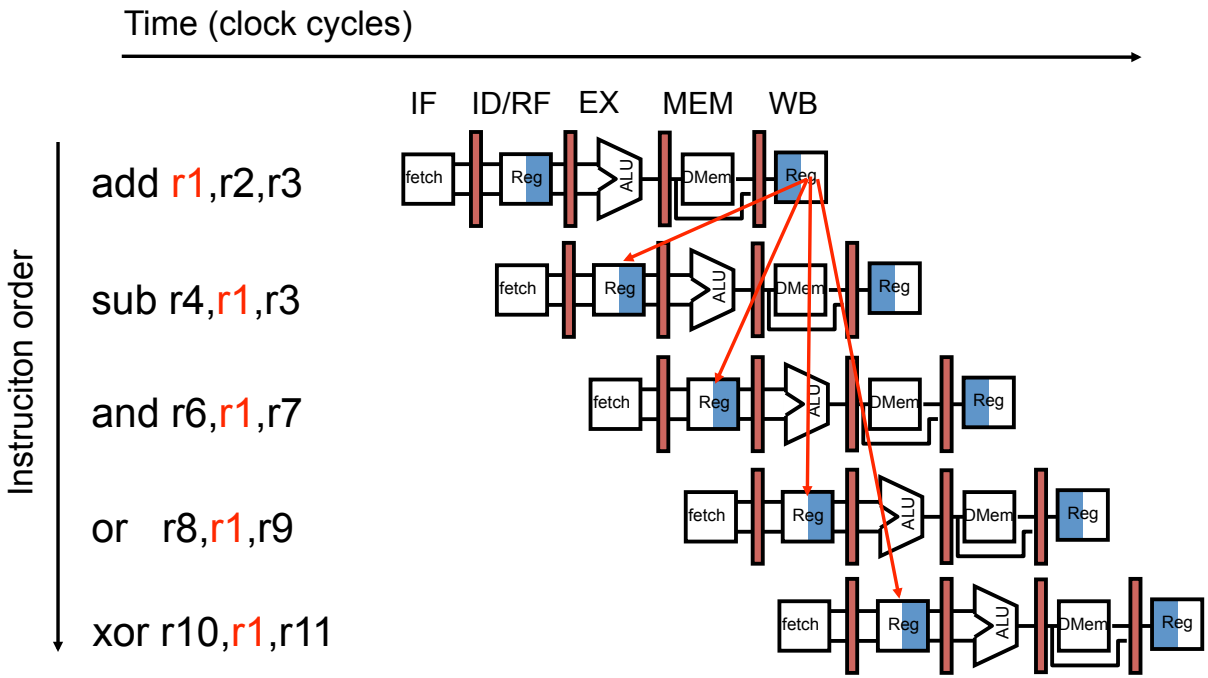
Ideal balanced pipeline

$$Clock\ cycle_{pipelined} = \frac{Clock\ cycle_{unpipelined}}{Pipeline\ depth}$$

$$Speedup_{pipelined} = \frac{1}{1 + Pipeline\ stall\ cycles\ per\ instruction} \times Pipeline\ depth$$

Data Hazards

Read-after-write data hazard through registers



Read-after-write (RAW) hazard

Details

Inst I: add r1, r2, r3
Inst J: sub r4, r1, r3

- Instruction I precedes instruction J in program order
- Instruction I produces result used by instruction J
- Instruction J is data-dependent on instruction I
- Result is not actually committed in register r1 in simple 5-stage pipeline until instruction I finishes the WB stage
- Value of result actually produced earlier, i.e. during the EX stage of instruction I

Write-after-read (WAR) hazard

Details

Inst I: sub r4, r1, r3
Inst J: add r1, r2, r3

- Instruction I precedes instruction J in program order
- Instruction I reads the register written by instruction J
- If instruction I reads r1 in cycle C, instruction J writes r1 in cycle C+4
- No hazard in simple 5-stage pipeline
- Hazard may occur if we attempt to reorder the two instructions. Will see examples later in the course ...

