18-640 Foundations of Computer Architecture

Lecture 2:
"Review of Pipelined Processor Design"

John Paul Shen
August 28, 2014

- Required Reading Assignment:
  - **Chapter 2 of Shen and Lipasti (SnL).**
- Recommended Reference:
  - "Optimum Power/Performance Pipeline Depth" by A. Hartstein and Thomas R. Puzak, MICRO 2003.

Electrical & Computer ENGINEERING
Carnegie Mellon University

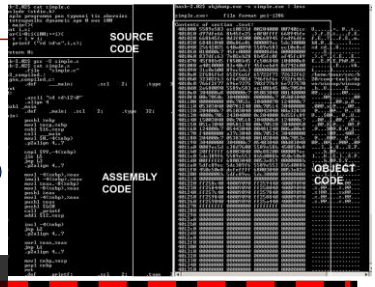
18-640 Foundations of Computer Architecture

Lecture 2:
"Review of Pipelined Processor Design"

A. Pipelining Fundamentals
B. Pipelined Processor Organization
   a. Balancing Pipe Stages
   b. Unifying Instruction Types
   c. Resolving Pipeline Hazards
C. Pipelined Processor Performance
   a. ALU Instruction Interlock & Penalty
   b. Load Instruction Interlock & Penalty
   c. Branch Instruction Interlock & Penalty

Electrical & Computer ENGINEERING
Carnegie Mellon University

Computer Architecture: Instruction Set Architecture

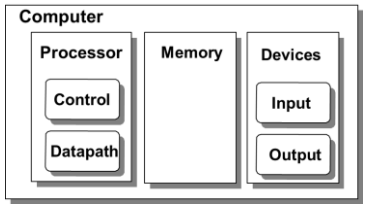


Application Software
- Program development
- Program compilation

SPECIFICATION — Instruction Set Architecture (ISA)

Instruction-Set Processor Design

Hardware Technology
- Program execution
- Machine performance

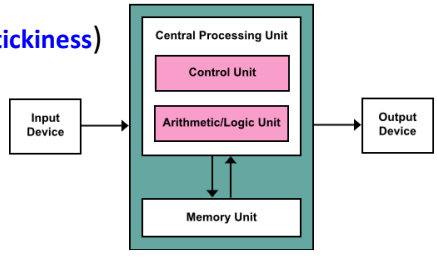


IMPLEMENTATION

18-640 Course Coverage: Processor Designs

Persistence of Von Neumann Model (**Legacy SW Stickiness**)

1. One CPU
2. Monolithic Memory
3. Sequential Execution Semantics



Evolution of Von Neumann Implementations:

- ~~SP: Sequential Processors~~ (direct implementation of sequential execution)
- PP: Pipelined Processors (overlapped execution of in-order instructions)
- SSP: Superscalar Processors (out-of-order execution of multiple instructions)
- MCP: Multi-core Processors = CMP: Chip Multiprocessors (concurrent multi-threads)
- SMP: Symmetric Multiprocessors (concurrent multi-threads and multi-programs)

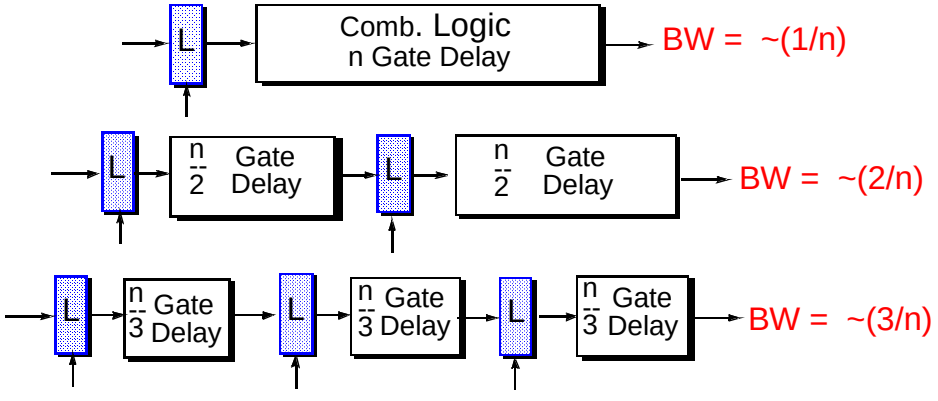
A. Pipelining Fundamentals

- Motivation:
  - Increase throughput with little increase in hardware.

Bandwidth or Throughput = Performance

- Bandwidth (BW) = no. of tasks/unit time
- For a system that operates on one task at a time:
  - BW = 1/delay (latency)
- BW can be increased by pipelining if many operands exist which need the same operation, i.e. many repetitions of the same task are to be performed.
- Latency required for each task remains the same or may even increase slightly.

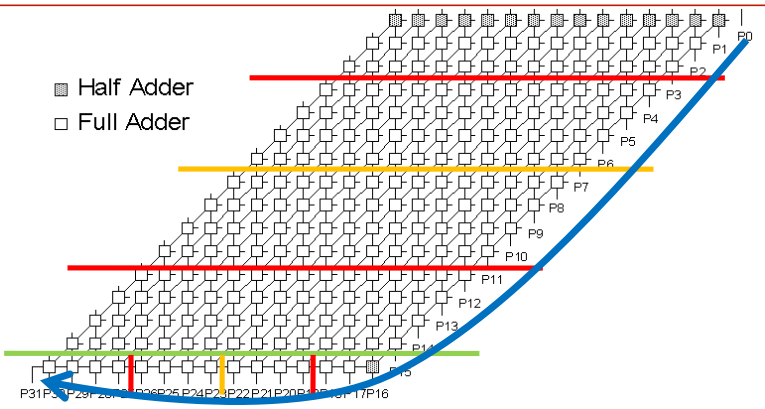
Ideal Pipelining Illustrated



- Bandwidth increases linearly with pipeline depth
- Latency increases by latch delays

Example: Integer Multiplier

[Source: J. Hayes, Univ. of Michigan]



- 16x16 combinational multiplier
  - ISCAS-85 C6288 standard benchmark
- Tools: Synopsys DC/LSI Logic 110nm gflxp ASIC

