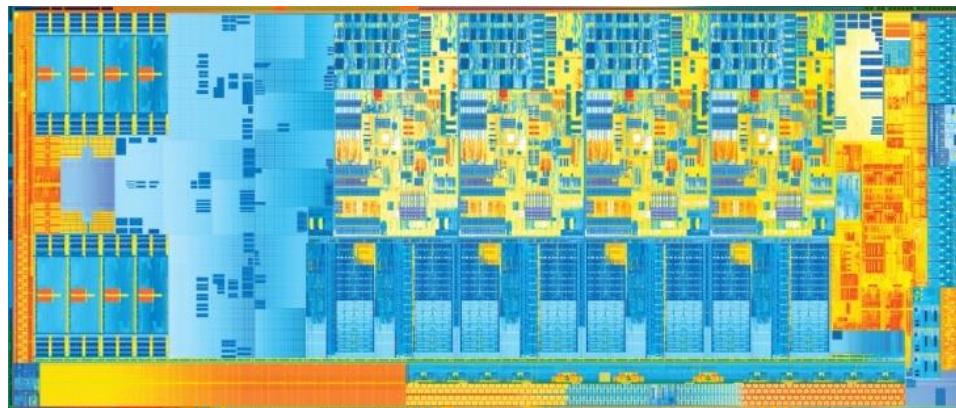


6.823 Computer System Architecture

Instructor: *Daniel Sanchez*

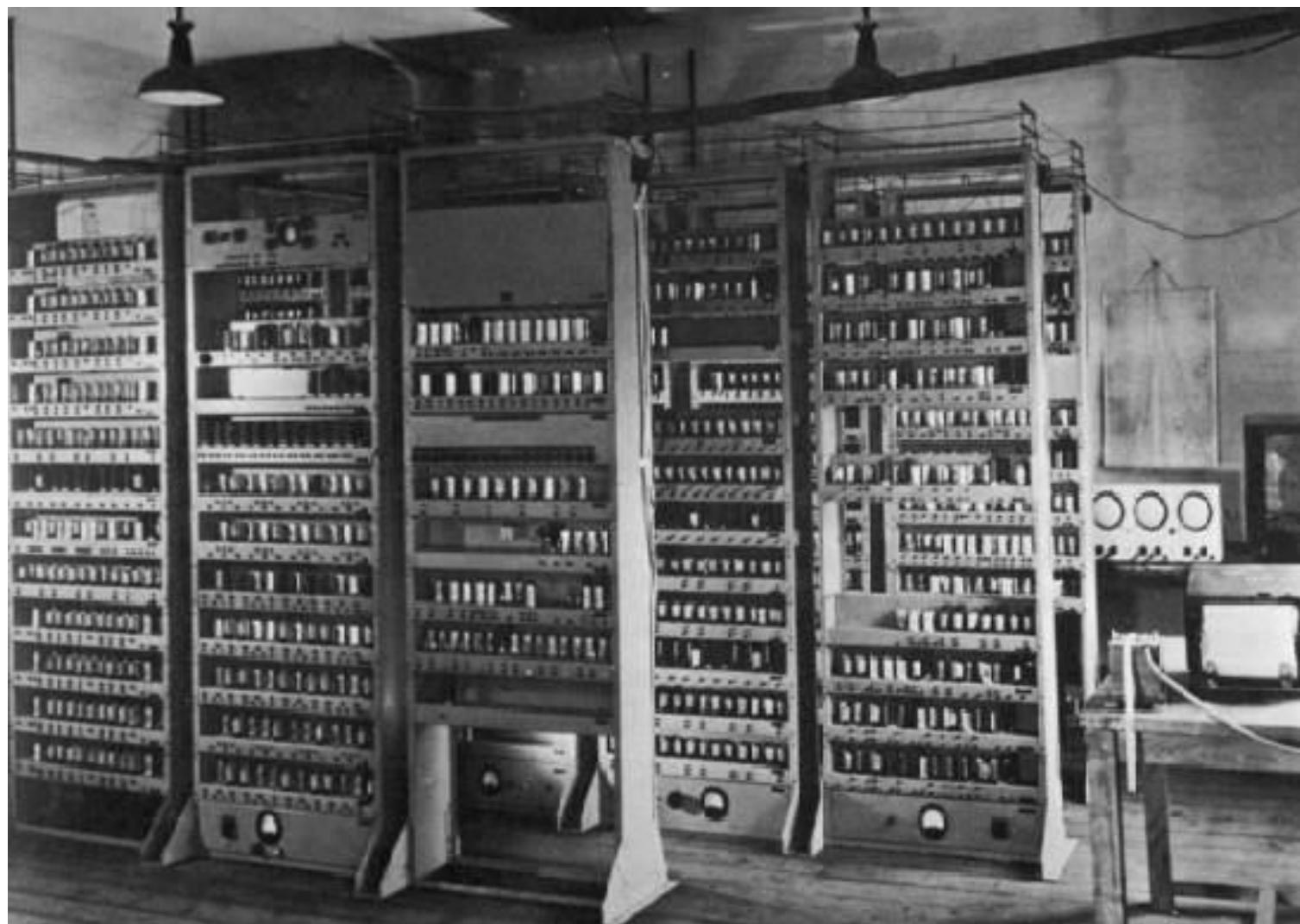
TA: *Hyun Ryong (Ryan) Lee*

The processor you
built in 6.004*

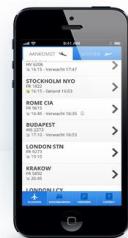


What you'll
understand after
taking 6.823

Computing devices then...



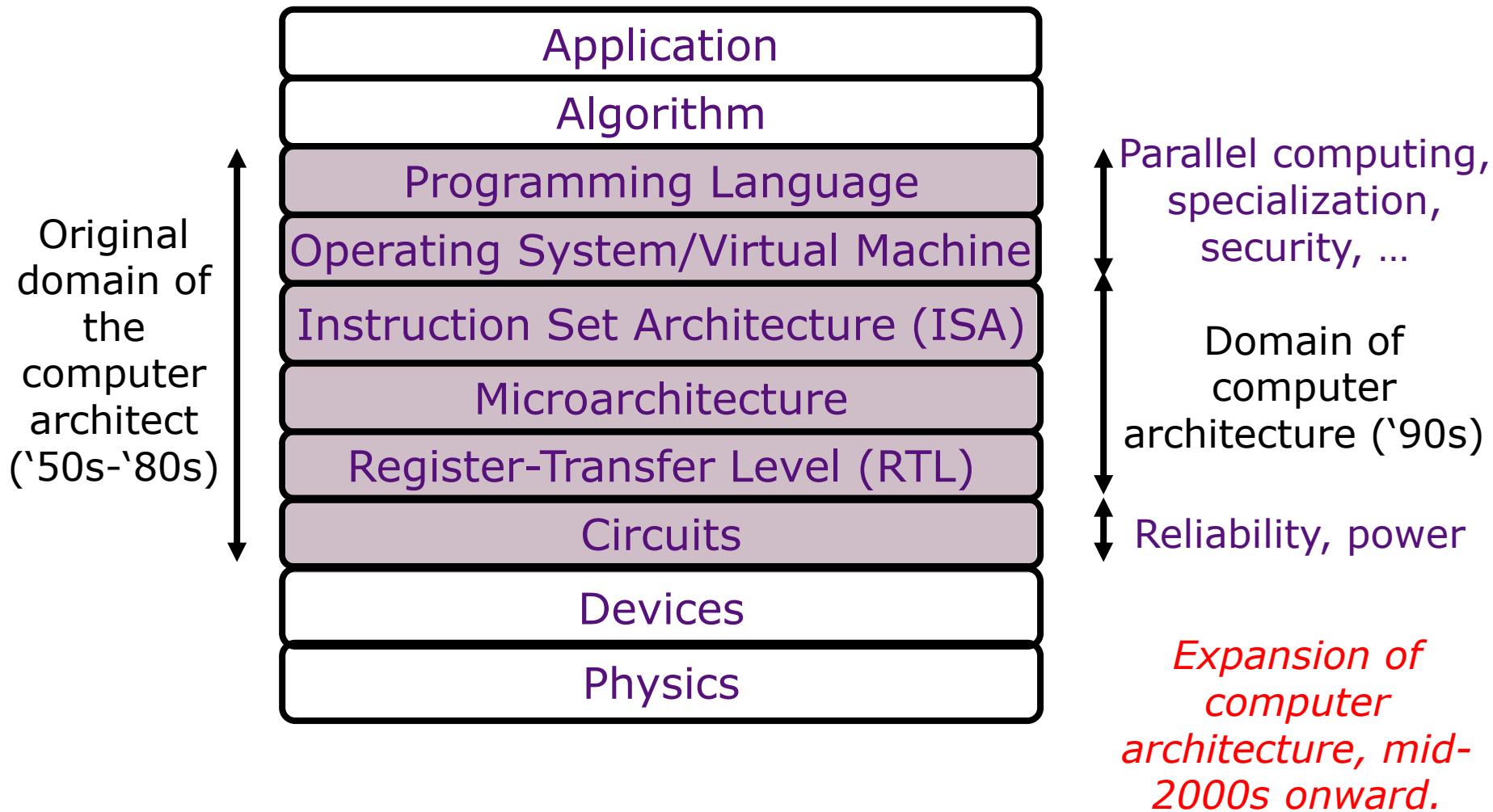
Computing devices now



A journey through this space

- What do computer architects actually do?
- Illustrate via historical examples
 - Early days: ENIAC, EDVAC, and EDSAC
 - Arrival of IBM 650 and then IBM 360
 - Seymour Cray – CDC 6600, Cray 1
 - Microprocessors and PCs
 - Multicores
 - Cell phones
- Focus on ideas, mechanisms, and principles, especially those that have withstood the test of time

Abstraction layers

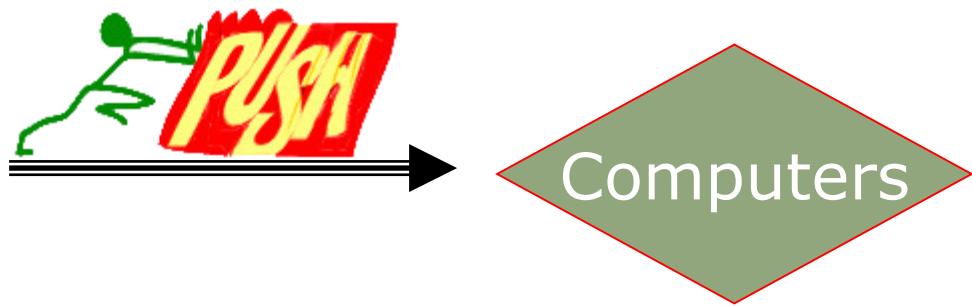


Computer Architecture is the design of abstraction layers

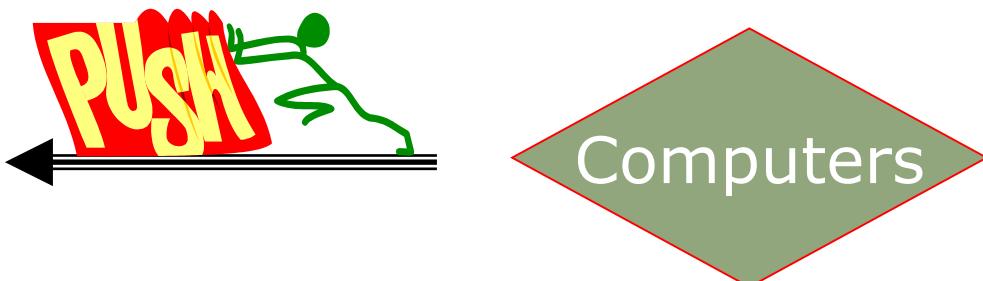
- What do abstraction layers provide?
 - Environmental stability within generation
 - Environmental stability across generations
 - Consistency across a large number of units
- What are the consequences?
 - *Encouragement to create reusable foundations:*
 - *Toolchains, operating systems, libraries*
 - Enticement for application innovation

Technology is the dominant factor in computer design

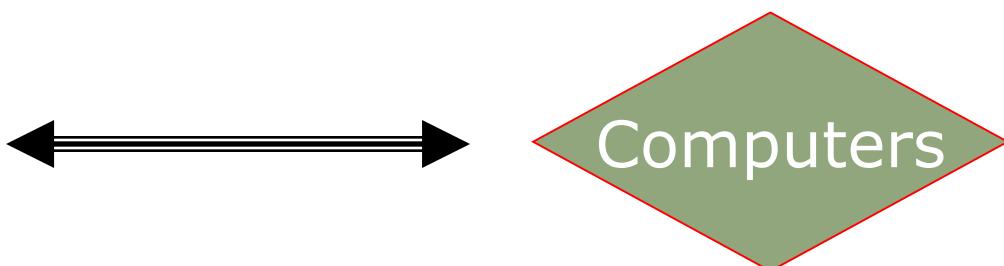
Technology
Transistors
Integrated circuits
VLSI (initially)
Flash memories, ...



Technology
Core memories
Magnetic tapes
Disk



Technology
ROMs, RAMs
VLSI
Packaging
Low Power

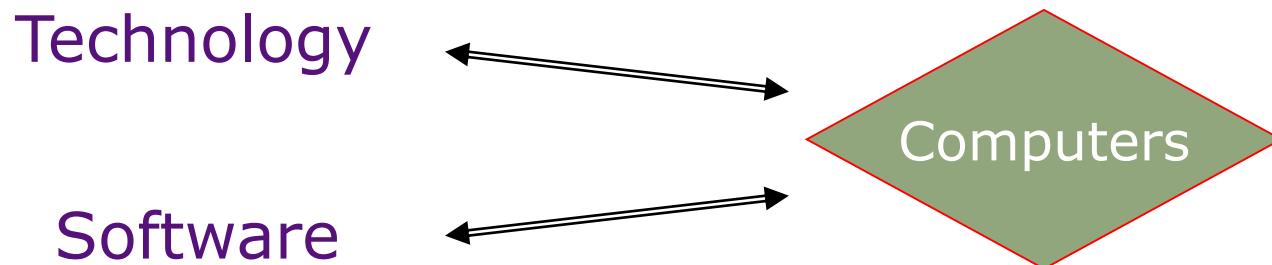


But Software...

As people write programs and use computers,
our understanding of *programming* and
program behavior improves.

*This has profound though slower impact
on computer architecture*

Modern architects must pay attention to
software and compilation issues.



Architecture is engineering design under constraints

Factors to consider:

- Performance of whole system on target applications
 - Average case & worst case
- Cost of manufacturing chips and supporting system
- Power to run system
 - Peak power & energy per operation
- Reliability of system
 - Soft errors & hard errors
- Cost to design chips (engineers, computers, CAD tools)
 - Becoming a limiting factor in many situations, fewer unique chips can be justified
- Cost to develop applications and system software
 - Often the dominant constraint for any programmable device

At different times, and for different applications at the same point in time, the relative balance of these factors can result in widely varying architectural choices

Course Information

All info kept up to date on the website:

<http://www.csg.csail.mit.edu/6.823>

Contact times

- Lectures Tuesdays and Thursdays
 - 1:00pm to 2:30pm
- Tutorial on Fridays
 - 1:00pm to 2:00pm
 - Attendance is optional
 - Additional tutorials will be held in evenings before quizzes
- Quizzes on Friday (*except last quiz*)
 - 1:00pm to 2:30pm
 - Attendance is NOT optional
- Instructor office hours
 - After class or by email appointment
- TA office hours
 - Wednesday 4-5:30pm

Lectures and tutorials

- Lectures/tutorials are synchronous, through Zoom
 - Video recordings will be available on the website
- Two ways of asking questions:
 - Unmute yourself and ask – for questions specific to lecture
 - Zoom chat – for relevant but less direct questions
- If you can, please enable video ☺
 - Helps you stay engaged, helps us get to know you and get nonverbal feedback like in an in-person lecture
 - Your video won't appear on recordings, and recordings won't be publicly available

Online resources & help

- We use Piazza extensively
 - Fastest way to get your questions answered
 - Links to lecture & tutorial videos will be posted on Piazza
 - All course announcements are made on Piazza
- This is not a normal term; if you need help, let us know!
 - We can be accommodating

The course has four modules

Module 1

- Instruction Set Architecture (ISA)
- Caches and Virtual Memory
- Simple Pipelining and Hazards

Module 2

- Complex Pipelining and Out of Order Execution
- Branch Prediction and Speculative Execution

Module 3

- Multithreading and Multiprocessors
- Coherence and consistency
- On-chip networks

Module 4

- VLIW, EPIC
- Vector machines and GPUs

Textbook and readings

- “Computer Architecture: A Quantitative Approach”, Hennessy & Patterson, 5th / 6th ed.
 - 5th edition available online through MIT Libraries
 - Recommended, but not necessary
- Course website lists H&P reading material for each lecture, and optional readings that provide more in-depth coverage

Grading

- Grades are not assigned based on a predetermined curve
 - Most of you are capable of getting an A
- 75% of the grade is based on four closed book 1.5 hour quizzes
 - The first three quizzes will be held during the tutorials; the last one during the last lecture (dates on web syllabus)
 - We'll have distant-timezone quizzes and makeups if needed
- 25% of the grade is based on four laboratory exercises
- No final exam
- No final project

Problem sets & labs

- Problem sets
 - One problem set per module, not graded
 - Intended for private study and for tutorials to help prepare for quizzes
 - Quizzes assume you are very familiar with the content of problem sets
- Labs
 - Four graded labs (Lab 0 is introductory)
 - Based on widely-used PIN tool
 - Labs 2 and 4 are open-ended challenges
- You must complete labs & quizzes individually
 - Please review the collaboration & academic honesty policy

Self evaluation take-home quiz

- Goal is to help you judge for yourself whether you have prerequisites for this class, and to help refresh your memory
- We assume that you understand digital logic, a simple 5-stage pipeline, and simple caches
- Please work by yourself on this quiz – not in groups
- Remember to complete self-evaluation section at end of the quiz
- Due by Friday (on recitation or send answers to TA mailing list)

*Please email us if you have concerns
about your ability to take the class*

Early Developments: From ENIAC to the mid 50's

Prehistory

- 1800s: Charles Babbage
 - Difference Engine (conceived in 1823, first implemented in 1855 by Scheutz)
 - Analytic Engine, the first conception of a general purpose computer (1833, never implemented)
- 1890: Tabulating machines
- Early 1900s: Analog computers
- 1930s: Early electronic (fixed-function) digital computers

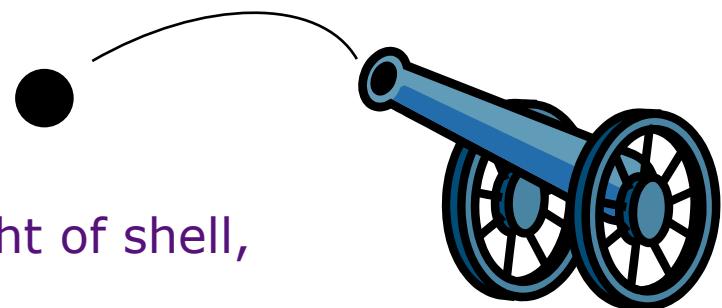
Electronic Numerical Integrator and Computer (ENIAC)

- Designed and built by Eckert and Mauchly at the University of Pennsylvania during 1943-45
- The first, completely electronic, operational, general-purpose analytical calculator!
 - 30 tons, 72 square meters, 200KW
- Performance
 - Read in 120 cards per minute
 - Addition took 200 μs , Division 6 ms
- Not very reliable!

WW-2 Effort

Application: Ballistic calculations

angle = f (location, tail wind, cross wind,
air density, temperature, weight of shell,
propellant charge, ...)



Electronic Discrete Variable Automatic Computer (EDVAC)

- ENIAC's programming system was external
 - Sequences of instructions were executed independently of the results of the calculation
 - Human intervention required to take instructions "out of order"
- EDVAC was designed by Eckert, Mauchly, and von Neumann in 1944 to solve this problem
 - Solution was the *stored program computer*
⇒ "*program can be manipulated as data*"
- *First Draft of a report on EDVAC* was published in 1945, but just had von Neumann's signature!
 - Without a doubt the most influential paper in computer architecture

Stored Program Computer

Program = A sequence of instructions

How to control instruction sequencing?

manual control calculators

automatic control

external (paper tape) Harvard Mark I, 1944
Zuse's Z1, WW2

internal

plug board

ENIAC 1946

read-only memory

ENIAC 1948

read-write memory

EDVAC 1947 (concept)

– The same storage can be used to store program and data

EDSAC

1950

Maurice Wilkes

The Spread of Ideas

ENIAC & EDVAC had immediate impact

brilliant engineering: Eckert & Mauchly

lucid paper: Burks, Goldstein & von Neumann

IAS	Princeton	46-52	Bigelow
EDSAC	Cambridge	46-50	Wilkes
MANIAC	Los Alamos	49-52	Metropolis
JOHNIAC	Rand	50-53	
ILLIAC	Illinois	49-52	
	Argonne	49-53	
SWAC	UCLA-NBS		

UNIVAC - the first commercial computer, 1951

*Alan Turing's direct influence on these developments
is often debated by historians.*

Dominant Technology Issue: *Reliability*

ENIAC

18,000 tubes

20 10-digit numbers

⇒

EDVAC

4,000 tubes

2000 word storage

mercury delay lines

Mean time between failures (MTBF)

MIT's Whirlwind with an MTBF of 20 min. was perhaps the most reliable machine!

Reasons for unreliability:

1. Vacuum tubes

2. Storage medium

Acoustic delay lines

Mercury delay lines

Williams tubes

Selections

CORE

J. Forrester

1954

Computers in the mid 50's

- Hardware was expensive
- Stores were small (1000 words)
 - ⇒ No resident system-software!
- Memory access time was 10 to 50 times slower than the processor cycle
 - ⇒ Instruction execution time was totally dominated by the *memory reference time*
- The *ability to design complex control circuits* to execute an instruction was the central design concern as opposed to *the speed of decoding or an ALU operation*
- Programmer's view of the machine was inseparable from the actual hardware implementation

Accumulator-based computing



- *Single Accumulator*
 - Calculator design carried over to computers

Why?

Registers expensive

The Earliest Instruction Sets

Burks, Goldstein & von Neumann ~1946

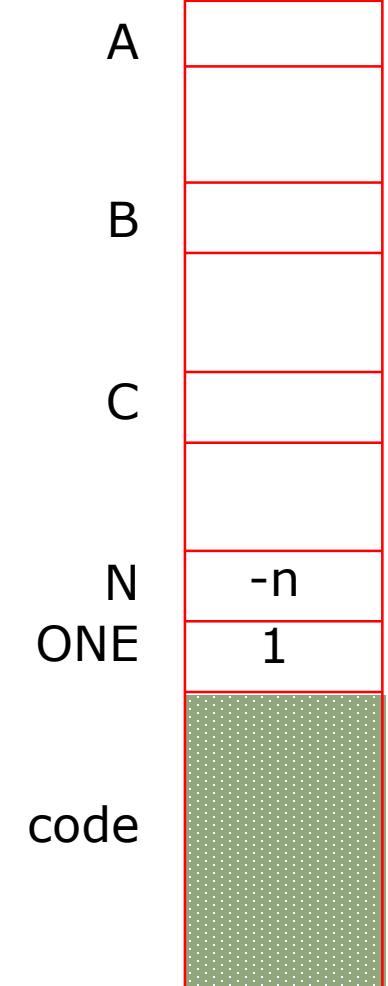
LOAD	x	$AC \leftarrow M[x]$
STORE	x	$M[x] \leftarrow (AC)$
ADD	x	$AC \leftarrow (AC) + M[x]$
SUB	x	
MUL	x	Involved a quotient register
DIV	x	
SHIFT LEFT		$AC \leftarrow 2 \times (AC)$
SHIFT RIGHT		
JUMP	x	$PC \leftarrow x$
JGE	x	if $(AC) \geq 0$ then $PC \leftarrow x$
LOAD ADR	x	$AC \leftarrow \text{Extract address field}(M[x])$
STORE ADR	x	

Typically less than 2 dozen instructions!

Programming: Single Accumulator Machine

$$C_i \leftarrow A_i + B_i, \quad 1 \leq i \leq n$$

LOOP	LOAD	N
	JGE	DONE
	ADD	ONE
	STORE	N
F1	LOAD	A
F2	ADD	B
F3	STORE	C
	JUMP	LOOP
DONE	HLT	



Problem?

How to modify the addresses A, B and C ?

Self-Modifying Code

LOOP	LOAD JGE ADD STORE	N DONE ONE N
F1	LOAD	A
F2	ADD	B
F3	STORE	C
	LOAD ADR ADD STORE ADR LOAD ADR ADD STORE ADR LOAD ADR ADD STORE ADR JUMP	F1 ONE F1 F2 ONE F2 F3 ONE F3 LOOP
DONE	HLT	

modify the program for the next iteration

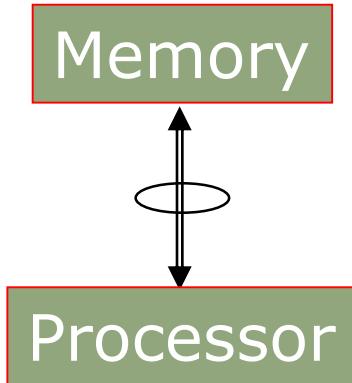
$$C_i \leftarrow A_i + B_i, \quad 1 \leq i \leq n$$

Each iteration involves total book-keeping instruction fetches operand fetches stores

Most of the executed instructions are for bookkeeping!

Processor-Memory Bottleneck: Early Solutions

- Indexing capability
 - to reduce bookkeeping instructions
- Fast local storage in the processor
 - 8-16 registers as opposed to one accumulator
 - to reduce loads/stores
- Complex instructions
 - to reduce instruction fetches
- Compact instructions
 - implicit address bits for operands
 - to reduce instruction fetch cost



Index Registers

Tom Kilburn, Manchester University, mid 50's

One or more specialized registers to simplify address calculation

Modify existing instructions

LOAD	x, IX	$AC \leftarrow M[x + (IX)]$
ADD	x, IX	$AC \leftarrow (AC) + M[x + (IX)]$

...

Add new instructions to manipulate *index registers*

JZi	x, IX	if $(IX)=0$ then $PC \leftarrow x$ else $IX \leftarrow (IX) + 1$
LOADi	x, IX	$IX \leftarrow M[x]$ (truncated to fit IX)

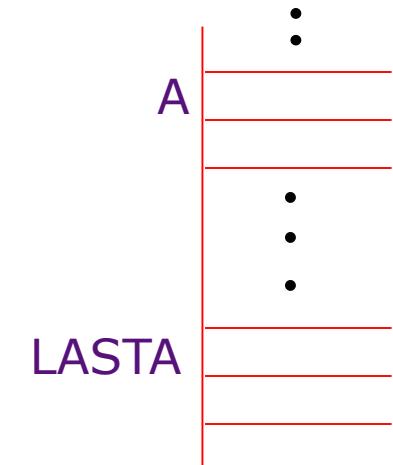
...

Index registers have accumulator-like characteristics

Using Index Registers

$$C_i \leftarrow A_i + B_i, \quad 1 \leq i \leq n$$

	LOADi	N, IX	
LOOP	JZi	DONE, IX	N starts with -n
	LOAD	LASTA, IX	
	ADD	LASTB, IX	
	STORE	LASTC, IX	
	JUMP	LOOP	
DONE		HALT	



- Program does not modify itself
- Efficiency has improved dramatically (ops / iter)

with index regs without index regs

instruction fetch	17 (14)
operand fetch	10 (8)
store	5 (4)

- Costs?

Operations on Index Registers

To increment index register by k

$AC \leftarrow (IX)$ *new instruction*

$AC \leftarrow (AC) + k$

$IX \leftarrow (AC)$ *new instruction*

also the AC must be saved and restored

It may be better to increment IX directly

$INCI$ k, IX $IX \leftarrow (IX) + k$

More instructions to manipulate index register

$STOREi$ x, IX $M[x] \leftarrow (IX)$ (extended to fit a word)

...

IX begins to look like an accumulator

⇒ several index registers

several accumulators

⇒ *General Purpose Registers*

Evolution of Addressing Modes

1. Single accumulator, absolute address

LOAD x

2. Single accumulator, index registers

LOAD x, IX

3. Indirection

LOAD (x)

4. Multiple accumulators, index registers, indirection

LOAD R, IX, x

or LOAD R, IX, (x)

the meaning?

$R \leftarrow M[M[x] + (IX)]$

or $R \leftarrow M[M[x + (IX)]]$

5. Indirect through registers

LOAD R_I, (R_J)

6. The works

LOAD R_I, R_J, (R_K)

R_J = index, R_K = base addr

Variety of Instruction Formats

- *Three address formats:* One destination and up to two operand sources per instruction

(Reg op Reg) to Reg
(Reg op Mem) to Reg

$$R_I \leftarrow (R_J) \text{ op } (R_K)$$
$$R_I \leftarrow (R_J) \text{ op } M[x]$$

- x can be specified directly or via a register
- effective address calculation for x could include indexing, indirection, ...

- *Two address formats:* the destination is same as one of the operand sources

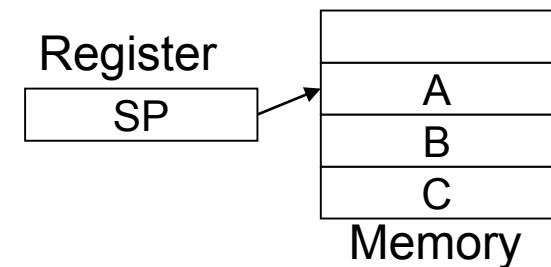
(Reg op Reg) to Reg
(Reg op Mem) to Reg

$$R_I \leftarrow (R_I) \text{ op } (R_J)$$
$$R_I \leftarrow (R_I) \text{ op } M[x]$$

More Instruction Formats

- *One address formats:* Accumulator machines
 - Accumulator is always other implicit operand
- *Zero address formats:* operands on a stack

add $M[sp-1] \leftarrow M[sp] + M[sp-1]$
load $M[sp] \leftarrow M[M[sp]]$



- Stack can be in registers or in memory
 - usually top of stack cached in registers

Many different formats are possible!

Instruction sets in the mid 50's

- Great variety of instruction sets, but all intimately tied to implementation details
- Programmer's view of the machine was inseparable from the actual hardware implementation!

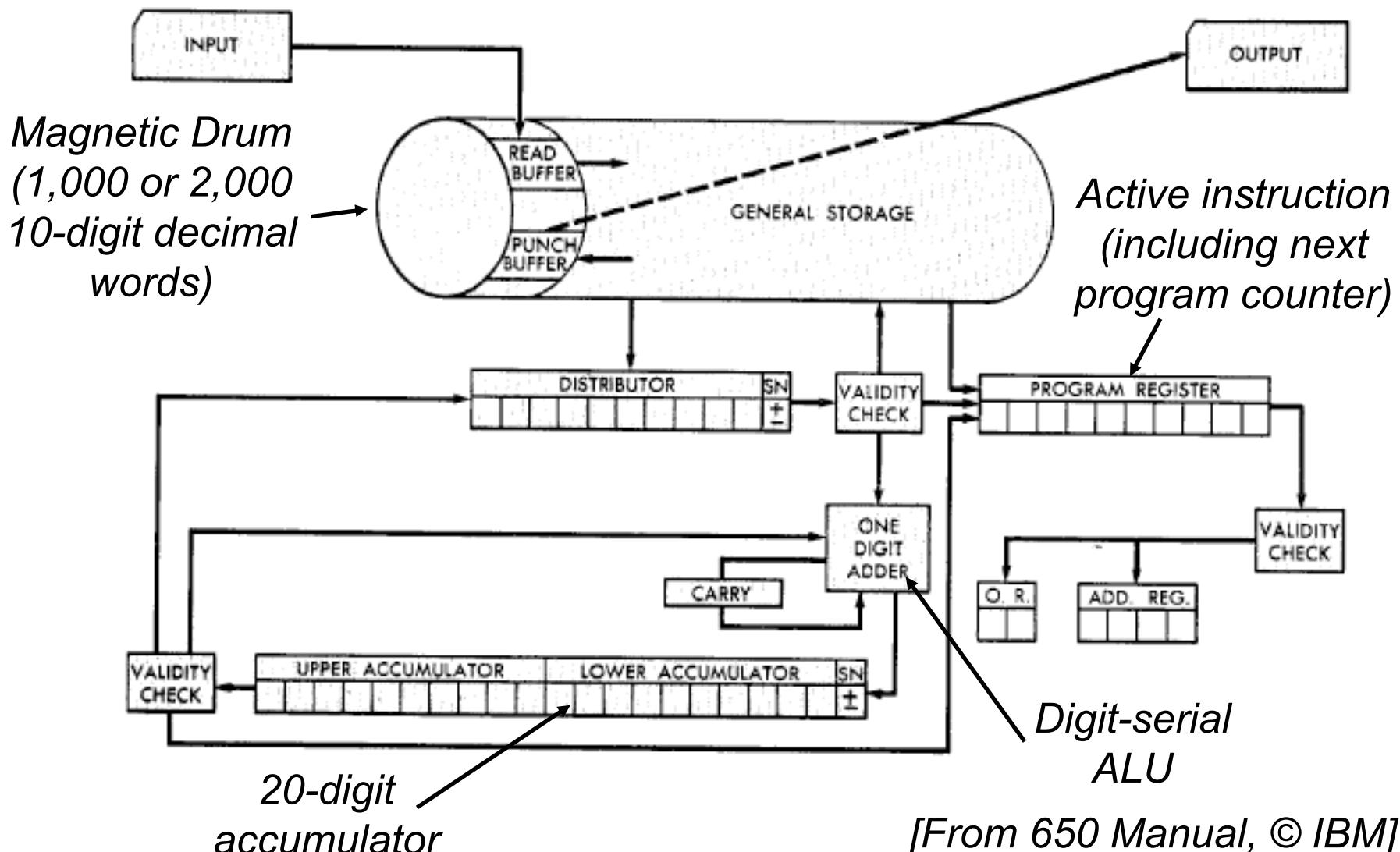
Next Lecture:
Instruction Set Architectures:
Decoupling Interface and
Implementation

Instruction Set Architecture & Hardwired, Non-pipelined ISA Implementation

Daniel Sanchez

Computer Science & Artificial Intelligence Lab
M.I.T.

The IBM 650 (1953-4)



Programmer's view of a machine: IBM 650

A drum machine with 44 instructions

Instruction: 60 1234 1009

- “Load the contents of location 1234 into the *distribution*; put it also into the *upper accumulator*; set *lower accumulator* to zero; and then go to location 1009 for the next instruction.”

- Programmer's view of the machine was inseparable from the actual hardware implementation
- Good programmers optimized the placement of instructions on the drum to reduce latency!

Compatibility Problem at IBM

By early 60's, *IBM had 4 incompatible lines of computers!*

701	→	7094
650	→	7074
702	→	7080
1401	→	7010

Each system had its own

- Instruction set
- I/O system and Secondary Storage:
magnetic tapes, drums and disks
- Assemblers, compilers, libraries,...
- Market niche
business, scientific, real time, ...

⇒ *IBM 360*

IBM 360: Design Premises

Amdahl, Blaauw, and Brooks, 1964

The design must lend itself to *growth and successor machines*

- General method for connecting I/O devices
- Total performance - answers per month rather than bits per microsecond ⇒ *programming aids*
- Machine must be capable of *supervising itself* without manual intervention
- Built-in *hardware fault checking* and locating aids to reduce down time
- Simple to assemble systems with redundant I/O devices, memories, etc. for *fault tolerance*
- Some problems required floating point words larger than 36 bits

Processor State and Data Types

The information held in the processor at the end of an instruction to provide the processing context for the next instruction.

Program Counter, Accumulator, ...

- The information held in the processor will be interpreted as having data types manipulated by the instructions.
- If the processing of an instruction can be interrupted then the *hardware* must save and restore the state in a transparent manner

Programmer's machine model is a **contract** between the hardware and software

Instruction Set

The control for changing the information held in the processor are specified by the instructions available in the instruction set architecture or ISA.

Some things an ISA must specify:

- *A way to reference registers and memory*
- *The computational operations available*
- *How to control the sequence of instructions*

- *A binary representation for all of the above*

ISA must satisfy the needs of the software:
- assembler, compiler, OS, VM

IBM 360: A General-Purpose Register (GPR) Machine

- Processor State
 - 16 General-Purpose 32-bit Registers
 - 4 Floating Point 64-bit Registers
 - A Program Status Word (PSW)
 - *PC, Condition codes, Control flags*
- Data Formats
 - 8-bit bytes, 16-bit half-words, 32-bit words, 64-bit double-words
 - 24-bit addresses
- A 32-bit machine with 24-bit addresses
 - *No instruction contains a 24-bit address!*
- Precise interrupts

IBM 360: Initial Implementations (1964)

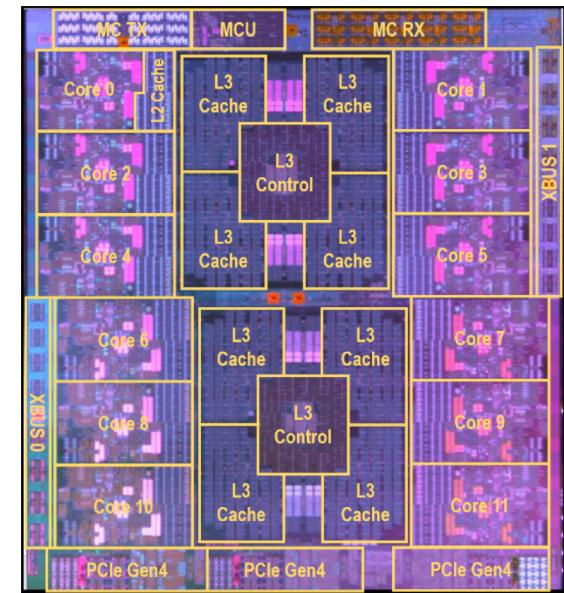
	<i>Model 30</i>	...	<i>Model 70</i>
<i>Memory Capacity</i>	8K - 64 KB		256K - 512 KB
<i>Memory Cycle</i>	2.0 μ s	...	1.0 μ s
<i>Datapath</i>	8-bit		64-bit
<i>Circuit Delay</i>	30 nsec/level		5 nsec/level
<i>Registers</i>	in Main Store		in Transistor
<i>Control Store</i>	Read only 1 μ sec		Dedicated circuits

- Six implementations (Models, 30, 40, 50, 60, 62, 70)
- 50x performance difference across models
- *ISA completely hid the underlying technological differences between various models*

With minor modifications, IBM 360 ISA is still in use

IBM 360: Fifty-five years later... z15 Microprocessor

- 9.2 billion transistors, 12-core design
- Up to 190 cores (2 spare) per system
- 5.2 GHz, 14nm CMOS technology
- 64-bit virtual addressing
 - Original 360 was 24-bit; 370 was a 31-bit extension
- Superscalar, out-of-order
 - 12-wide issue
 - Up to 180 instructions in flight
- 16K-entry Branch Target Buffer
 - Very large buffer to support commercial workloads
- Four Levels of caches
 - 128KB L1 I-cache, 128KB L1 D-cache
 - 4MB L2 cache per core
 - 256MB shared on-chip L3 cache
 - 960MB shared off-chip L4 cache
- Up to 40TB of main memory per system



September 2019
Image credit: IBM

Instruction Set Architecture (ISA) versus Implementation

- ISA is the hardware/software interface
 - Defines set of programmer visible state
 - Defines data types
 - Defines instruction semantics (operations, sequencing)
 - Defines instruction format (bit encoding)
 - Examples: *MIPS, RISC-V, Alpha, x86, IBM 360, VAX, ARM, JVM*
- Many possible implementations of one ISA
 - 360 implementations: model 30 (c. 1964), z15 (c. 2019)
 - x86 implementations: *8086 (c. 1978), 80186, 286, 386, 486, Pentium, Pentium Pro, Pentium-4, Core i7, AMD Athlon, AMD Opteron, Transmeta Crusoe, SoftPC*
 - MIPS implementations: *R2000, R4000, R10000, ...*
 - JVM: *HotSpot, PicoJava, ARM Jazelle, ...*

Processor Performance

$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} * \frac{\text{Cycles}}{\text{Instruction}} * \frac{\text{Time}}{\text{Cycle}}$$

- Instructions per program depends on source code, compiler technology and ISA
- Cycles per instructions (CPI) depends upon the ISA and the microarchitecture
- Time per cycle depends upon the microarchitecture and the base technology

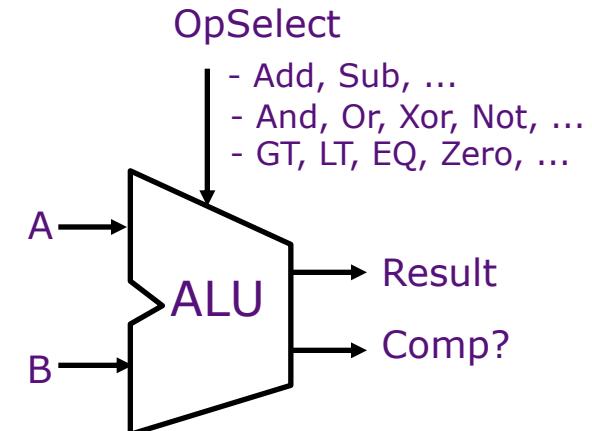
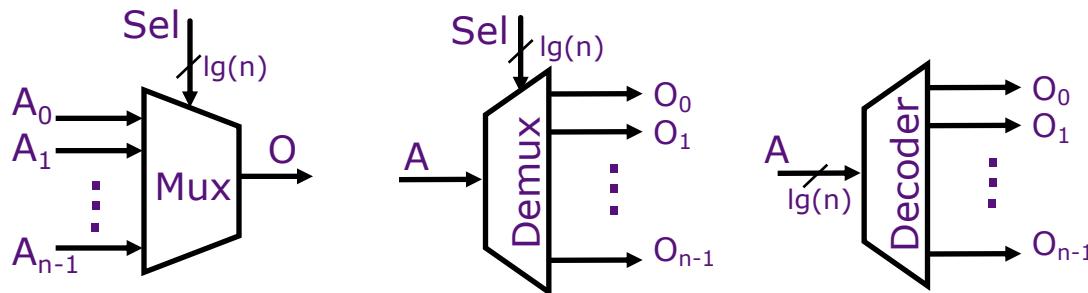
rest of
this lecture →

Microarchitecture	CPI	cycle time
Microcoded	>1	short
Single-cycle unpipelined	1	long
Pipelined	1	short

Hardware Elements

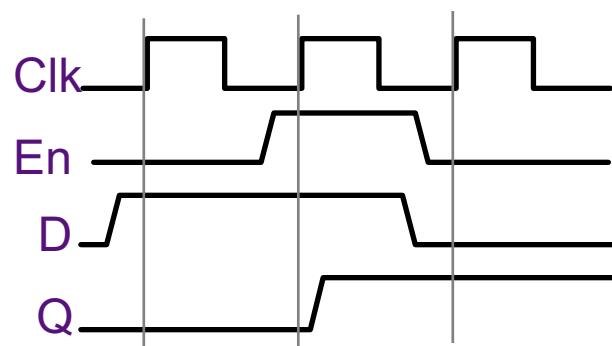
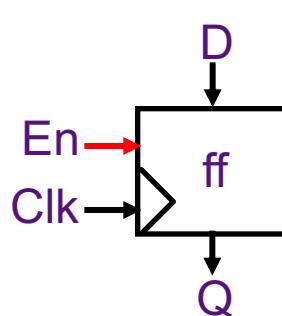
- Combinational circuits

- Mux, Demux, Decoder, ALU, ...