Write-after-write (WAW) hazard

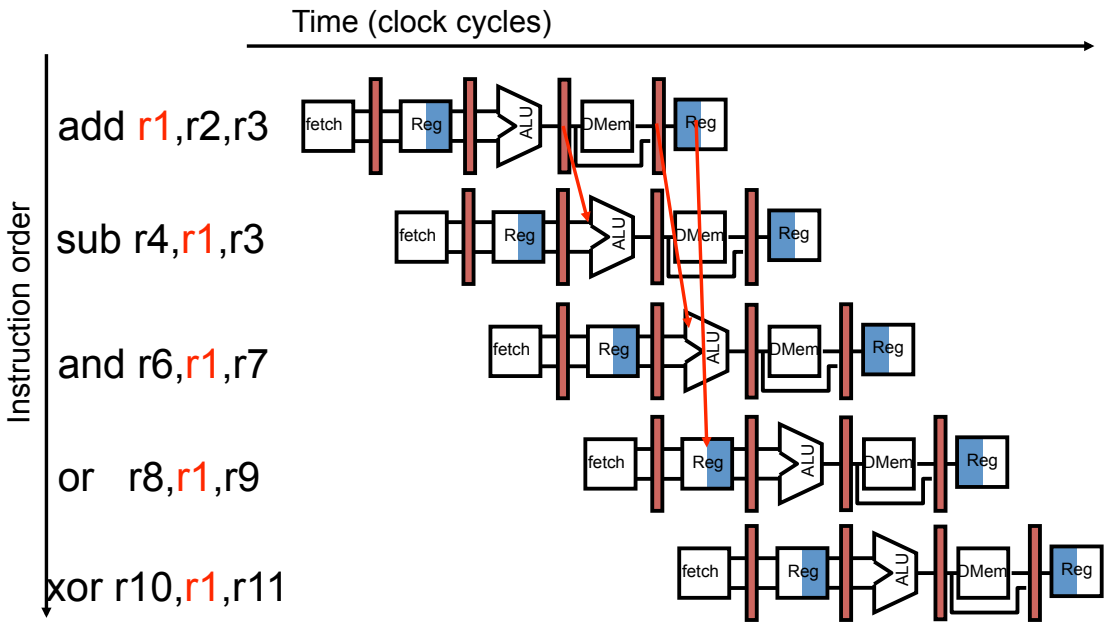
Details

Inst I: add r1, r2, r3
Inst J: add r1, r1, r4

- Instruction I precedes instruction J in program order
- Instruction I writes in the same register as instruction J
- If instruction I writes r1 in cycle C, instruction J writes r1 in cycle C+1
- No actual hazard in simple 5-stage pipeline
- Hazard may occur if we attempt to reorder the two instructions.