Example: Integer Multiplier

Configuration	Delay	MPS	Area (FF/wiring)	Area Increase
Combinational	3.52ns	284	7535 (--/1759)	
2 Stages	1.87ns	534 (1.9x)	8725 (1078/1870)	16%
4 Stages	1.17ns	855 (3.0x)	11276 (3388/2112)	50%
8 Stages	0.80ns	1250 (4.4x)	17127 (8938/2612)	127%

- Pipeline efficiency
  - 2-stage: nearly double throughput; marginal area cost
  - 4-stage: 75% efficiency; area still reasonable
  - 8-stage: 55% efficiency; area more than doubles
- Tools: Synopsys DC/LSI Logic 110nm gflxp ASIC

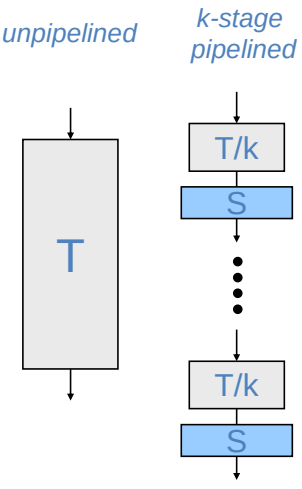
Pipelining Performance Model

- Starting from an un-pipelined version with propagation delay T and BW = 1/T

$P_{pipelined}=BW_{pipelined} = 1 / (T/ k +S)$

where

S = delay through latch and overhead



Hardware Cost Model

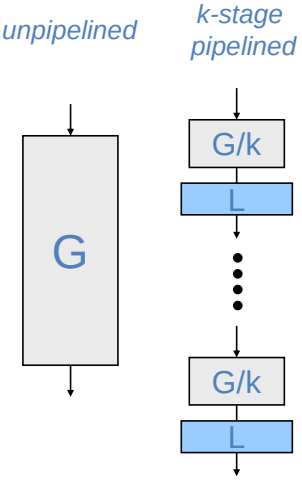
- Starting from an un-pipelined version with hardware cost G

$Cost_{pipelined} = kL + G$

where

L = cost of adding each latch, and

k = number of stages



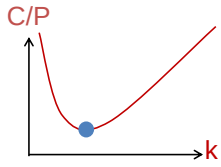
[Peter M. Kogge, 1981]

Cost/Performance Trade-off

Cost/Performance:

C/P = [Lk + G] / [1/(T/k + S)] = (Lk + G) (T/k + S)

= LT + GS + LSk + GT/k



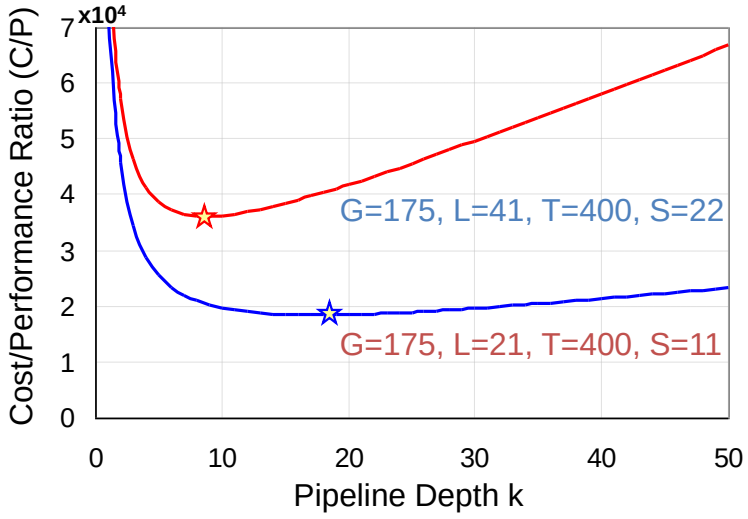
Optimal Cost/Performance: find min. C/P w.r.t. choice of k

$$\frac{d}{dk}\left(\frac{Lk+G}{\frac{1}{\frac{T}{k}+S}}\right) = 0 + 0 + LS - \frac{GT}{k^2}$$

$$LS - \frac{GT}{k^2} = 0$$

$$k_{opt} = \sqrt{\frac{GT}{LS}}$$

"Optimal" Pipeline Depth (k_{opt}) Examples



Pipelining Idealistic Assumptions

- Uniform Sub-computations

  The computation to be pipelined can be evenly partitioned into uniform-latency sub-operations

- Repetition of Identical Computations

  The same computations are to be performed repeatedly on a large number of different inputs

- Repetition of Independent Computations

  All the repetitions of the same computation are mutually independent, i.e. no data dependence and no resource conflicts

Instruction Pipeline Design

- Uniform Sub-computations ... NOT!

  ⇒ balancing pipeline stages

  - stage quantization to yield balanced pipe stages
  - minimize internal fragmentation (some waiting stages)

- Identical Computations ... NOT!

  ⇒ unifying instruction types

  - coalescing instruction types into one multi-function pipe
  - minimize external fragmentation (some idling stages)

- Independent Computations ... NOT!

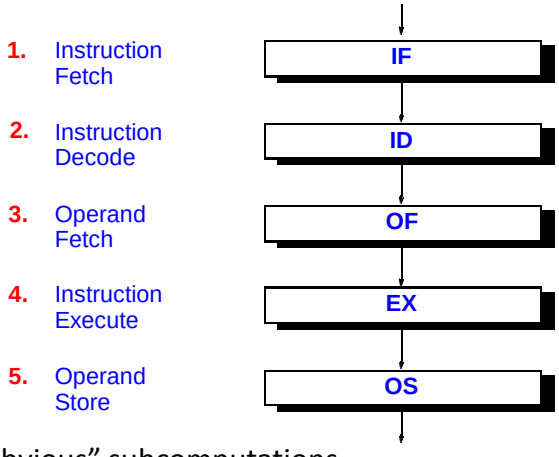
  ⇒ resolving pipeline hazards

  - inter-instruction dependence detection and resolution
  - minimize performance lose due to pipeline stalls