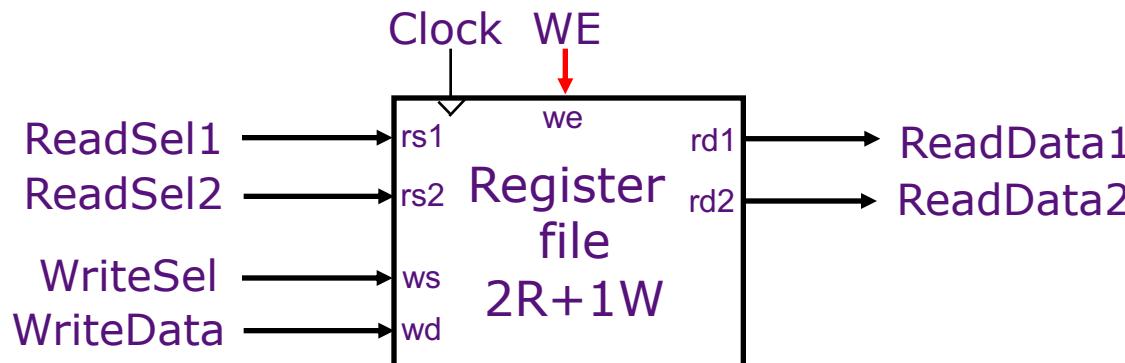
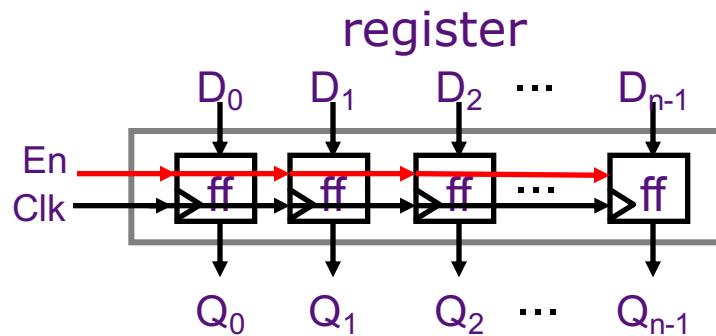
- Synchronous state elements

- Flipflop, Register, Register file, SRAM, DRAM



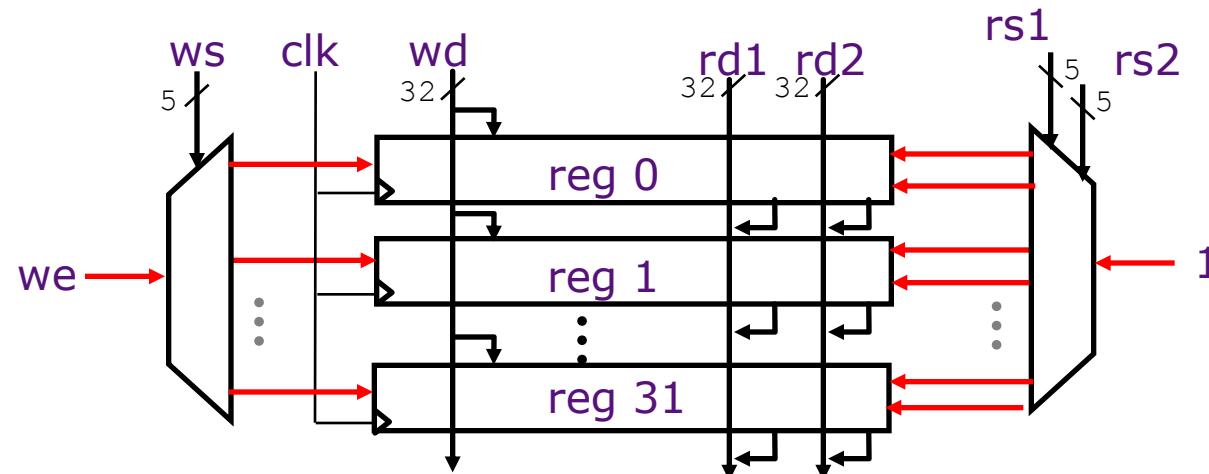
Edge-triggered: Data is sampled at the rising edge

Register Files



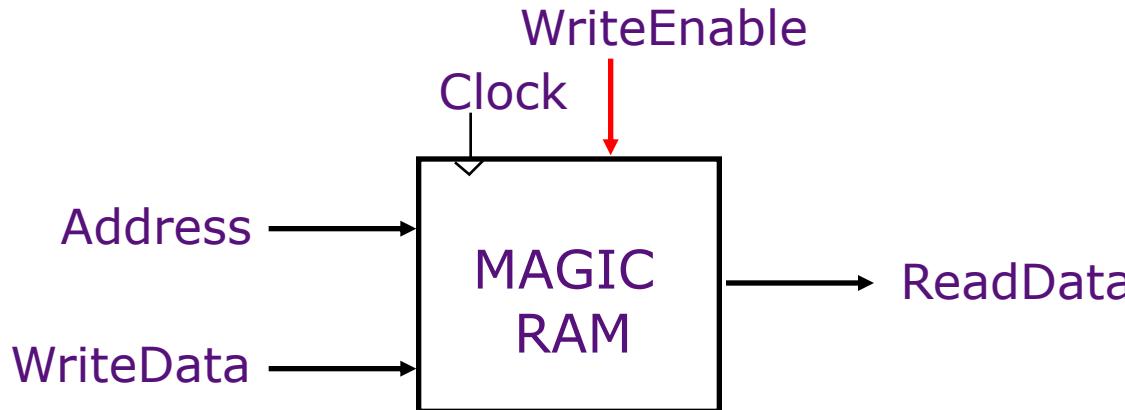
No timing issues when reading and writing the same register
(writes happen at the end of the cycle)

Register File Implementation



- Register files with a large number of ports are difficult to design
 - Area scales with ports²
 - Almost all Alpha instructions have exactly 2 register source operands
 - *Intel's Itanium GPR File has 128 registers with 8 read ports and 4 write ports!!*

A Simple Memory Model



- Reads and writes are always completed in one cycle
 - A Read can be done any time (i.e., combinational)
 - If enabled, a Write is performed at the rising clock edge
(the write address and data must be stable at the clock edge)

Later in the course we will present a more realistic model of memory

Implementing MIPS: Single-cycle per instruction datapath & control logic

The MIPS ISA

Processor State

32 32-bit GPRs, R0 always contains a 0

32 single precision FPRs, may also be viewed as
16 double precision FPRs

FP status register, used for FP compares & exceptions

PC, the program counter

Some other special registers

Data types

8-bit byte, 16-bit half word

32-bit word for integers

32-bit word for single precision floating point

64-bit word for double precision floating point

Load/Store style instruction set

Data addressing modes: immediate & indexed

Branch addressing modes: PC relative & register indirect

Byte-addressable memory, big-endian mode

All instructions are 32 bits

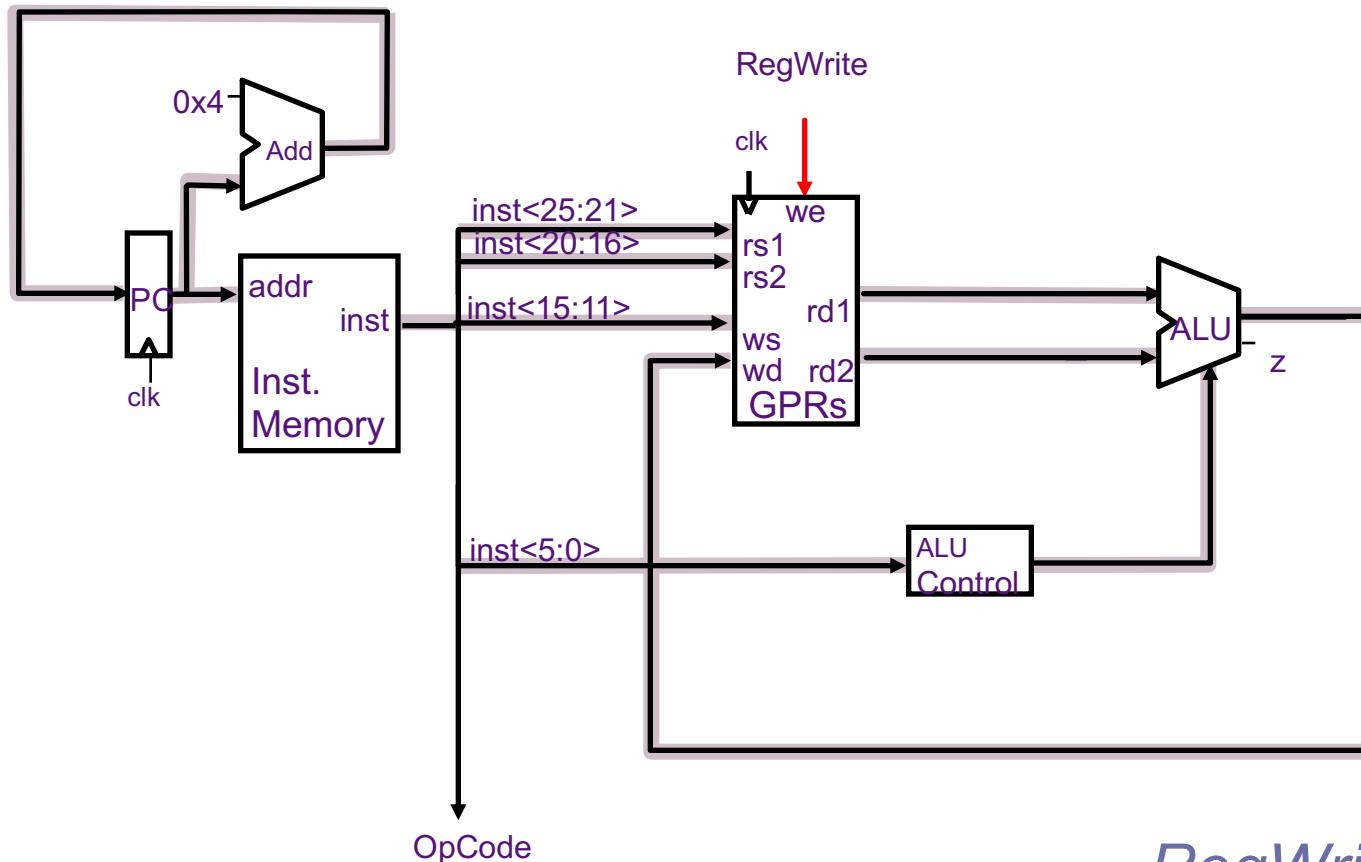
Instruction Execution

Execution of an instruction involves

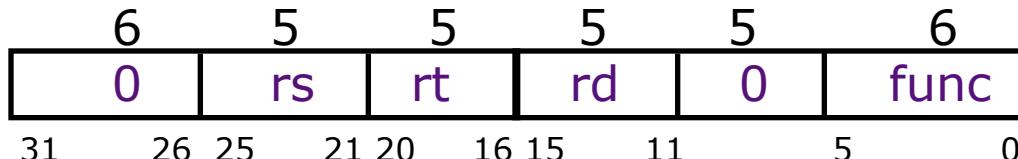
1. Instruction fetch
2. Decode
3. Register fetch
4. ALU operation
5. Memory operation (optional)
6. Write back

And computing the address of the
next instruction (next PC)

Datapath: Reg-Reg ALU Instructions

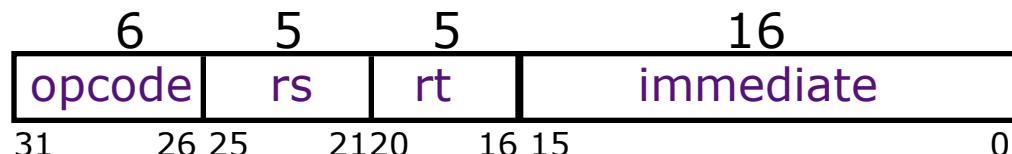
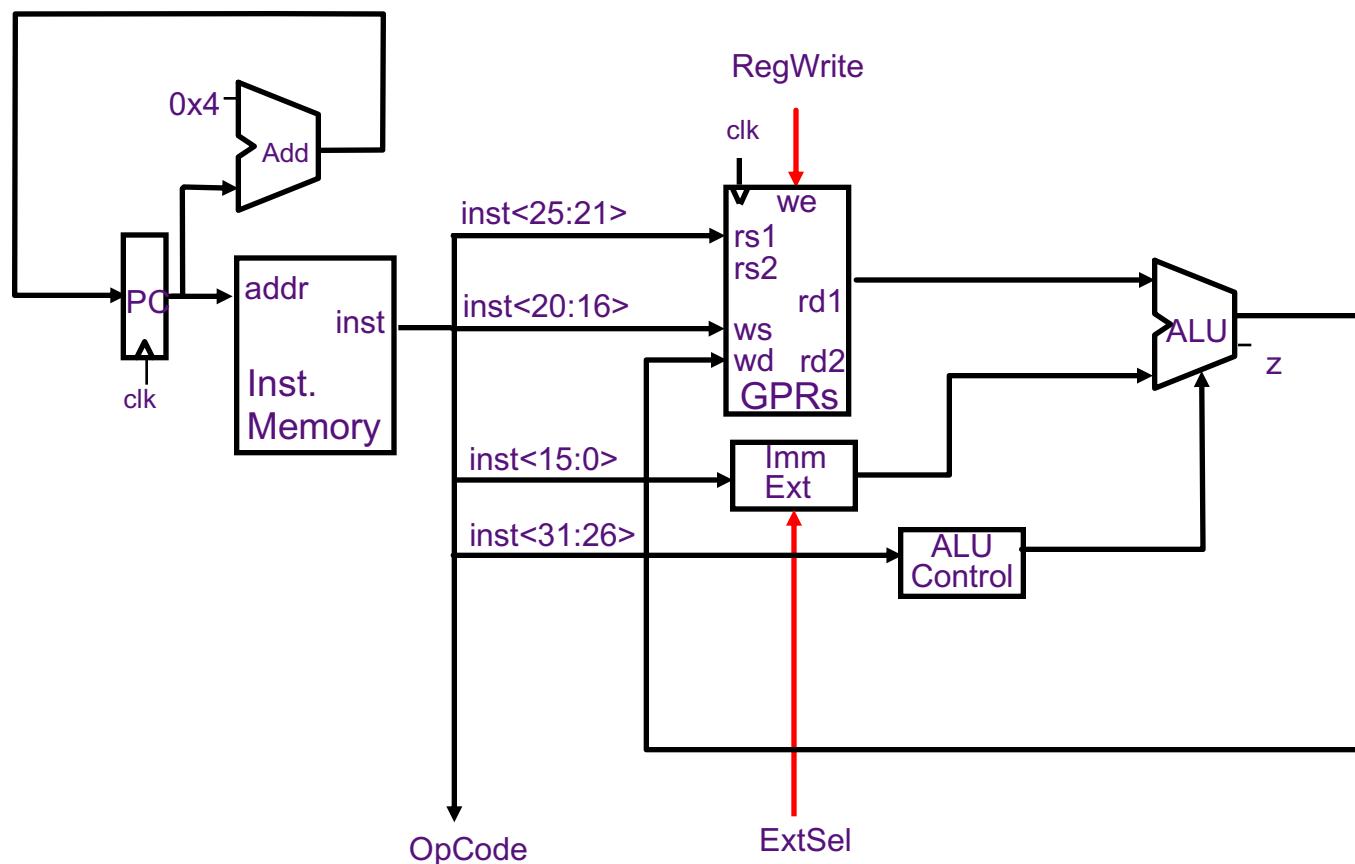


RegWrite Timing?



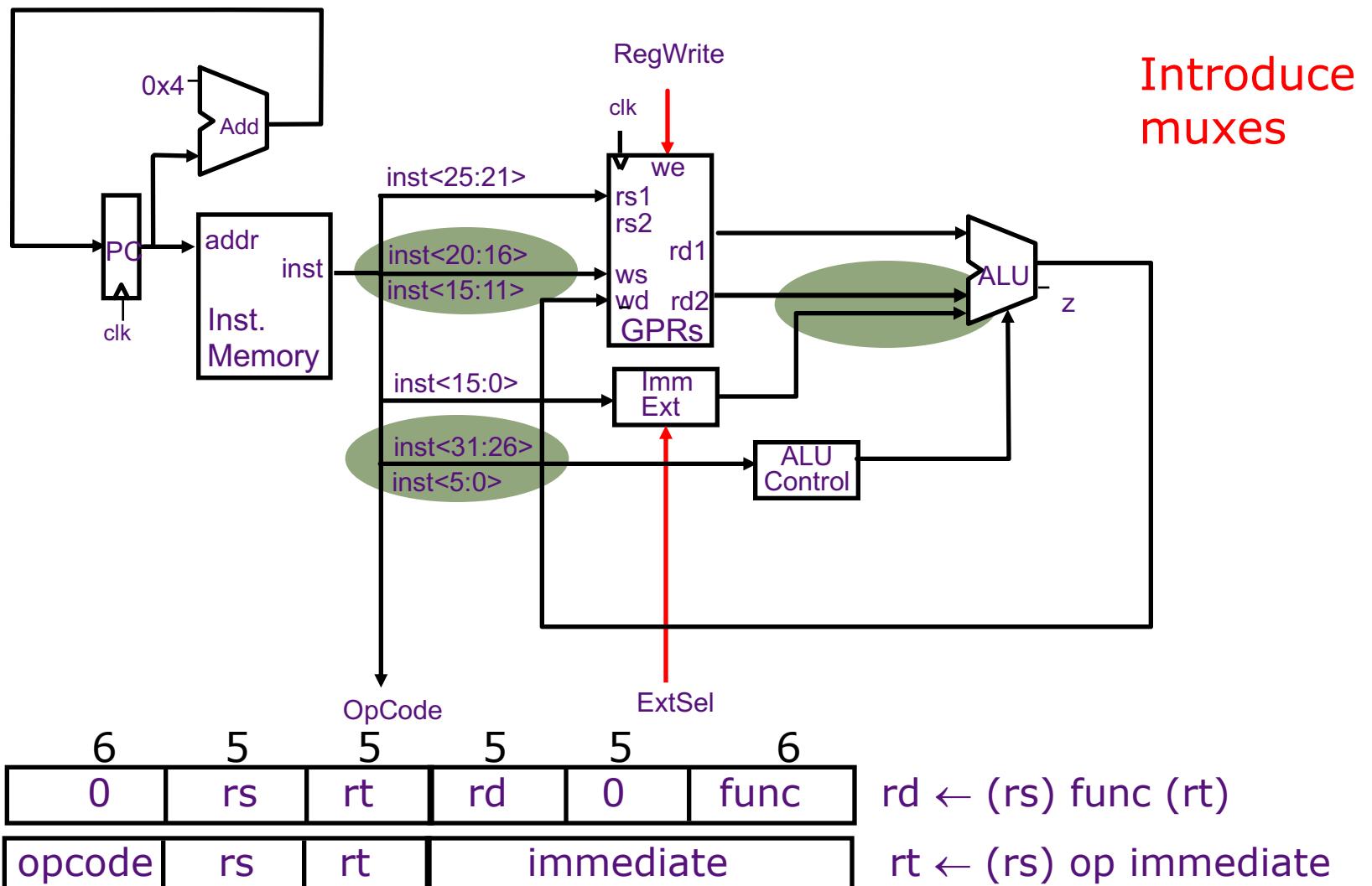
$$rd \leftarrow (rs) \text{ func } (rt)$$

Datapath: Reg-Imm ALU Instructions

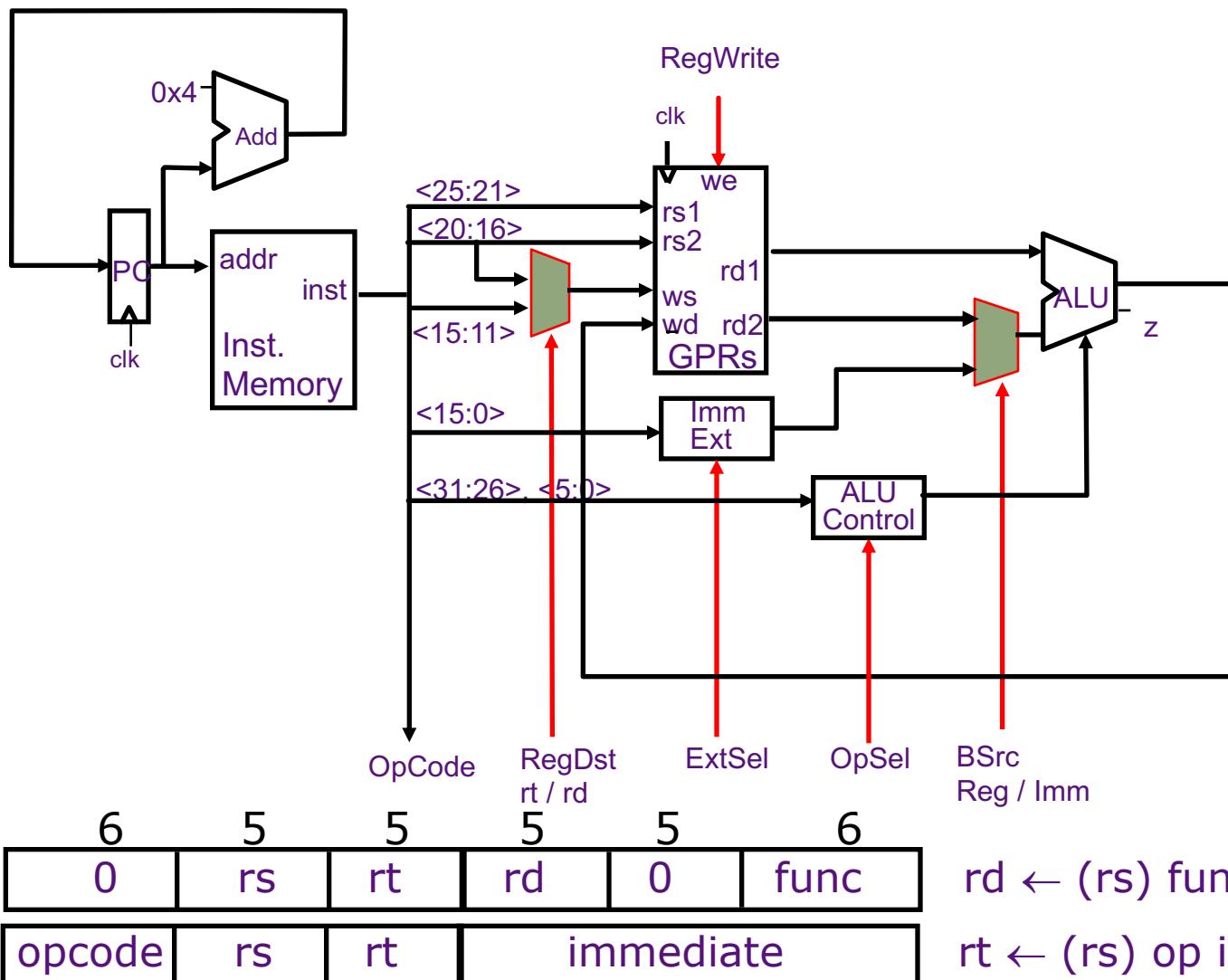


$rt \leftarrow (rs) \text{ op immediate}$

Conflicts in Merging Datapath



Datapath for ALU Instructions



Datapath for Memory Instructions

Should program and data memory be separate?

Harvard style: separate (Aiken and Mark 1 influence)

- read-only program memory
- read/write data memory

- Note:

There must be a way to load the program memory

Princeton style: the same (von Neumann's influence)

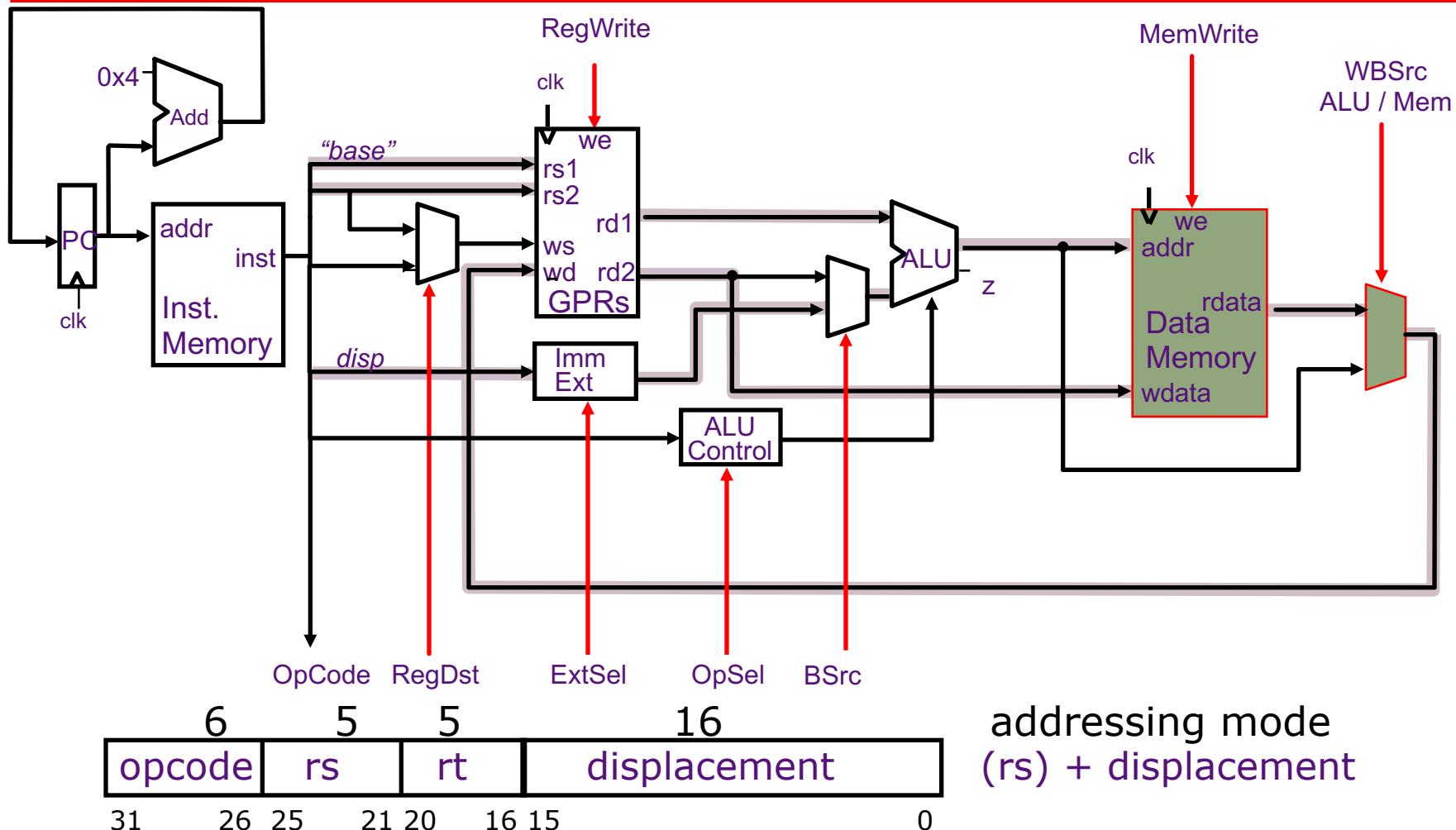
- single read/write memory for program and data

- Note:

Executing a Load or Store instruction requires accessing the memory more than once

Load/Store Instructions

Harvard Datapath



rs is the base register

rt is the destination of a Load or the source for a Store

MIPS Control Instructions

Conditional (on GPR) PC-relative branch



BEQZ, BNEZ

Unconditional register-indirect jumps



JR, JALR

Unconditional absolute jumps



J, JAL

Target PC

Condition

BEQZ

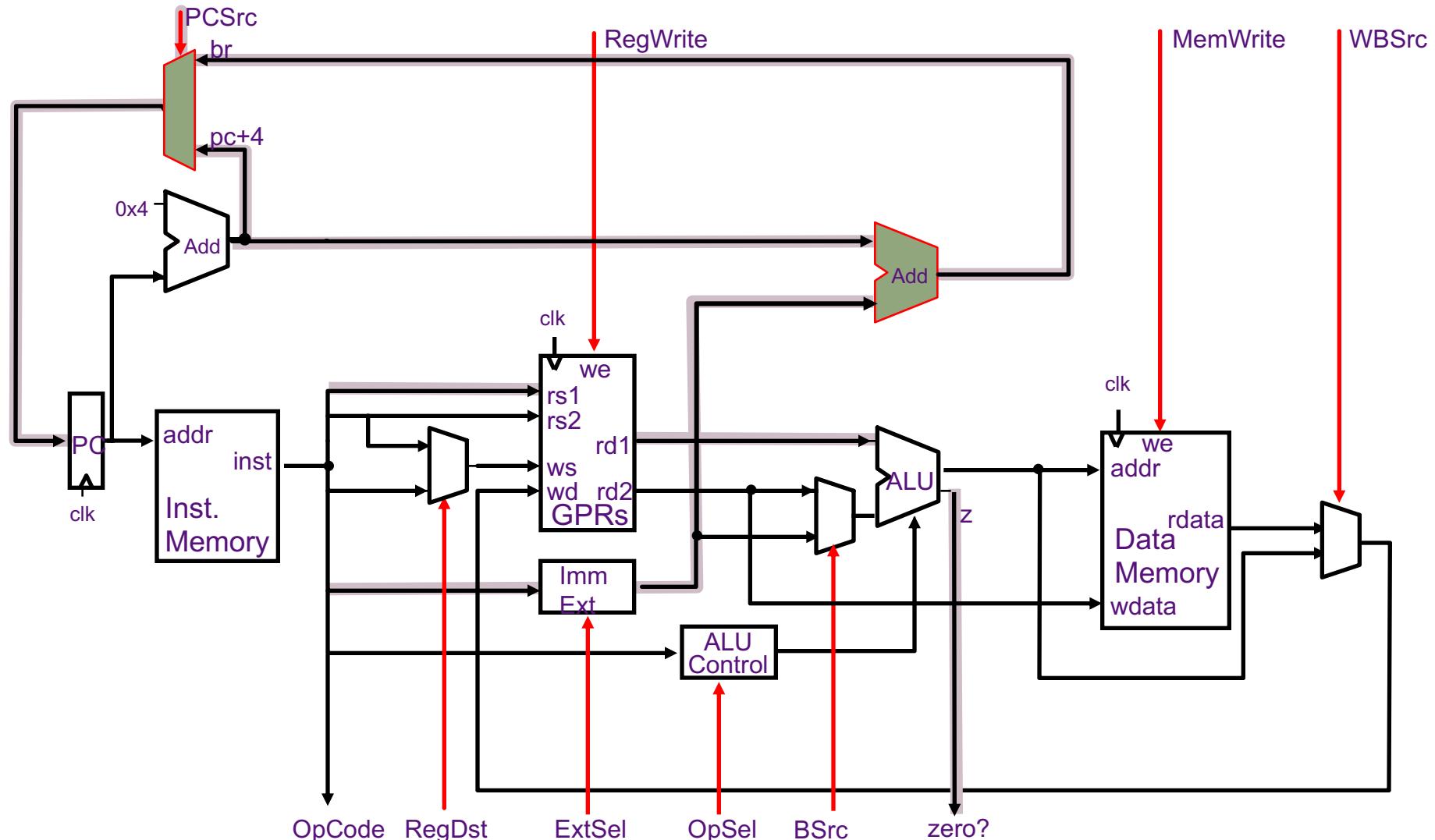
BNEZ

JR, JALR

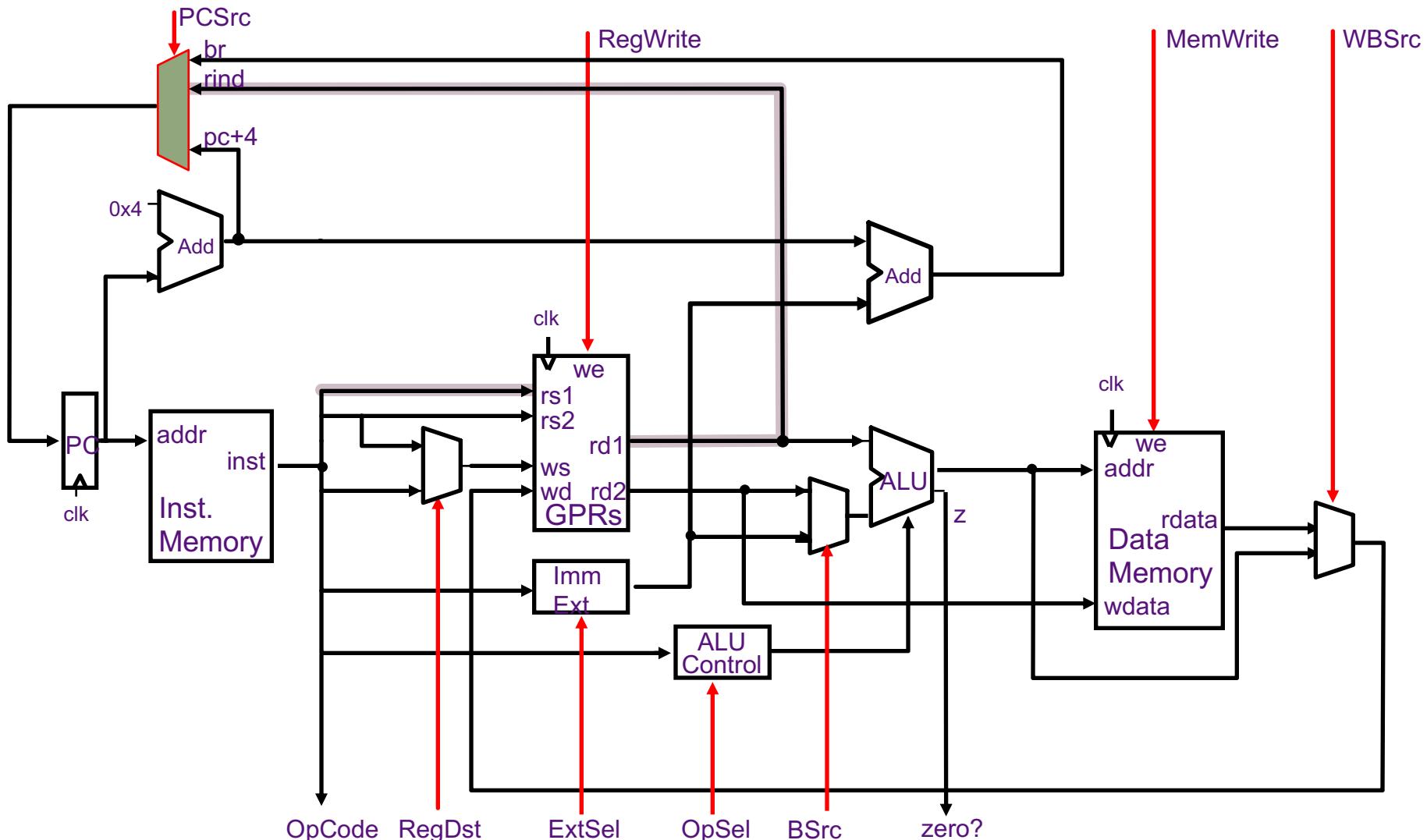
J, JAL

- Jump-&-link stores PC+4 into the link register (R31)
- Control transfers are not delayed
we will worry about the branch delay slot later

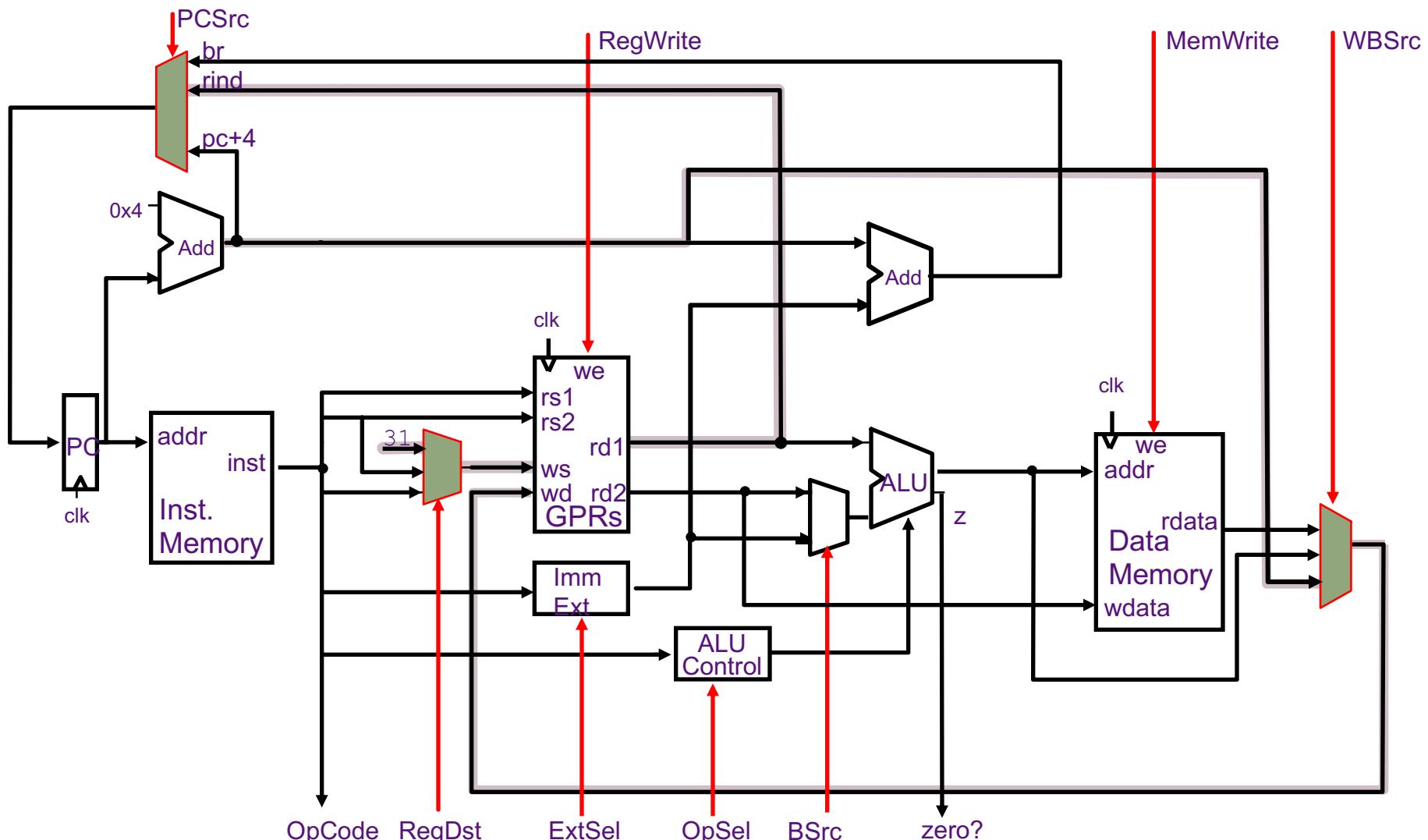
Conditional Branches (BEQZ, BNEZ)