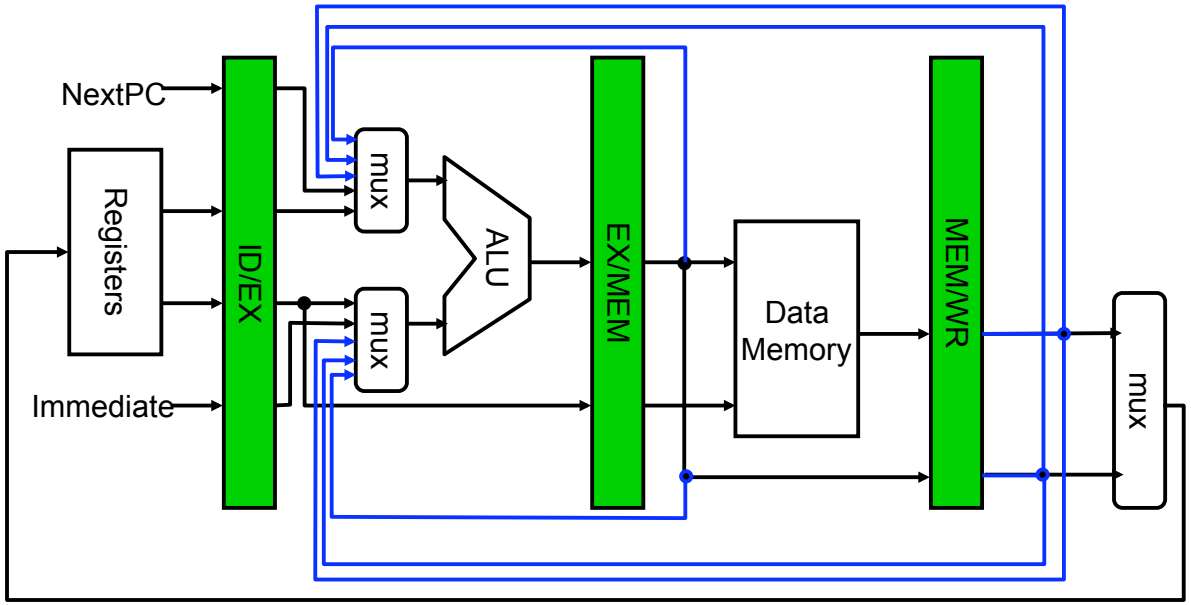
Forwarding

Exploit early production of results in pipeline



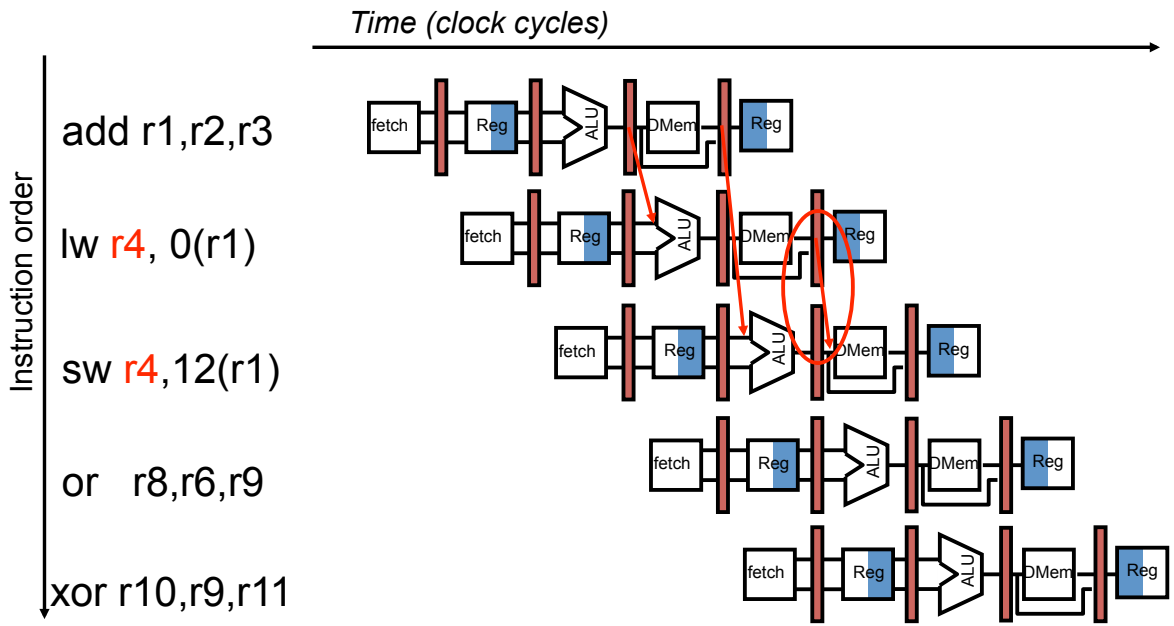
Forwarding control

Additional multiplexers select ALU input



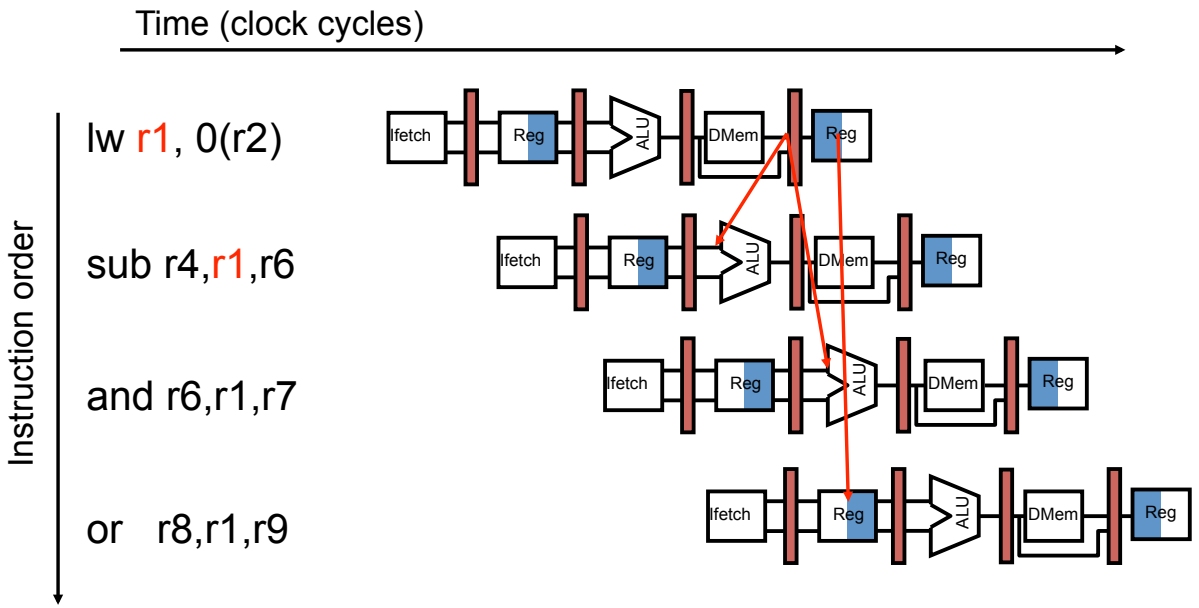
RAW hazards through memory

Store following load to same memory location



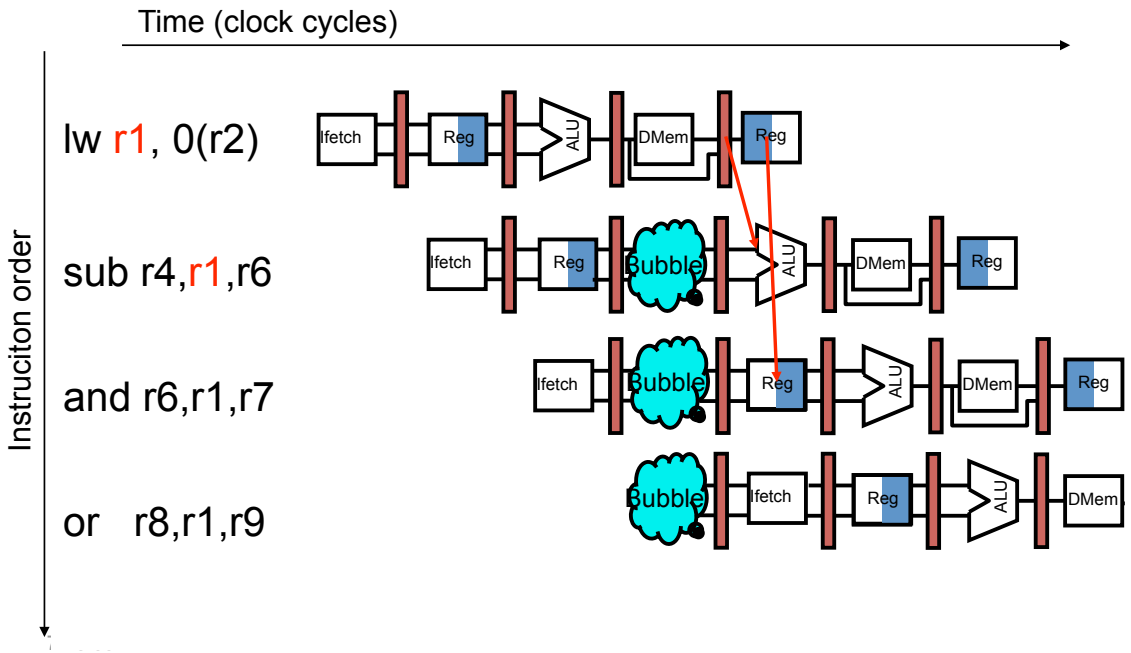
Forwarding can not resolve all hazards

Load-use hazard



Resolving load-use hazard

Bubble to provide chance for forwarding



Resolving load-use hazard

Control logic

//Instruction needs to stall in the EX stage
if (ID/EX.MemRead and // A load instruction has been issued a cycle ago
((ID/EX.RegisterRd = IF/ID.RegisterRs) or // ld destination is source A
(ID/EX.RegisterRd = IF/ID.RegisterRt))) // or ld destination is source B
stall pipeline // zero out all control signals, thwarts EX, MEM, WB stages

Pipeline forwarding data logic summary

Forwarding from ALU (EX/MEM) and memory (MEM/WB)

Pipeline register containing source instruction	Opcode of source instruction	Pipeline register containing destination instruction	Opcode of destination instruction	Destination of the forwarded result	Comparison (if equal then forward)
EX/MEM	Register-register ALU	ID/EX	Register-register ALU, ALU immediate, load, store, branch	Top ALU input	EX/MEM.IR16..20 = ID/EX.IR6..10
EX/MEM	Register-register ALU	ID/EX	Register-register ALU	Bottom ALU input	EX/MEM.IR16..20 = ID/EX.IR11..15
MEM/WB	Register-register ALU	ID/EX	Register-register ALU, ALU immediate, load, store, branch	Top ALU input	MEM/WB.IR16..20 = ID/EX.IR6..10
MEM/WB	Register-register ALU	ID/EX	Register-register ALU	Bottom ALU input	MEM/WB.IR16..20 = ID/EX.IR11..15
EX/MEM	ALU immediate	ID/EX	Register-register ALU, ALU immediate, load, store, branch	Top ALU input	EX/MEM.IR11..15 = ID/EX.IR6..10
EX/MEM	ALU immediate	ID/EX	Register-register ALU	Bottom ALU input	EX/MEM.IR11..15 = ID/EX.IR11..15