Instruction Pipelining

- The "computation" to be pipelined.
  - Instruction Fetch (IF)
  - Instruction Decode (ID)
  - Operand(s) Fetch (OF)
  - Instruction Execution (EX)
  - Operand Store (OS)
  - Update Program Counter (PC)

A Generic Processor Pipeline

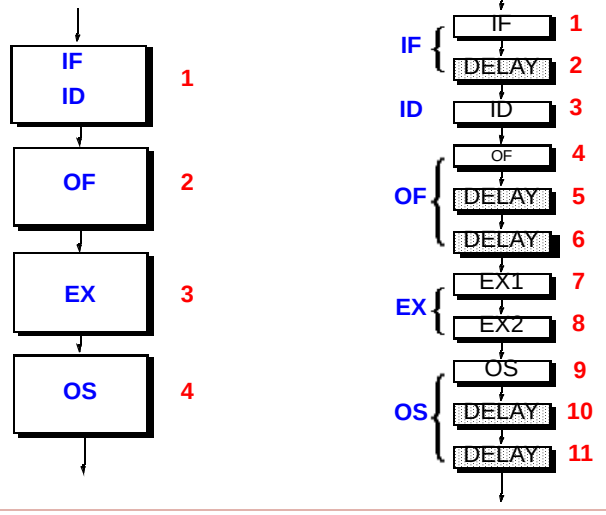


- Based on "obvious" subcomputations

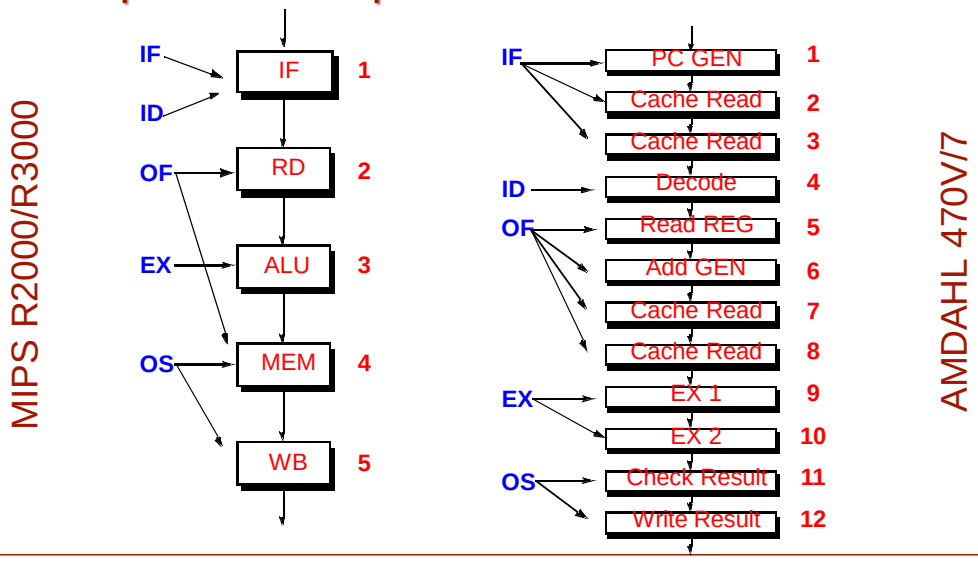
B.a. Balancing Pipeline Stages

- Two Methods for Stage Quantization:
  - Merging of multiple sub-computations into one.
  - Subdividing a sub-computation into multiple sub-computations.
- Recent Trends:
  - Deeper pipelines with more and more stages.
  - Multiplicity of different sub-pipelines.
  - Pipelining of memory access becomes tricky.

Granularity of Pipeline Stages Can Vary



Actual Pipeline Examples



B.b. Unifying Instruction Types

- Procedure:
  1. Classification of Instruction Types: Analyze the sequence of register transfers required by each instruction type.
  2. Coalescing Resource Requirements: Find commonality across instruction types and merge them to share the same pipeline stage and hardware resource.
  3. Instruction Pipeline Implementation: If there exists flexibility, shift or reorder some register transfers to facilitate further merging.

ALU Instruction Specification

Generic subcomputations	1. ALU Instruction Type:	
	Integer instruction	Floating-point instruction
IF	- Fetch instruction (access I-cache)	- Fetch instruction (access I-cache)
ID	- Decode instruction	- Decode instruction
OF	- Access register file	- Access FP register file
EX	- Perform ALU operation	- Perform FP operation
OS	- Write back to reg. file	- Write back to FP reg. file

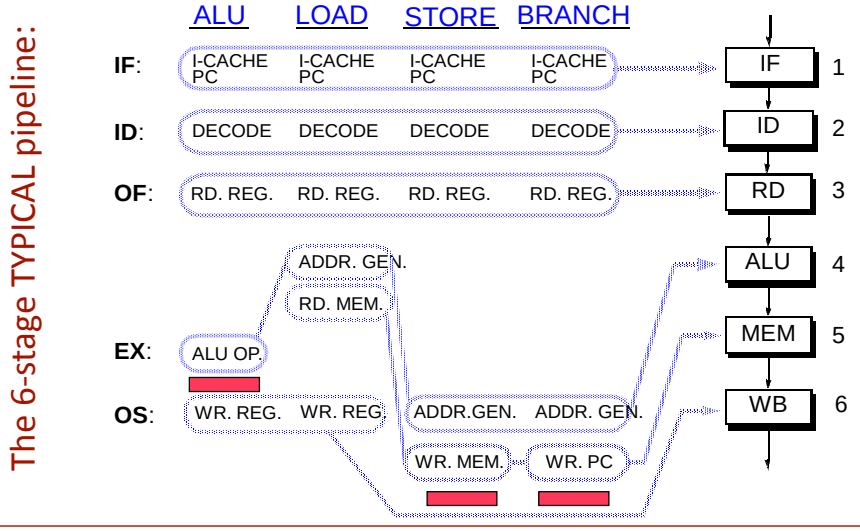
Memory Instruction Specification

Generic subcomputations	2. Load/Store Instruction Type:	
	Load instruction	Store instruction
IF	**- Fetch instruction (access I-cache)**	**- Fetch instruction (access I-cache)**
ID	**- Decode instruction**	**- Decode instruction**
OF	**- Access register file (base address) - Generate effective address (base + offset) - Access (read) memory location (D-cache)**	**- Access register file (register operand, and base address)**
EX	-	-
OS	**- Write back to reg. file**	**- Generate effective address (base + offset) - Access (write) memory location (D-cache)**