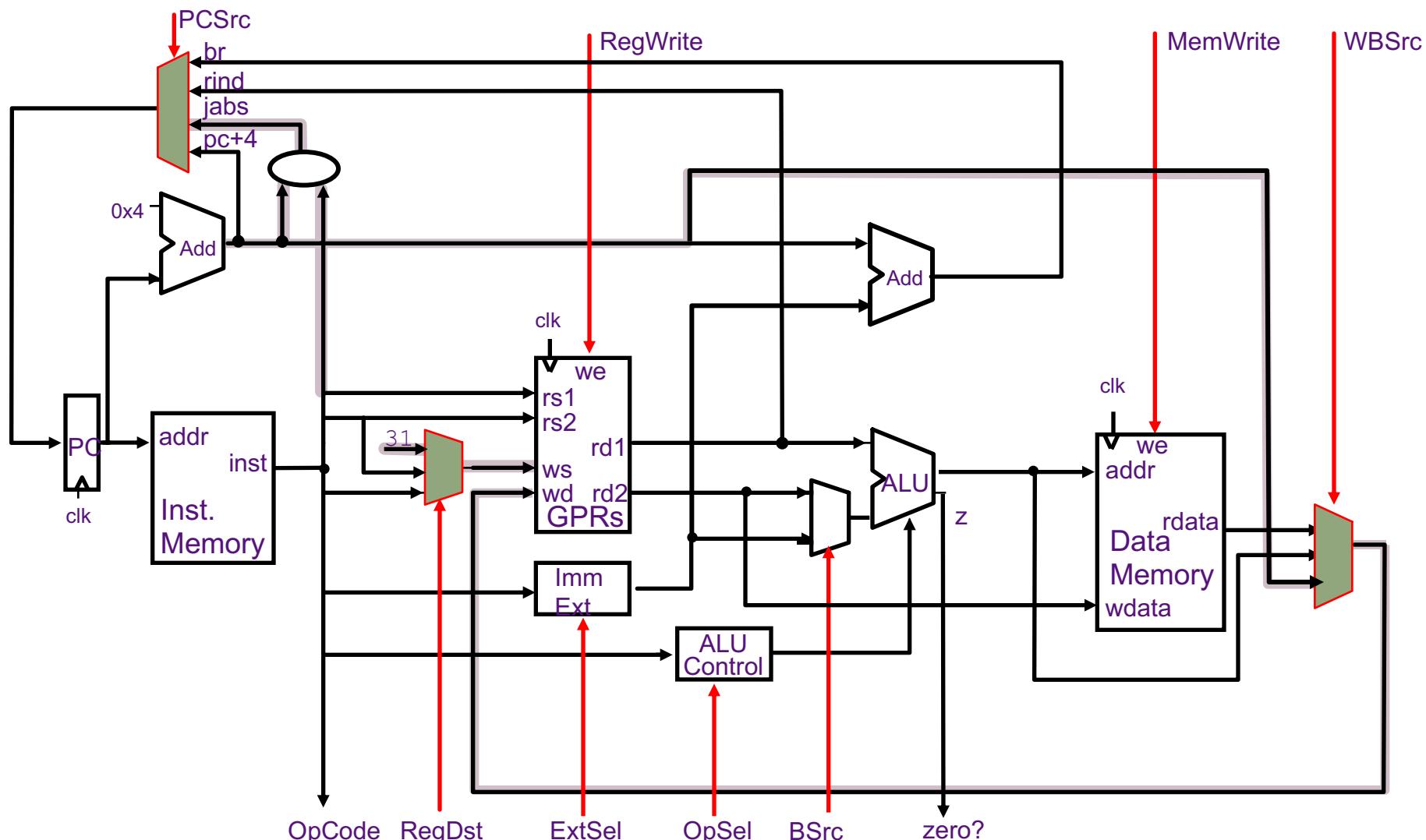
Register-Indirect Jumps (JR)



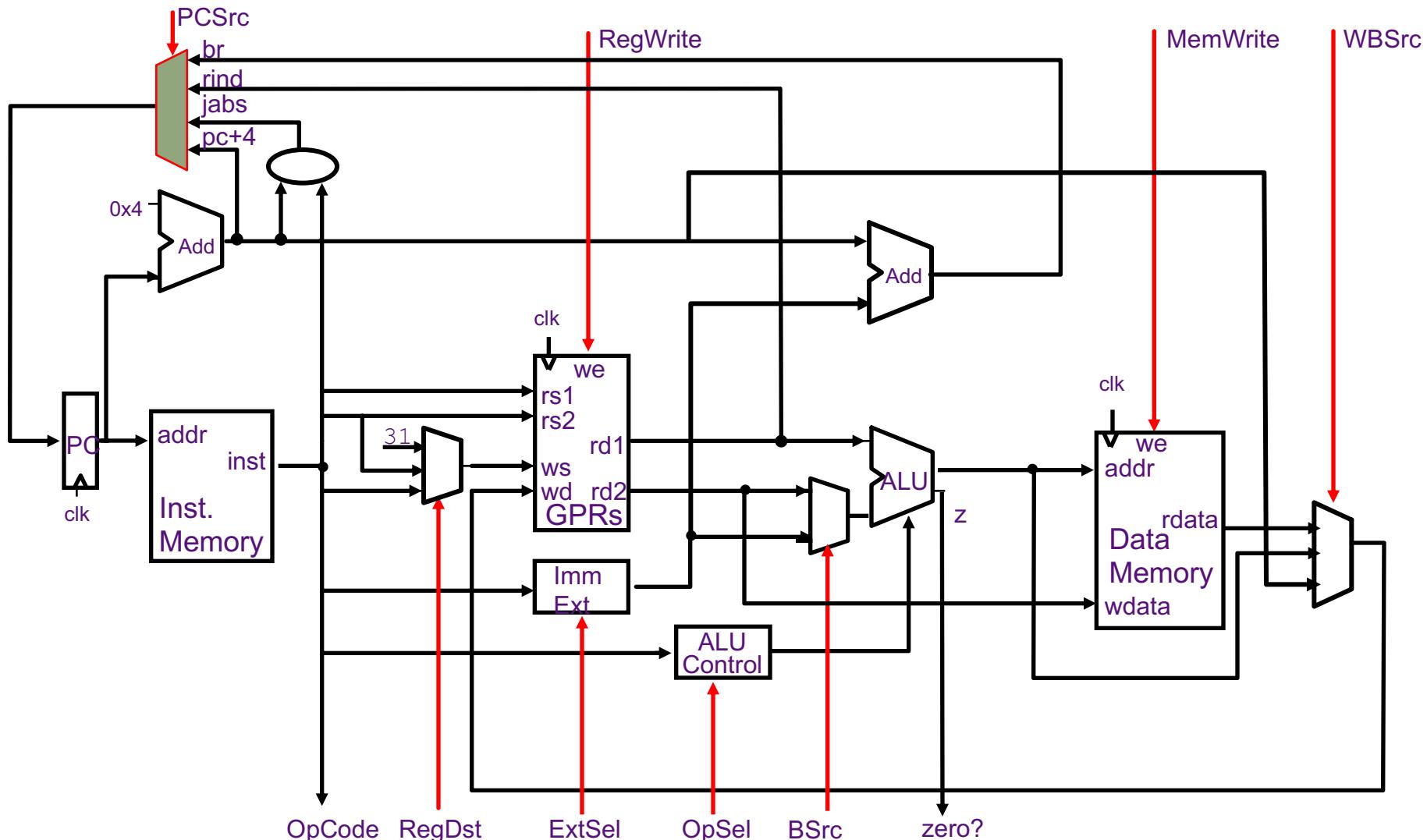
Register-Indirect Jump-&-Link (JALR)



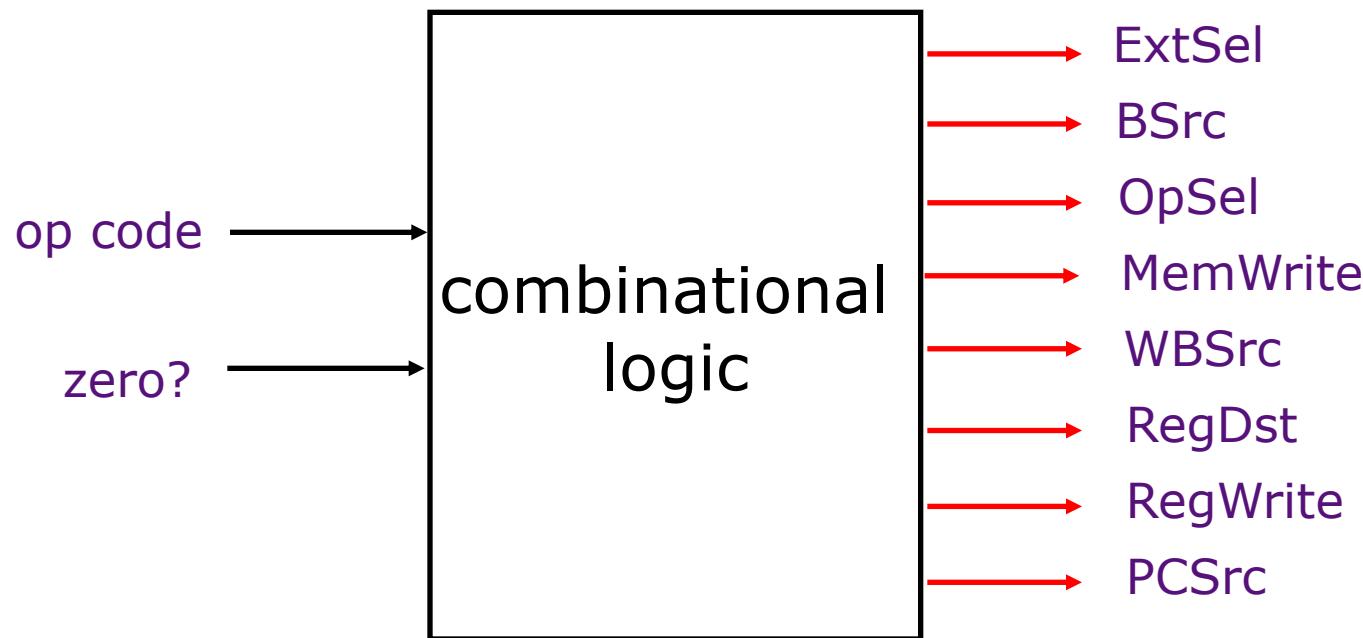
Absolute Jumps (J , JAL)



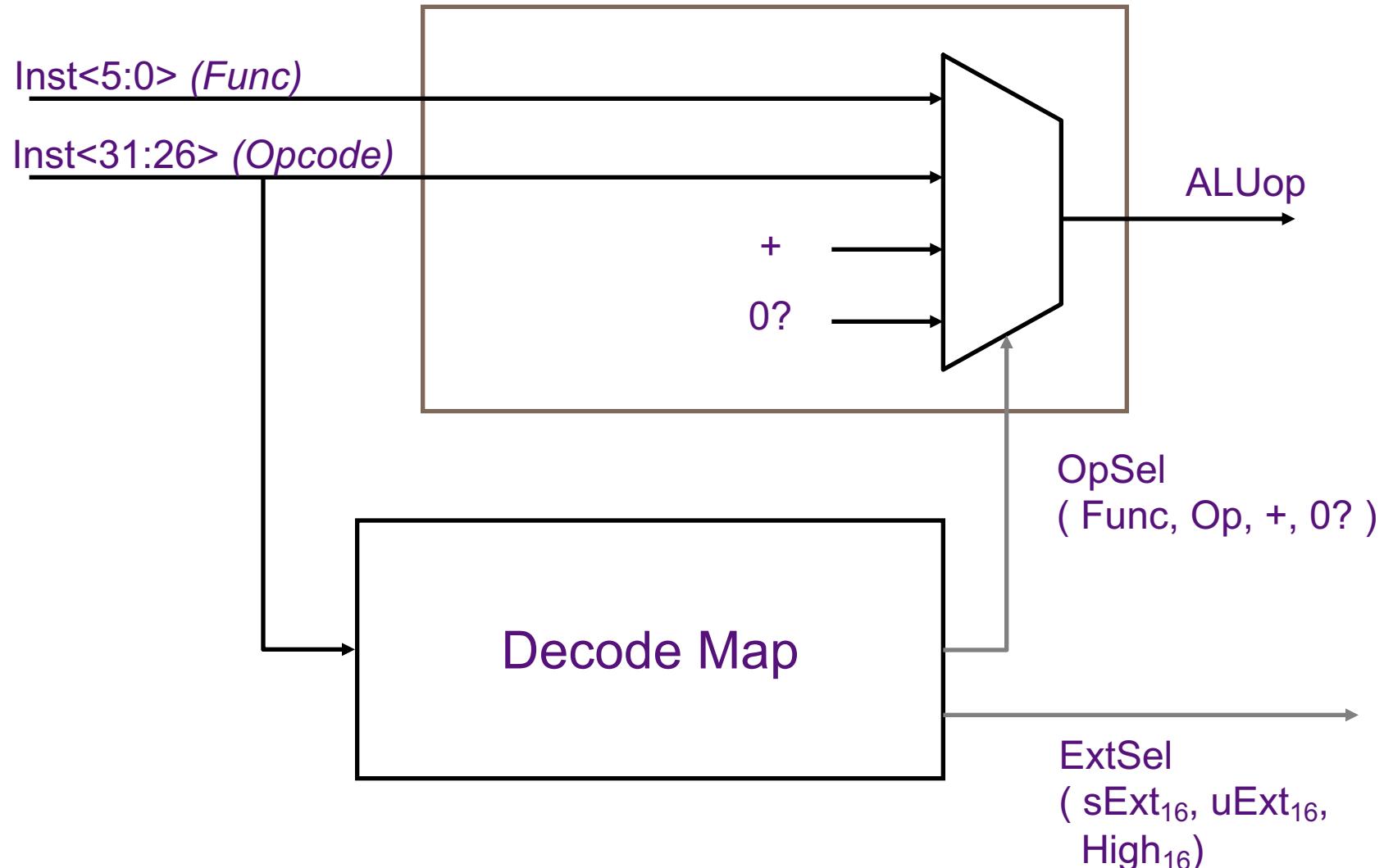
Harvard-Style Datapath for MIPS



Hardwired Control is pure Combinational Logic



ALU Control & Immediate Extension



Hardwired Control Table

Opcode	ExtSel	BSrc	OpSel	MemW	RegW	WBSrc	RegDst	PCSsrc
ALU								
ALUi								
ALUiu								
LW								
SW								
BEQZ _{z=0}								
BEQZ _{z=1}								
J								
JAL								
JR								
JALR								

BSrc = Reg / Imm

RegDst = rt / rd / R31

WBSrc = ALU / Mem / PC

PCSsrc = pc+4 / br / rind / jabs

Single-Cycle Hardwired Control:

Harvard architecture

We will assume

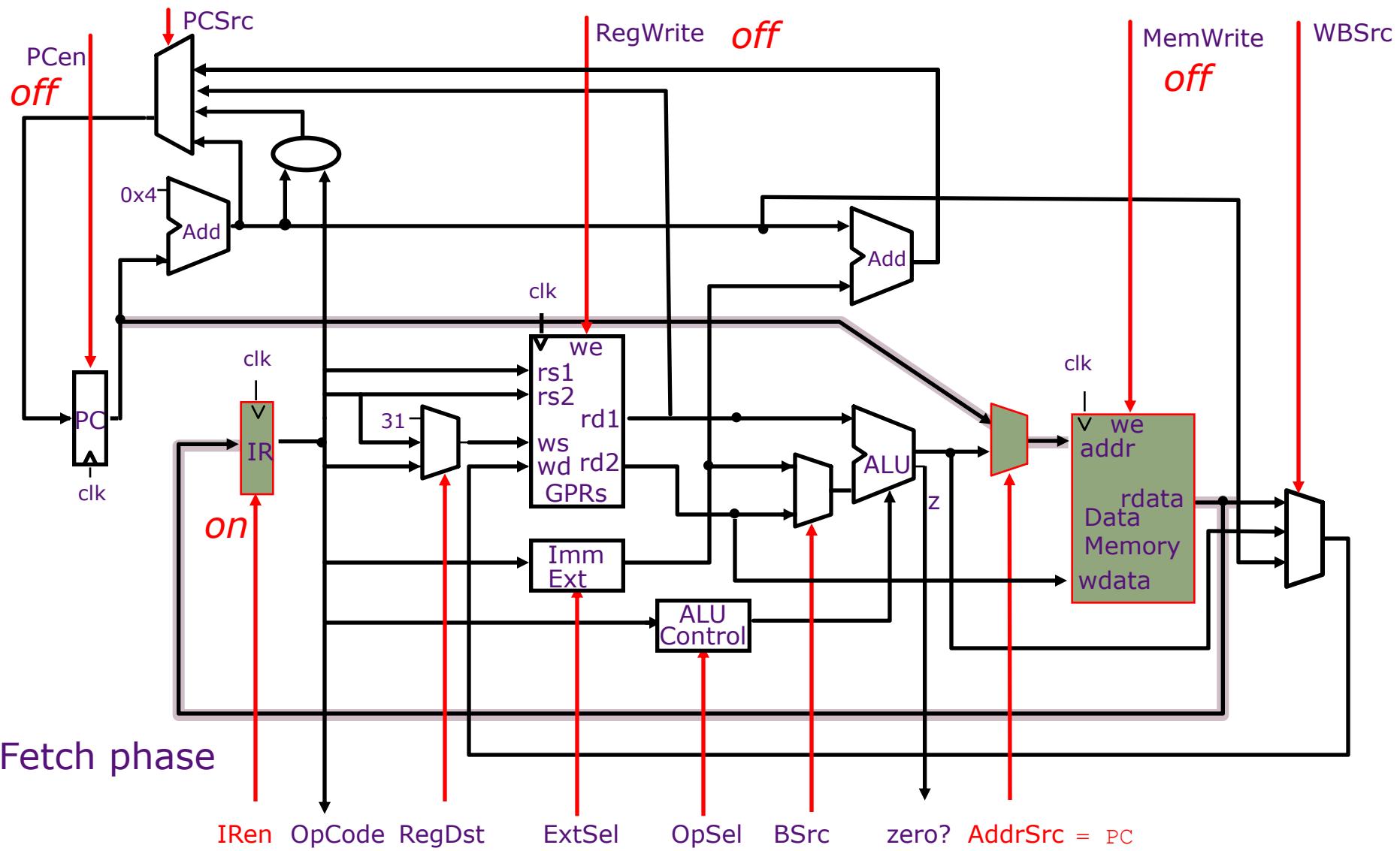
- Clock period is sufficiently long for all of the following steps to be “completed”:
 1. instruction fetch
 2. decode and register fetch
 3. ALU operation
 4. data fetch if required
 5. register write-back setup time
$$\Rightarrow t_C > t_{IFetch} + t_{RFetch} + t_{ALU} + t_{DMem} + t_{RWB}$$
- At the rising edge of the following clock, the PC, the register file and the memory are updated

Princeton challenge

- What problem arises if instructions and data reside in the same memory?

Princeton Microarchitecture

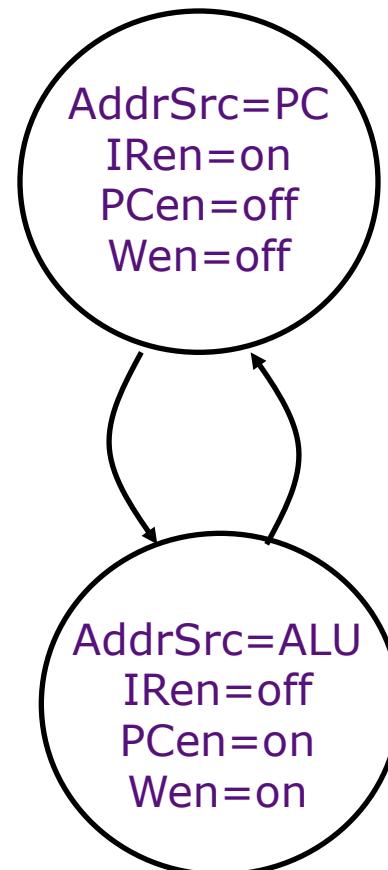
Datapath & Control



Two-State Controller: *Princeton Architecture*

fetch phase

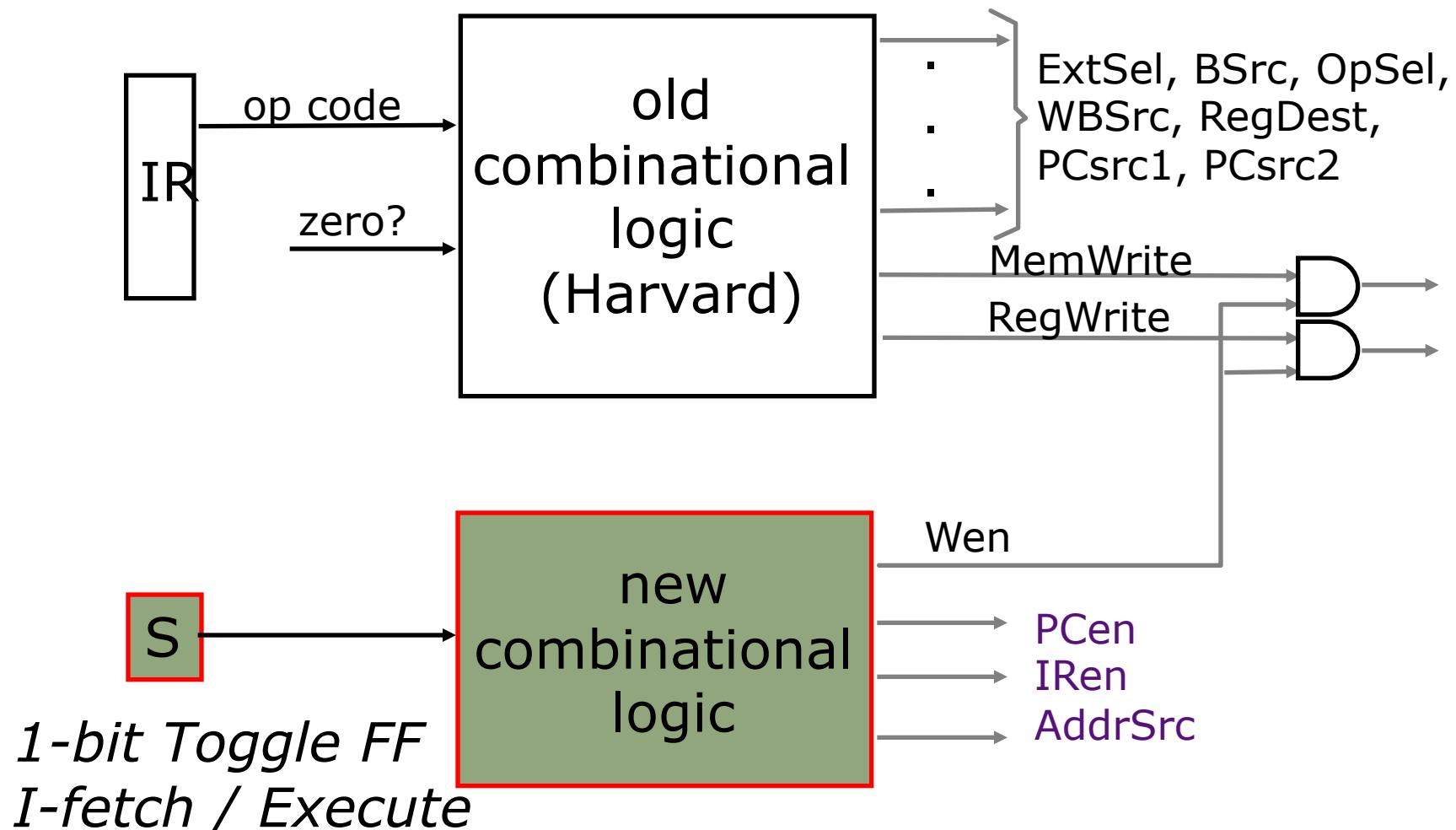
execute phase



A flipflop can be used to remember the phase

Hardwired Controller:

Princeton Architecture



Clock Rate vs CPI

$$t_{C\text{-Princeton}} > \max \{t_M, t_{RF} + t_{ALU} + t_M + t_{WB}\}$$

$$t_{C\text{-Princeton}} > t_{RF} + t_{ALU} + t_M + t_{WB}$$

$$t_{C\text{-Harvard}} > t_M + t_{RF} + t_{ALU} + t_M + t_{WB}$$

Suppose $t_M \gg t_{RF} + t_{ALU} + t_{WB}$

$$t_{C\text{-Princeton}} = 0.5 * t_{C\text{-Harvard}}$$

$$CPI_{\text{Princeton}} = 2$$

$$CPI_{\text{Harvard}} = 1$$

No difference in performance!

Is it possible to design a controller for the Princeton architecture with $CPI < 2$?

CPI = Clock cycles Per Instruction

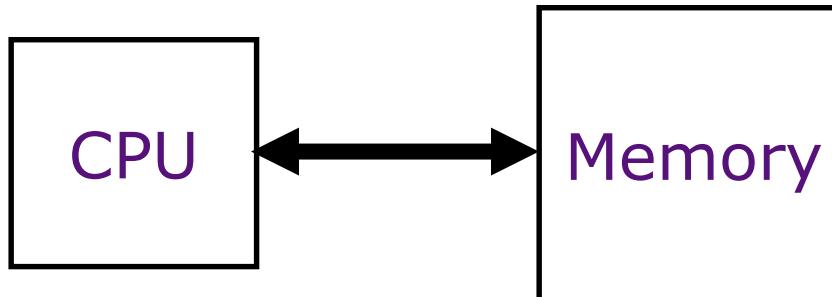
Stay tuned!

Cache Organization

Daniel Sanchez

Computer Science and Artificial Intelligence Laboratory
M.I.T.

CPU-Memory Bottleneck



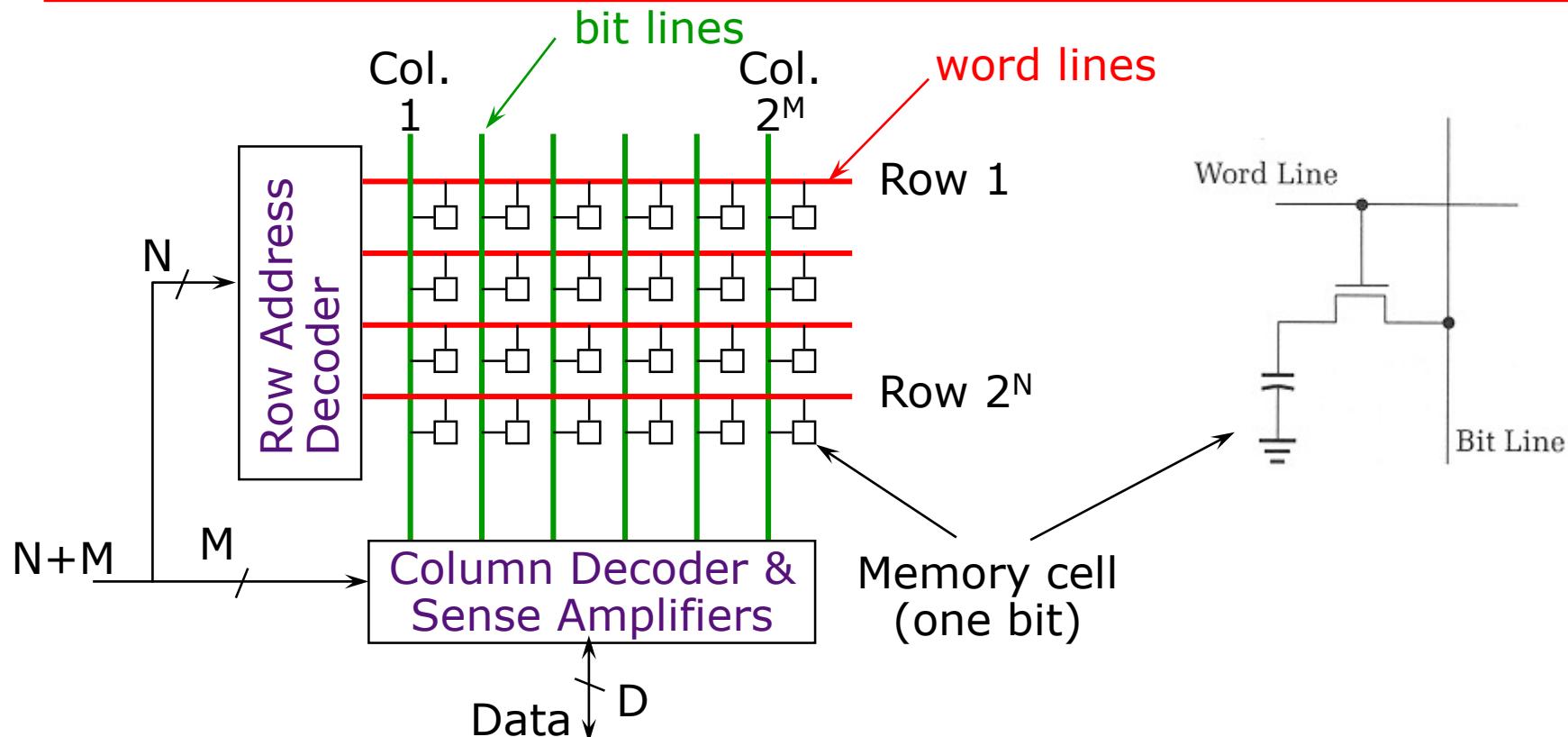
Performance of high-speed computers is usually limited by memory *bandwidth* & *latency*

- Latency (time for a single access)
Memory access time $>>$ Processor cycle time
- Bandwidth (number of accesses per unit time)
if fraction m of instructions access memory,
 $\Rightarrow 1+m$ memory references / instruction
 $\Rightarrow \text{CPI} = 1$ requires $1+m$ memory refs / cycle

Memory Technology

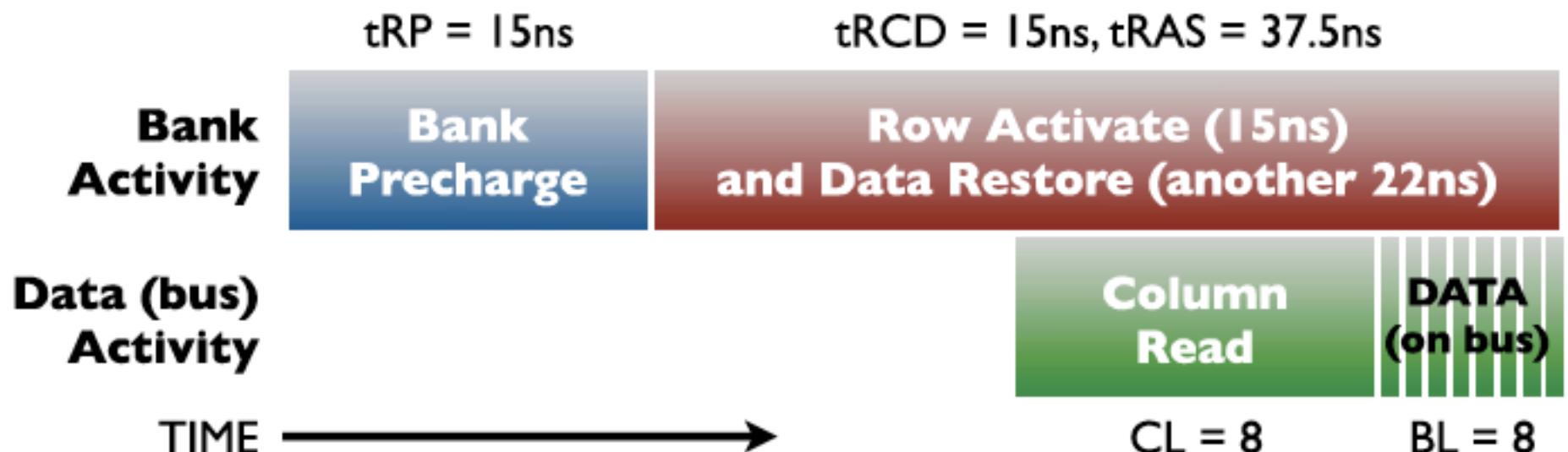
- Early machines used a variety of memory technologies
 - Manchester Mark I used CRT Memory Storage
 - EDVAC used a mercury delay line
- Core memory was first large scale reliable main memory
 - Invented by Forrester in late 40s at MIT for Whirlwind project
 - Bits stored as magnetization polarity on small ferrite cores threaded onto 2 dimensional grid of wires
- First commercial DRAM was Intel 1103
 - 1Kbit of storage on single chip
 - charge on a capacitor used to hold value
- Semiconductor memory quickly replaced core in 1970s
 - Intel formed to exploit market for semiconductor memory
- Flash memory
 - Slower, but denser than DRAM. Also non-volatile, but with wearout issues
- Phase change memory (PCM, 3D XPoint)
 - Slightly slower, but much denser than DRAM and non-volatile

DRAM Architecture



- Bits stored in 2-dimensional arrays on chip
- Modern chips have around 8 logical banks on each chip
 - Each logical bank physically implemented as many smaller arrays

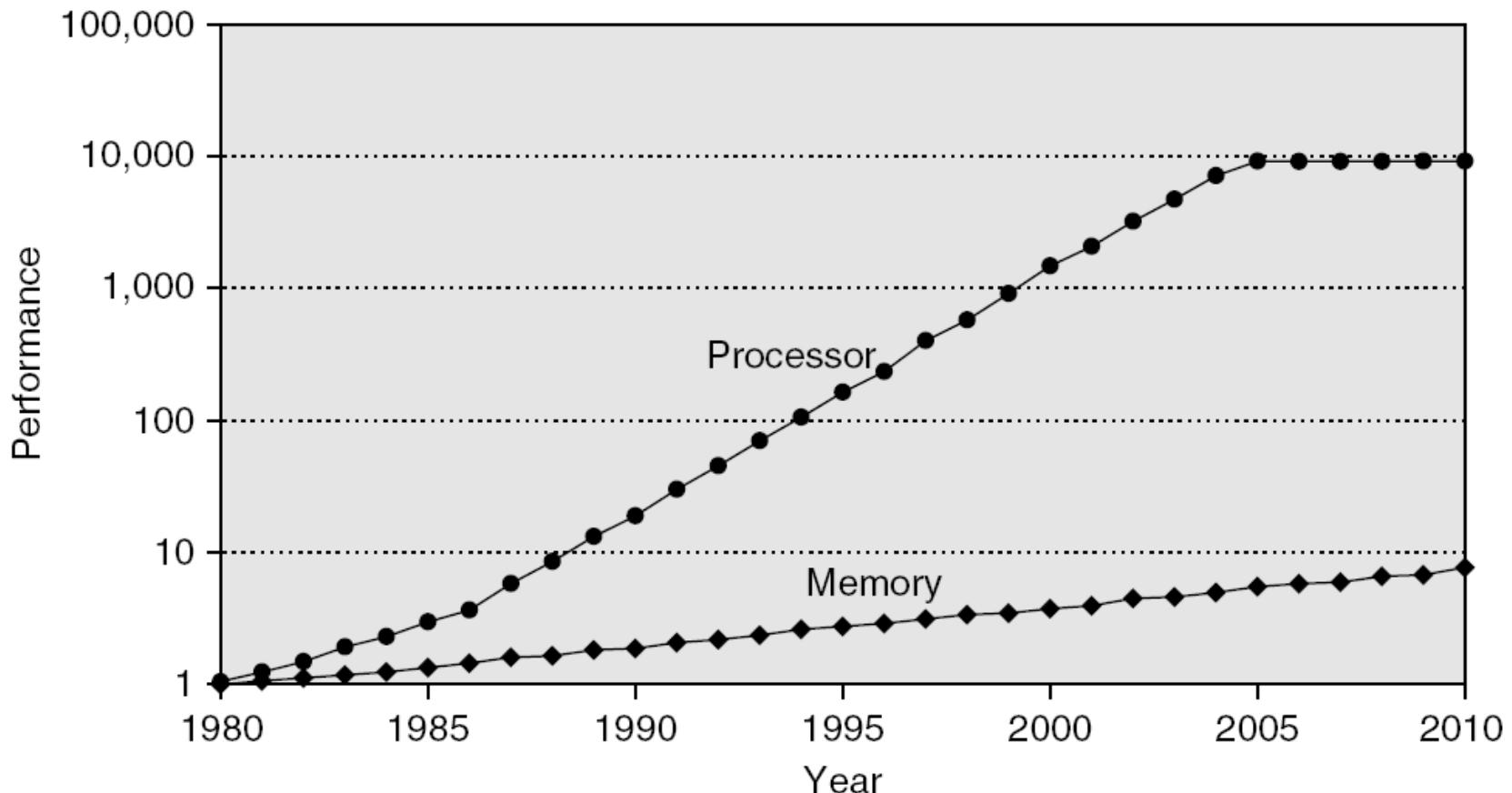
DRAM timing



DRAM Spec:

CL, tRCD, tRP, tRAS, e.g., 9-9-9-24

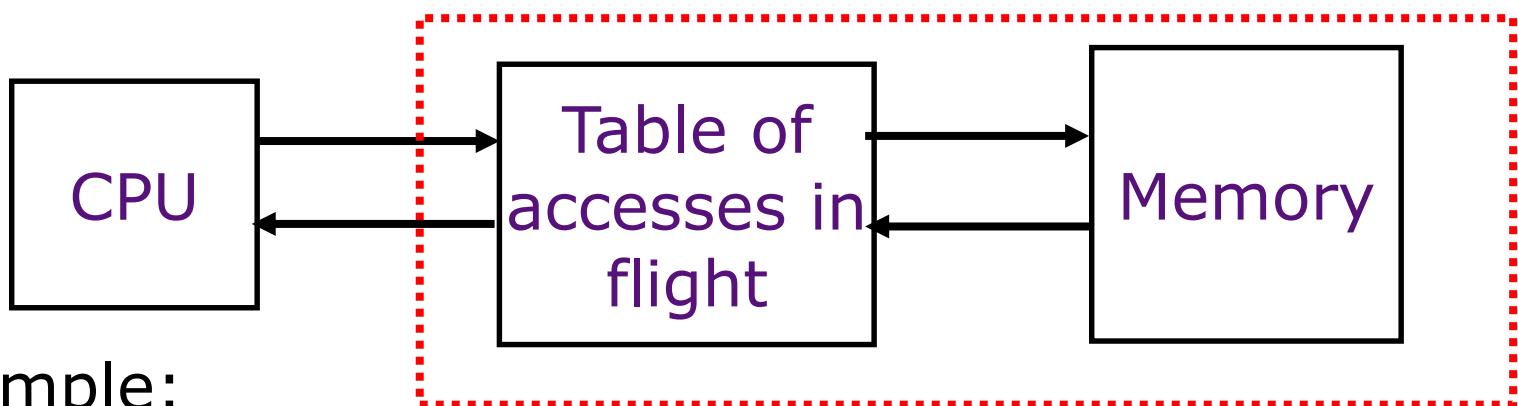
Processor-DRAM Gap (latency)



Four-issue 2GHz superscalar accessing 100ns DRAM could execute 800 instructions during time for one memory access!