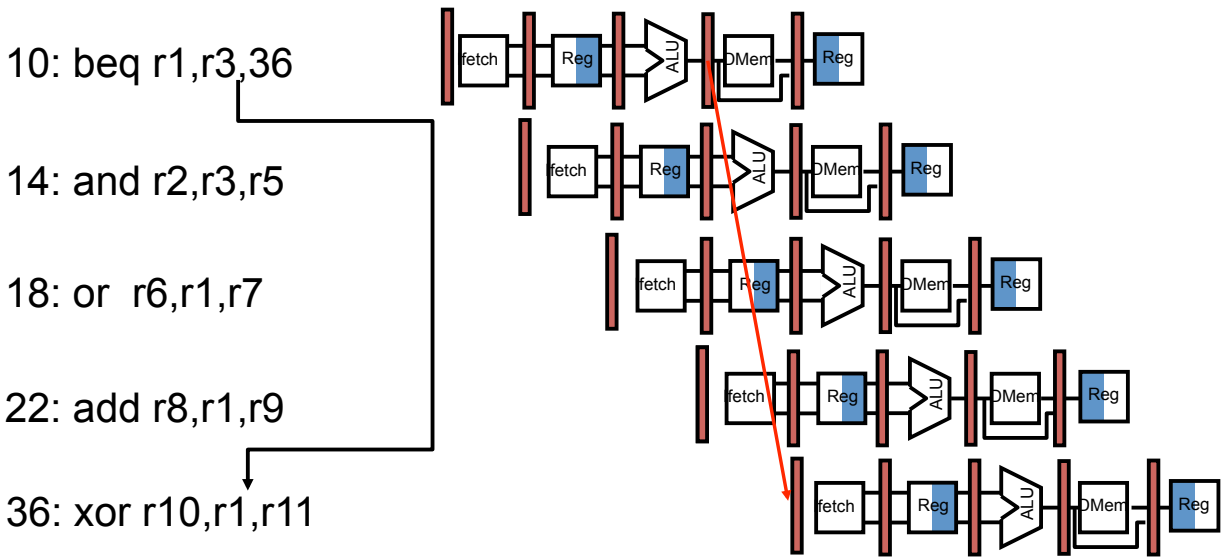
Pipeline forwarding data logic summary (cont.)

Forwarding from ALU (EX/MEM) and memory (MEM/WB)

Pipeline register containing source instruction	Opcode of source instruction	Pipeline register containing destination instruction	Opcode of destination instruction	Destination of the forwarded result	Comparison (if equal then forward)
MEM/WB	ALU immediate	ID/EX	Register-register ALU, ALU immediate, load,	Top ALU input	MEM/WB.IR11..15 = ID/EX.IR6..10
MEM/WB	ALU immediate	ID/EX	Register-register ALU	Bottom ALU input	MEM/WB.IR11..15 = ID/EX.IR11..15
MEM/WB	Load	ID/EX	Register-register ALU, ALU immediate, load,	Top ALU input	MEM/WB.IR11..15 = ID/EX.IR6..10
MEM/WB	Load	ID/EX	Register-register ALU	Bottom ALU input	MEM/WB.IR11..15 = ID/EX.IR11..15