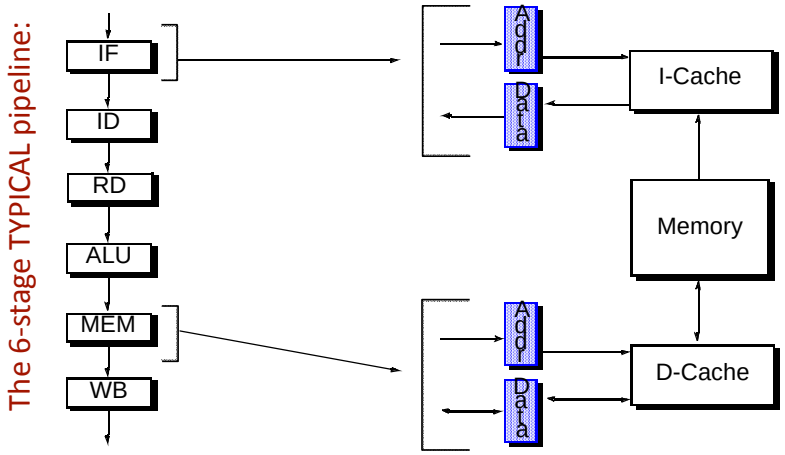
Branch Instruction Specification

Generic subcomputations	3. Branch Instruction Type:	
	Jump (uncond.) instruction	Conditional branch instr.
IF	**- Fetch instruction (access I-cache)**	**- Fetch instruction (access I-cache)**
ID	**- Decode instruction**	**- Decode instruction**
OF	**- Access register file (base address) - Generate effective address (base + offset)**	**- Access register file (base address) - Generate effective address (base + offset)**
EX	**-**	**- Evaluate branch condition**
OS	**- Update program counter with target address**	**- If condition is true, update program counter with target address**

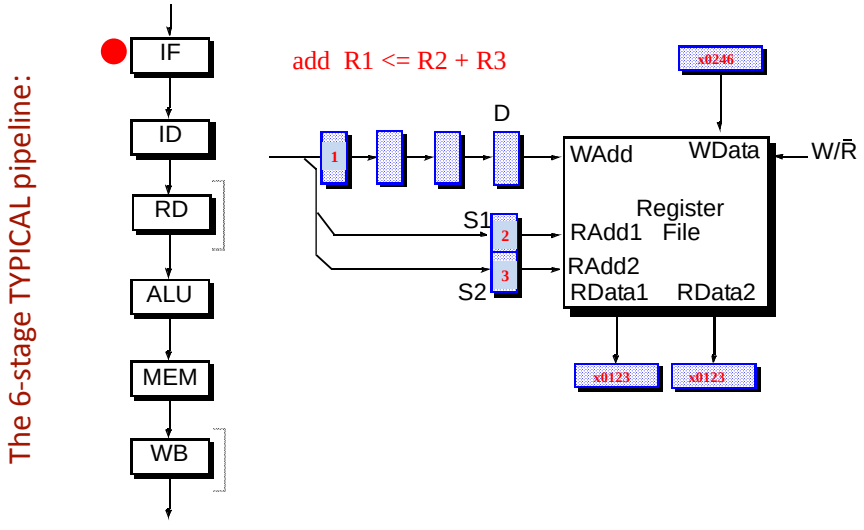
Unifying Instruction Types in a Pipelined Processor



Pipeline Interface to Memory Subsystem

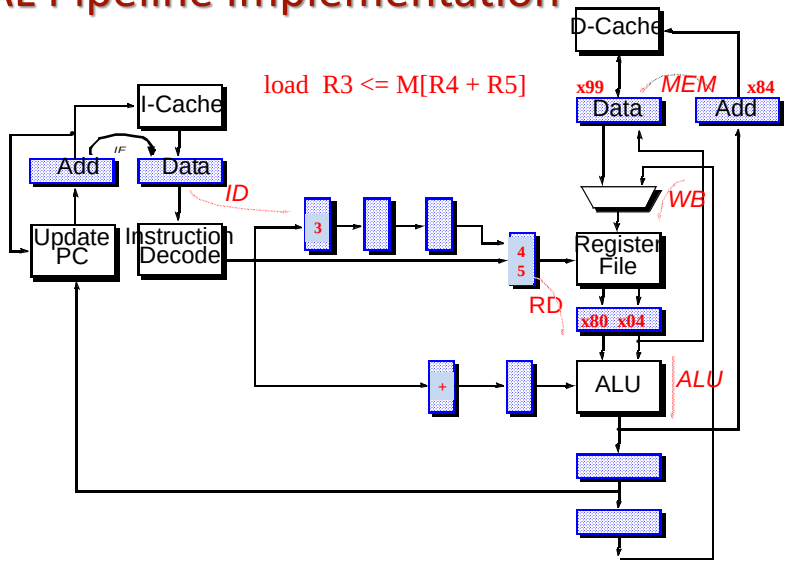


Pipeline Interface to Register File



TYPICAL Pipeline Implementation

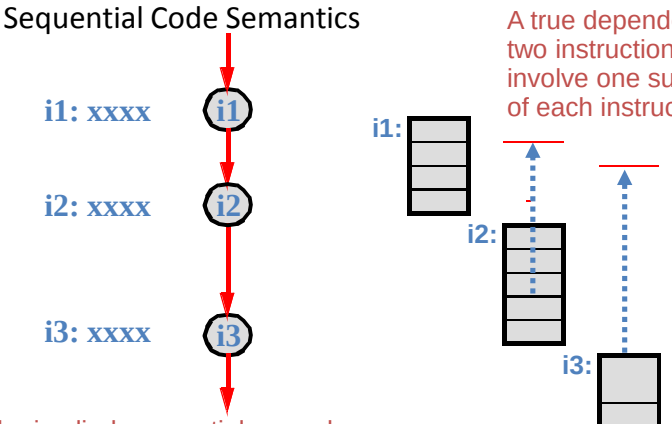
The 6-stage TYPICAL pipeline:



Pipelining Idealistic Assumptions

- ☑ Uniform subcomputations
  - Can pipeline into stages with equal delay
  - Balance pipeline stages
- ☑ Identical computations
  - Can fill pipeline with identical work
  - Unify instruction types
- Independent computations
  - No relationships between work units
  - Resolve Pipeline Hazards
  - Minimize pipeline stalls

Instruction Dependences and Pipeline Hazards



A true dependence between two instructions may only involve one subcomputation of each instruction.