Little's Law

$$\text{Throughput } (T) = \text{Number in Flight } (N) / \text{Latency } (L)$$



Example:

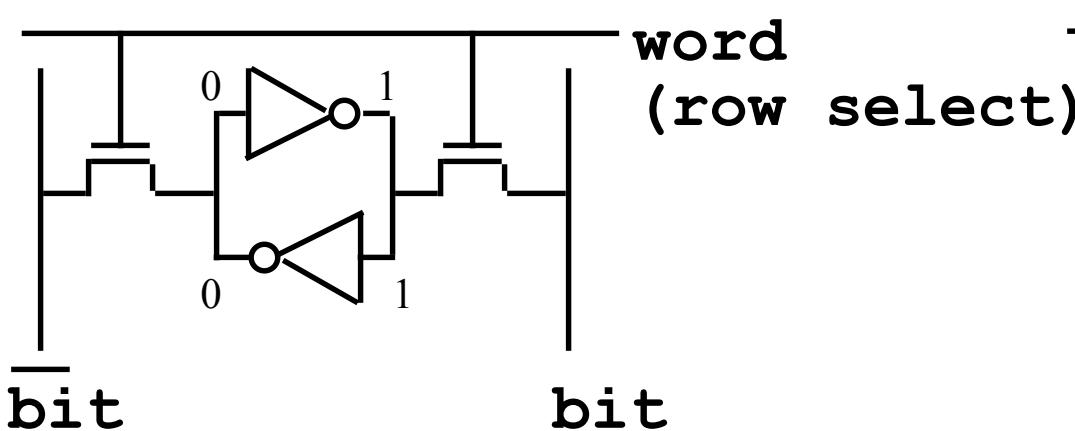
- Assume infinite-bandwidth memory
- 100 cycles / memory reference
- 1 + 0.2 memory references / instruction

$$\Rightarrow \text{Table size} = 1.2 * 100 = 120 \text{ entries}$$

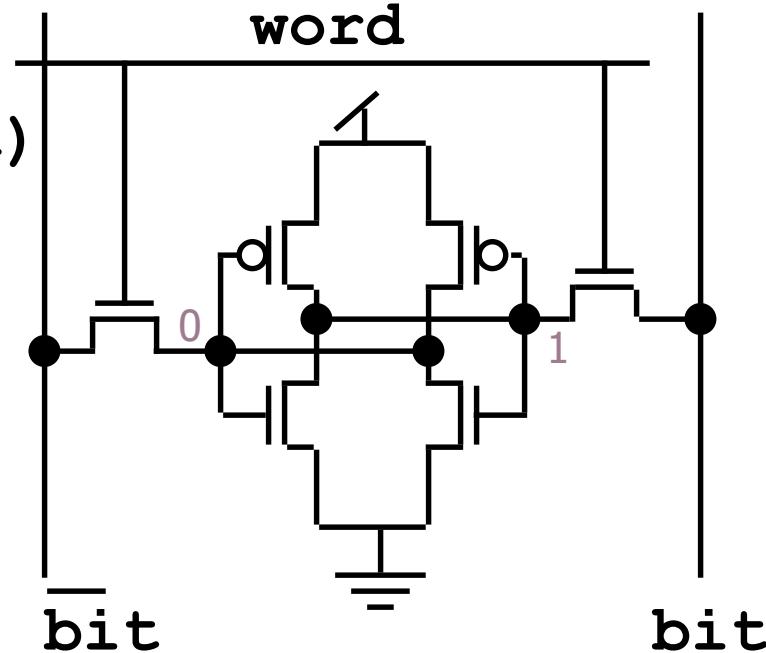
120 independent memory operations in flight!

Basic Static RAM Cell

6-Transistor SRAM Cell



- Write:
 1. Drive bit lines ($\text{bit}=1$, $\overline{\text{bit}}=0$)
 2. Select word line
 - Read:
 1. Precharge bit and $\overline{\text{bit}}$ to Vdd
 2. Select word line
 3. Cell pulls one bit line low
 4. Column sense amp detects difference between bit & $\overline{\text{bit}}$



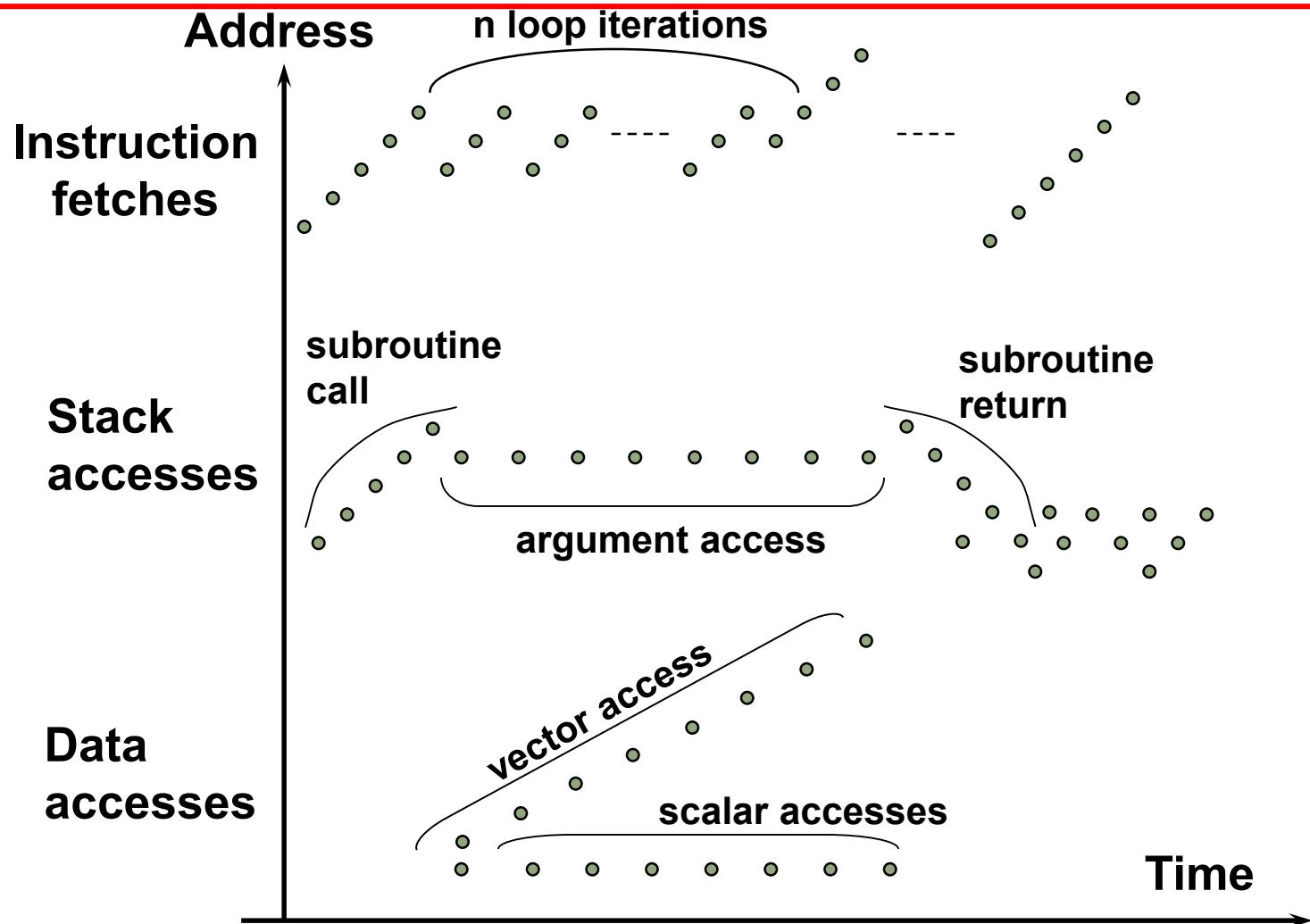
Multilevel Memory

Strategy: Reduce average latency using small, fast memories called caches.

Caches are a mechanism to reduce memory latency based on the empirical observation that the patterns of memory references made by a processor are often highly predictable:

	<i>PC</i>
...	96
<i>Loop:</i>	
<i>add r2, r1, r1</i>	<i>100</i>
<i>subi r3, r3, #1</i>	<i>104</i>
<i>bnez r3, Loop</i>	<i>108</i>
...	112

Typical Memory Reference Patterns

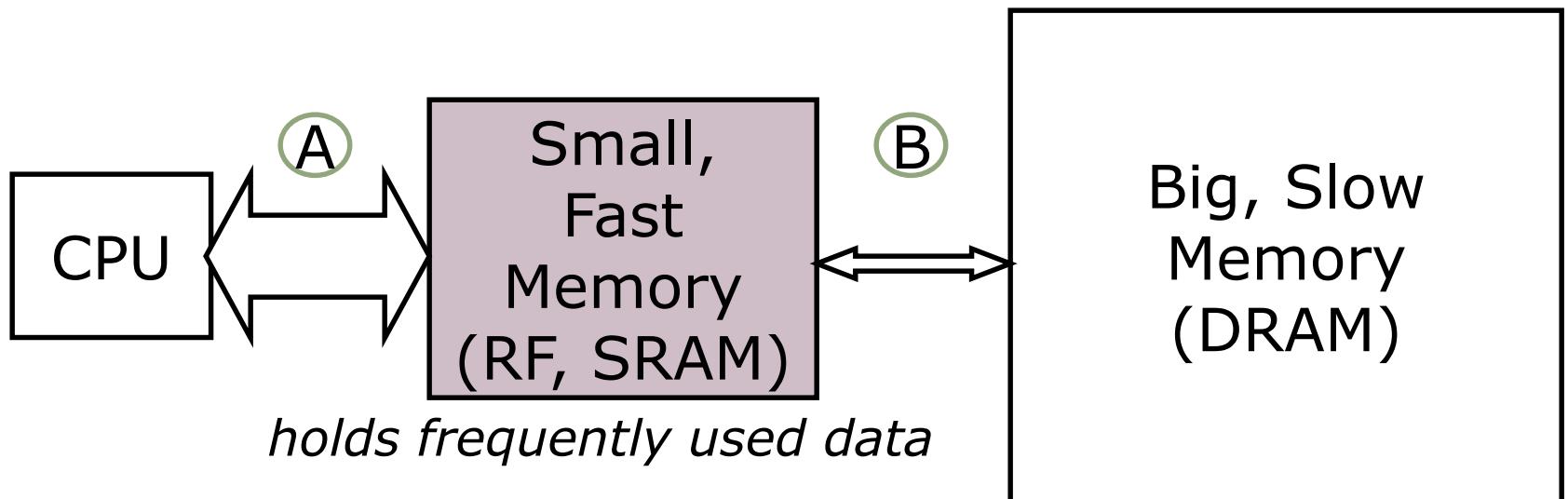


Common Predictable Patterns

Two predictable properties of memory references:

- *Temporal Locality*: If a location is referenced, it is likely to be referenced again in the near future
- *Spatial Locality*: If a location is referenced, it is likely that locations near it will be referenced in the near future

Memory Hierarchy



- *size:* Register << SRAM << DRAM *why?*
- *latency:* Register << SRAM << DRAM *why?*
- *bandwidth:* on-chip >> off-chip *why?*

On a data access:

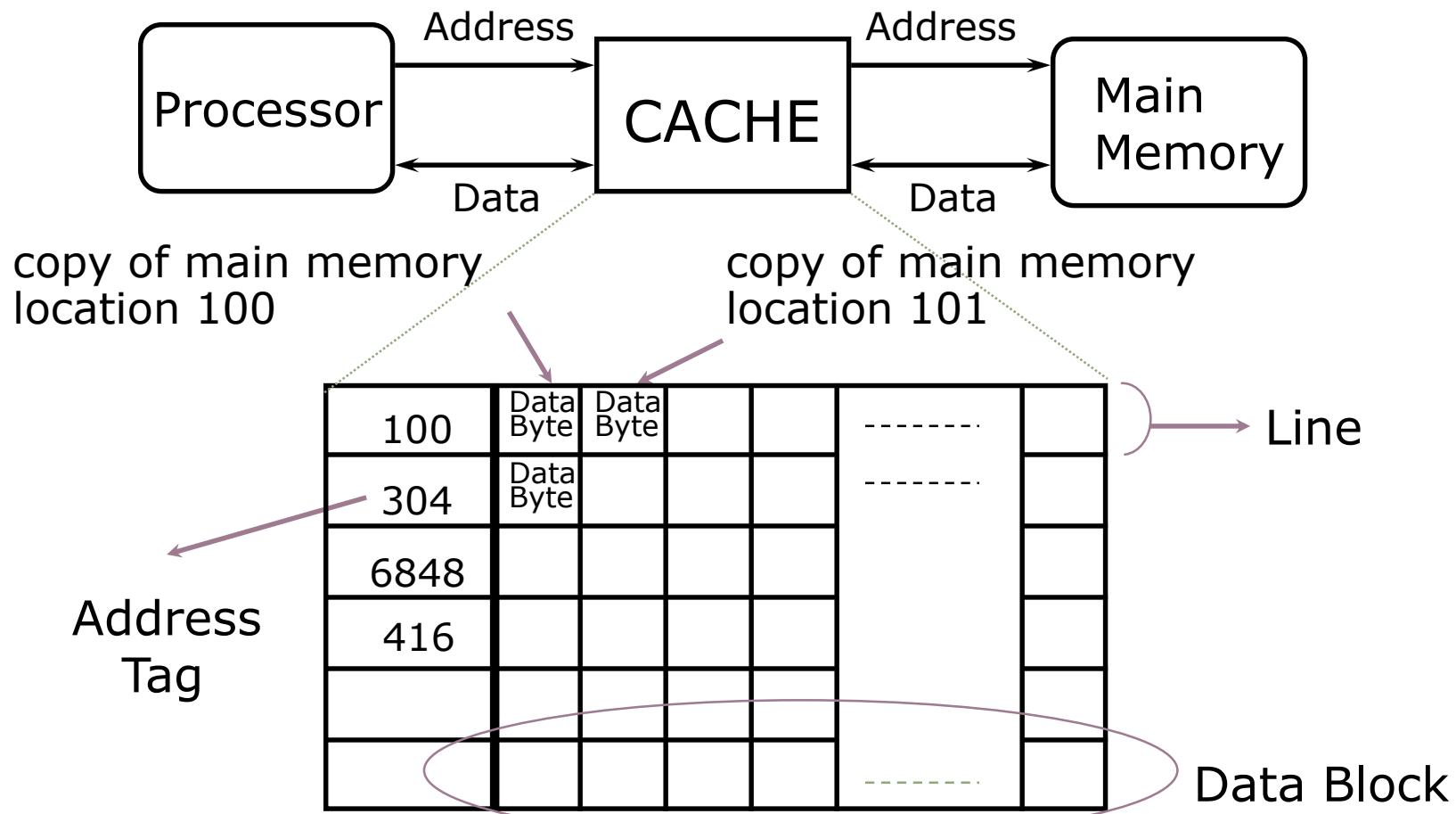
hit (data \in fast memory) \Rightarrow low latency access

miss (data \notin fast memory) \Rightarrow long latency access (*DRAM*)

Management of Memory Hierarchy

- Small/fast storage, e.g., registers
 - Address usually specified in instruction
 - Generally implemented directly as a register file
 - but hardware might do things behind software's back, e.g., stack management, register renaming
- Large/slower storage, e.g., memory
 - Address usually computed from values in register
 - Generally implemented as a cache hierarchy
 - hardware decides what is kept in fast memory
 - but software may provide "hints", e.g., don't cache or prefetch

Inside a Cache



Q: How many bits needed in tag?

Cache Algorithm (Read)

Look at Processor Address, search cache tags to find match.
Then either

Found in cache
a.k.a. HIT

Return copy
of data from
cache

Not in cache
a.k.a. MISS

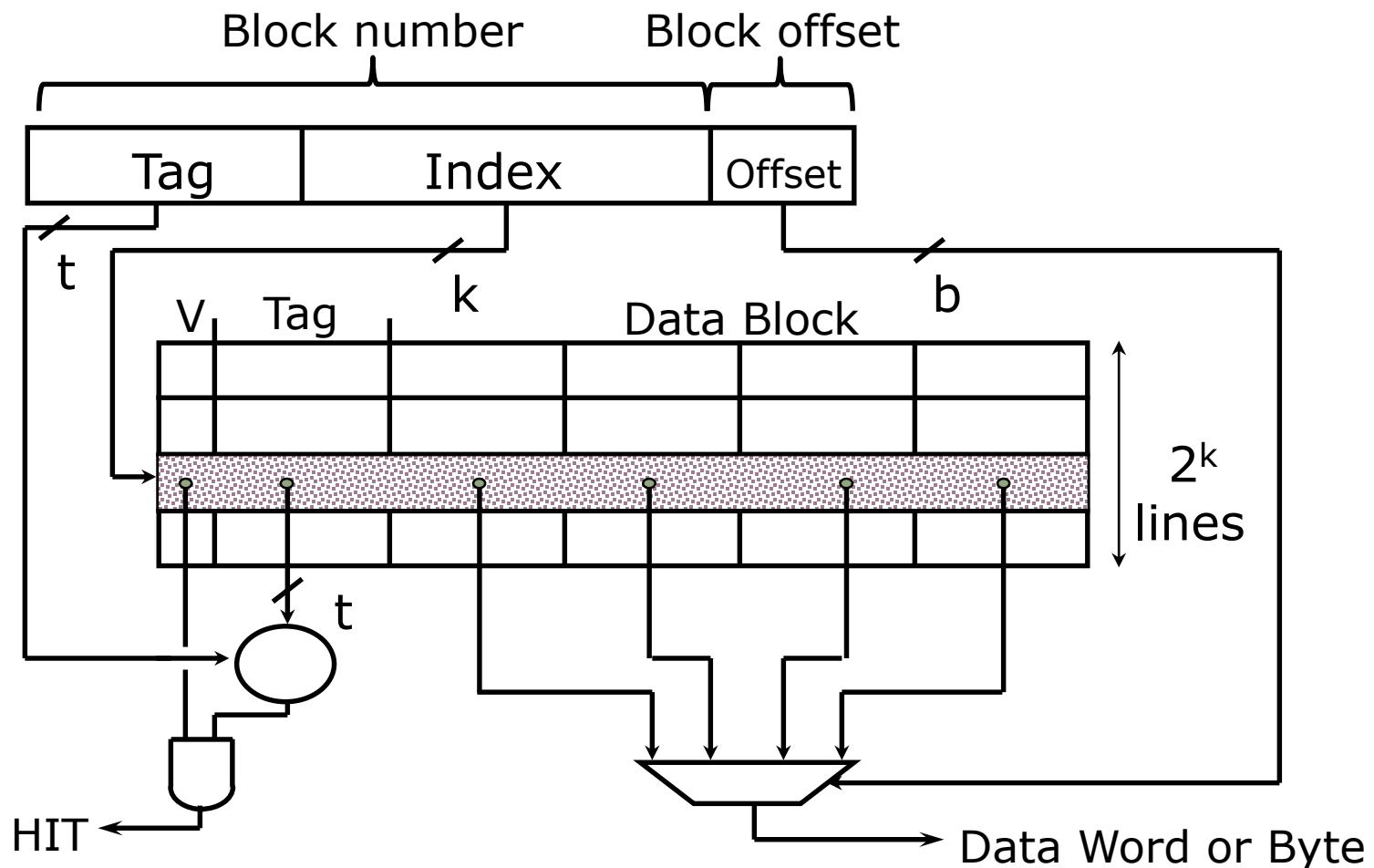
Read block of data from
Main Memory

Wait ...

Return data to processor
and update cache

Which line do we replace?

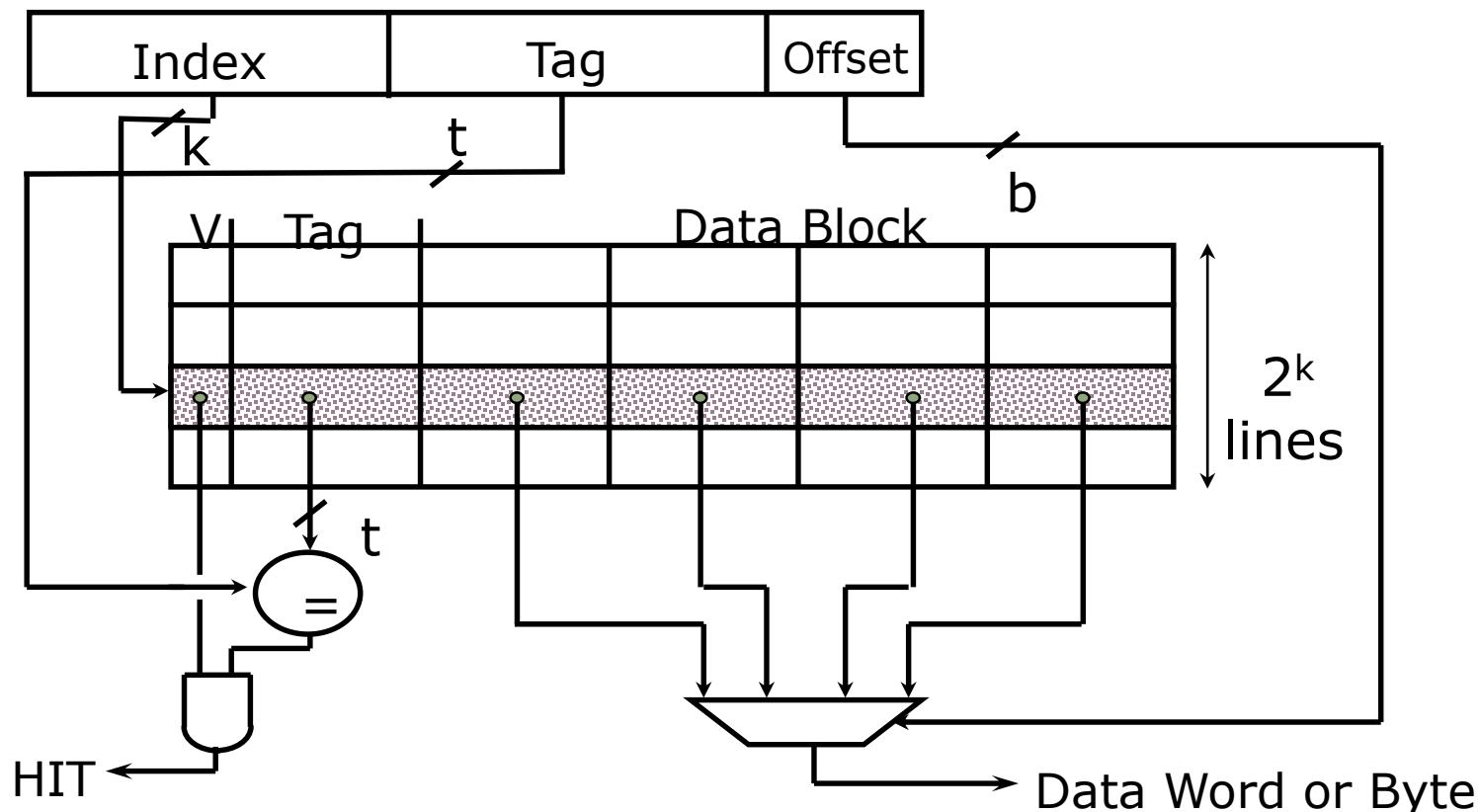
Direct-Mapped Cache



Q: What is a bad reference pattern?

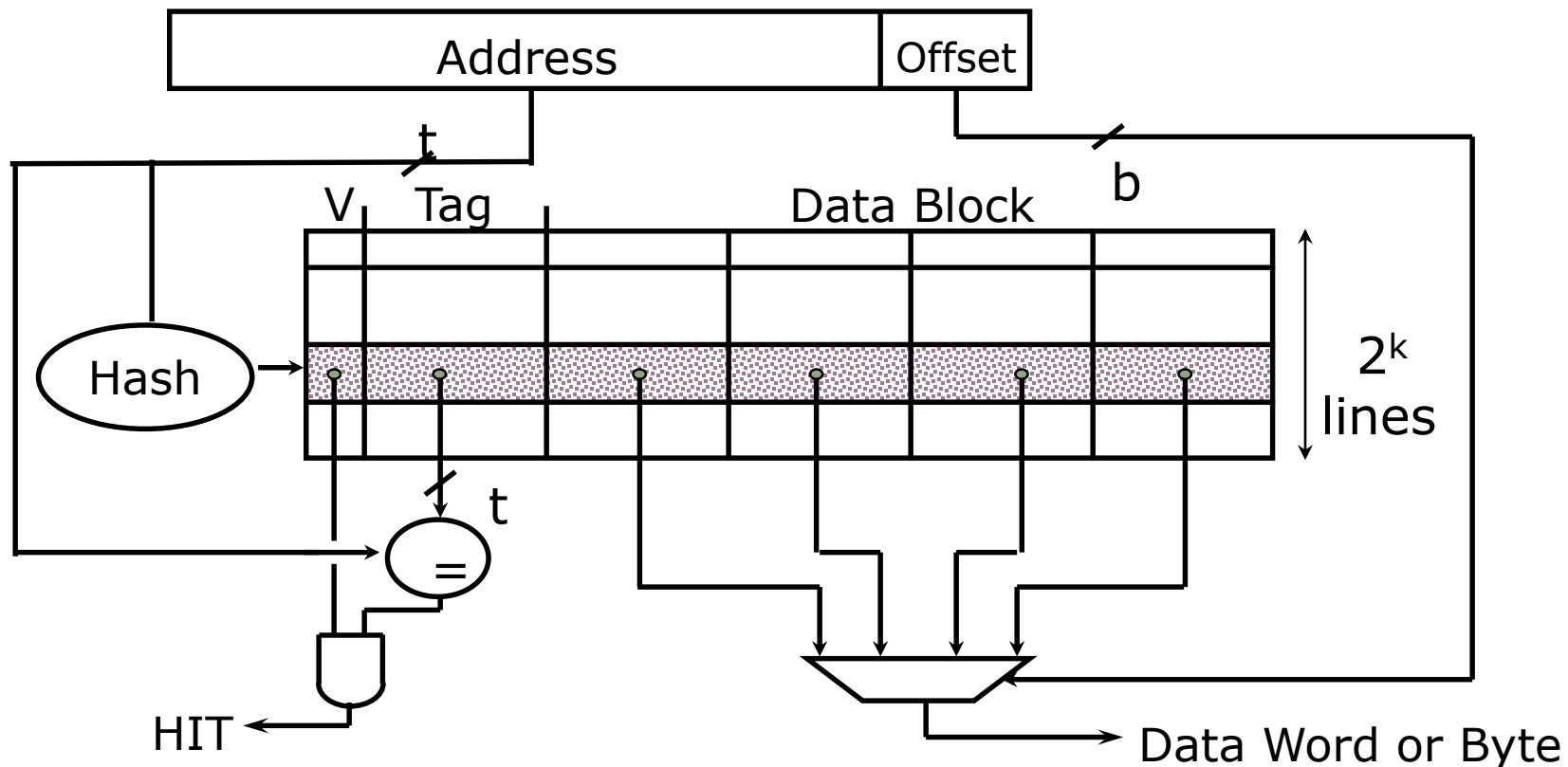
Direct Map Address Selection

higher-order vs. lower-order address bits



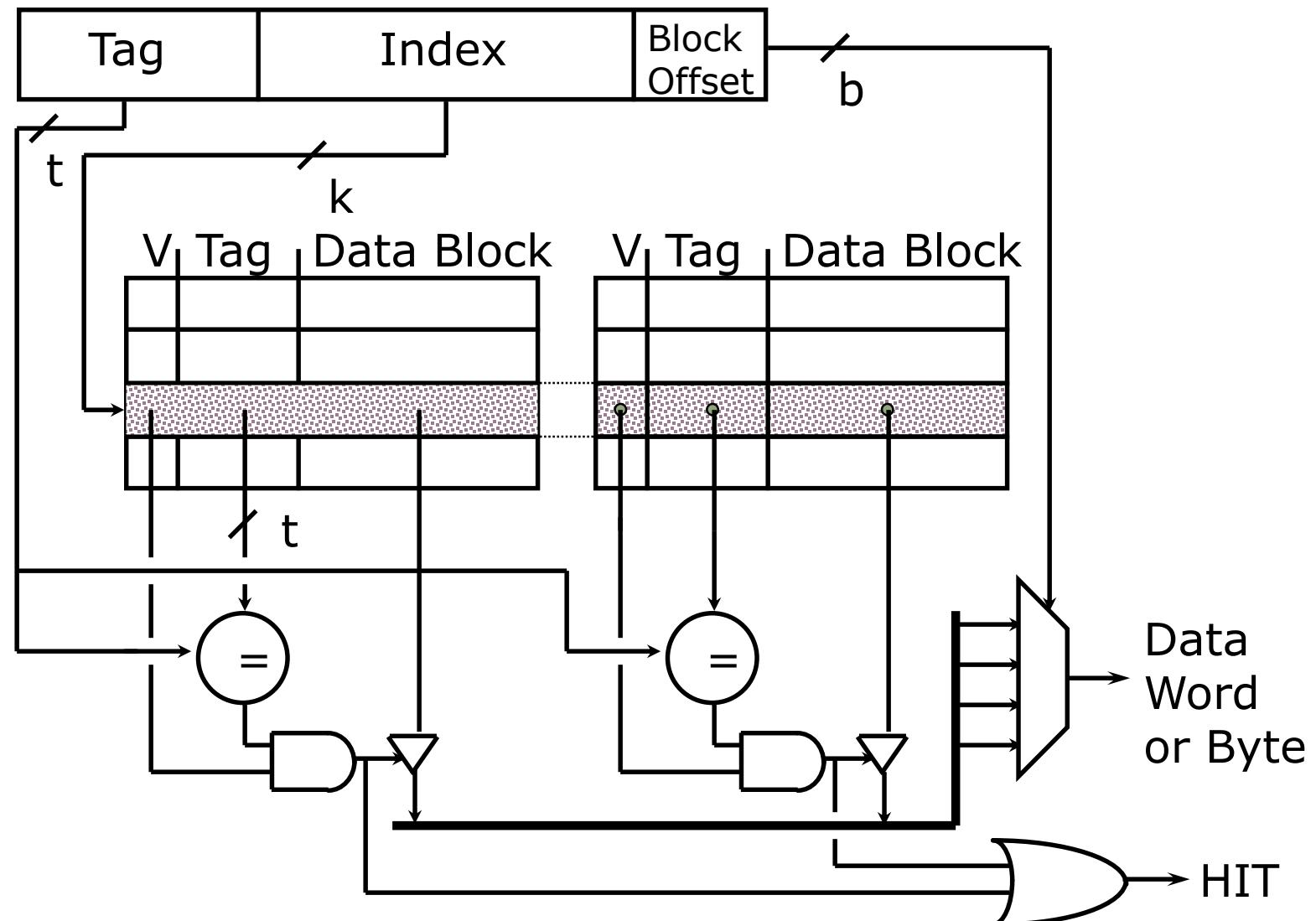
Q: Why might this be undesirable? _____

Hashed Address Mapping

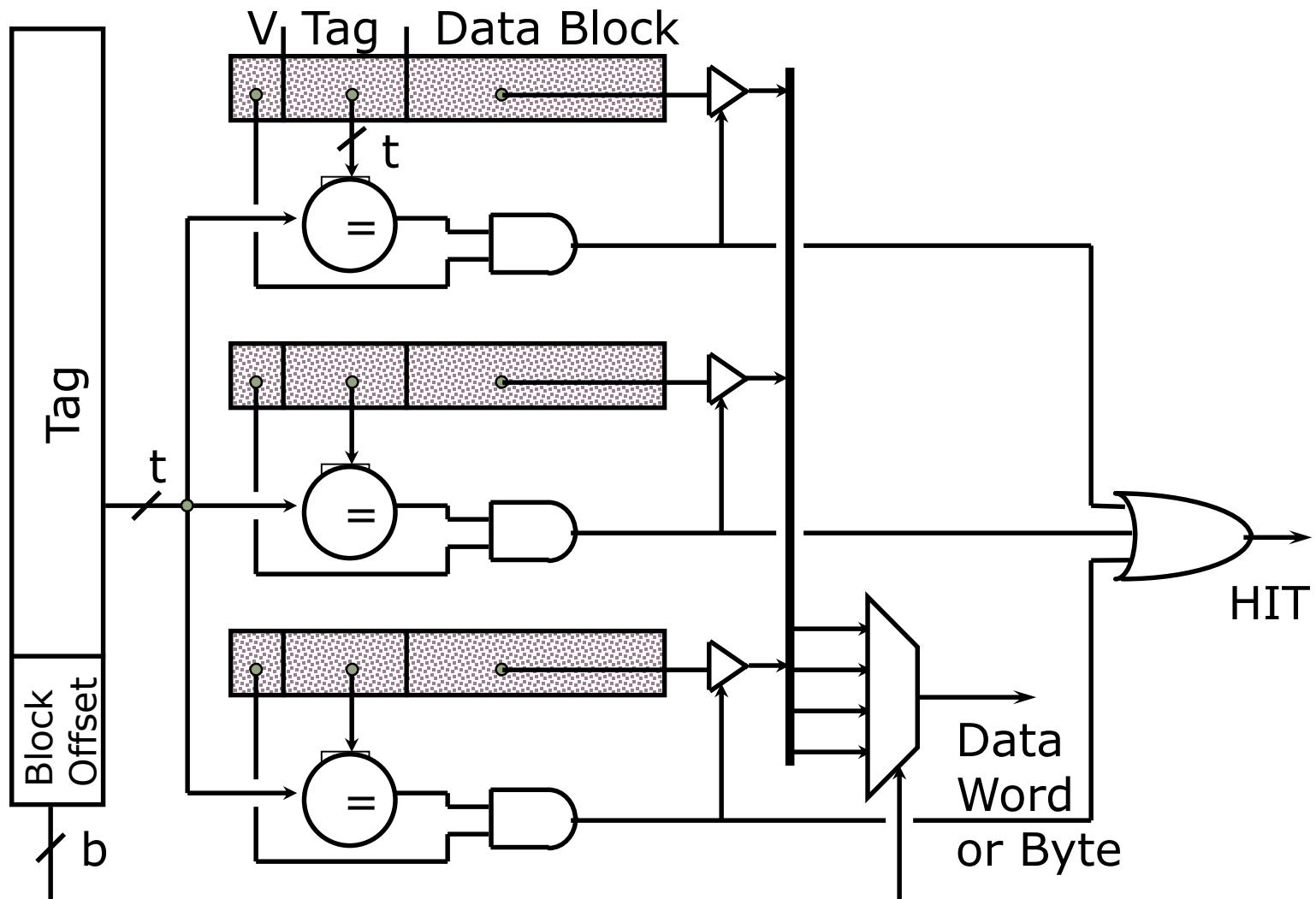


Q: What are the tradeoffs of hashing?

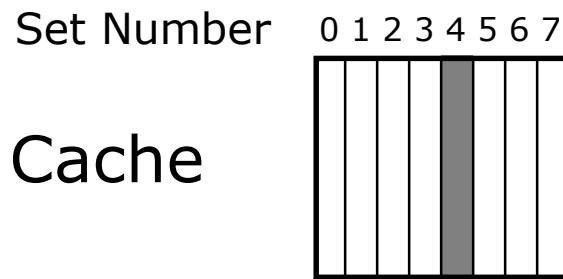
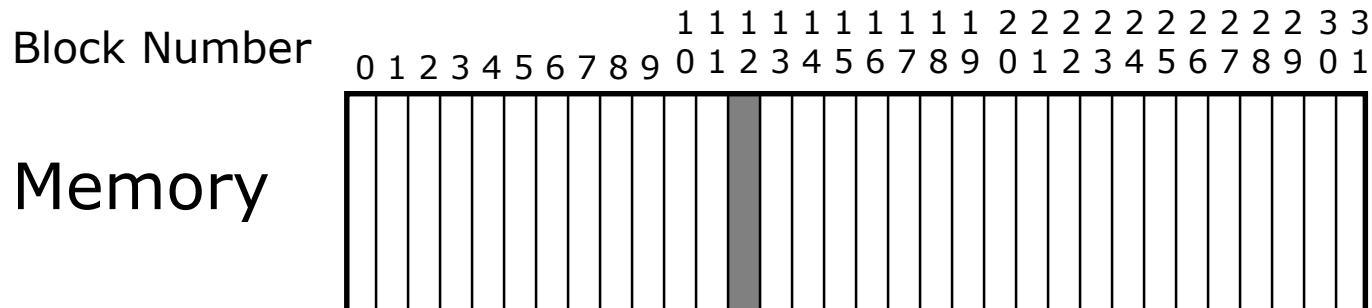
2-Way Set-Associative Cache



Fully Associative Cache

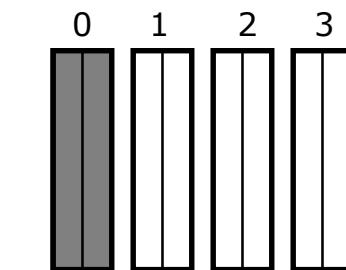


Placement Policy

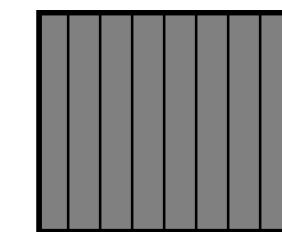


block 12
can be placed

Direct
Mapped
only into
block 4
(12 mod 8)



(2-way) Set
Associative
anywhere in
set 0
(12 mod 4)



Fully
Associative
anywhere

Improving Cache Performance

Average memory access time =
 Hit time + Miss rate × Miss penalty

To improve performance:

- reduce the hit time
- reduce the miss rate (e.g., larger, better policy)
- reduce the miss penalty (e.g., L2 cache)

What is the simplest design strategy?

Causes for Cache Misses

- *Compulsory:*
 - First reference to a block a.k.a. cold start misses
 - misses that would occur even with infinite cache
- *Capacity:*
 - cache is too small to hold all data the program needs
 - misses that would occur even under perfect placement & replacement policy
- *Conflict:*
 - misses from collisions due to block-placement strategy
 - misses that would not occur with full associativity

Effect of Cache Parameters on Performance

	Larger capacity cache	Higher associativity cache	Larger block size cache *
Compulsory misses			
Capacity misses			
Conflict misses			
Hit latency			
Miss latency			

* Assume substantial spatial locality

Block-level Optimizations

- Tags are too large, i.e., too much overhead
 - Simple solution: Larger blocks, but miss penalty could be large.
- Sub-block placement (aka sector cache)
 - A valid bit added to units smaller than the full block, called sub-blocks
 - Only read a sub-block on a miss
 - *If a tag matches, is the sub-block in the cache?*

100
300
204

1	1	1	1	1
1	1	0	0	
0	1	0		1

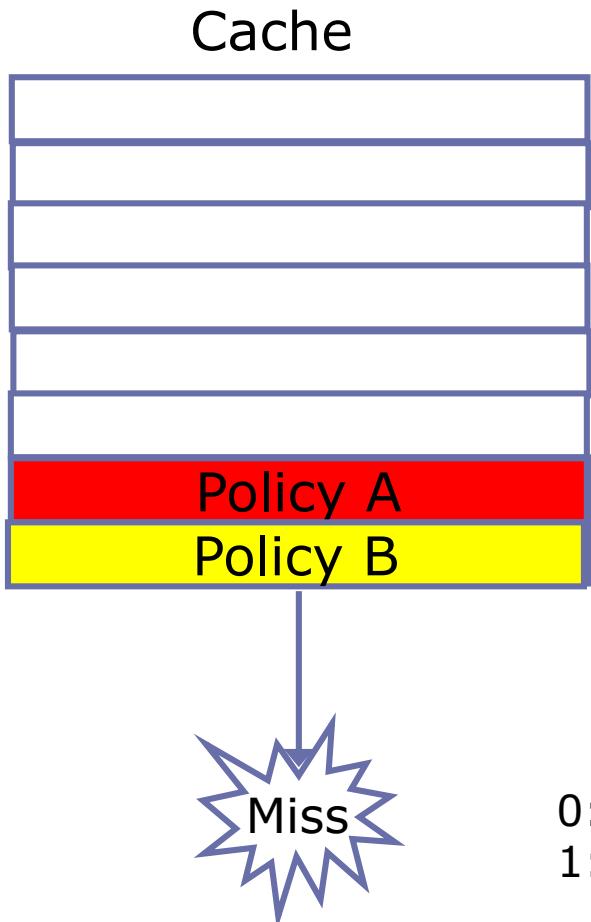
Replacement Policy

Which block from a set should be evicted?

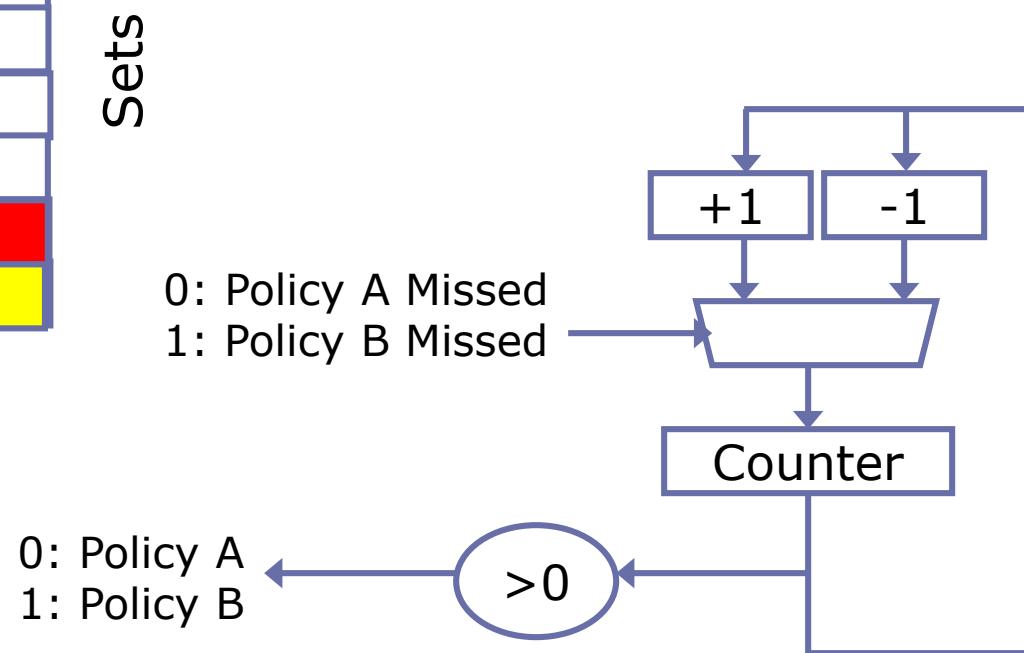
- Random
- Least Recently Used (LRU)
 - LRU cache state must be updated on every access
 - true implementation only feasible for small sets (2-way)
 - pseudo-LRU binary tree was often used for 4-8 way
- First In, First Out (FIFO) a.k.a. Round-Robin
 - used in highly associative caches
- Not Least Recently Used (NLRU)
 - FIFO with exception for most recently used block or blocks
- One-bit LRU
 - Each way represented by a bit. Set on use, replace first unused.