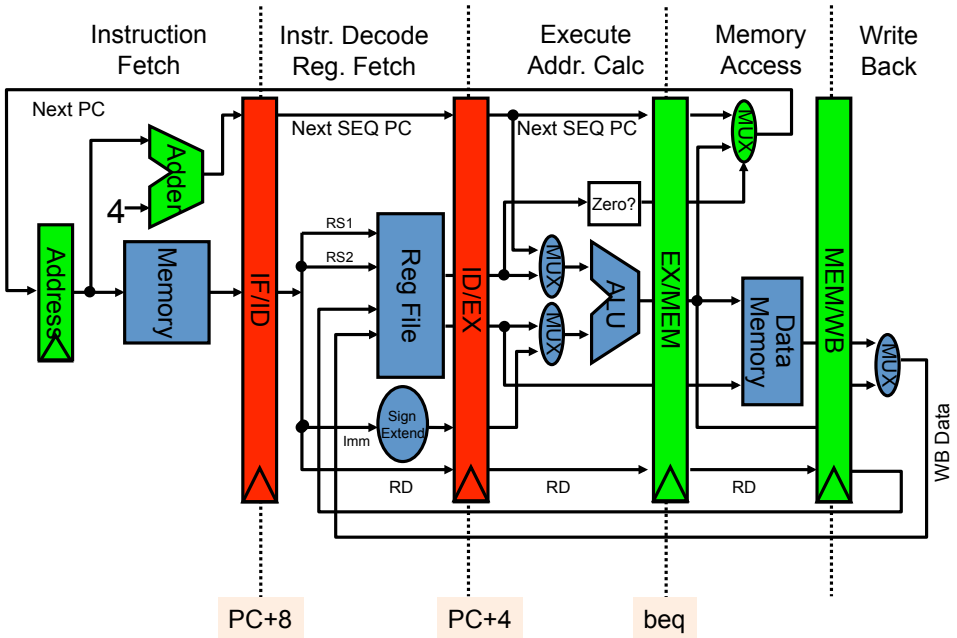
Resolving branches in pipeline

Control-dependent instructions



Understanding control hazards

MIPS datapath

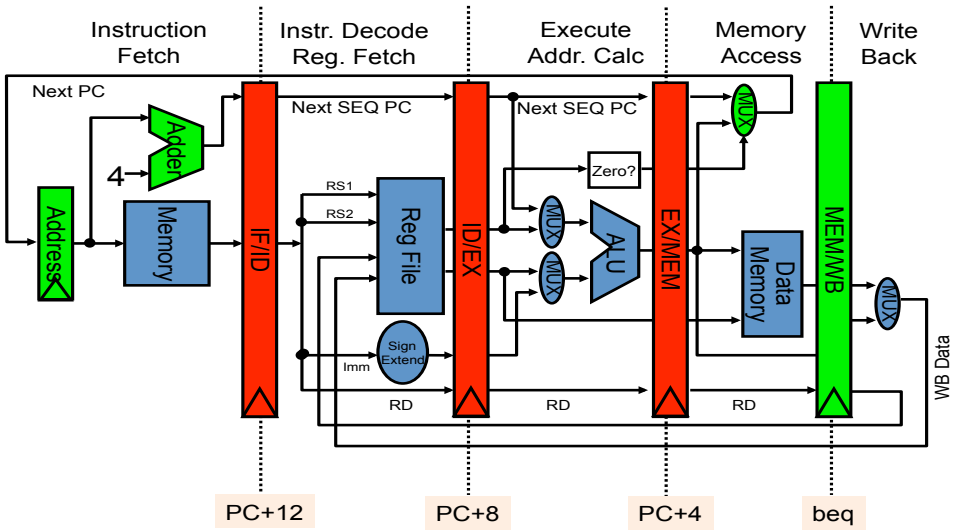


Branch execution

- Comparison with zero and target address calculation at EX stage
- 2 stall cycles

Understanding control hazards

MIPS datapath



Branch execution

- Branch taken decision plus potential branch target out of EX stage
- Next PC forwarded from MEM stage through multiplexer
- 1 more stall cycle for a total of 3

Reducing branch stall impact

Impact of branches on performance

- Branch frequency (conditional, unconditional)
  - ca. 20% for integer programs
  - ca. 10% for floating point programs

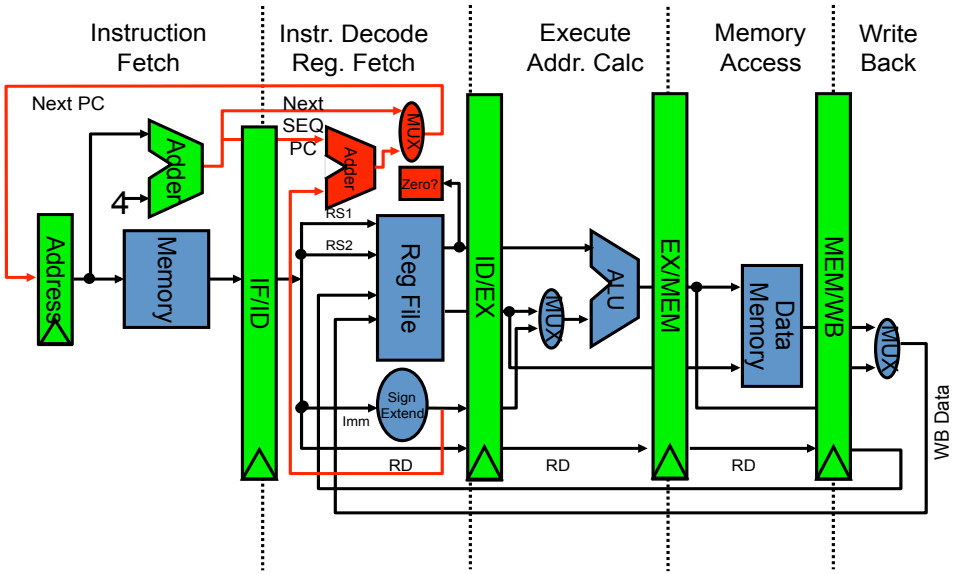
$$Stall\ CPI\ from\ branches = branch\ frequency \times branch\ penalty$$

$$Speedup_{pipeline} = \frac{Pipeline\ depth}{1 + branch\ frequency \times branch\ penalty}$$

- Max speedup drops from 5.0 to 3.1 (int), or 3.8 (fp)

Reducing branch stall impact

HW solution



Explanation

- Comparison with zero happens at EX stage
- Move comparison to EX stage
- May increase cycle time!