The implied sequential precedences are over-specifications. It is sufficient but not necessary to ensure program correctness.

Inter-Instruction Dependences

- Data dependence

$r_3 \leftarrow r_1$ op r_2
$r_5 \leftarrow r_3$ op r_4

Read-after-Write (RAW)

- Anti-dependence

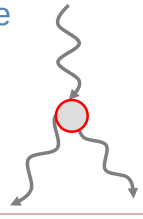
$r_3 \leftarrow r_1$ op r_2
$r_1 \leftarrow r_4$ op r_5

Write-after-Read (WAR)

- Output dependence

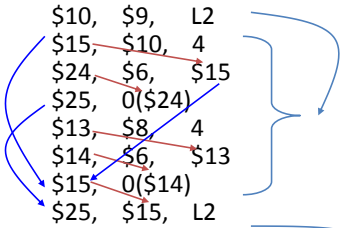
$r_3 \leftarrow r_1$ op r_2
$r_5 \leftarrow r_3$ op r_4
$r_3 \leftarrow r_6$ op r_7

Write-after-Write (WAW)



- Control dependence

Example: Quick Sort for MIPS



```
        bge     $10,  $9,   L2
        mul     $15,  $10,  4
        addu    $24,  $6,   $15
        lw      $25,  0($24)
        mul     $13,  $8,   4
        addu    $14,  $6,   $13
        lw      $15,  0($14)
        bge     $25,  $15,  L2
L1:
        addu    $10,  $10,  1
        . . .
L2:
        addu    $11,  $11,  -1
        . . .
#           for (;(j<high)&&(array[j]<array[low]);++j);
#           $10 = j; $9 = high; $6 = array; $8 = low
```


B.c. Resolving Pipeline Hazards

- Pipeline Hazards:
  - Potential violations of program dependences
  - Must ensure program dependences are not violated
- Hazard Resolution:
  - Static Method: Performed at compiled time in software
  - Dynamic Method: Performed at run time using hardware
- Pipeline Interlock:
  - Hardware mechanisms for dynamic hazard resolution
  - Must detect and enforce dependences at run time

Pipeline Hazards

- Necessary conditions:
  - WAR: write stage earlier than read stage
    - Is this possible in IF-ID-RD-ALU-MEM-WB ?
  - WAW: write stage earlier than write stage
    - Is this possible in IF-ID-RD-ALU-MEM-WB ?
  - RAW: read stage earlier than write stage
    - Is this possible in IF-ID-RD-ALU-MEM-WB?
- If conditions not met, no need to resolve
- Check for both **register** and **memory**

Hazards due to Memory Data Dependences

Pipe Stage	ALU Inst.	Load inst.	Store inst.	Branch inst.
1. IF	I-cache PC<PC+4	I-cache PC<PC+4	I-cache PC<PC+4	I-cache PC<PC+4
2. ID	decode	decode	decode	decode
3. RD	read reg.	read reg.	read reg.	read reg.
4. ALU	ALU op.	addr. gen.	addr. gen.	addr. gen. cond. gen.
5. MEM	-----	read mem.	write mem.	PC<-br. addr.
6. WB	write reg.	write reg.	-----	-----

Hazards due to Register Data Dependences

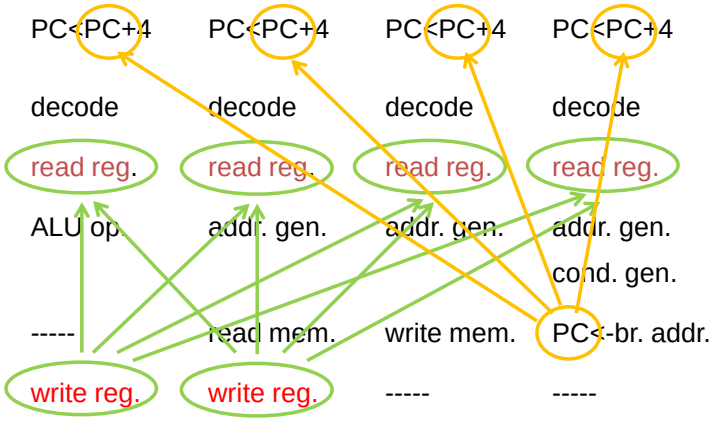
Pipe Stage	ALU Inst.	Load inst.	Store inst.	Branch inst.
1. IF	I-cache PC<PC+4	I-cache PC<PC+4	I-cache PC<PC+4	I-cache PC<PC+4
2. ID	decode	decode	decode	decode
3. RD	read reg.	read reg.	read reg.	read reg.
4. ALU	ALU op.	addr. gen.	addr. gen.	addr. gen. cond. gen.
5. MEM	-----	read mem.	write mem.	PC<-br. addr.
6. WB	write reg.	write reg.	-----	-----

Hazards due to Control Dependences

Pipe Stage	ALU Inst.	Load inst.	Store inst.	Branch inst.
1. IF	I-cache PC<PC+4	I-cache PC<PC+4	I-cache PC<PC+4	I-cache PC<PC+4
2. ID	decode	decode	decode	decode
3. RD	read reg.	read reg.	read reg.	read reg.
4. ALU	ALU op.	addr. gen.	addr. gen.	addr. gen. cond. gen.
5. MEM	-----	read mem.	write mem.	PC<-br. addr.
6. WB	write reg.	write reg.	-----	-----