Multiple replacement policies

Use the best replacement policy for a program

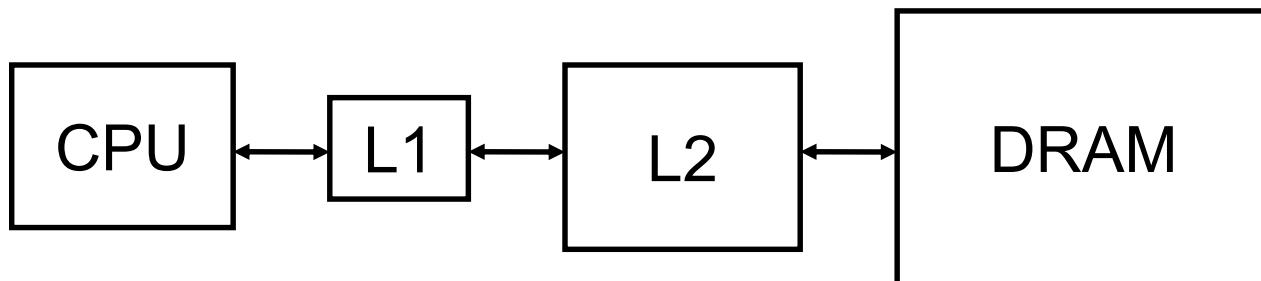


How do we decide
which policy to use?



Multilevel Caches

- A memory cannot be large and fast
- Add level of cache to reduce miss penalty
 - Each level can have longer latency than level above
 - So, increase sizes of cache at each level



Metrics:

Local miss rate = misses in cache / accesses to cache

Global miss rate = misses in cache / CPU memory accesses

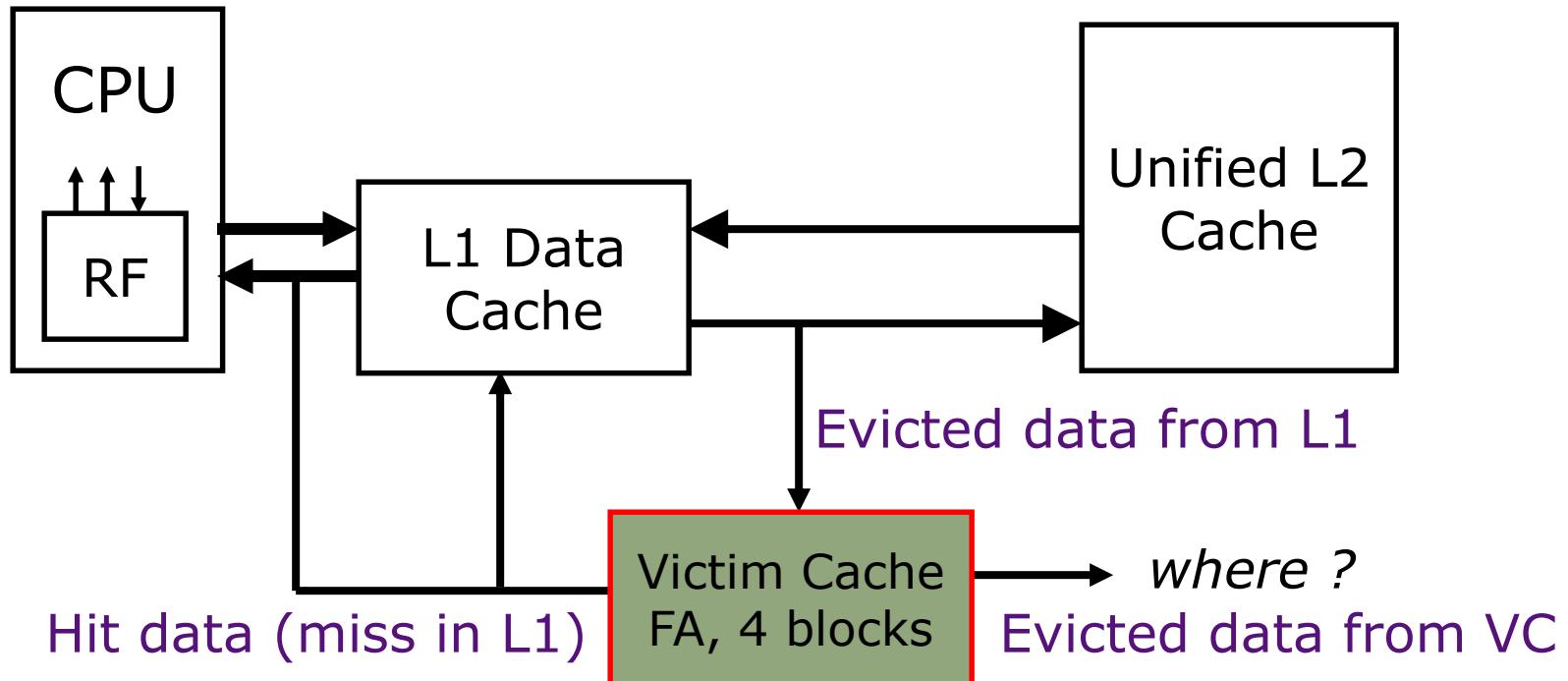
Misses per instruction (MPI) = misses in cache / number of instructions

Inclusion Policy

- Inclusive multilevel cache:
 - Inner cache holds copies of data in outer cache
 - On miss, line inserted in inner and outer cache; replacement in outer cache invalidates line in inner cache
 - External accesses need only check outer cache
 - Commonly used (e.g., Intel CPUs up to Broadwell)
- Non-inclusive multilevel caches:
 - Inner cache may hold data not in outer cache
 - Replacement in outer cache doesn't invalidate line in inner cache
 - Used in Intel Skylake, ARM
- Exclusive multilevel caches:
 - Inner cache and outer cache hold different data
 - Swap lines between inner/outer caches on miss
 - Used in AMD processors

Why choose one type or the other?

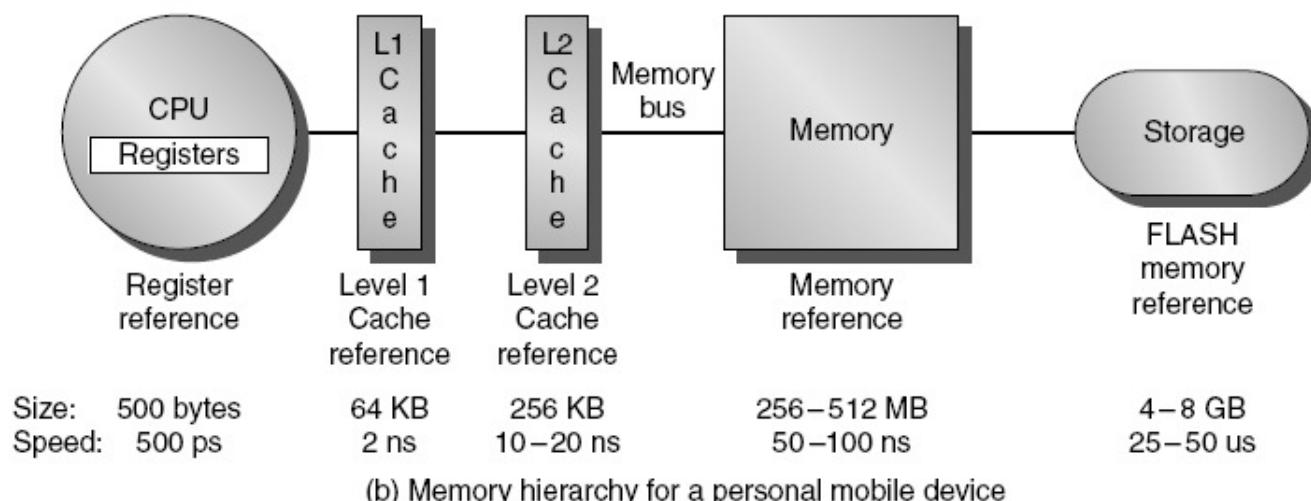
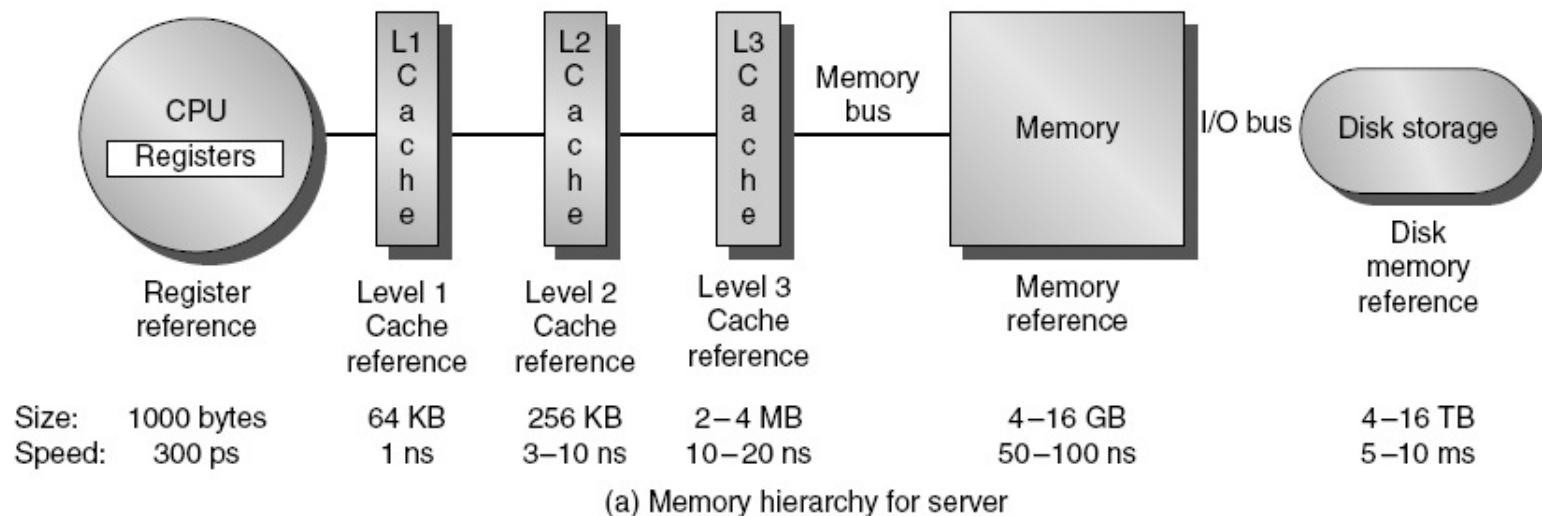
Victim Caches (HP 7200)



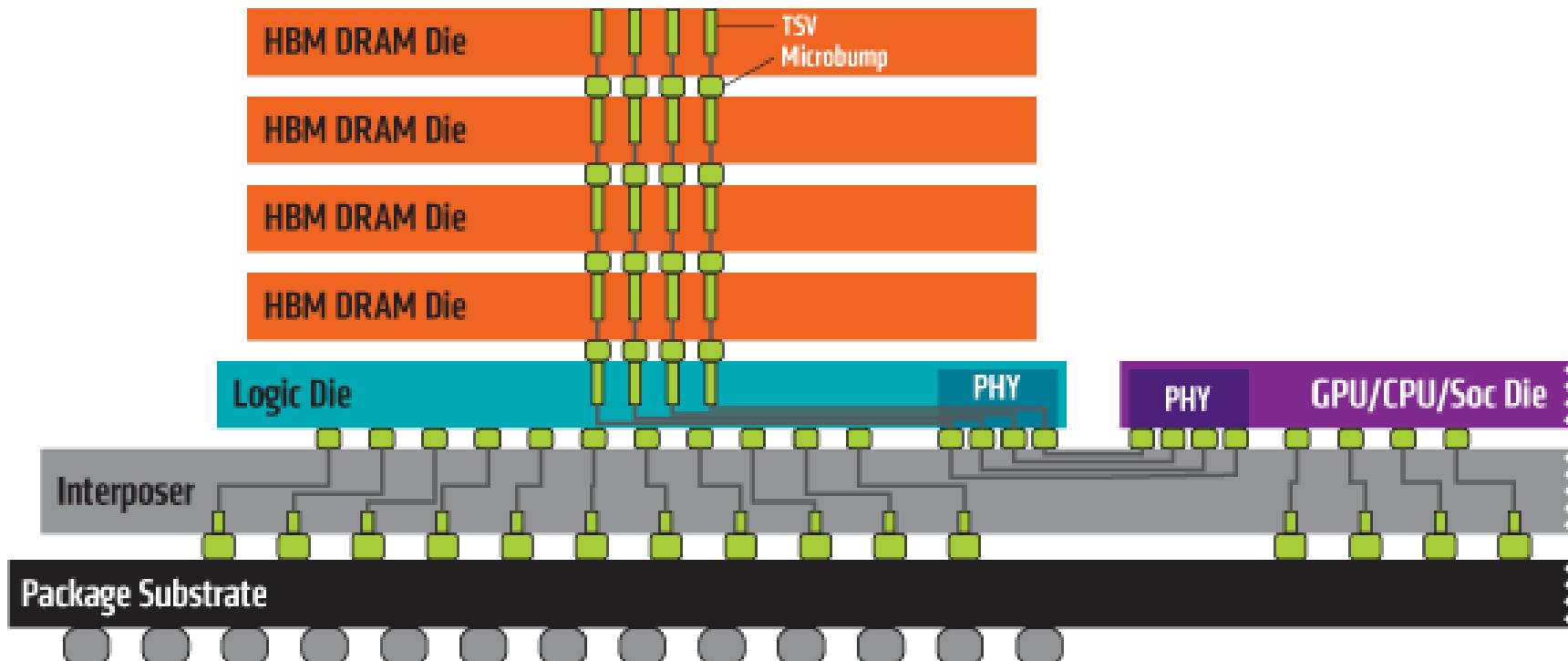
Victim cache is a small associative back up cache, added to a direct mapped cache, which holds recently evicted lines

- First look up in direct mapped cache
 - If miss, look in victim cache
 - If hit in victim cache, swap hit line with line now evicted from L1
 - If miss in victim cache, L1 victim -> VC, VC victim->?
- Fast hit time of direct mapped but with reduced conflict misses

Typical memory hierarchies

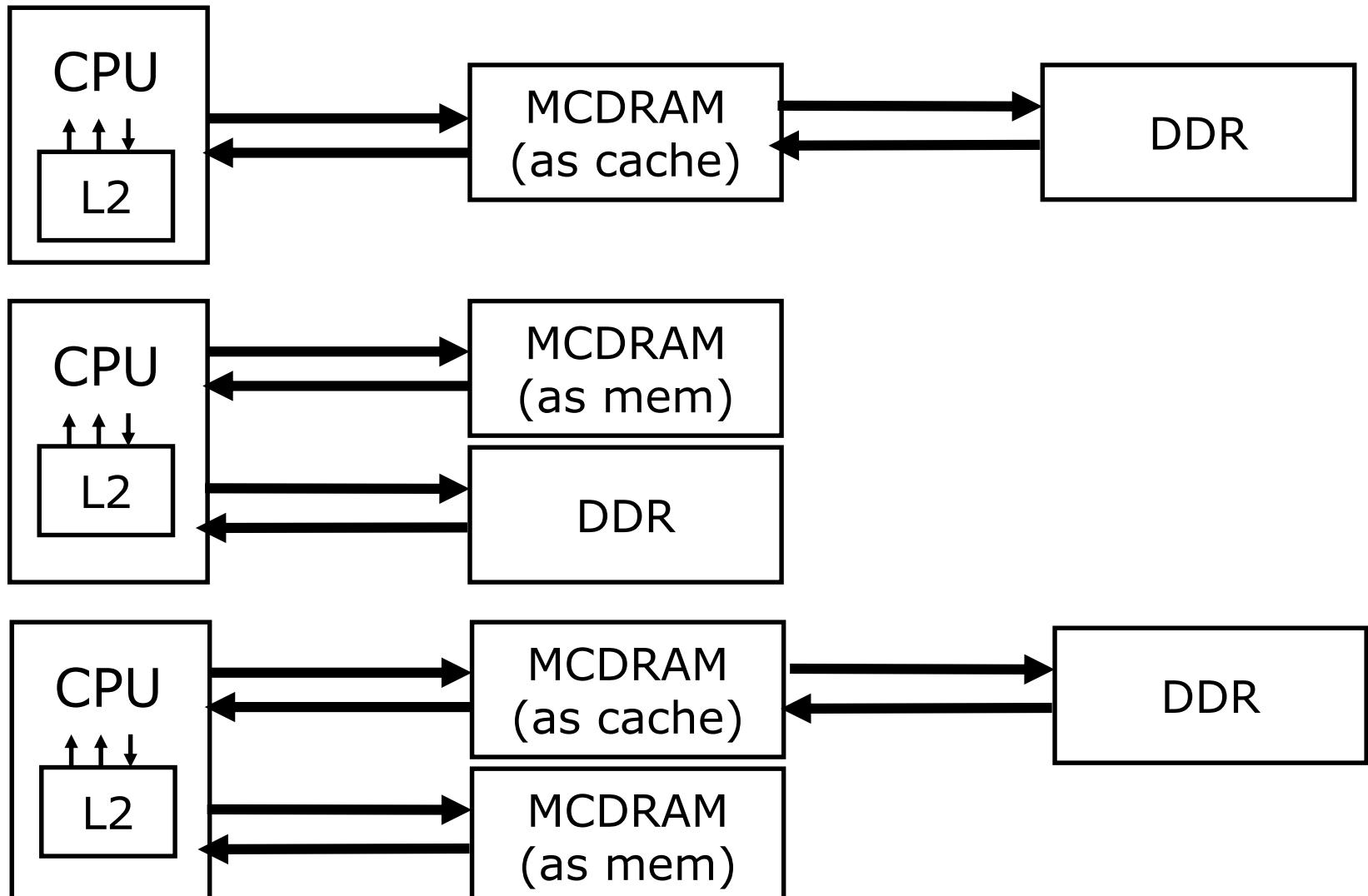


HBM DRAM or MCDRAM



Source: AMD

Mixed technology caching (Intel Knights Landing)



Thank you!

*Next lecture:
Virtual memory*

Memory Management: *From Absolute Addresses to Demand Paging*

Daniel Sanchez

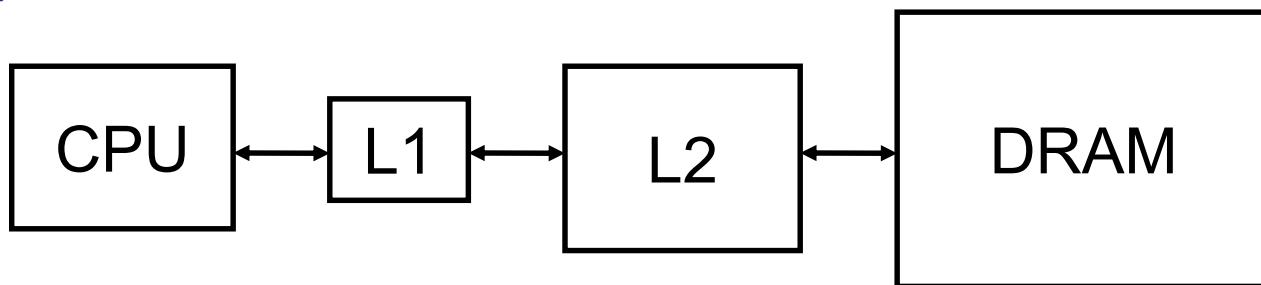
Computer Science and Artificial Intelligence Laboratory
M.I.T.

Recap: Cache Organization

- Caches are small and fast memories that transparently retain recently accessed data
- Cache organizations
 - Direct-mapped
 - Set-associative
 - Fully associative
- Cache performance
 - $AMAT = HitLatency + MissRate * MissLatency$
 - Minimizing AMAT requires balancing competing tradeoffs

Multilevel Caches

- A memory cannot be large and fast
- Add level of cache to reduce miss penalty
 - Each level can have longer latency than level above
 - So, increase sizes of cache at each level



Metrics:

Local miss rate = misses in cache / accesses to cache

Global miss rate = misses in cache / CPU memory accesses

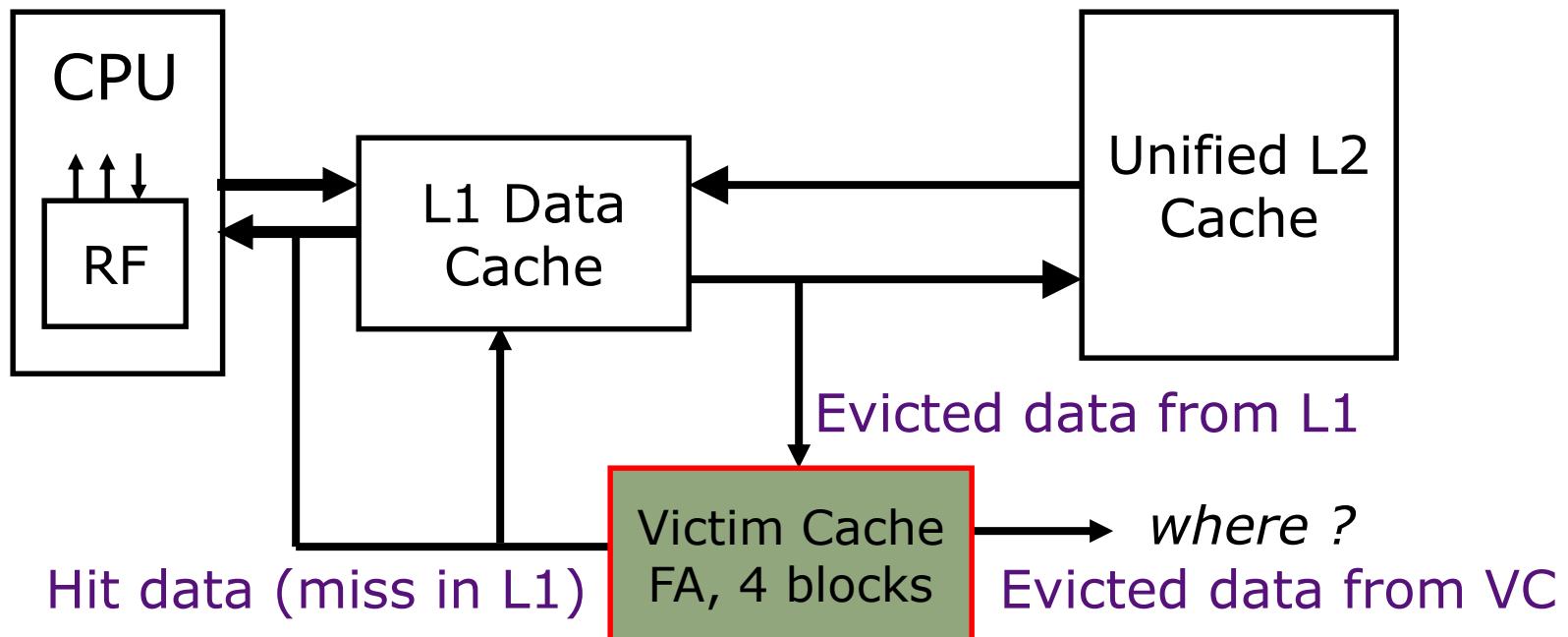
Misses per instruction = misses in cache / number of instructions

Inclusion Policy

- Inclusive multilevel cache:
 - Inner cache holds copies of data in outer cache
 - On miss, line inserted in inner and outer cache; replacement in outer cache invalidates line in inner cache
 - External accesses need only check outer cache
 - Commonly used (e.g., Intel CPUs up to Broadwell)
- Non-inclusive multilevel caches:
 - Inner cache may hold data not in outer cache
 - Replacement in outer cache doesn't invalidate line in inner cache
 - Used in Intel Skylake, ARM
- Exclusive multilevel caches:
 - Inner cache and outer cache hold different data
 - Swap lines between inner/outer caches on miss
 - Used in AMD processors

Why choose one type or the other?

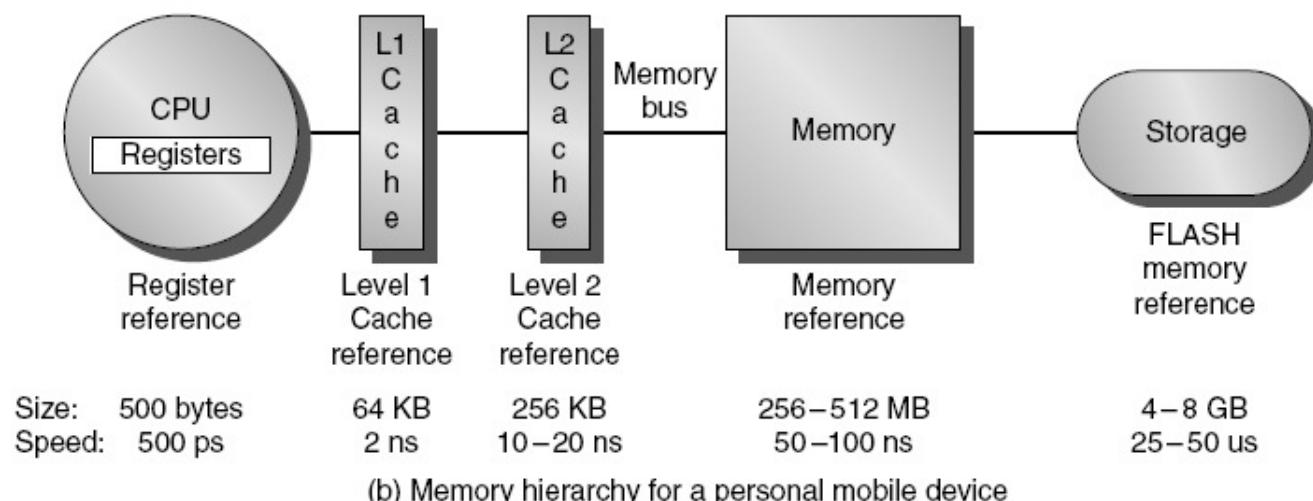
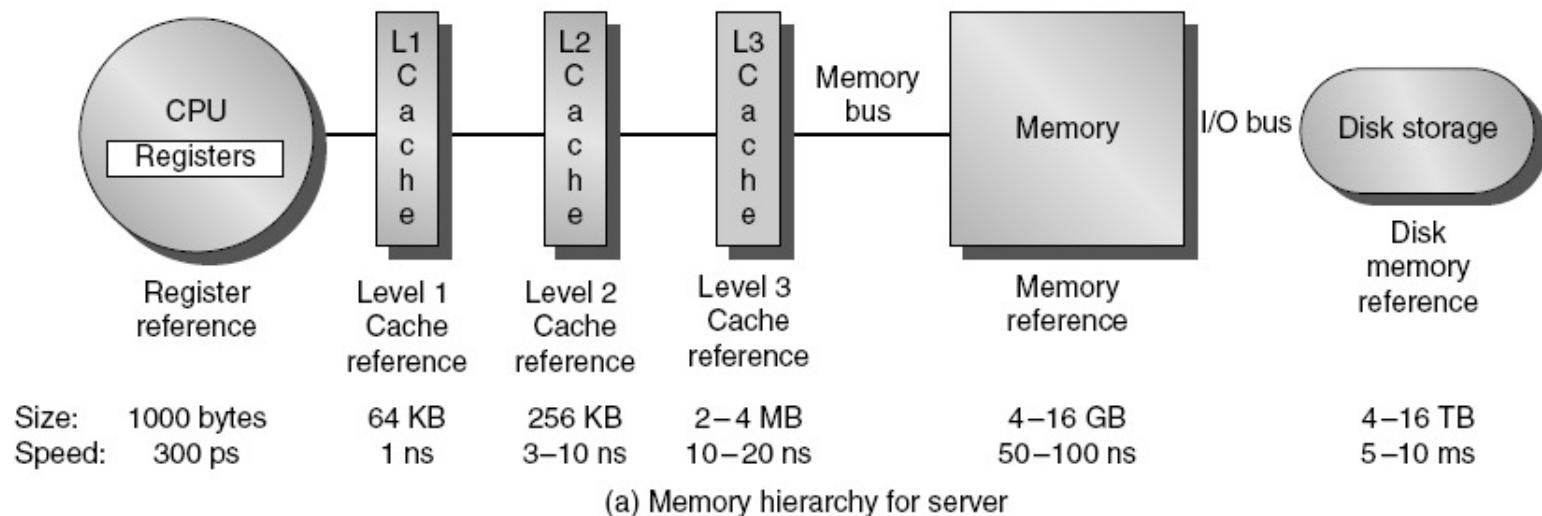
Victim Caches (HP 7200)



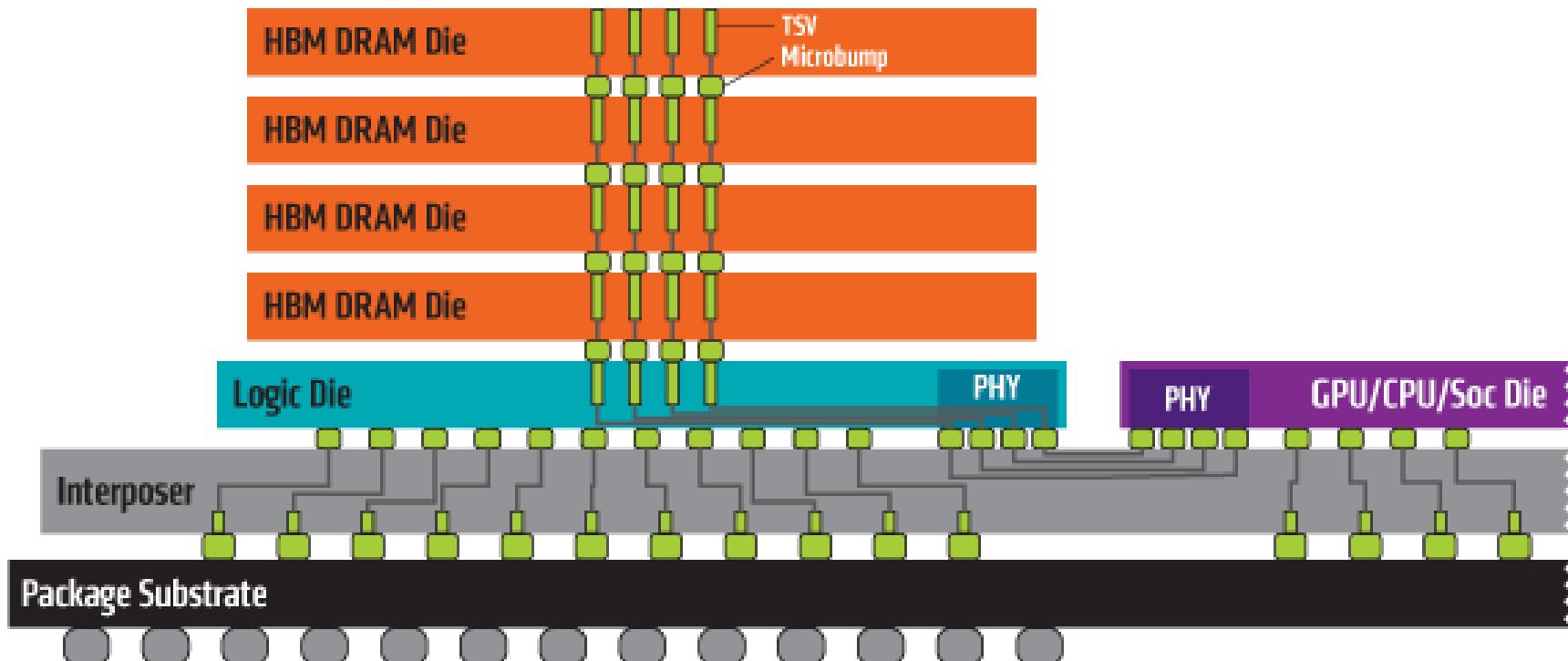
Victim cache is a small associative back up cache, added to a direct mapped cache, which holds recently evicted lines

- First look up in direct-mapped cache
 - If miss, look in victim cache
 - If hit in victim cache, swap hit line with line now evicted from L1
 - If miss in victim cache, L1 victim -> VC, VC victim->?
- Fast hit time of direct-mapped but with reduced conflict misses

Typical memory hierarchies



HBM DRAM or MCDRAM

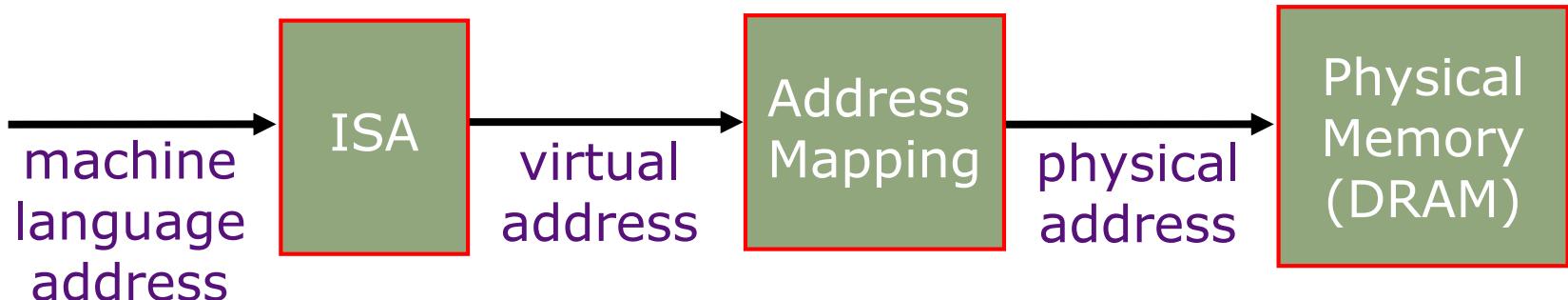


Source: AMD

Memory Management

- The Fifties
 - Absolute Addresses
 - Dynamic address translation
- The Sixties
 - Atlas and Demand Paging
 - Paged memory systems and TLBs
- Modern Virtual Memory Systems

Names for Memory Locations



- Machine language address
 - as specified in machine code
- Virtual address
 - ISA specifies translation of machine code address into virtual address of program variable (sometimes called *effective address*)
- Physical address
 - ⇒ operating system specifies mapping of virtual address into name for a physical memory location

Absolute Addresses

EDSAC, early 50's

virtual address = physical memory address

- Only one program ran at a time, with unrestricted access to entire machine (RAM + I/O devices)
- Addresses in a program depended upon where the program was to be loaded in memory
- *But it was more convenient for programmers to write location-independent subroutines*

How could location independence be achieved?