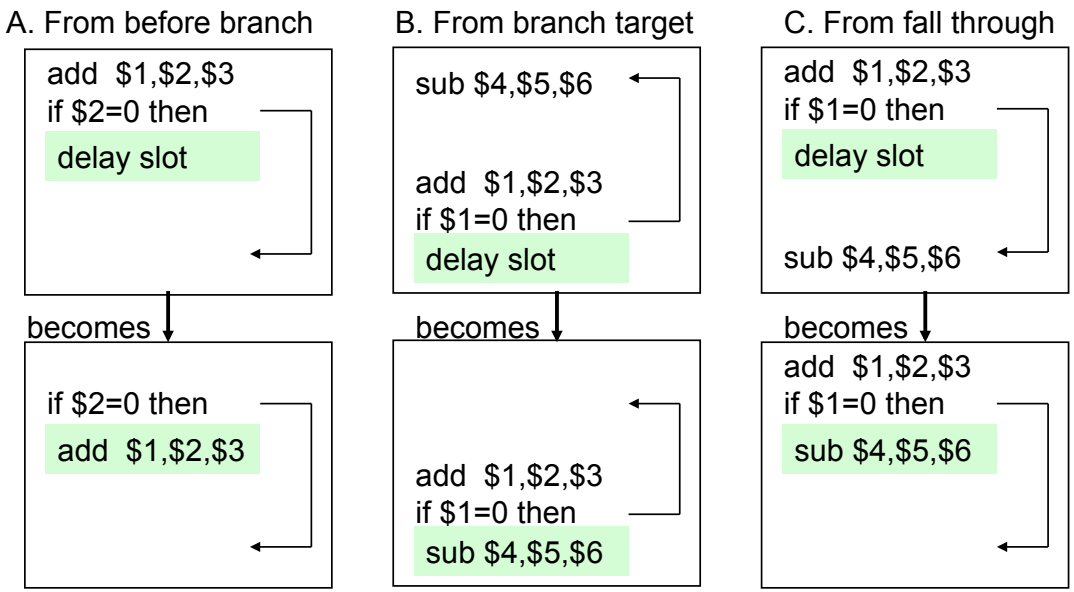
Reducing branch stall impact

HW/SW solution

- Delayed branches always execute the instruction in the slot following the branch (PC+4)
- Instruction two slots down (PC+8) affected by the branch
- Software (compiler) tasked with filling delay slots
- Choices are from before the branch, from the target (branch taken), or from the fall through path (branch not taken)