Inter-Instruction Hazards

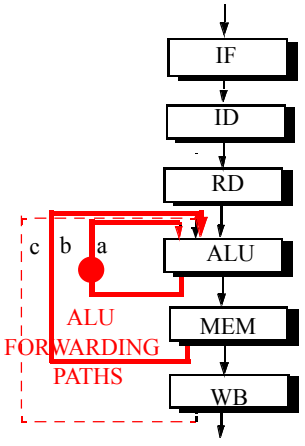
Pipe Stage	ALU Inst.	Load inst.	Store inst.	Branch inst.
1. IF	I-cache PC<PC+4	I-cache PC<PC+4	I-cache PC<PC+4	I-cache PC<PC+4
2. ID	decode	decode	decode	decode
3. RD	read reg.	read reg.	read reg.	read reg.
4. ALU	ALU op.	addr. gen.	addr. gen.	addr. gen. cond. gen.
5. MEM	-----	read mem.	write mem.	PC<-br. addr.
6. WB	write reg.	write reg.	-----	-----



Dealing with Data Hazards

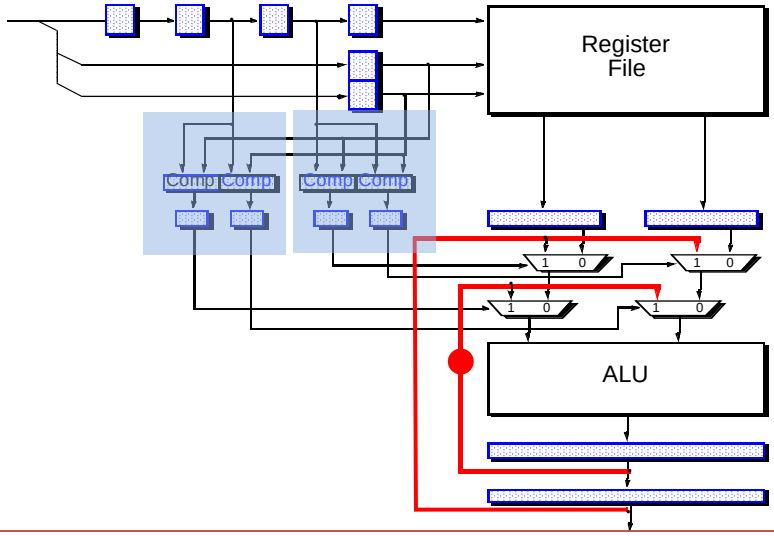
- Must first detect RAW hazards
  - Compare read register specifiers for newer instructions with write register specifiers for older instructions
  - Newer instruction in ID; older instructions in EX, MEM
- Resolve hazard dynamically
  - **Stall** or **forward**
- Not all hazards because
  - No register written (store or branch)
  - No register is read (e.g. addi, jump)
  - Do something only if necessary
    - Use special encodings for these cases to prevent spurious detection

ALU (Leading Instruction) Interlock and Penalty

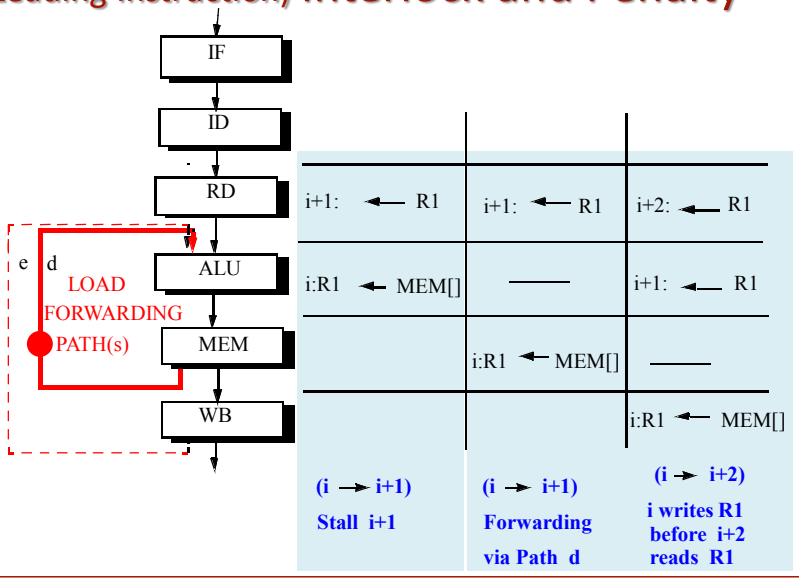


i+1: ← R1	i+2: ← R1	i+3: ← R1
i: R1 ←	i+1: ← R1	i+2: ← R1
	i: R1 ←	i+1: ←
		i: R1 ←
(i → i+1) Forwarding via Path a	(i → i+2) Forwarding via Path b	(i → i+3) i writes R1 before i+3 reads R1