Linker and/or loader modify addresses of subroutines and callers when building a program memory image

Multiprogramming

Motivation

In the early machines, I/O operations were slow and each word transferred involved the CPU

Higher throughput if CPU and I/O of 2 or more programs were overlapped. *How?*

⇒ *multiprogramming*

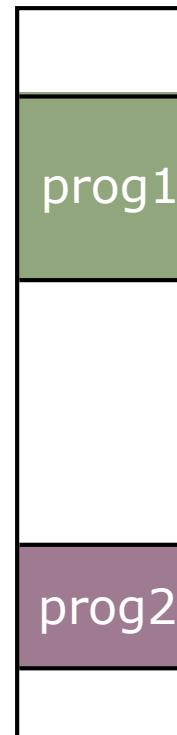
Location-independent programs

Programming and storage management ease
⇒ need for a *base register*

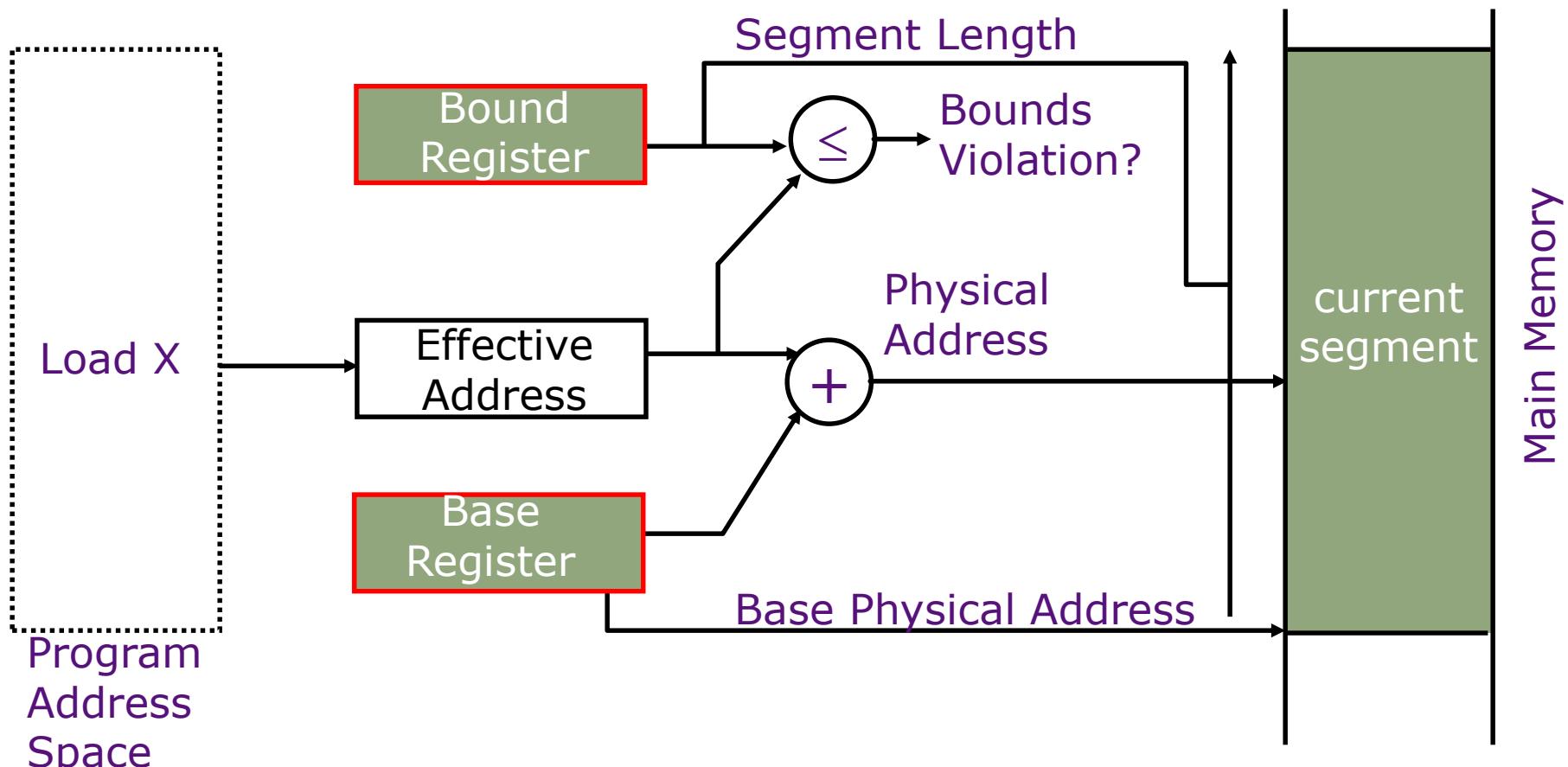
Protection

Independent programs should not affect each other inadvertently

⇒ need for a *bound register*

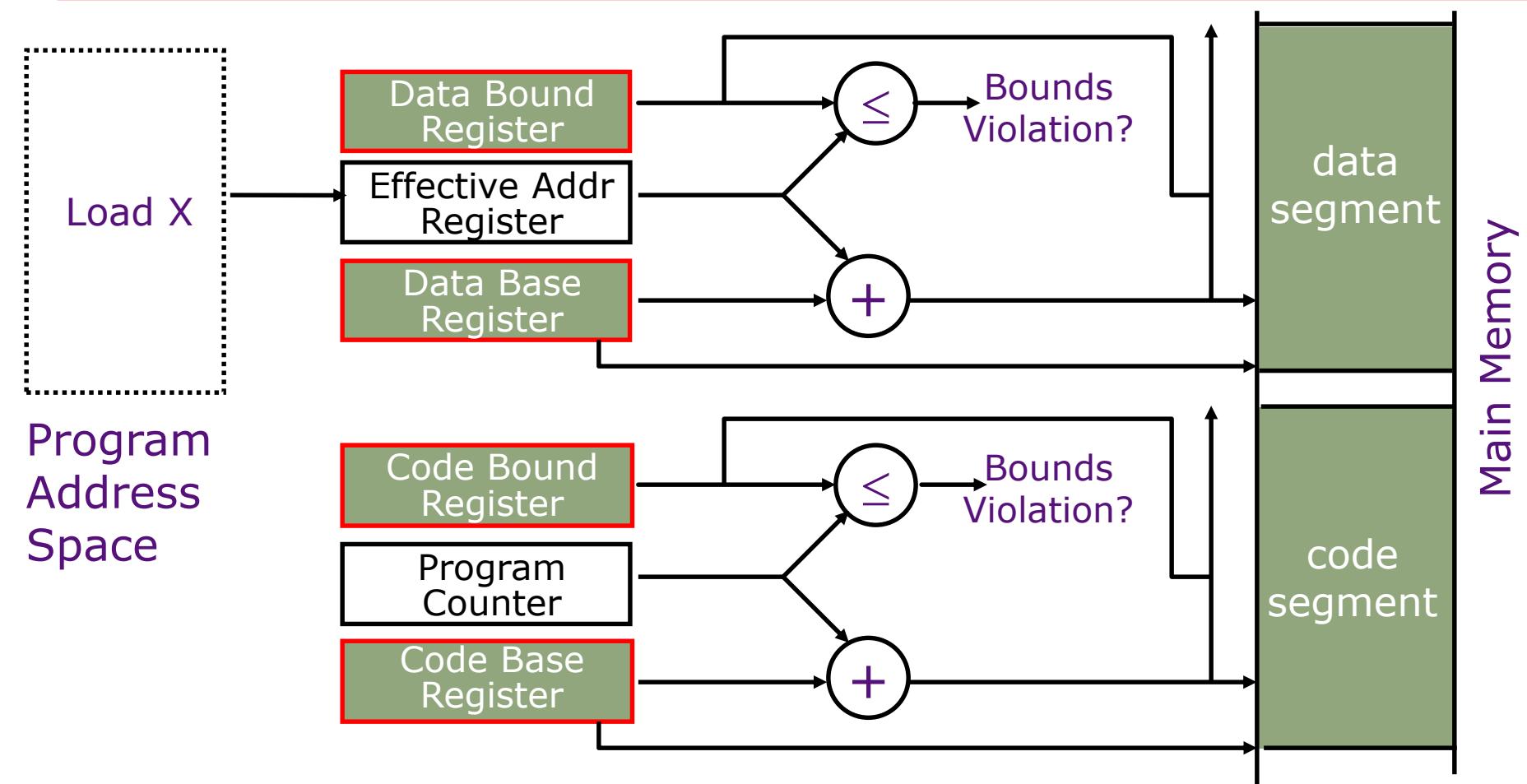


Simple Base and Bound Translation



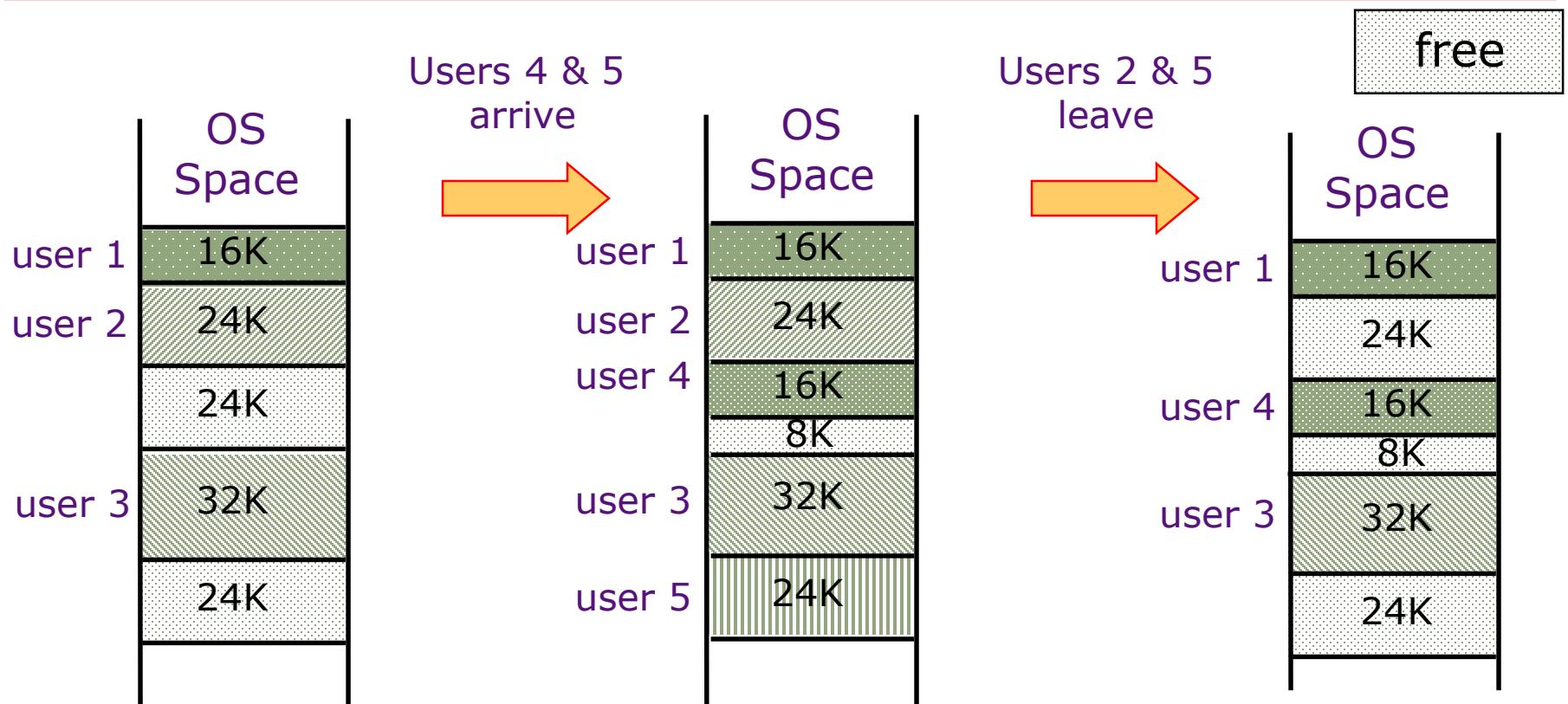
Base and bounds registers are visible/accessible only when processor is running in *supervisor mode*

Separate Areas for Code and Data



What is an advantage of this separation?
(Scheme used on all Cray vector supercomputers prior to X1, 2002)

Memory Fragmentation



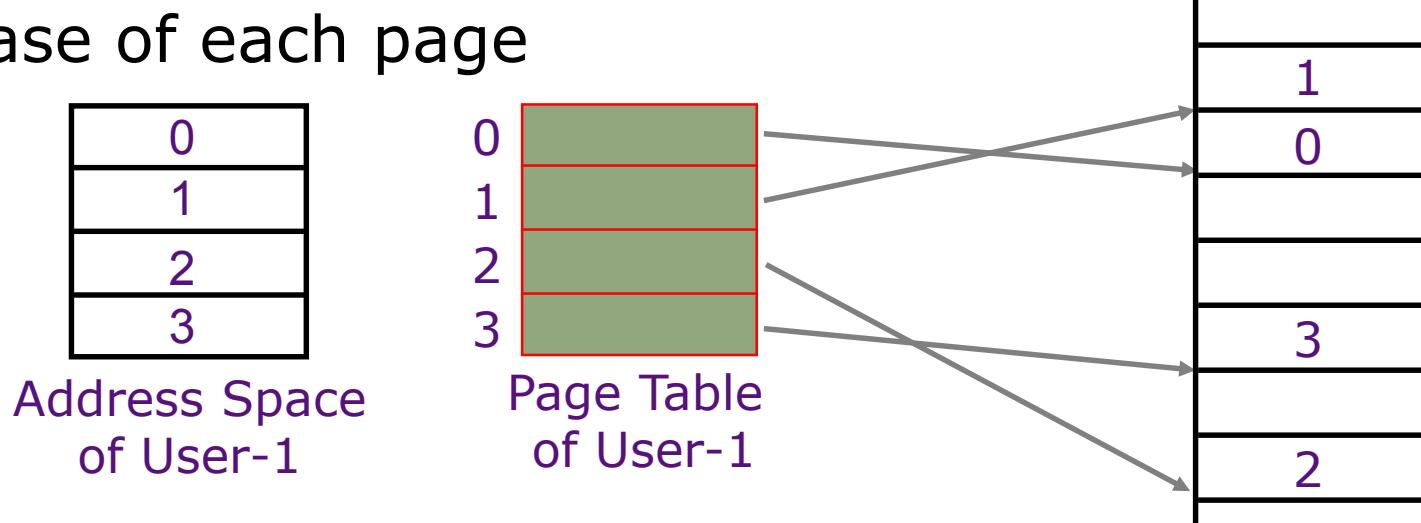
As users come and go, the storage is “fragmented”. Therefore, at some stage programs have to be moved around to compact the storage.

Paged Memory Systems

- Processor-generated address can be interpreted as a pair <page number, offset>

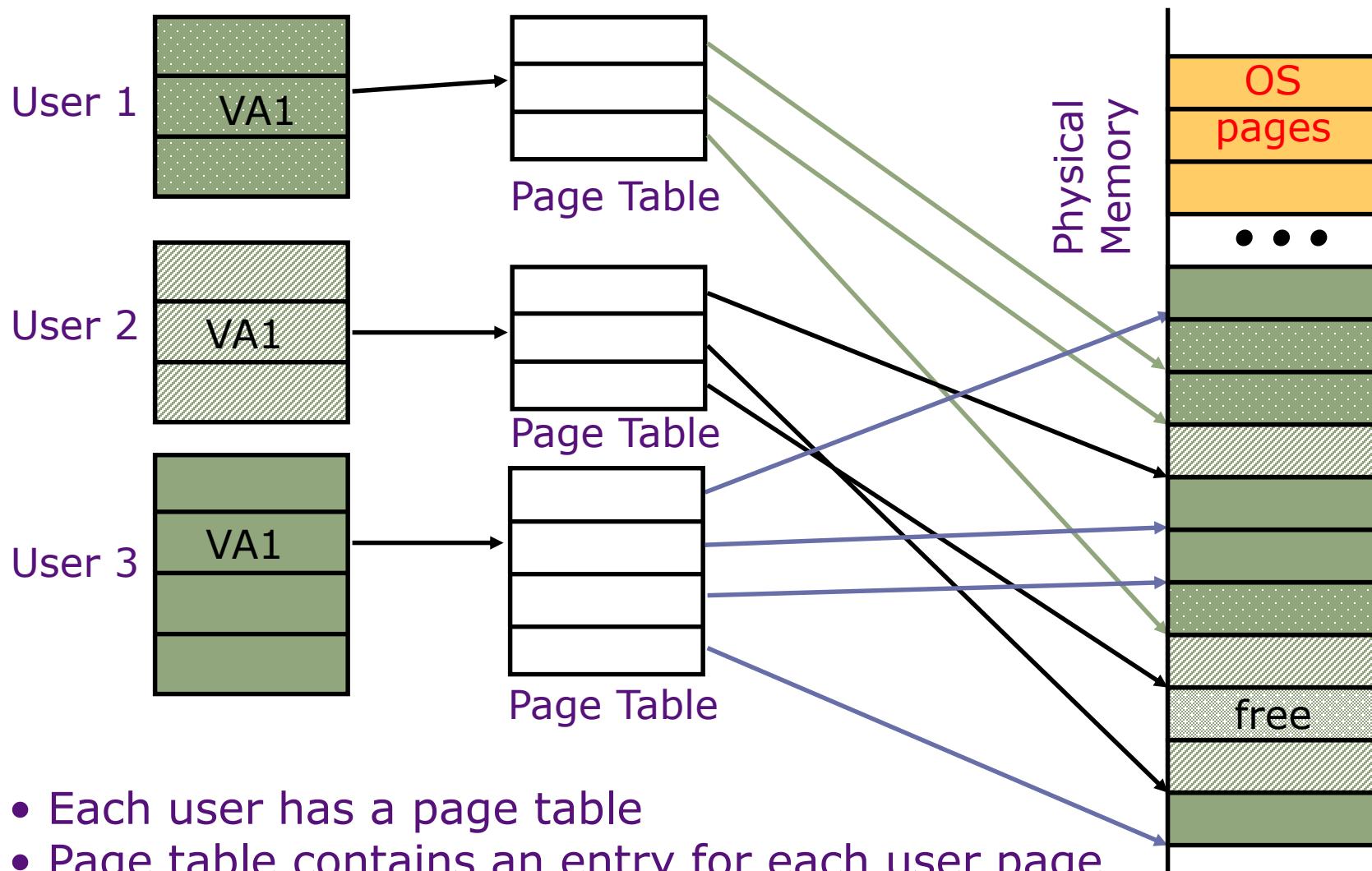


- A page table contains the physical address of the base of each page



Page tables make it possible to store the pages of a program non-contiguously.

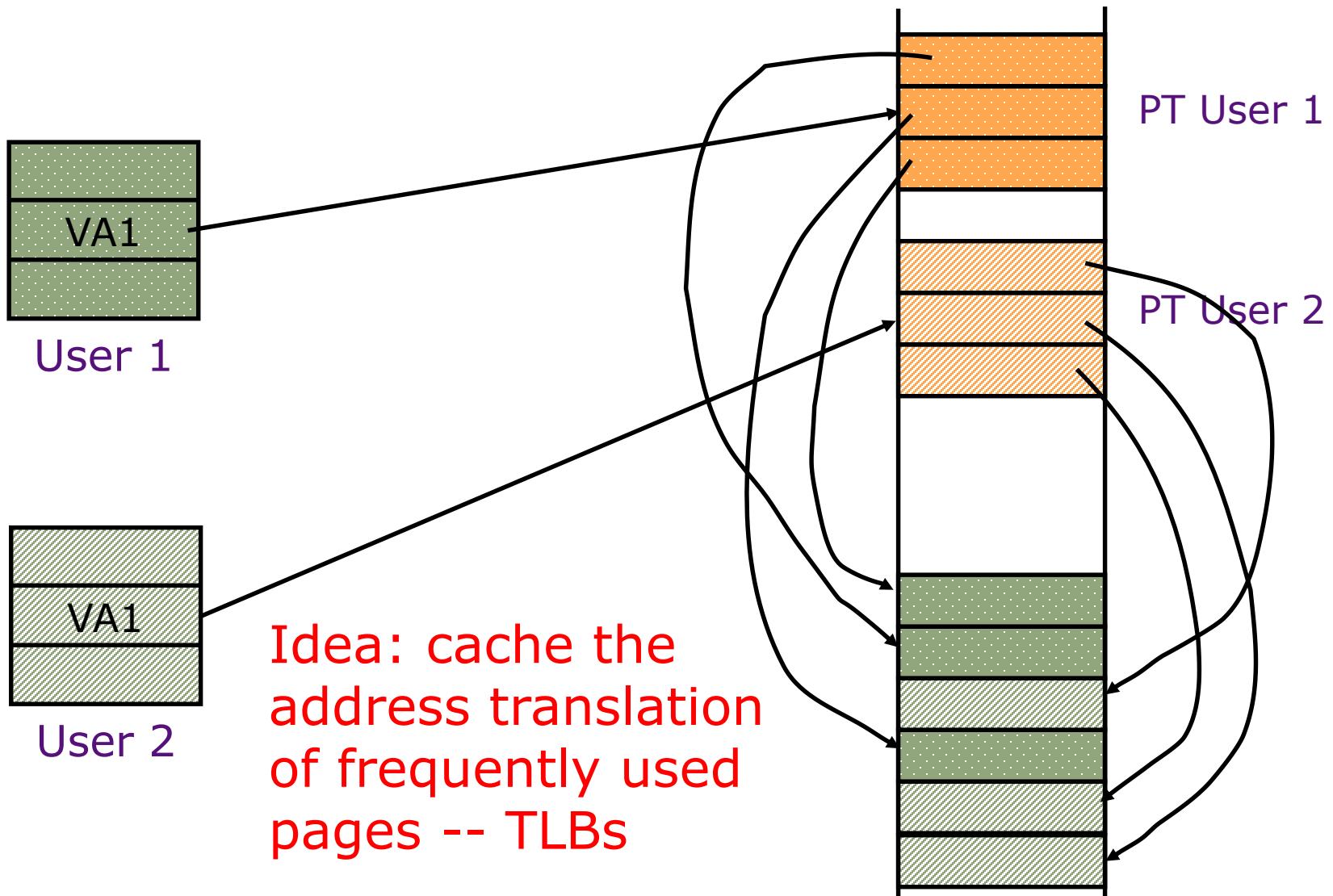
Private Address Space per User



Where Should Page Tables Reside?

- Space required by the page tables (PT) is proportional to the address space, number of users, ...
 - ⇒ Space requirement is large
 - ⇒ Too expensive to keep in registers
- Idea: Keep PT of the current user in special registers
 - may not be feasible for large page tables
 - Increases the cost of context swap
- Idea: Keep PTs in the main memory
 - needs one reference to retrieve the page base address and another to access the data word
 - ⇒ *doubles the number of memory references!*

Page Tables in Physical Memory



A Problem in Early Sixties

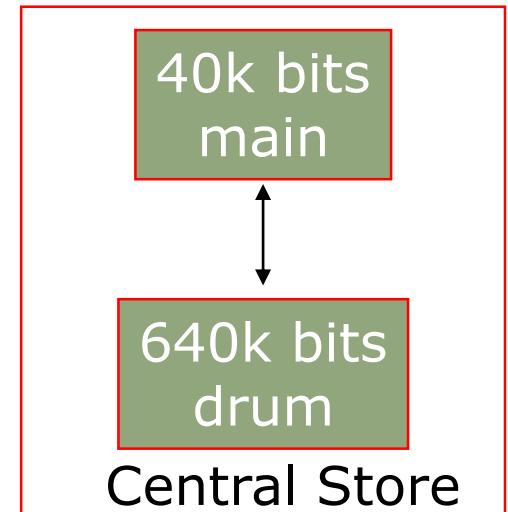
- There were many applications whose data could not fit in the main memory, e.g., payroll
 - *Paged memory system reduced fragmentation but still required the whole program to be resident in the main memory*
- Programmers moved the data back and forth from the secondary store by *overlaying* it repeatedly on the primary store

tricky programming!

Manual Overlays

- Assume an instruction can address all the storage on the drum
- *Method 1:* programmer keeps track of addresses in the main memory and initiates an I/O transfer when required
- *Method 2:* automatic initiation of I/O transfers by software address translation

Brooker's interpretive coding, 1960



Ferranti Mercury
1956

Problems?

Method1: Difficult, error prone
Method2: Inefficient

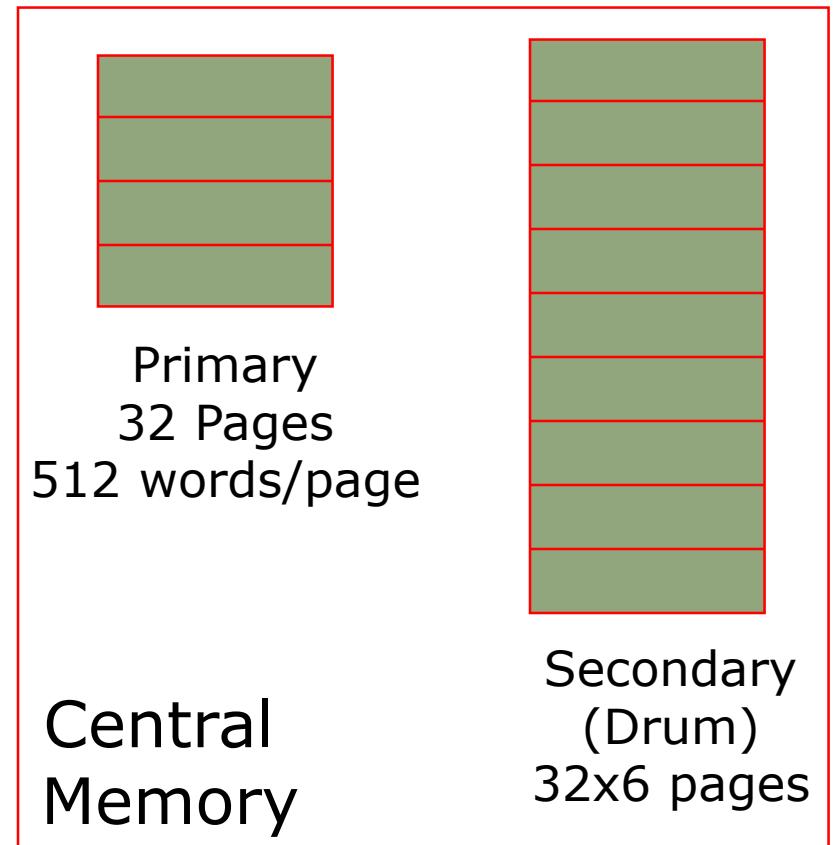
Demand Paging in Atlas (1962)

"A page from secondary storage is brought into the primary storage whenever it is (implicitly) demanded by the processor."

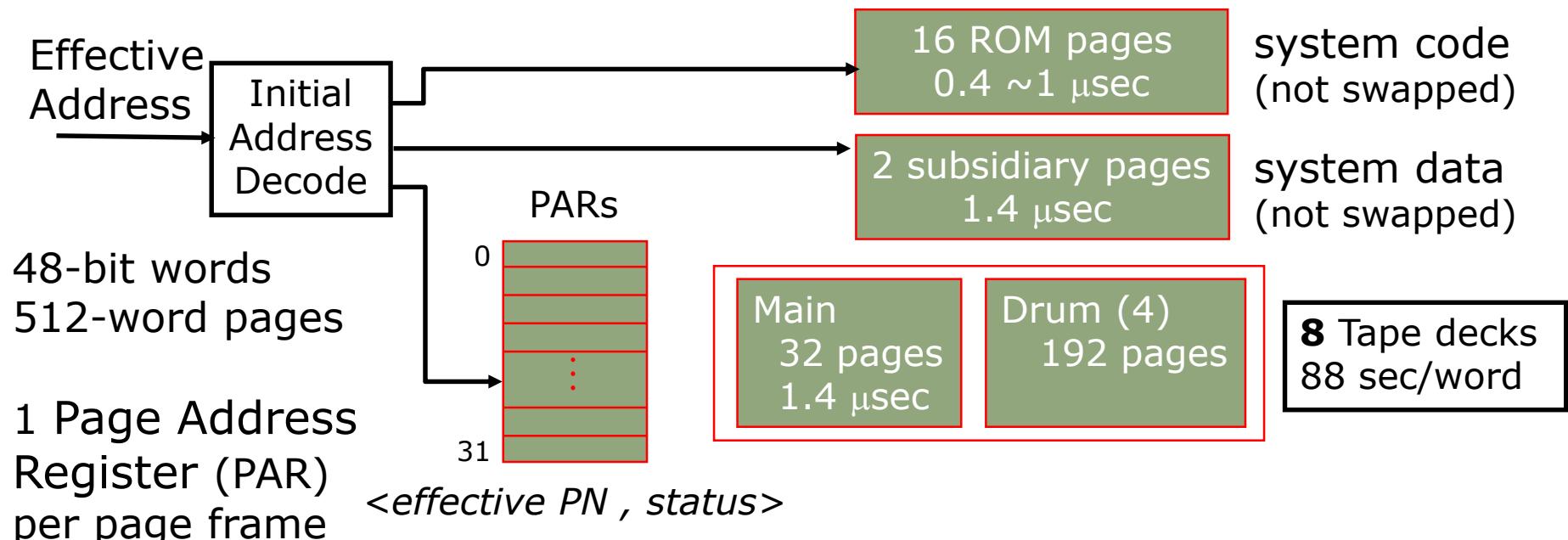
Tom Kilburn

Primary memory as a *cache* for secondary memory

User sees $32 \times 6 \times 512$ words of storage



Hardware Organization of Atlas



48-bit words
512-word pages

1 Page Address Register (PAR)
per page frame

Compare the effective page address against all 32 PARs

match

⇒ normal access

no match

⇒ *page fault*

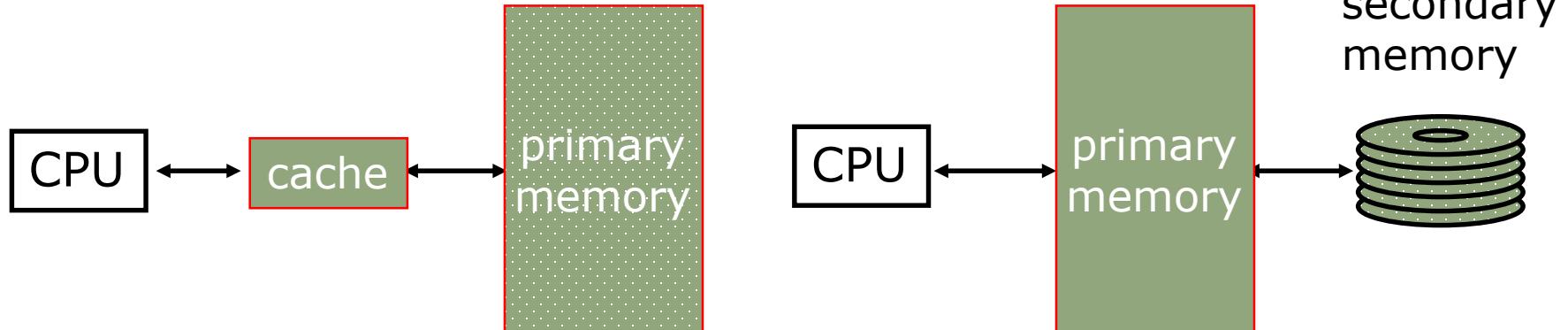
save the state of the partially executed instruction

8 Tape decks
88 sec/word

Atlas Demand Paging Scheme

- On a page fault:
 - Input transfer into a free page is initiated
 - The Page Address Register (PAR) is updated
 - If no free page is left, a *page is selected to be replaced* (based on usage)
 - The replaced page is written on the drum
 - to minimize the drum latency effect, the first empty page on the drum was selected
 - The *page table is updated* to point to the new location of the page on the drum

Caching vs. Demand Paging



Caching

- cache entry
- cache block (~32 bytes)
- cache miss rate (1% to 20%)
- cache hit (~1 cycle)
- cache miss (~100 cycles)
- a miss is handled
in hardware

Demand paging

- page frame
- page (~4K bytes)
- page miss rate (<0.001%)
- page hit (~100 cycles)
- page miss (~5M cycles)
- a miss is handled
mostly in software

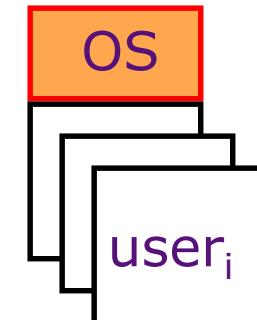
Modern Virtual Memory Systems

Illusion of a large, private, uniform store

Protection & Privacy

several users, each with their private address space and one or more shared address spaces

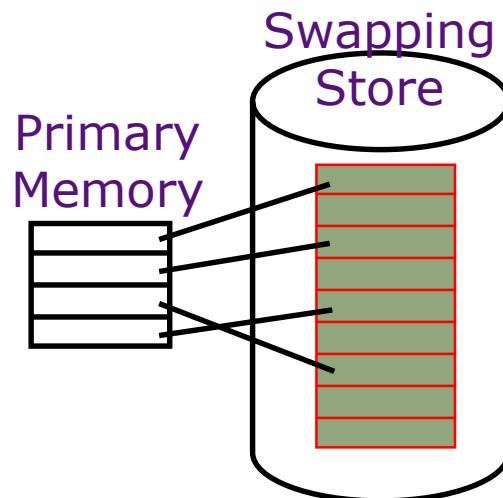
page table \equiv name space



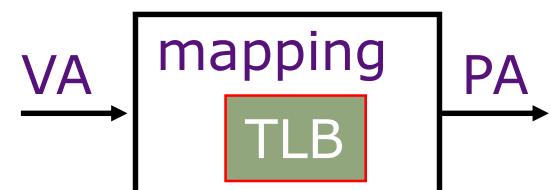
Demand Paging

Provides the ability to run programs larger than the primary memory

Hides differences in machine configurations

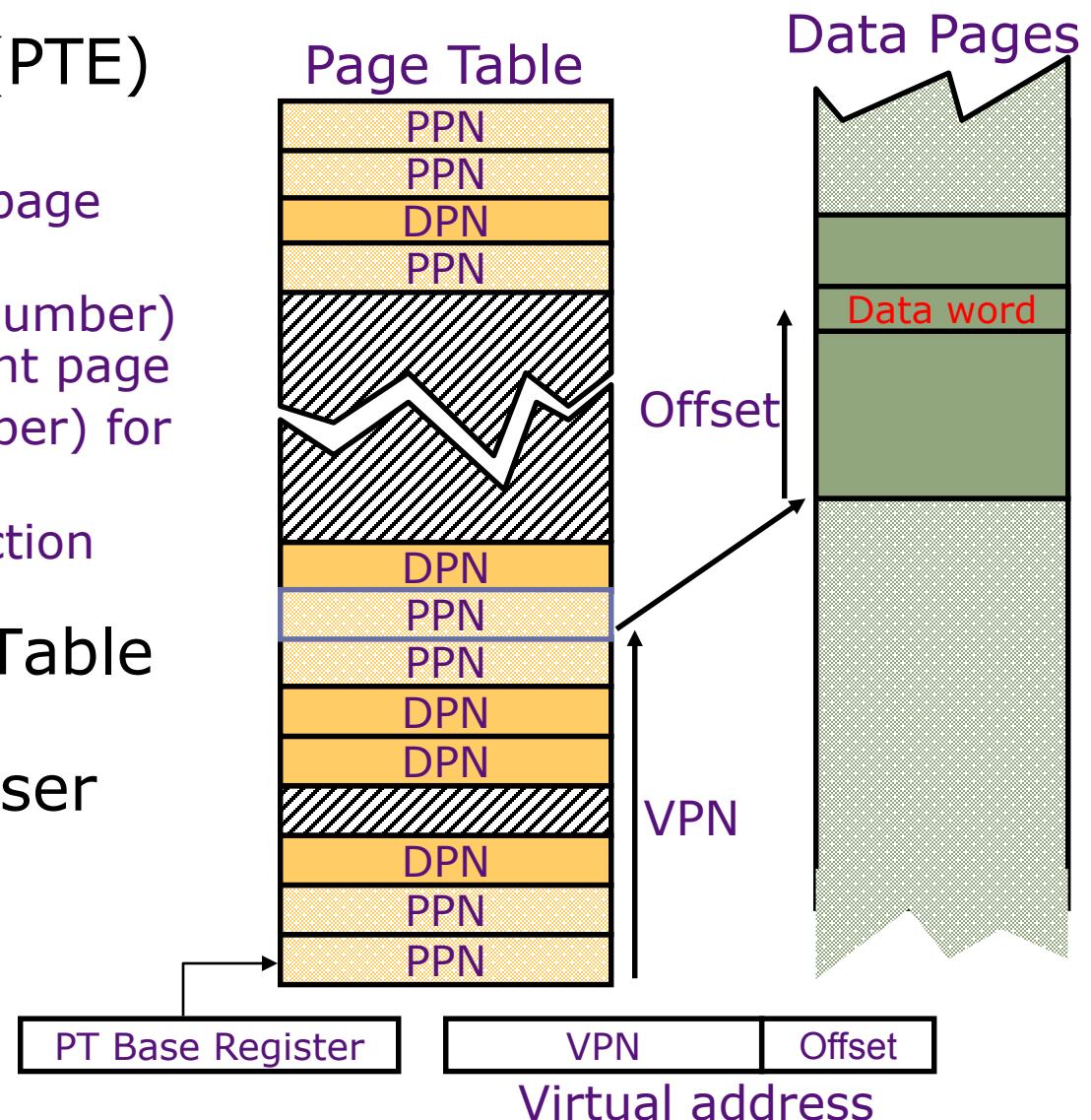


The price is address translation on each memory reference



Linear Page Table

- Page Table Entry (PTE) contains:
 - A bit to indicate if a page exists
 - PPN (physical page number) for a memory-resident page
 - DPN (disk page number) for a page on the disk
 - Status bits for protection and usage
- OS sets the Page Table Base Register whenever active user process changes



Size of Linear Page Table

With 32-bit addresses, 4 KB pages & 4-byte PTEs:

- ⇒ 2^{20} PTEs, i.e, 4 MB page table per user
- ⇒ 4 GB of swap space needed to back up the full virtual address space

Larger pages?

- Internal fragmentation (Not all memory in a page is used)
- Larger page fault penalty (more time to read from disk)

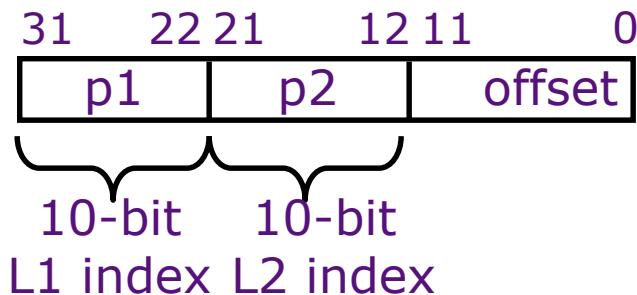
What about 64-bit virtual address space???

- Even 1MB pages would require 2^{44} 8-byte PTEs (35 TB!)

What is the “saving grace”?

Hierarchical Page Table

Virtual Address

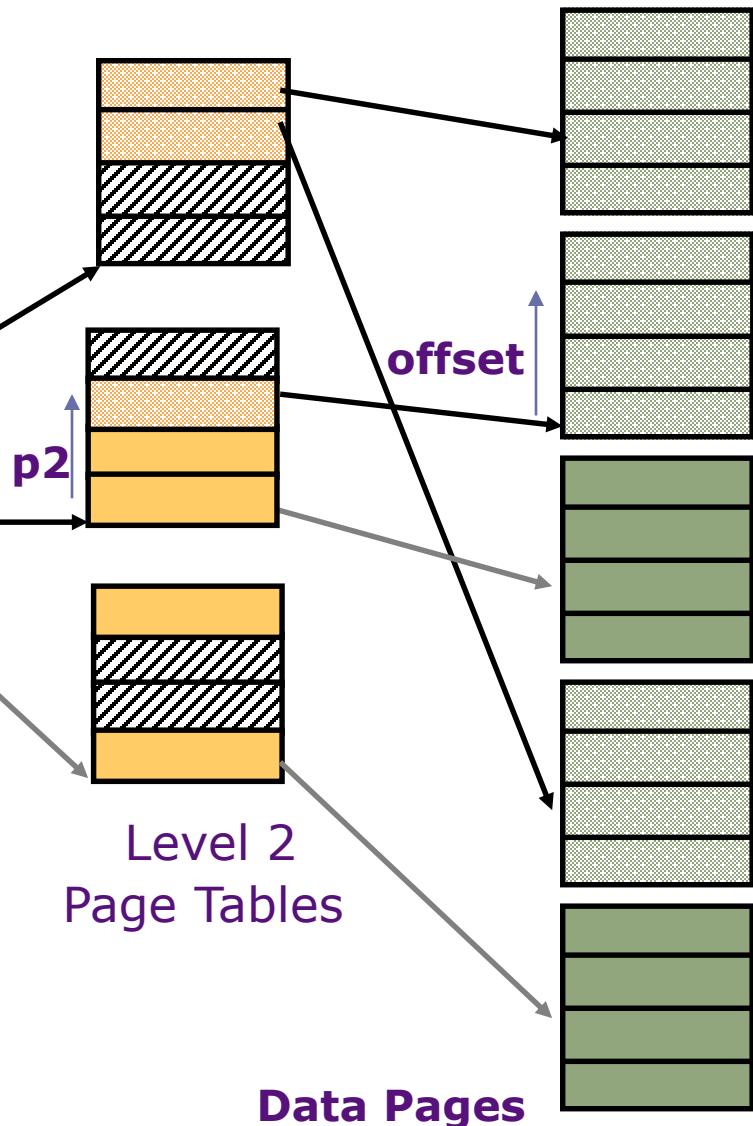


Root of the Current
Page Table



(Processor
Register)

Level 1
Page Table



page in primary memory

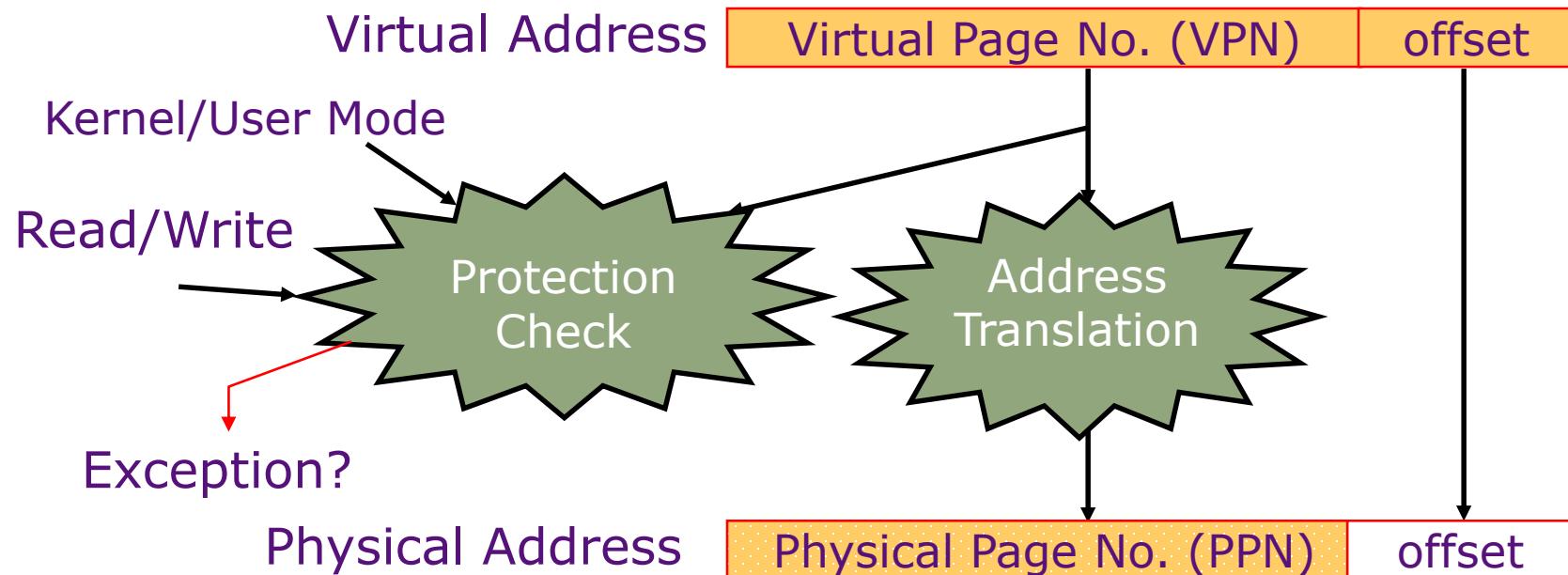


page in secondary memory



PTE of a nonexistent page

Address Translation & Protection



- Every instruction and data access needs address translation and protection checks

A good VM design needs to be fast (~ one cycle) and space-efficient

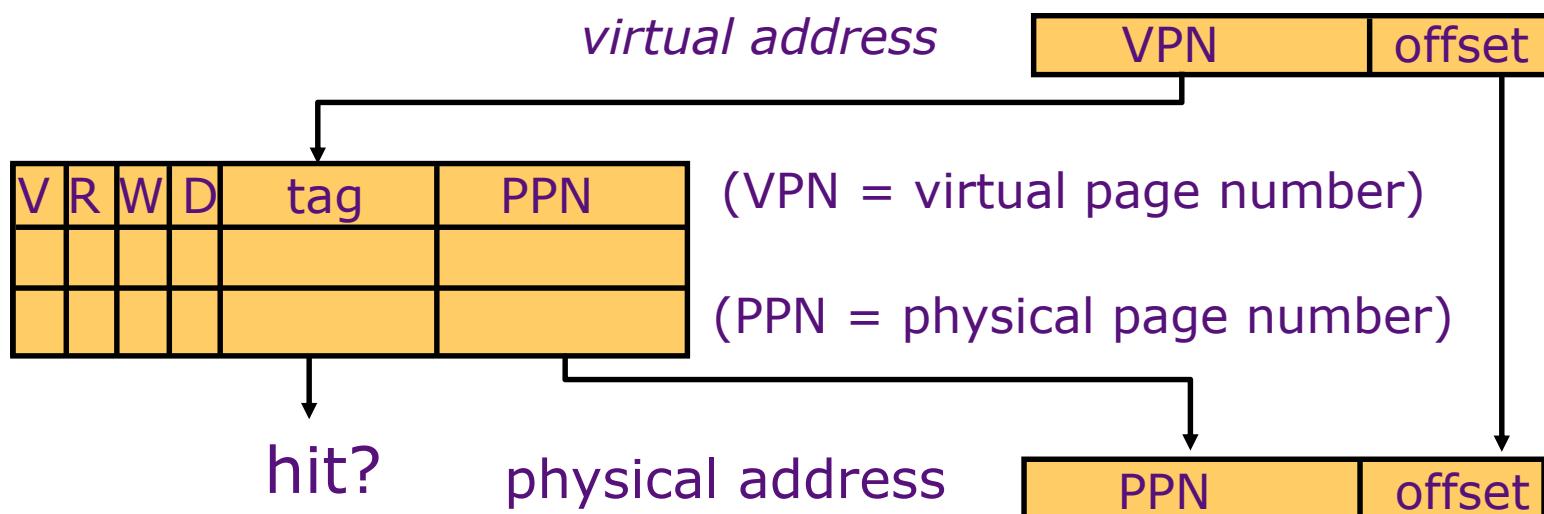
Translation Lookaside Buffers

Address translation is very expensive!