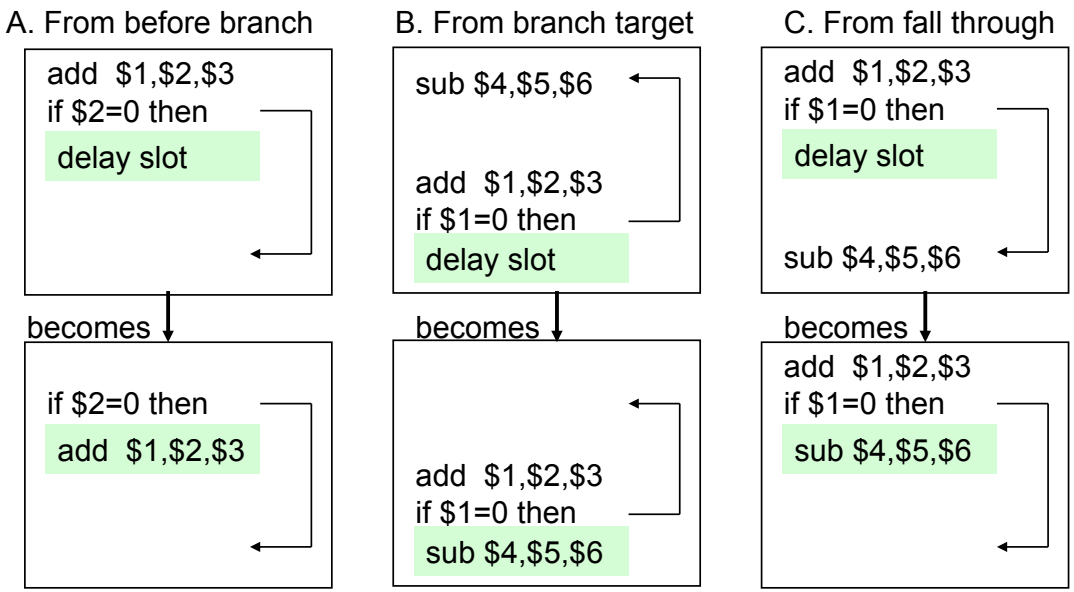
Options for filling delay slot

Three paths to look for instructions



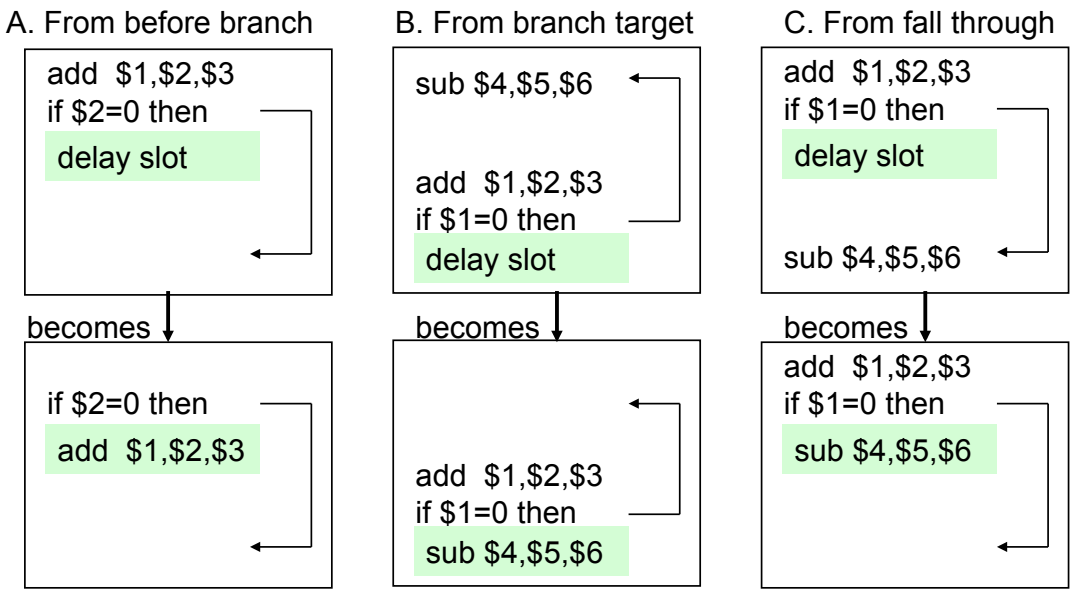
Options for filling delay slot

A: reduces instructions and improves performance



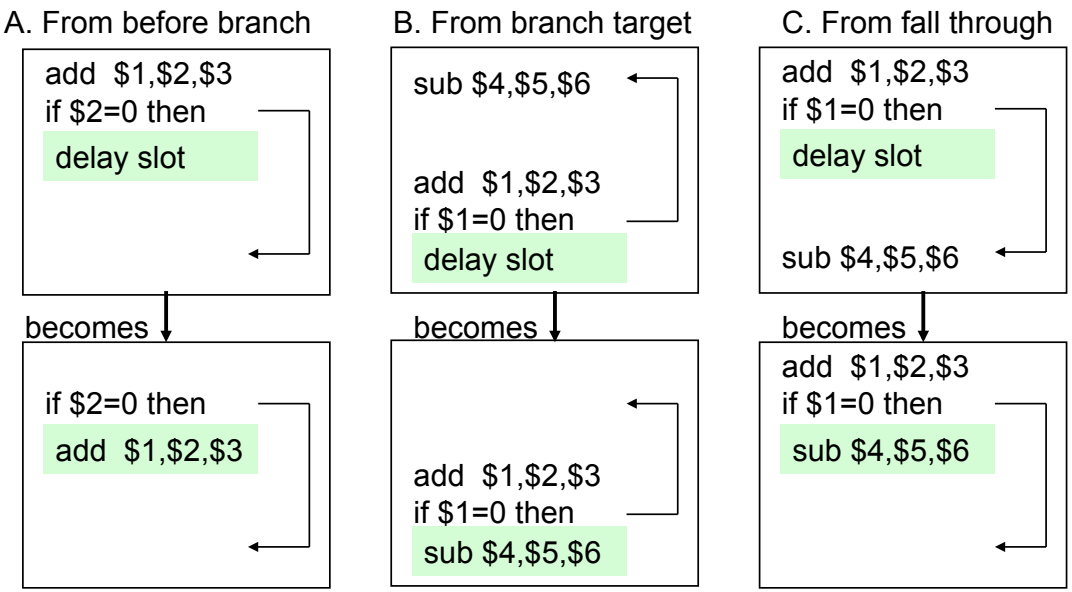
Options for filling delay slot

B: may require to copy instruction if branch taken



Options for filling delay slot

C: conditionally dependent instruction should not execute



Exception handling in pipelines

Exception difficulties in pipelining

- Exceptions may occur in different stages (e.g. overflows at EX, page faults at MEM, I/O device requests anywhere)
- Some exceptions are restartable

Instruction flush and restart

- Flush instructions following instruction causing the exception
- Start execution of exception handler from new address
  - Instruction flush is done using nop or trap (IF) or zeroing of control signals (ID, EX, MEM)
- Save address of offending instruction plus 4, if restartable

Precise vs. imprecise exceptions

- Instructions before offending instruction have committed results to registers/memory
- Offending and following instructions execute from the beginning
- Exceptions may happen out-of-order
- LW followed by an ADD
- HW maintains exception status vector for ”early” exceptions
- Exceptions are ”processed” in WB stage
- Status vector is read to cancel register or memory update