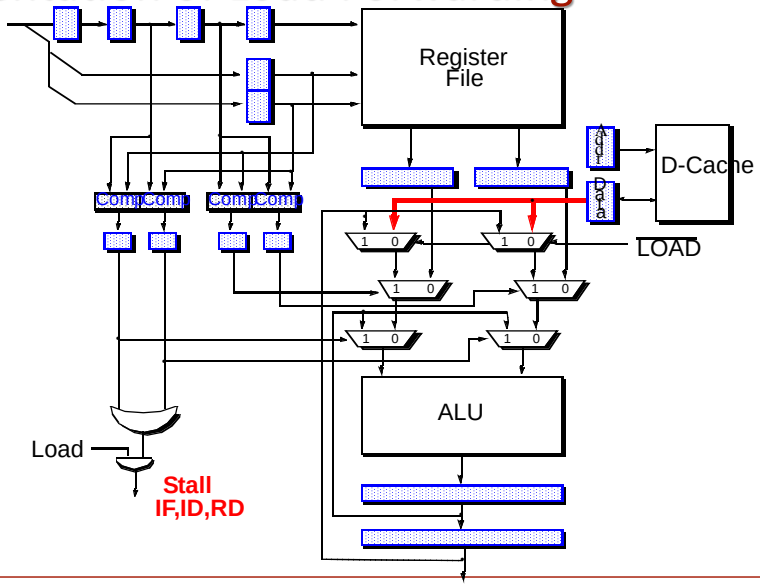
Implementation of ALU Forwarding



Load (Leading Instruction) Interlock and Penalty



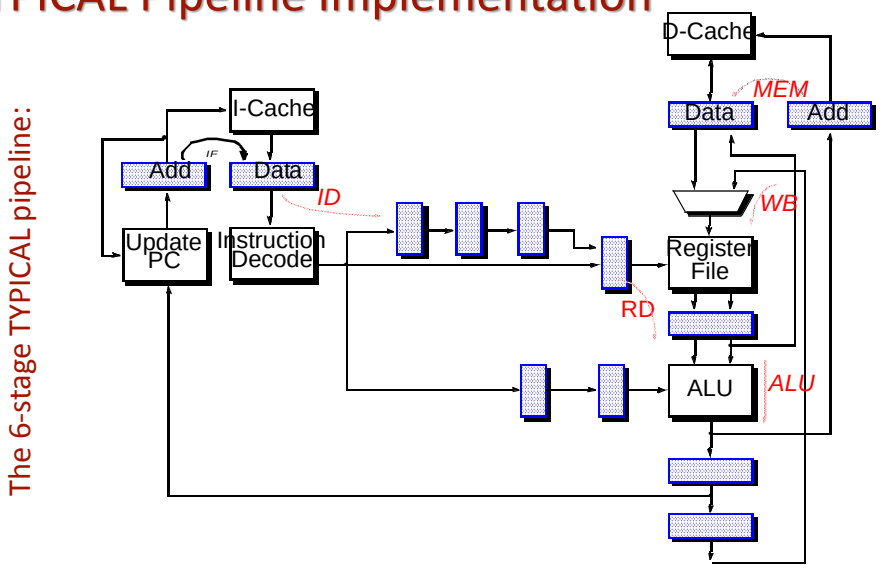
Implementation of Load Forwarding



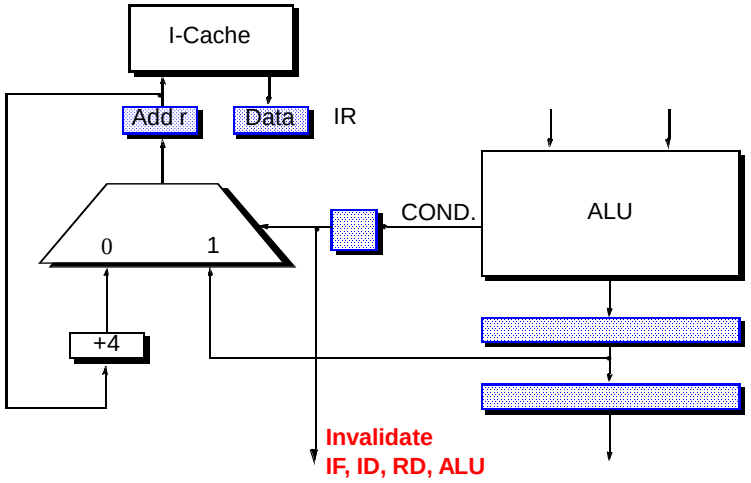
Branch Instruction Hazards

Pipe Stage	ALU Inst.	Load inst.	Store inst.	Branch inst.
1. IF	I-cache PC<PC+4	I-cache PC<PC+4	I-cache PC<PC+4	I-cache PC<PC+4
2. ID	decode	decode	decode	decode
3. RD	read reg.	read reg.	read reg.	read reg.
4. ALU	ALU op.	addr. gen.	addr. gen.	addr. gen. cond. gen.
5. MEM	-----	read mem.	write mem.	PC<-br. addr.
6. WB	write reg.	write reg.	-----	-----

TYPICAL Pipeline Implementation



Implementation of Branch Instruction Interlock



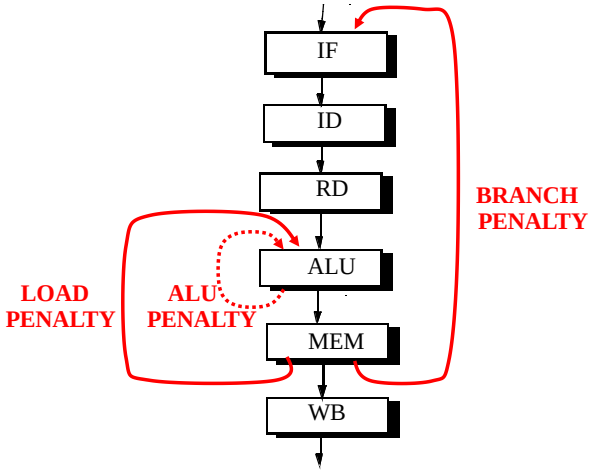
Penalties Due to Stalling for RAW

Leading $Inst_i$	ALU	Load	Branch
Trailing $Inst_j$	ALU, L/S, Br.	ALU, L/S, Br.	ALU, L/S, Br.
Hazard register	Int. Reg. (Ri)	Int. Reg. (Ri)	PC
Register WRITE stage (i)	WB (stage 6)	WB (stage 6)	MEM (stage 5)
Register READ stage (j)	RD (stage 3)	RD (stage 3)	IF (stage 1)
RAW distance or penalty:	3 cycles	3 cycles	4 cycles

Penalties with Forwarding Paths

Leading $Inst_i$	ALU	Load	Branch
Trailing $Inst_j$	ALU, L/S, Br.	ALU, L/S, Br.	ALU, L/S, Br.
Hazard register	Int. Reg. (Ri)	Int. Reg. (Ri)	PC
Register WRITE stage (i)	WB (stage 6)	WB (stage 6)	MEM (stage 5)
Register READ stage (j)	RD (stage 3)	RD (stage 3)	IF (stage 1)
Forward from outputs of:	ALU,MEM,WB	MEM,WB	MEM
Forward to input of:	ALU	ALU	IF
RAW distance or penalty:	0 cycles	1 cycles	4 cycles