In a two-level page table, each reference becomes several memory accesses

Solution: *Cache translations in TLB*

TLB hit	⇒ Single-cycle Translation
TLB miss	⇒ Page Table Walk to refill



TLB Designs

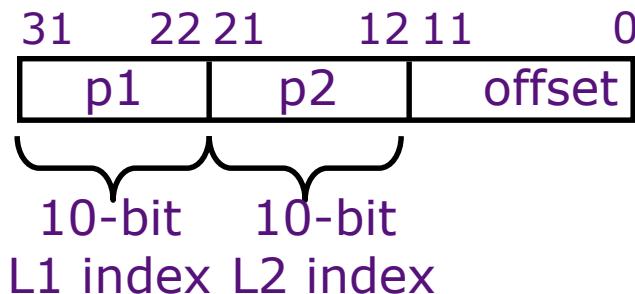
- Typically 32-128 entries, usually highly associative
 - Each entry maps a large page, hence less spatial locality across pages → more likely that two entries conflict
 - Sometimes larger TLBs (256-512 entries) are 4-8 way set-associative
- Random or FIFO replacement policy
- No process information in TLB?
- TLB Reach: Size of largest virtual address space that can be simultaneously mapped by TLB

Example: 64 TLB entries, 4KB pages, one page per entry

TLB Reach = _____ ?

Variable-Sized Page Support

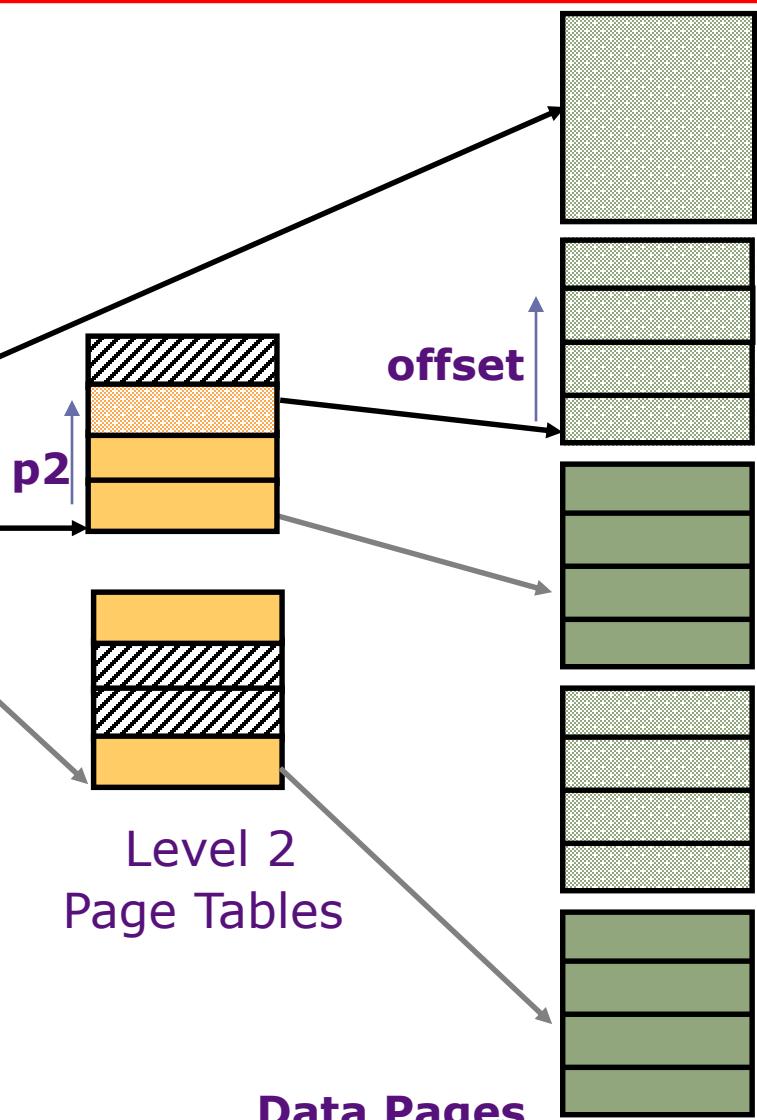
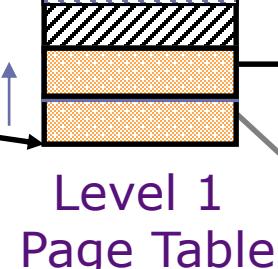
Virtual Address



Root of the Current
Page Table

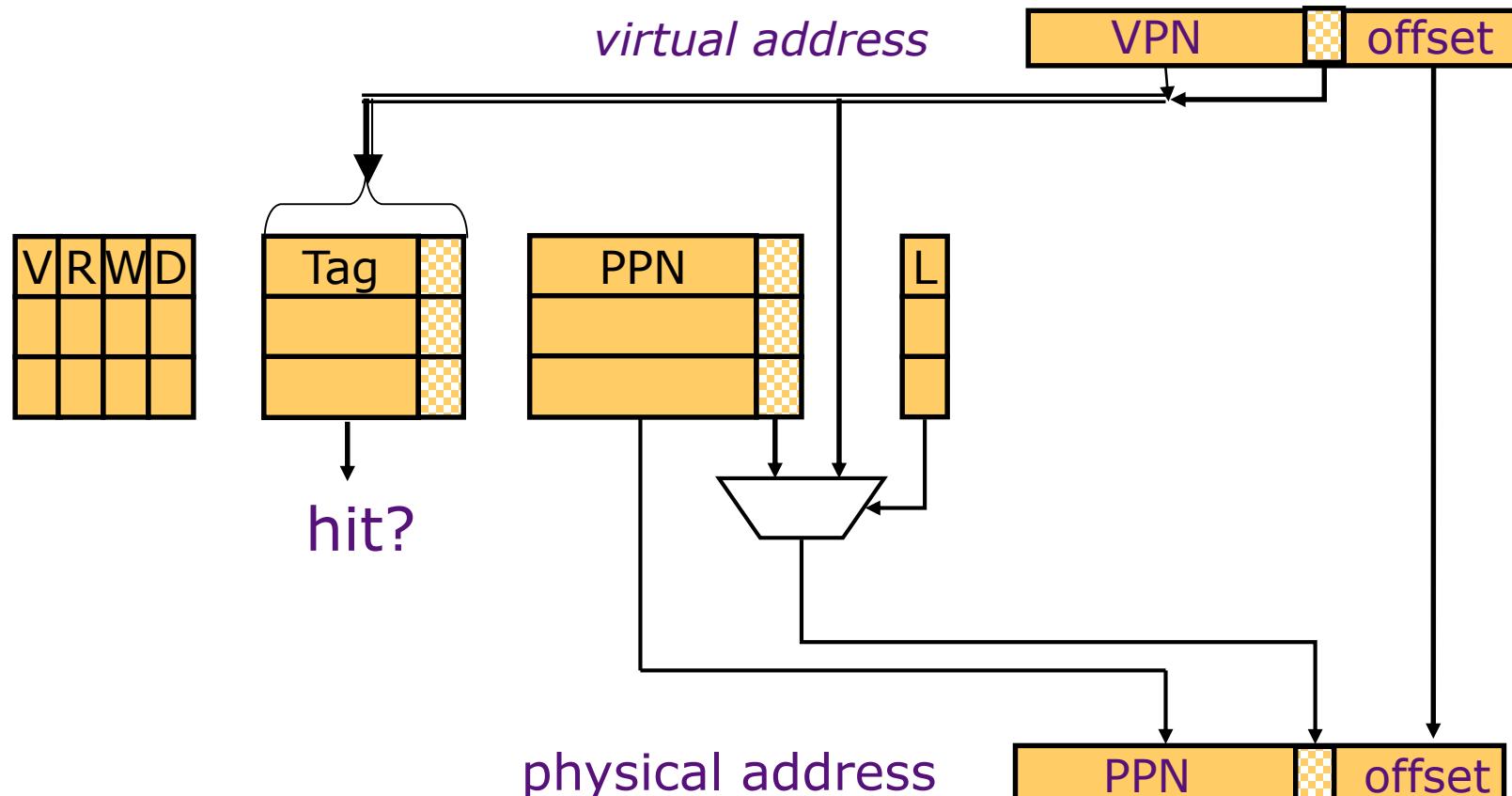


(Processor
Register)



- page in primary memory
- large page in primary memory
- page in secondary memory
- PTE of a nonexistent page

Variable-Size Page TLB



Alternatively, have a separate TLB
for each page size (pros/cons?)

Handling a TLB Miss

Software (MIPS, Alpha)

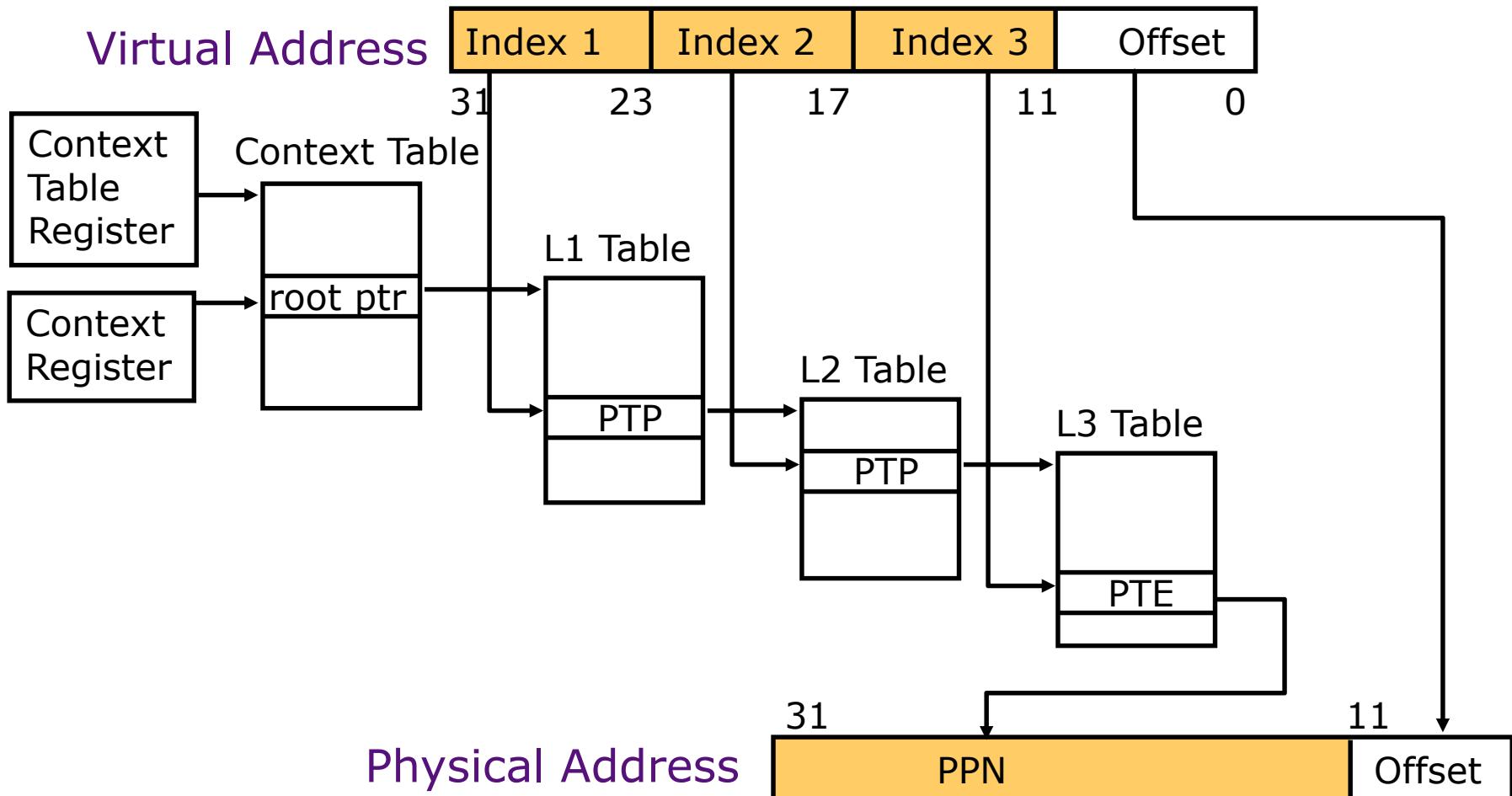
TLB miss causes an exception and the operating system walks the page tables and reloads TLB. A *privileged “untranslated” addressing mode used for walk*

Hardware (SPARC v8, x86, PowerPC)

A memory management unit (MMU) walks the page tables and reloads the TLB

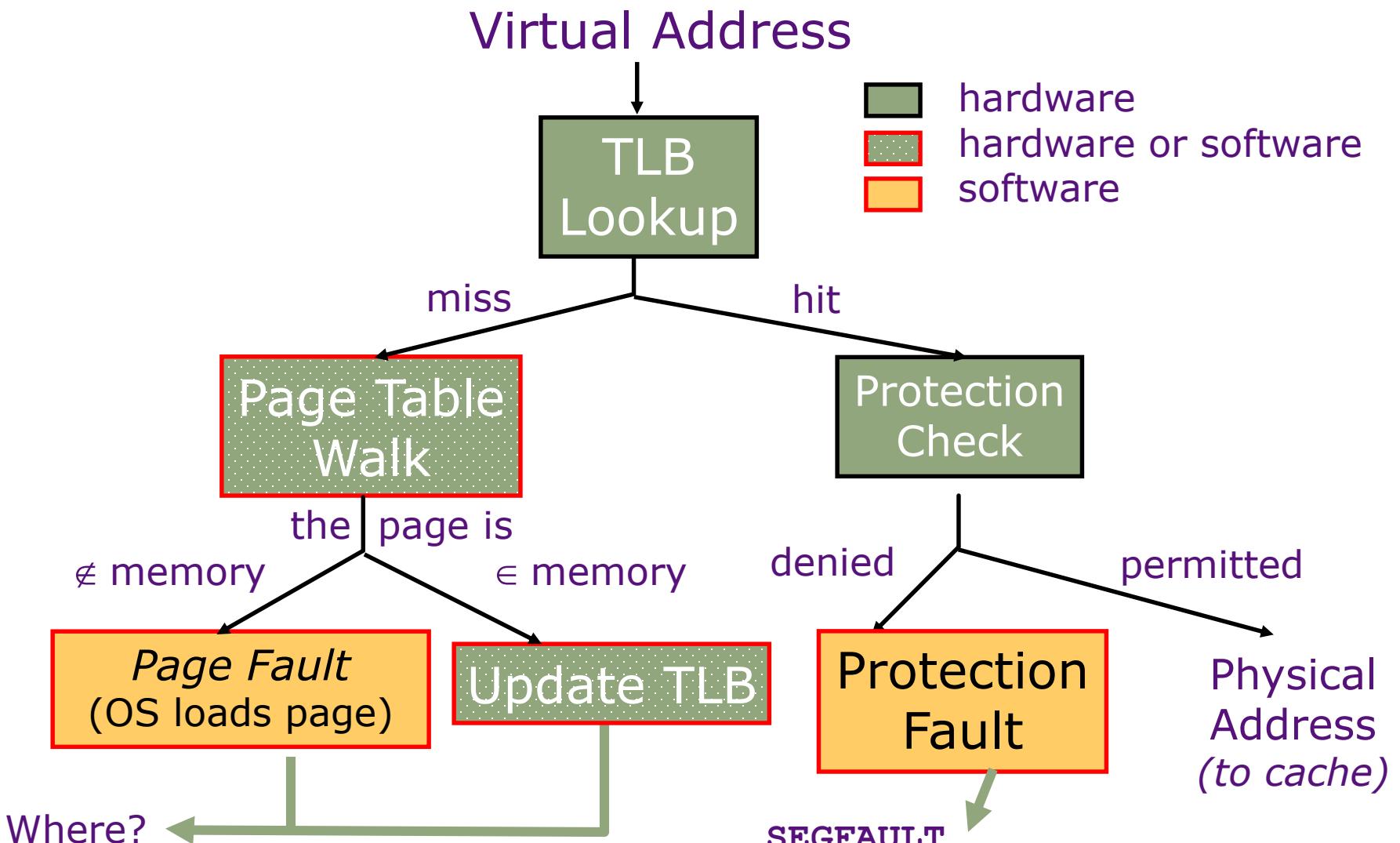
If a missing (data or PT) page is encountered during the TLB reloading, MMU gives up and signals a Page-Fault exception for the original instruction

Hierarchical Page Table Walk: SPARC v8



MMU does this table walk in hardware on a TLB miss

Address Translation: *putting it all together*



Next lecture:

Modern Virtual Memory Systems

Modern Virtual Memory Systems

Daniel Sanchez

Computer Science and Artificial Intelligence Laboratory
M.I.T.

Recap: Virtual Memory Systems

Illusion of a large, private, uniform store

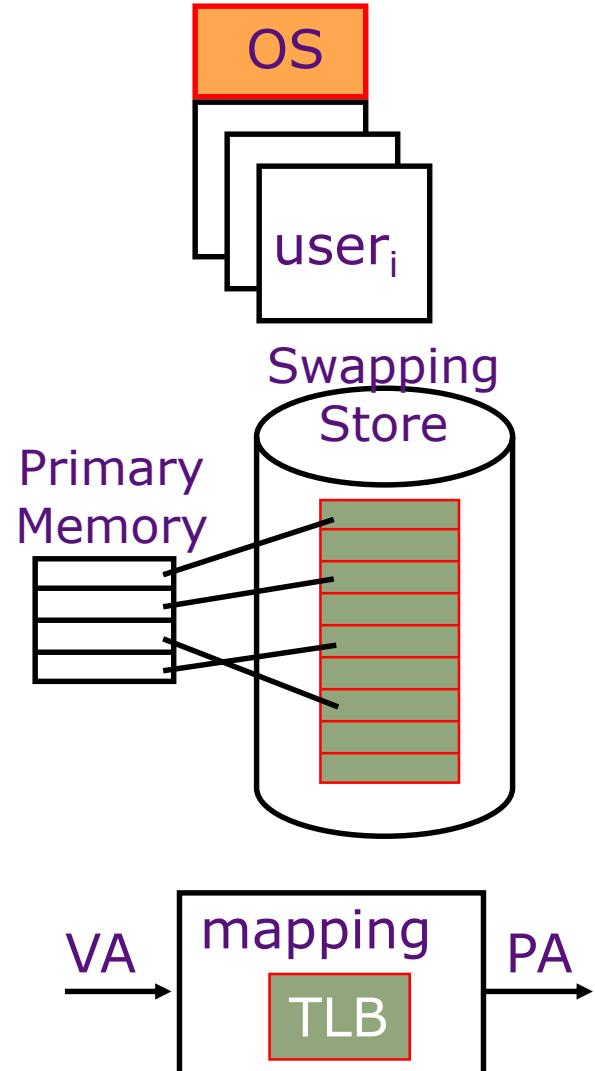
Protection & Privacy

- several users, each with their private address space and one or more shared address spaces
 - page table \equiv name space

Demand Paging

- Provides the ability to run programs larger than the primary memory
- Hides differences in machine configurations

The price is address translation on each memory reference



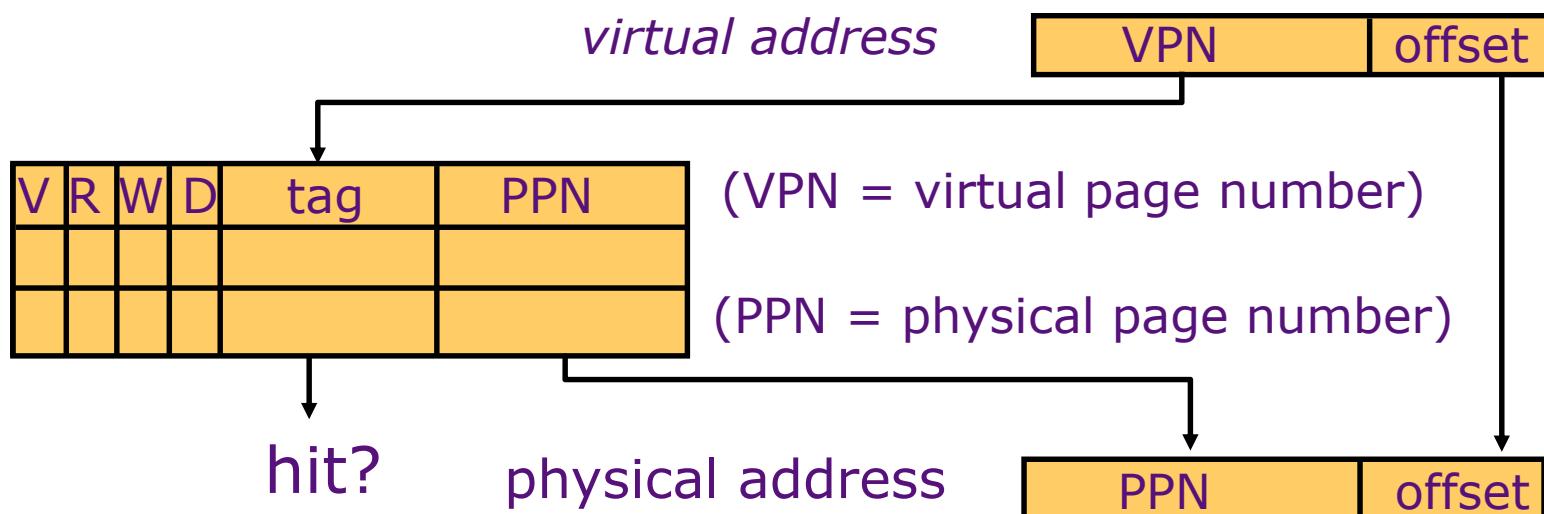
Reminder: Translation Lookaside Buffers

Address translation is very expensive!

In a hierarchical page table, each reference becomes several memory accesses

Solution: *Cache translations in TLB*

TLB hit	⇒ Single-cycle Translation
TLB miss	⇒ Page Table Walk to refill

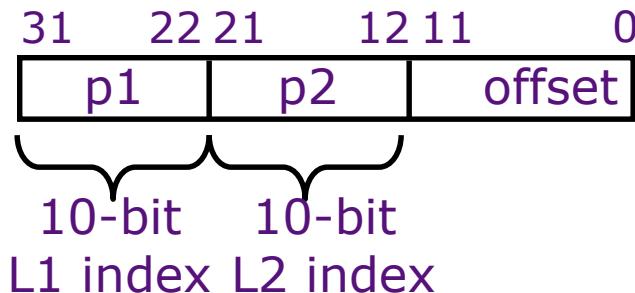


Reminder: TLB Designs

- Typically 32-128 entries, usually highly associative
- Keep process information in TLB?
 - No process id → Must flush on context switch
 - Tag each entry with process id → No flush, but costlier
- TLB Reach: Size of largest virtual address space that can be simultaneously mapped by TLB
 - Example: 64 TLB entries, 4KB pages, one page per entryTLB Reach = _____ ?
- Ways to increase TLB reach
 - Multi-level TLBs (e.g., Intel Skylake: 64-entry L1 data TLB, 128-entry L1 instruction TLB, 1.5K-entry L2 TLB)
 - Multiple page sizes (e.g., x86-64: 4KB, 2MB, 1GB)

Variable-Sized Page Support

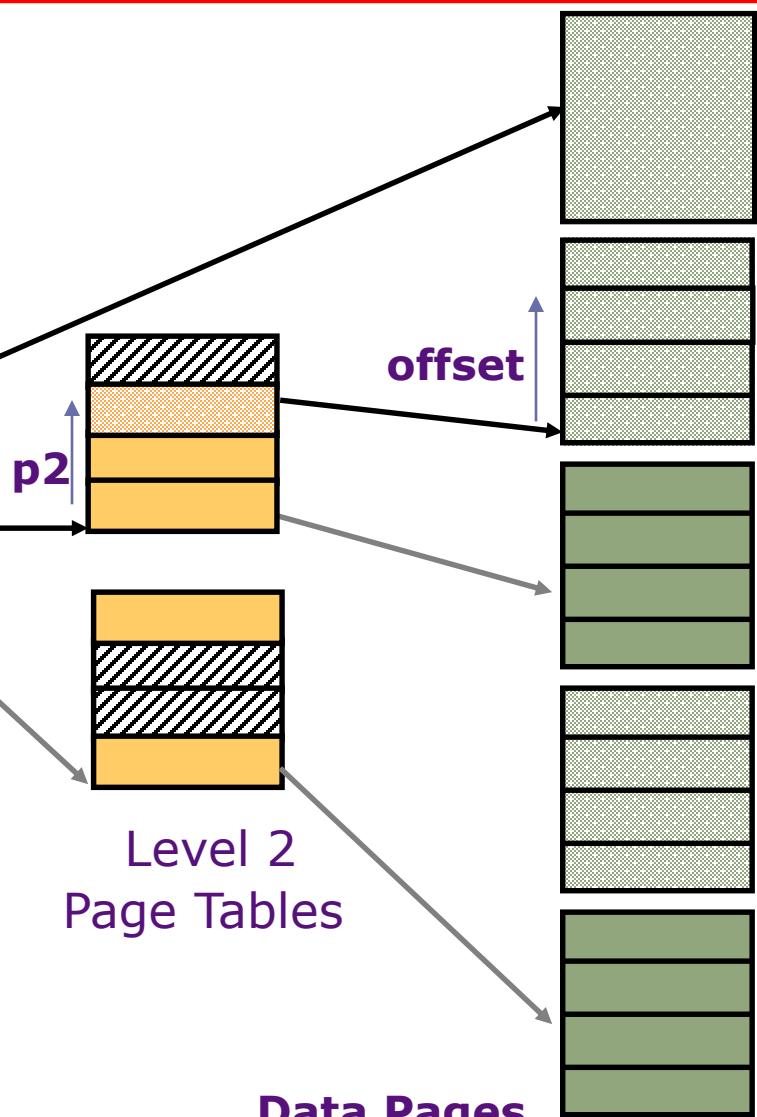
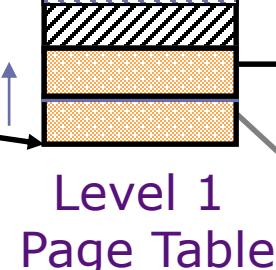
Virtual Address



Root of the Current
Page Table



(Processor
Register)



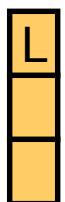
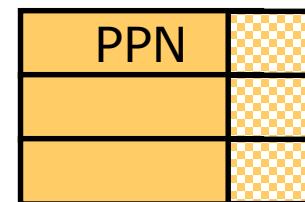
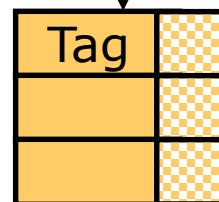
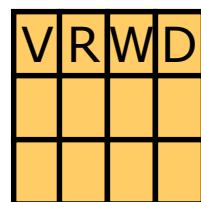
- page in primary memory
- large page in primary memory
- page in secondary memory
- PTE of a nonexistent page

Variable-Size Page TLB

virtual address – small page



large page



hit?

physical address



Large
page?

Alternatively, have a separate TLB
for each page size (pros/cons?)

Handling a TLB Miss

Software (MIPS, Alpha)

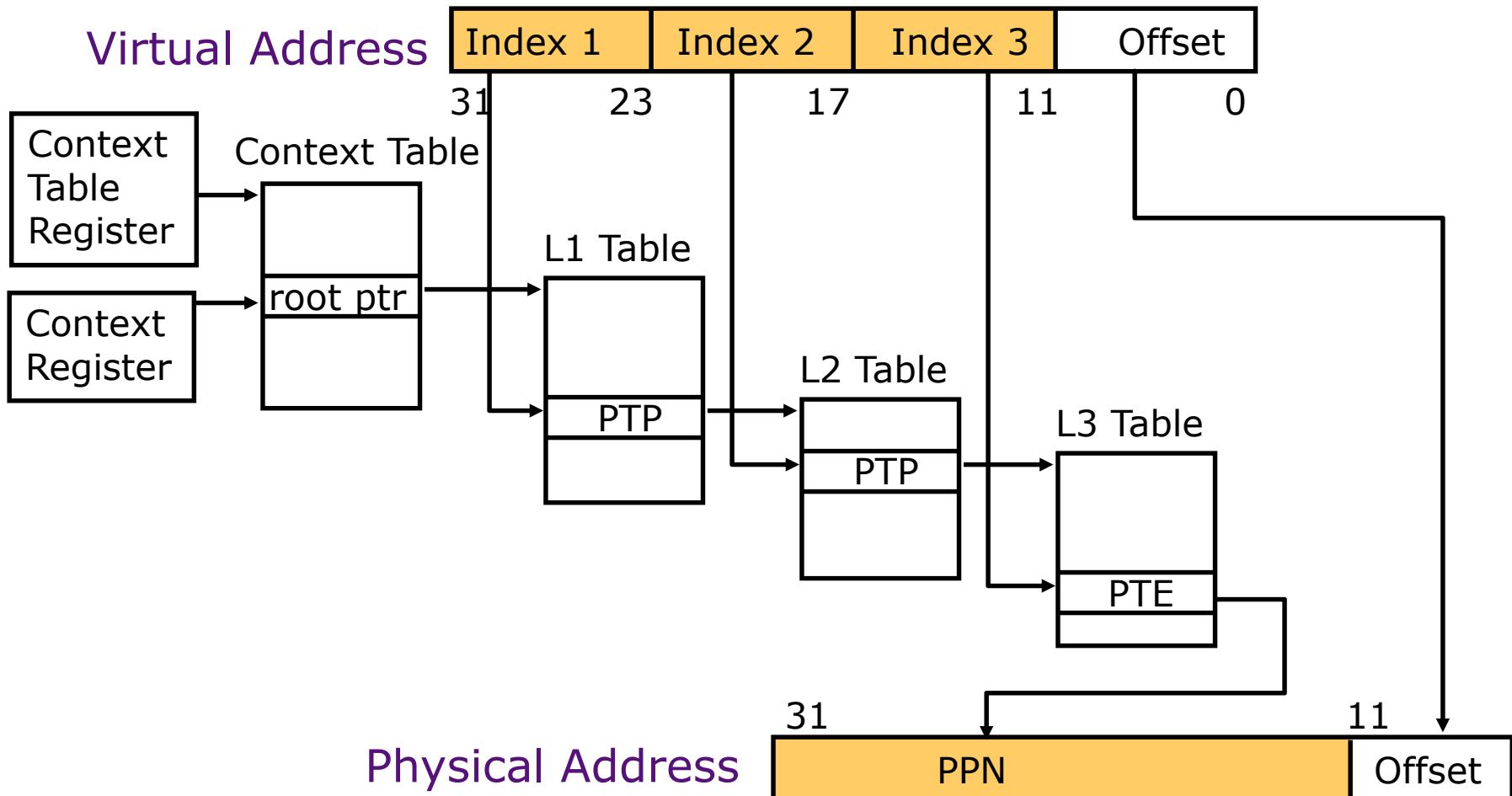
TLB miss causes an exception and the operating system walks the page tables and reloads TLB. A *privileged “untranslated” addressing mode used for walk*

Hardware (SPARC v8, x86, PowerPC)

A memory management unit (MMU) walks the page tables and reloads the TLB

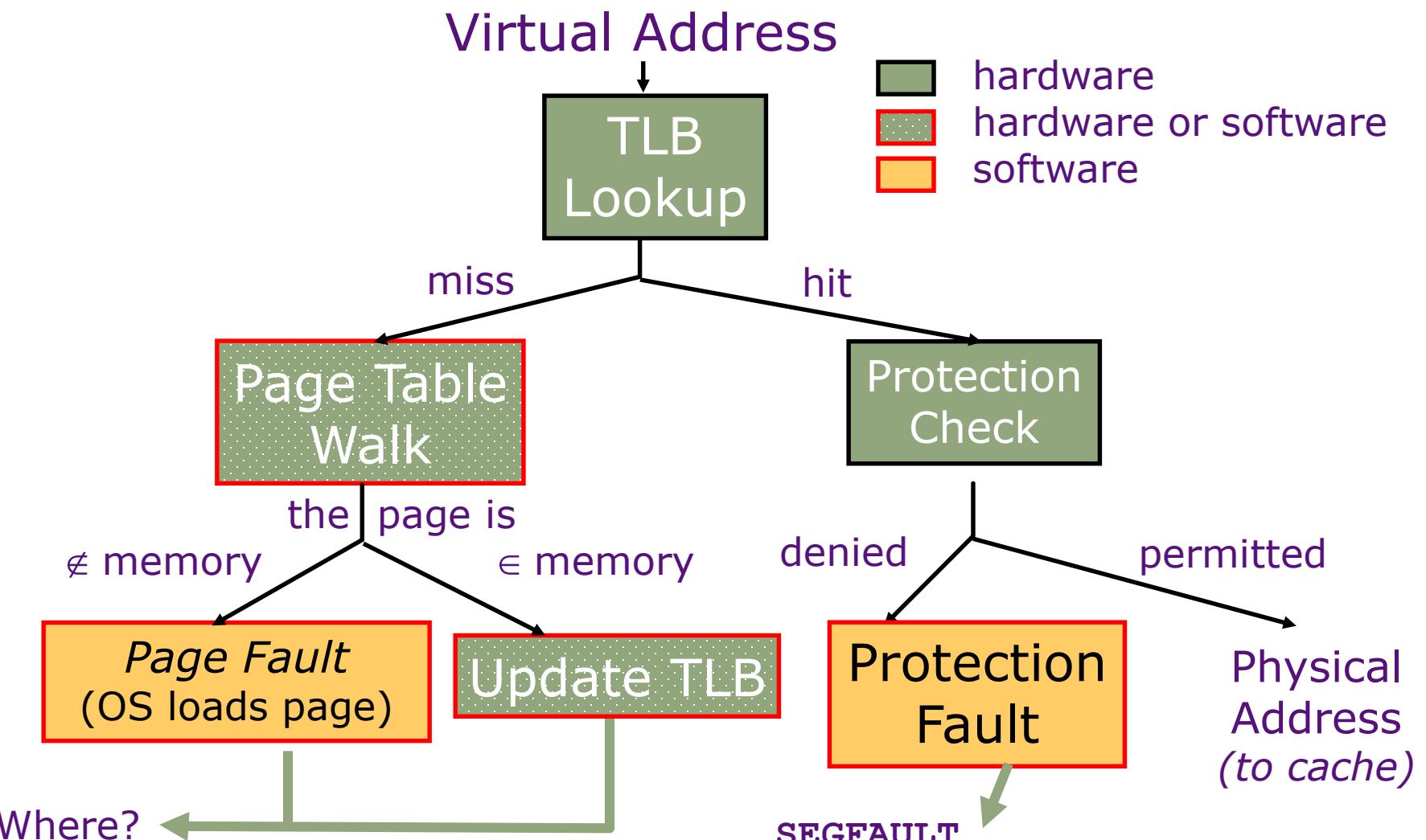
If a missing (data or PT) page is encountered during the TLB reloading, MMU gives up and signals a Page-Fault exception for the original instruction

Hierarchical Page Table Walk: SPARC v8



MMU does this table walk in hardware on a TLB miss

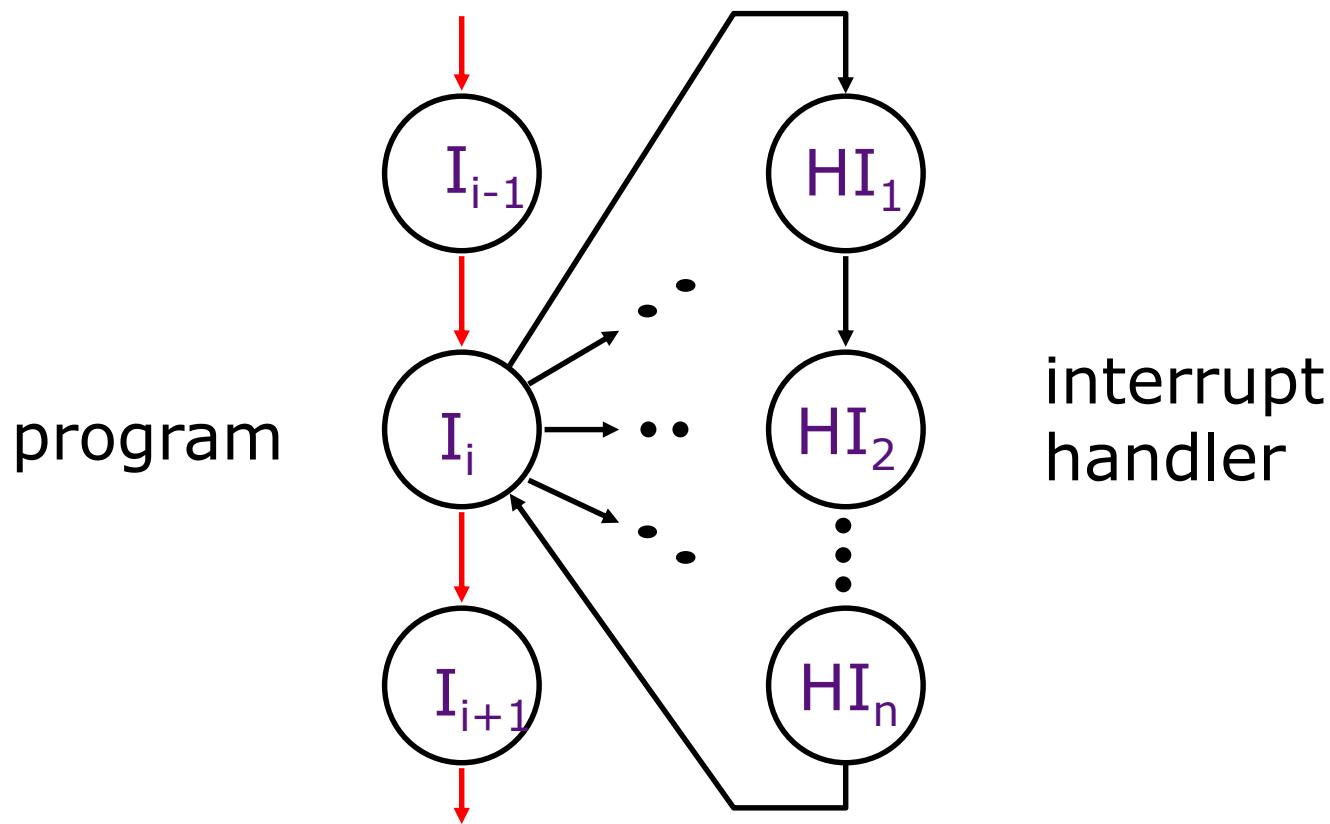
Address Translation: *putting it all together*



Topics

- Interrupts
- Speeding up the common case:
 - TLB & Cache organization
- Speeding up page table walks
- Modern Usage

Interrupts: altering the normal flow of control



An *external or internal event* that needs to be processed by another (system) program. The event is usually unexpected or rare from program's point of view.