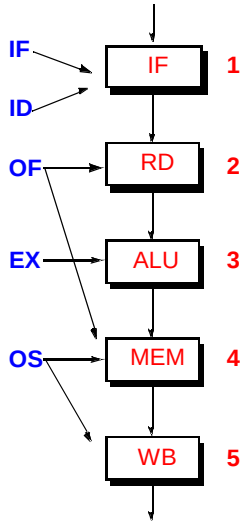
3 Major Penalty Loops of (Scalar) Pipelining

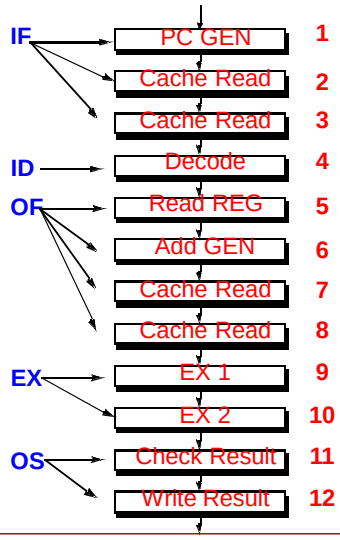


Performance Objective: Reduce CPI as close to 1 as possible.

Actual Pipelined Processor Examples

MIPS R2000/R3000





AMDAHL 470V/7

MIPS R2000/R3000 Pipeline

Separate Adder

Stage	Phase	Function performed
IF	φ_1	Translate virtual instr. addr. using TLB
	φ_2	Access I-cache
RD	φ_1	Return instruction from I-cache, check tags & parity
	φ_2	Read RF; if branch, generate target
ALU	φ_1	Start ALU op; if branch, check condition
	φ_2	Finish ALU op; if ld/st, translate addr
MEM	φ_1	Access D-cache
	φ_2	Return data from D-cache, check tags & parity
WB	φ_1	Write RF
	φ_2	

IBM RISC Experience [Agerwala and Cocke 1987]

- Internal IBM study: Limits of a scalar pipeline?
- Memory Bandwidth
  - Fetch 1 instr/cycle from I-cache
  - 40% of instructions are load/store (D-cache)
- Code characteristics (dynamic)
  - Loads – 25%
  - Stores 15%
  - ALU/RR – 40%
  - Branches – 20%
    - 1/3 unconditional (always taken
    - 1/3 conditional taken, 1/3 conditional not taken

IBM Experience

- Cache Performance
  - Assume 100% hit ratio (upper bound)
  - Cache latency: I = D = 1 cycle default
- Load and branch scheduling
  - Loads
    - 25% cannot be scheduled (delay slot empty)
    - 65% can be moved back 1 or 2 instructions
    - 10% can be moved back 1 instruction
  - Branches
    - Unconditional – 100% schedulable (fill one delay slot)
    - Conditional – 50% schedulable (fill one delay slot)

CPI Optimizations

- Goal and impediments
  - CPI = 1, prevented by pipeline stalls
- No cache bypass of RF, no load/branch scheduling
  - Load penalty: 2 cycles: 0.25 x 2 = 0.5 CPI
  - Branch penalty: 2 cycles: 0.2 x 2/3 x 2 = 0.27 CPI
  - Total CPI: 1 + 0.5 + 0.27 = 1.77 CPI
- Bypass, no load/branch scheduling
  - Load penalty: 1 cycle: 0.25 x 1 = 0.25 CPI
  - Total CPI: 1 + 0.25 + 0.27 = 1.52 CPI

More CPI Optimizations

- Bypass, scheduling of loads/branches
  - Load penalty:
    - 65% + 10% = 75% moved back, no penalty
    - 25% => 1 cycle penalty
    - 0.25 x 0.25 x 1 = 0.0625 CPI
  - Branch Penalty
    - 1/3 unconditional 100% schedulable => 1 cycle
    - 1/3 cond. not-taken, => no penalty (predict not-taken)
    - 1/3 cond. Taken, 50% schedulable => 1 cycle
    - 1/3 cond. Taken, 50% unschedulable => 2 cycles
    - 0.25 x [1/3 x 1 + 1/3 x 0.5 x 1 + 1/3 x 0.5 x 2] = 0.167
- Total CPI: 1 + 0.063 + 0.167 = 1.23 CPI

Simplify Branches

- Assume 90% can be PC-relative
  - No register indirect, no register access
  - Separate adder (like MIPS R3000)
  - Branch penalty reduced
- Total CPI: 1 + 0.063 + 0.085 = 1.15 CPI = 0.87 IPC

15% Overhead from program dependences

PC-relative	Schedulable	Penalty
Yes (90%)	Yes (50%)	0 cycle
Yes (90%)	No (50%)	1 cycle
No (10%)	Yes (50%)	1 cycle
No (10%)	No (50%)	2 cycles

Limits of Pipelining

- IBM RISC Experience
  - Control and data dependences add 15%
  - Best case CPI of 1.15, IPC of 0.87
  - Deeper pipelines (higher frequency) magnify dependence penalties
- This analysis assumes 100% cache hit rates
  - Hit rates approach 100% for some programs
  - Many important programs have much worse hit rates

Processor Performance

$$\text{1/Processor Performance} = \frac{\text{Time}}{\text{Program}}$$

$$= \underbrace{\frac{\text{Instructions}}{\text{Program}}}_{\text{(path length)}} \times \underbrace{\frac{\text{Cycles}}{\text{Instruction}}}_{\text{(CPI)}} \times \underbrace{\frac{\text{Time}}{\text{Cycle}}}_{\text{(cycle time)}}$$

- In the 1980's (decade of pipelining):
  - CPI: 5.0 => 1.15
- In the 1990's (decade of superscalar):
  - CPI: 1.15 => 0.5 (best case)
- In the 2000's (decade of multicore):
  - Core CPI unchanged; chip CPI scales with #cores

Limitations of (Scalar) Pipelined Processors

- Inefficiencies of Very Deep pipelines
  Clocking overheads
  Longer hazards and stalls
- Upper Bound on Scalar Pipeline Throughput
  Limited by IPC = 1
- Inefficient Unification Into Single Pipeline
  Long latency for each instruction
  Hazards and associated stalls
- Performance Lost Due to In-order Pipeline
  Unnecessary stalls