Causes of Interrupts

Interrupt: an *event* that requests the attention of the processor

- Asynchronous: an *external event*
 - input/output device service-request
 - timer expiration
 - power disruptions, hardware failure
- Synchronous: an *internal event (a.k.a. exception)*
 - undefined opcode, privileged instruction
 - arithmetic overflow, FPU exception
 - misaligned memory access
 - *virtual memory exceptions*: page faults, TLB misses, protection violations
 - *traps*: system calls, e.g., jumps into kernel

Asynchronous Interrupts

Invoking the interrupt handler

- An I/O device requests attention by asserting one of the *prioritized interrupt request lines*
- When the processor decides to process interrupt
 - It stops the current program at instruction I_i , completing all the instructions up to I_{i-1} (*precise interrupt*)
 - It saves the PC of instruction I_i in a special register (EPC)
 - It disables interrupts and transfers control to a designated interrupt handler running in kernel mode

Interrupt Handler

- Saves EPC before enabling interrupts to allow nested interrupts ⇒
 - need an instruction to move EPC into GPRs
 - need a way to mask further interrupts at least until EPC can be saved
- Needs to read a *status register* that indicates the cause of the interrupt
- Uses a special indirect jump instruction RFE (*return-from-exception*) that
 - enables interrupts
 - restores the processor to the user mode
 - restores hardware status and control state

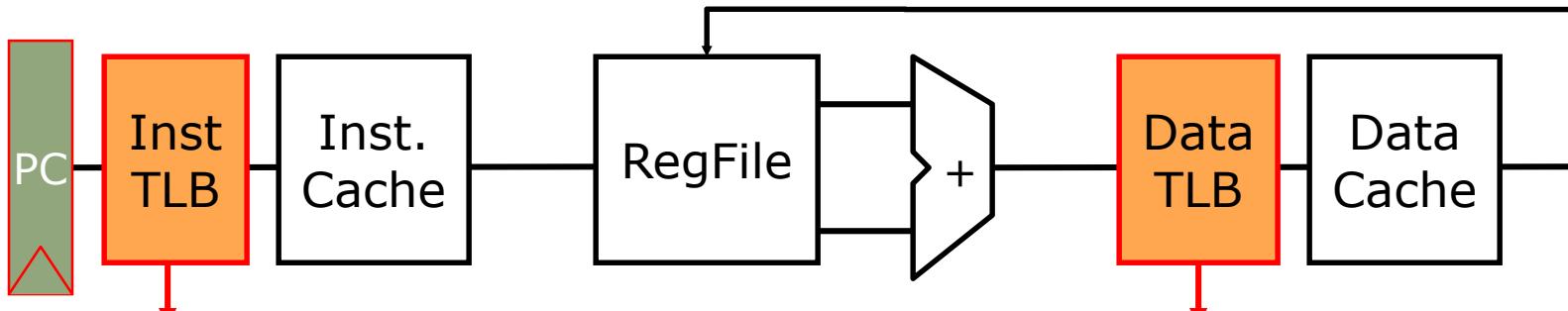
Synchronous Interrupts

- A synchronous interrupt (exception) is caused by a *particular instruction*
- In general, the instruction cannot be completed and needs to be *restarted* after the exception has been handled
 - With pipelining, requires undoing the effect of one or more partially executed instructions
- In case of a trap (system call), the instruction is considered to have been completed
 - A special jump instruction involving a change to privileged kernel mode

Topics

- Interrupts
- Speeding up the common case:
 - TLB & Cache organization
- Speeding up page table walks
- Modern Usage

Address Translation in CPU

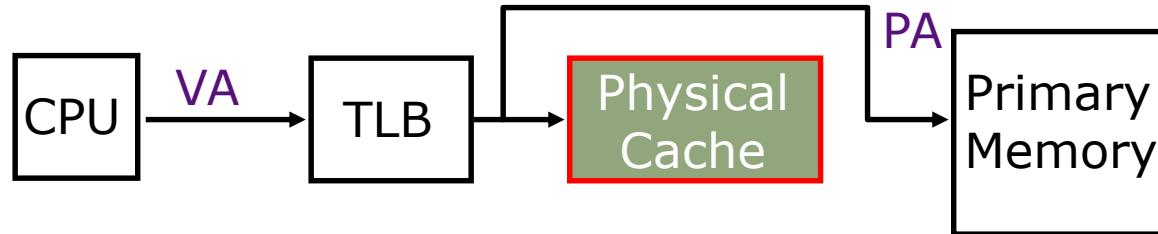


*TLB miss? Page Fault?
Protection violation?*

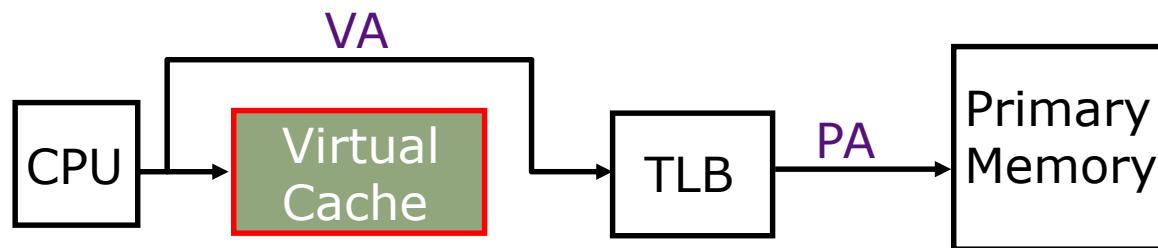
*TLB miss? Page Fault?
Protection violation?*

- Software handlers need a *restartable* exception on page fault or protection violation
- Handling a TLB miss needs a *hardware* or *software* mechanism to refill TLB
- Need mechanisms to cope with the additional latency of TLB:
 - slow down the clock
 - pipeline the TLB and cache access
 - virtual-address caches
 - parallel TLB/cache access

Virtual-Address Caches

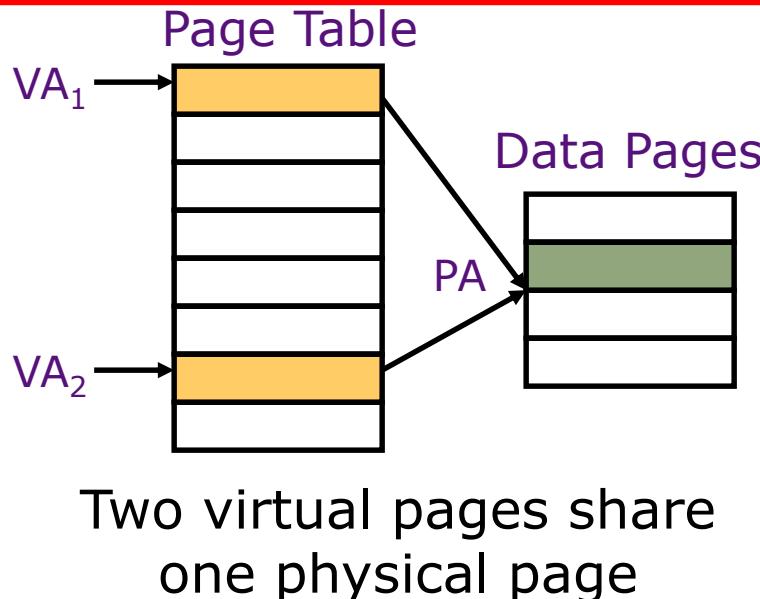


Alternative: place the cache before the TLB



- one-step process in case of a hit (+)
- cache needs to be flushed on a context switch unless address space identifiers (ASIDs) included in tags (-)
- *aliasing problems* due to the sharing of pages (-)

Aliasing in Virtual-Address Caches



Tag	Data
VA_1	1st Copy of Data at PA
VA_2	2nd Copy of Data at PA

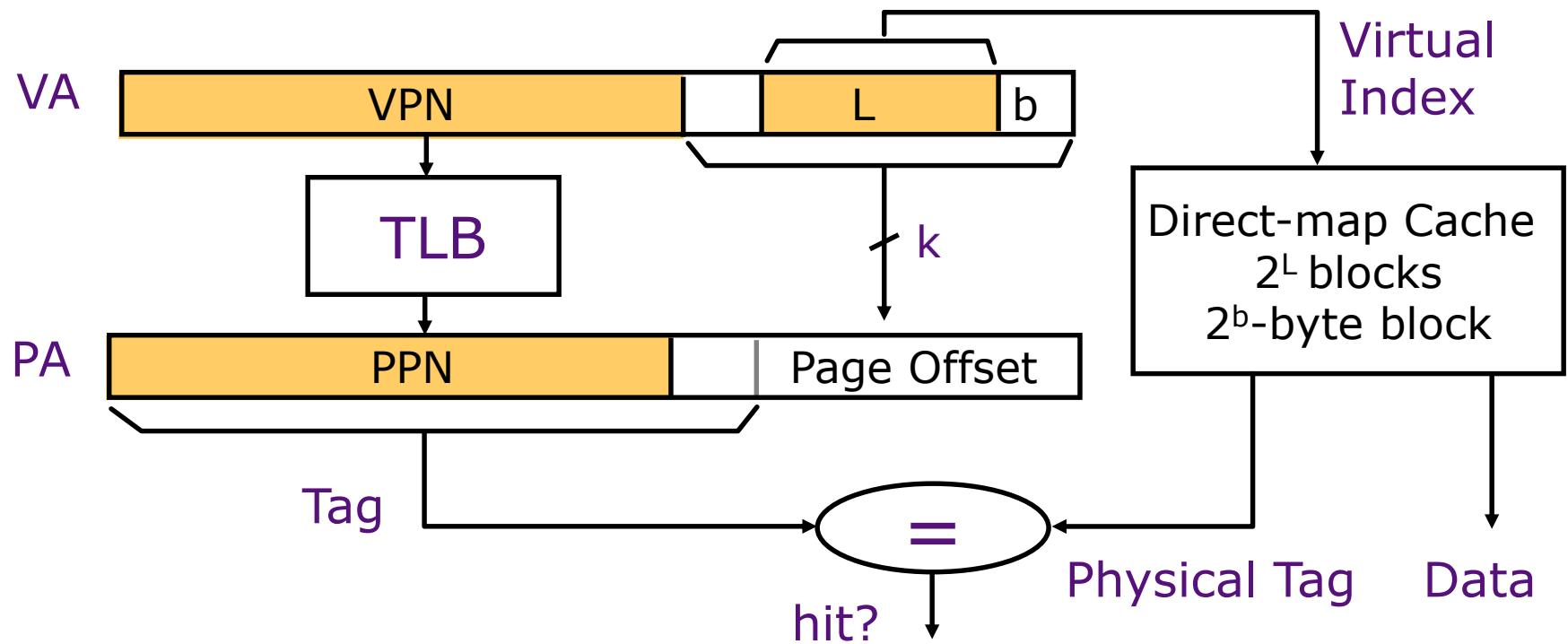
Virtual cache can have two copies of same physical data. Writes to one copy not visible to reads of other!

General Solution: *Disallow aliases to coexist in cache*

Software (i.e., OS) solution for direct-mapped cache

VAs of shared pages must agree in cache index bits; this ensures all VAs accessing same PA will conflict in direct-mapped cache (early SPARCcs)

Concurrent Access to TLB & Cache



Index L is available without consulting the TLB

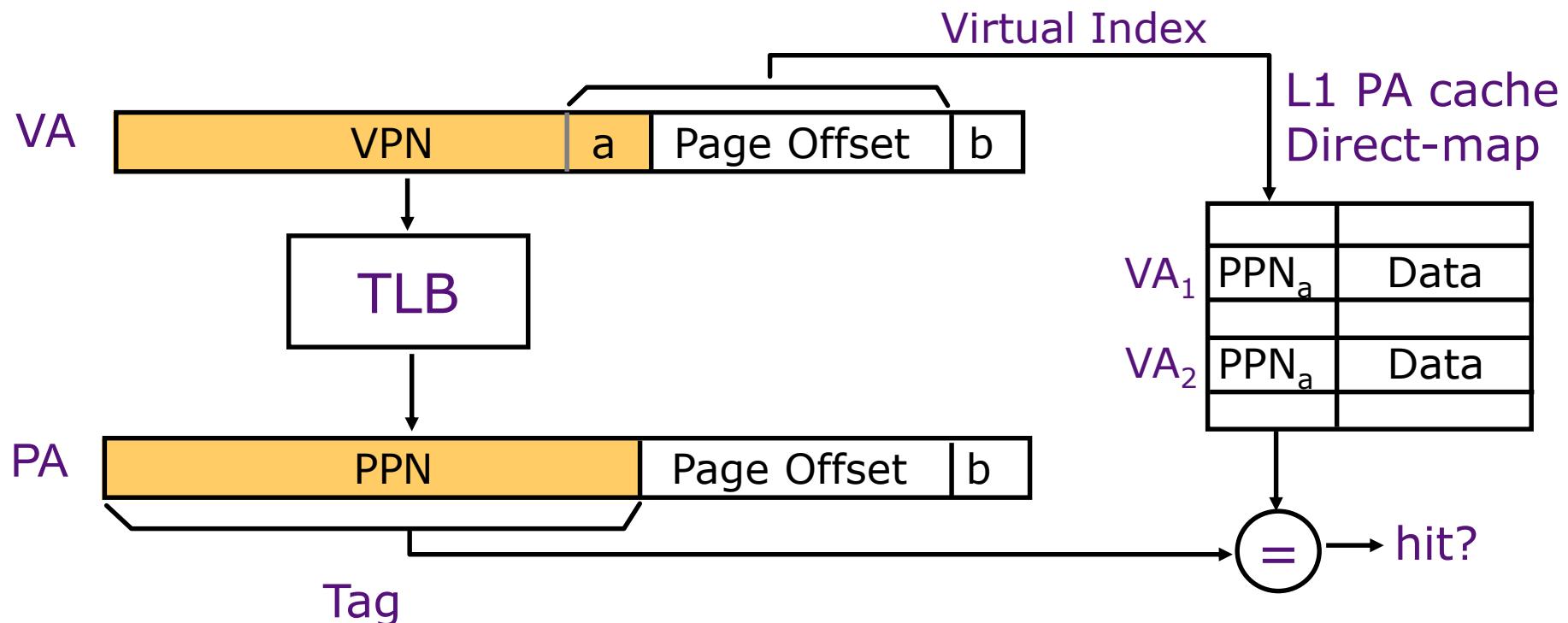
⇒ *cache and TLB accesses can begin simultaneously*

Tag comparison is made after both accesses are completed

When does this work? $L + b < k$ $L + b = k$ $L + b > k$

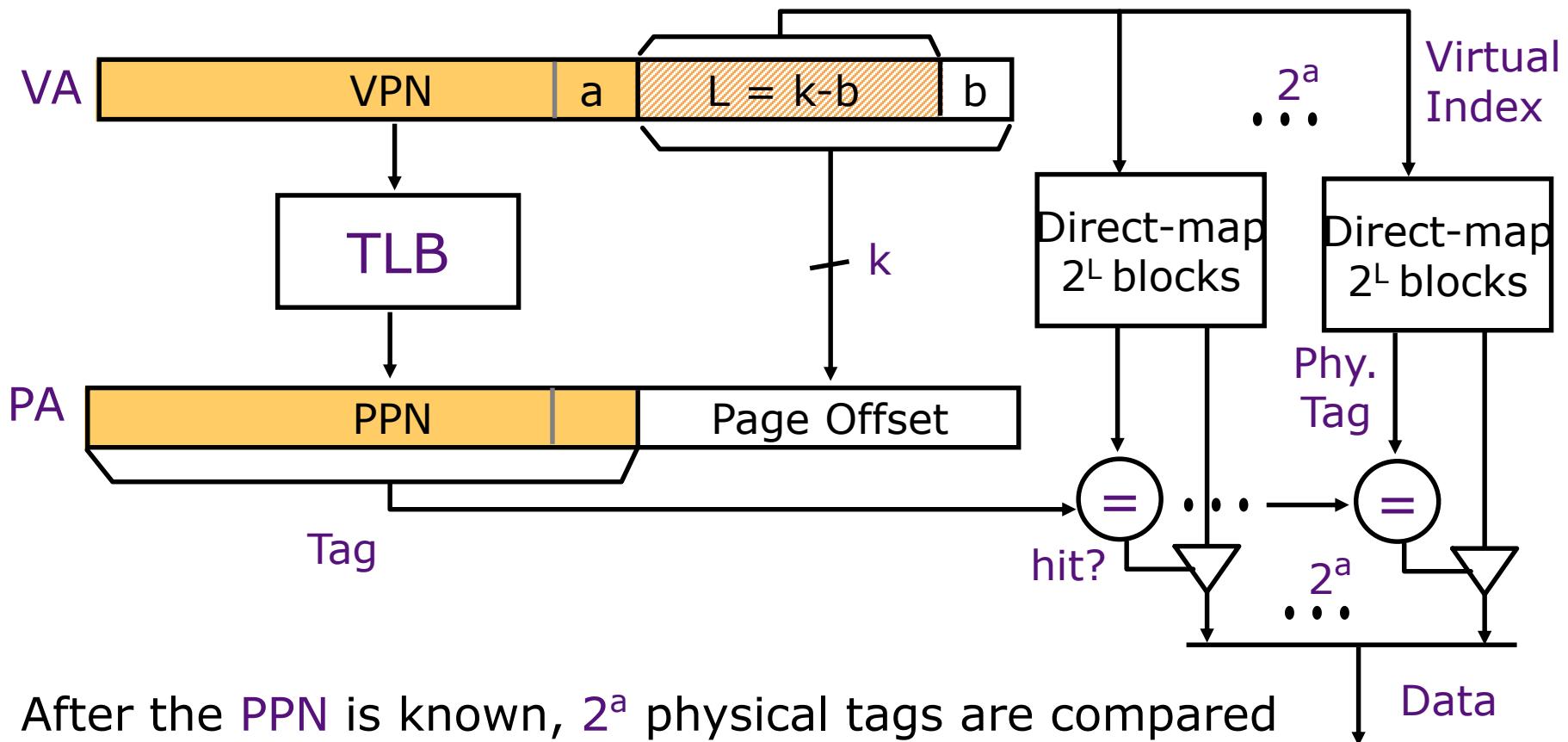
Concurrent Access to TLB & Large L1

The problem with $L1 > \text{Page size}$



Can VA₁ and VA₂ both map to PA?

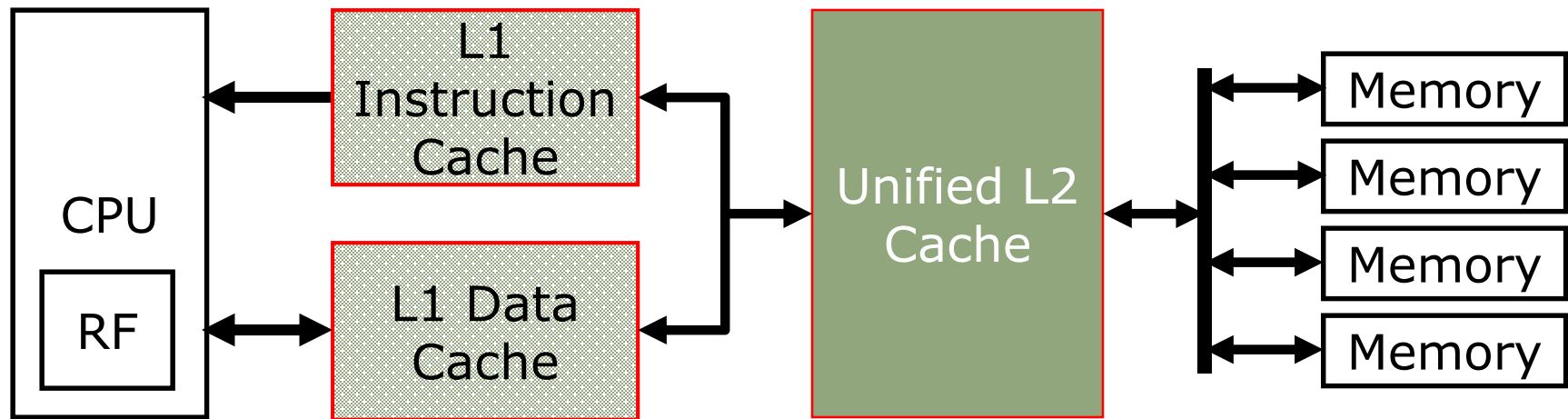
Virtual-Index Physical-Tag Caches: Associative Organization



After the PPN is known, 2^a physical tags are compared

Is this scheme realistic?

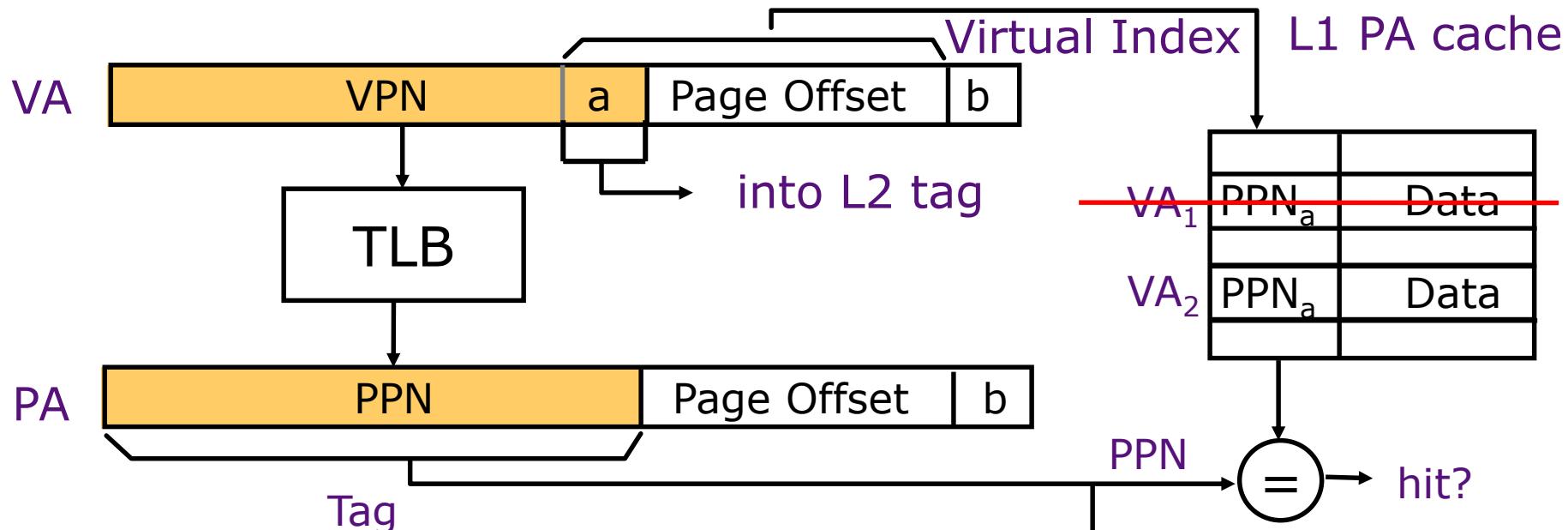
A solution via Second-Level Cache



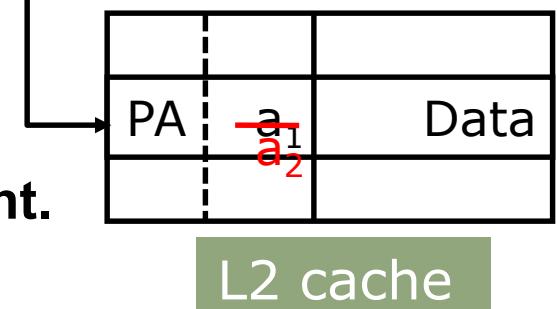
Usually a common L2 cache backs up both
Instruction and Data L1 caches

L2 is “inclusive” of both Instruction and Data caches

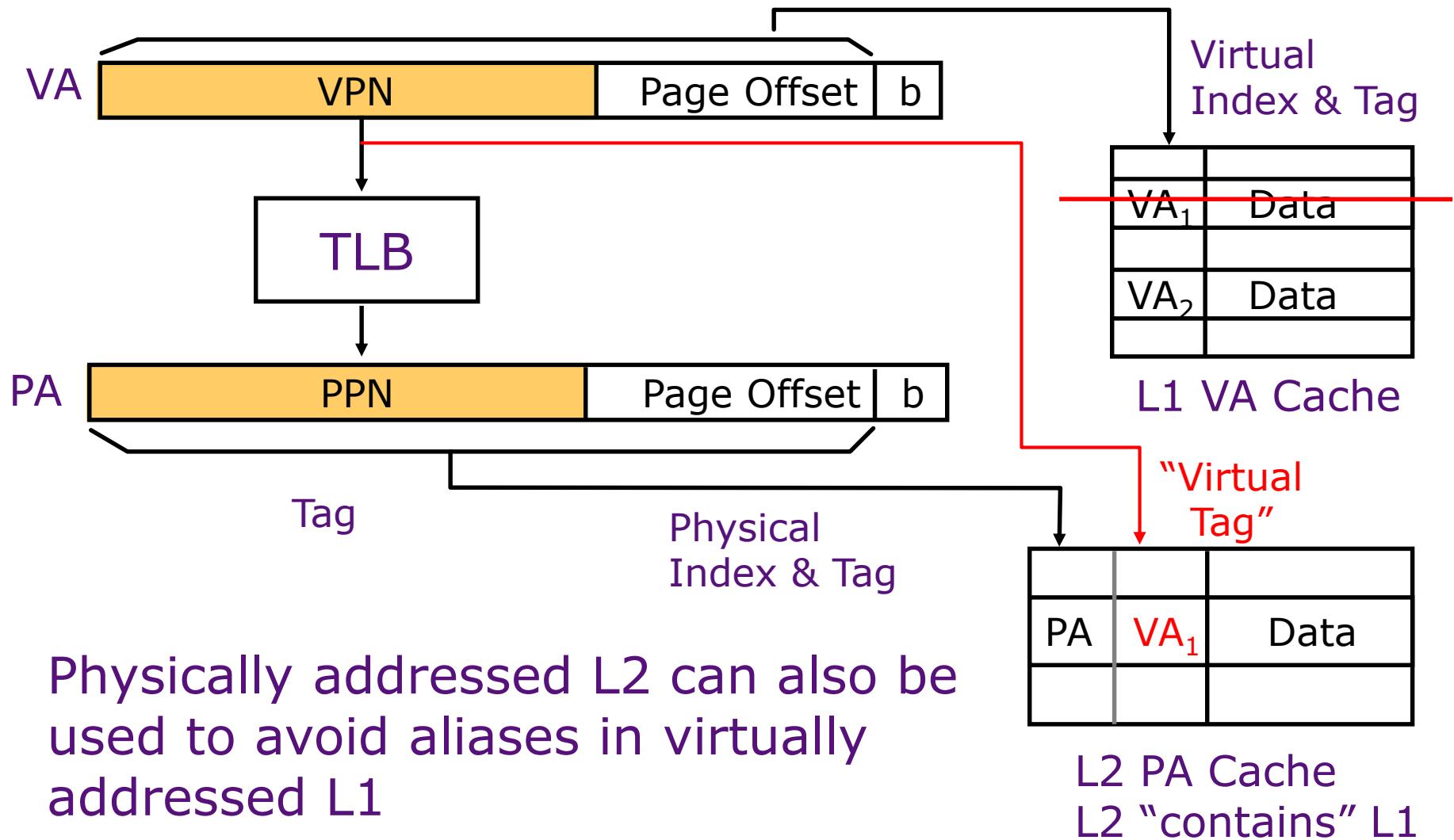
Anti-Aliasing Using L2: MIPS R10000



- Suppose VA1 and VA2 both map to PA and VA1 is already in L1, L2 ($VA_1 \neq VA_2$)
- After VA2 is resolved to PA, collision is detected in L2. Collision \rightarrow **Field a is different.**
- VA1 will be purged from L1, and VA2 will be loaded \Rightarrow *no aliasing!*



Virtually Addressed L1: Anti-Aliasing using L2



Topics

- Interrupts
- Speeding up the common case:
 - TLB & Cache organization
- Speeding up page table walks
- Modern Usage

Page Fault Handler

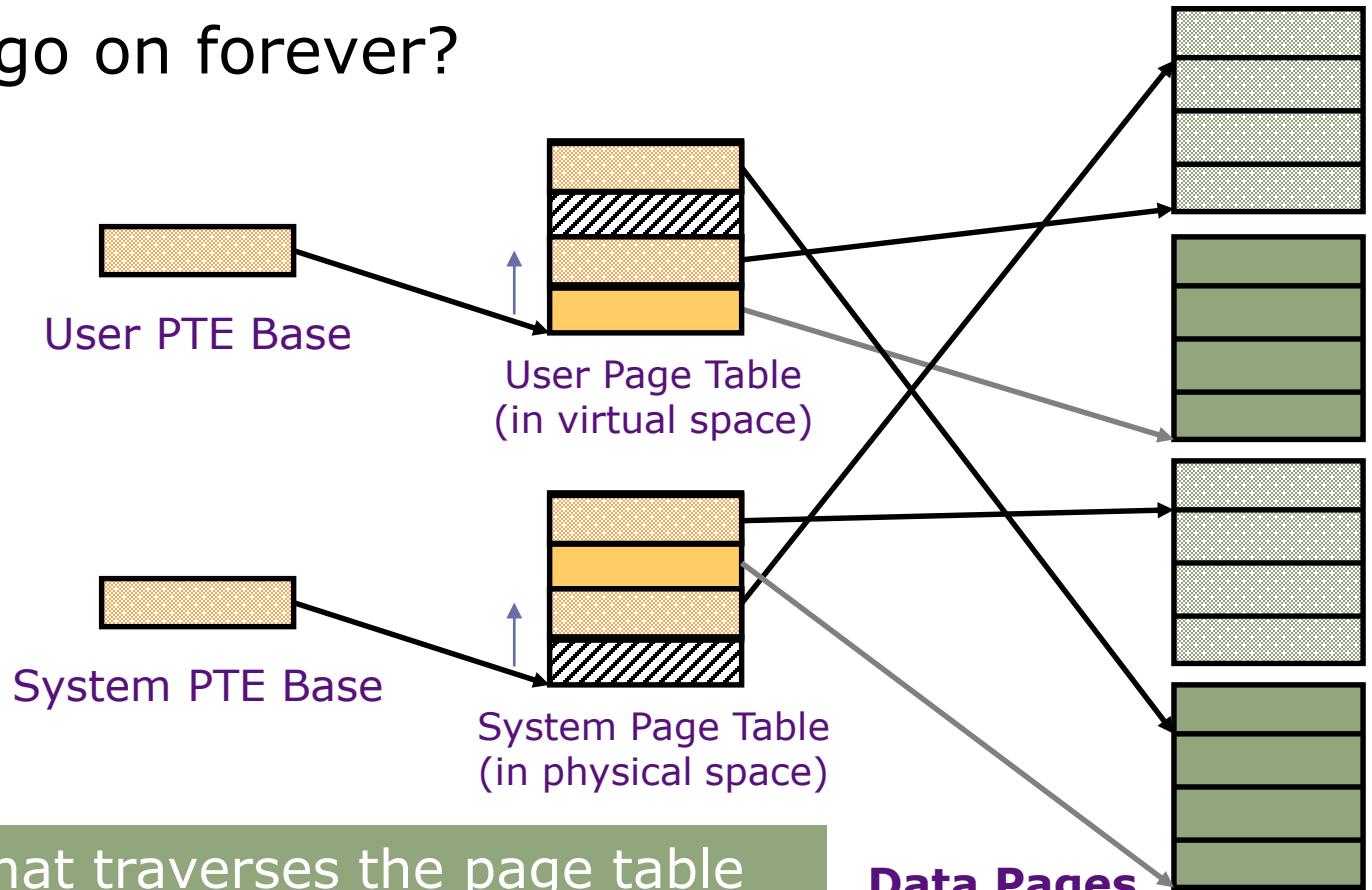
- When the referenced page is not in DRAM:
 - The missing page is located (or created)
 - It is brought in from disk, and page table is updated

Another job may be run on the CPU while the first job waits for the requested page to be read from disk
 - If no free pages are left, a page is swapped out

Pseudo-LRU replacement policy
- Since it takes a long time to transfer a page (msecs), page faults are handled completely in software by the OS
 - Untranslated addressing mode is essential to allow kernel to access page tables

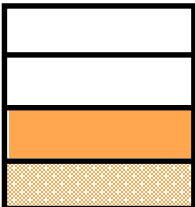
Translation for Page Tables

- Can references to page tables cause TLB misses?
- Can this go on forever?

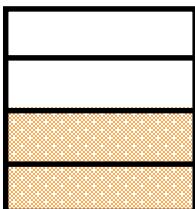


A program that traverses the page table needs a “no translation” addressing mode.

Swapping a Page of a Page Table



A PTE in primary memory contains primary or secondary memory addresses



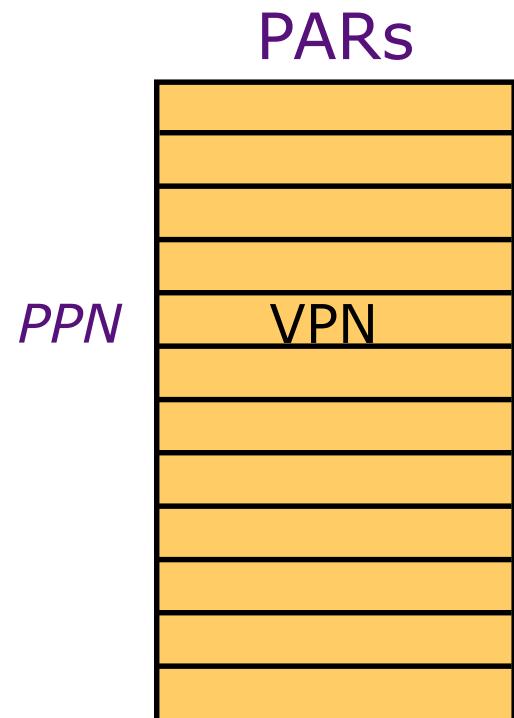
A PTE in secondary memory contains *only* secondary memory addresses

⇒ a page of a PT can be swapped out only if none of its PTE's point to pages in the primary memory

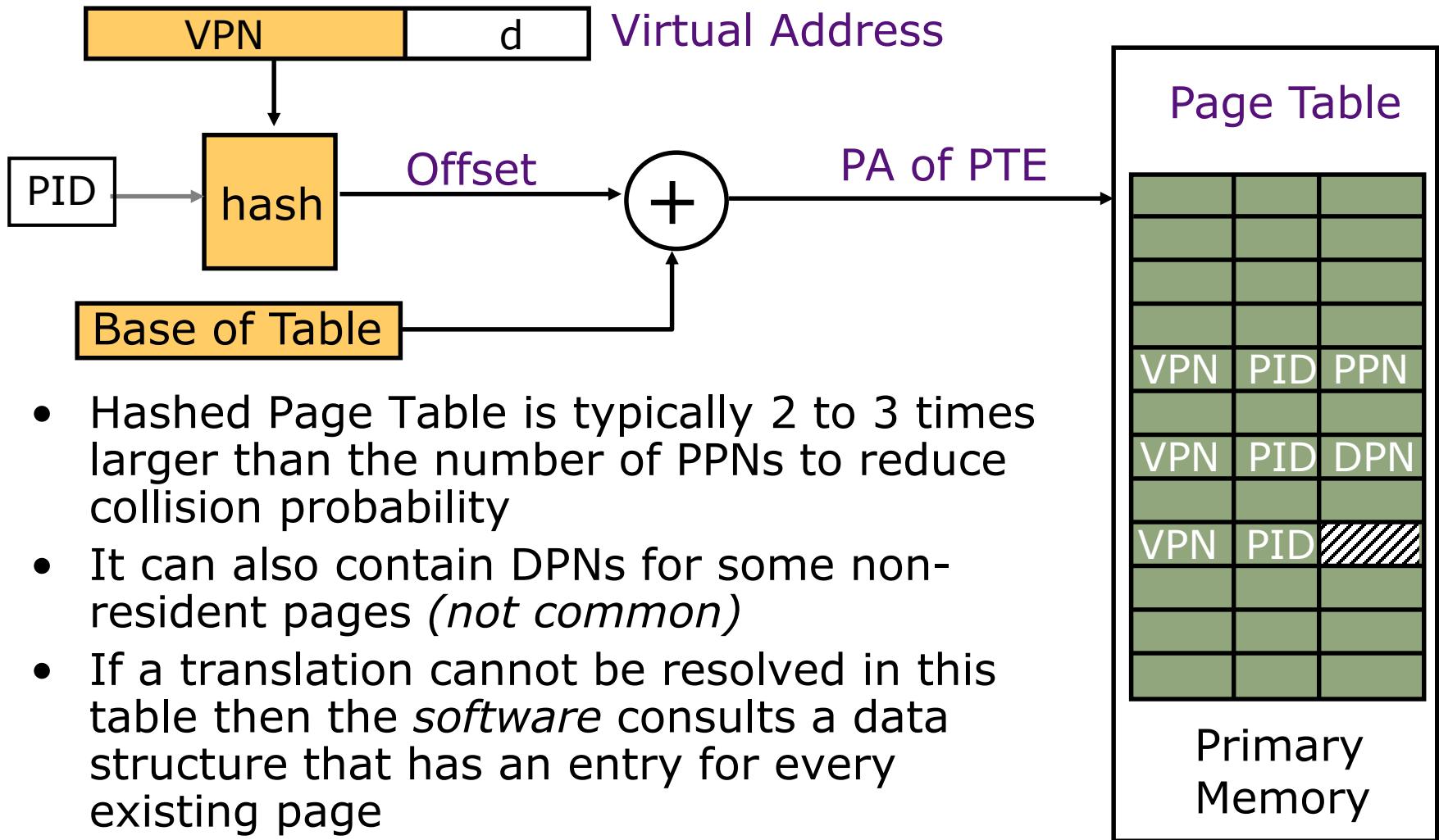
Why?

Atlas Revisited

- One PAR for each physical page
- PAR's contain the VPN's of the pages *resident in primary memory*
- *Advantage:* The size is proportional to the size of the primary memory
- *What is the disadvantage?*



Hashed Page Table: Approximating Associative Addressing



Virtual Memory Use Today - 1

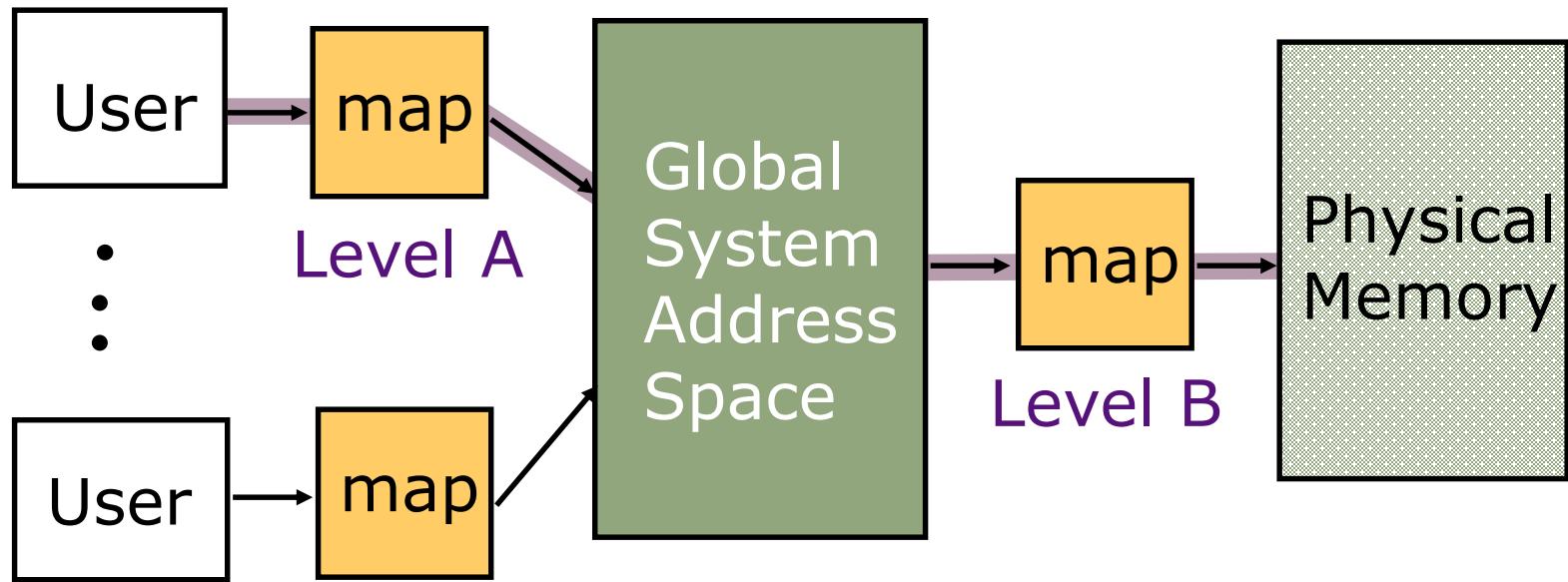
- Desktop/server/cellphone processors have full demand-paged virtual memory
 - Portability between machines with different memory sizes
 - Protection between multiple users or multiple tasks
 - Share small physical memory among active tasks
 - Simplifies implementation of some OS features
- Vector supercomputers and GPUs have translation and protection but not demand paging
(Older Crays: base&bound, Japanese & Cray X1: pages)
 - Don't waste expensive processor time thrashing to disk (make jobs fit in memory)
 - Mostly run in batch mode (run set of jobs that fits in memory)
 - Difficult to implement restartable vector instructions

Virtual Memory Use Today - 2

- Most embedded processors and DSPs provide physical addressing only
 - Can't afford area/speed/power budget for virtual memory support
 - Often there is no secondary storage to swap to!
 - Programs custom-written for particular memory configuration in product
 - Difficult to implement restartable instructions for exposed architectures

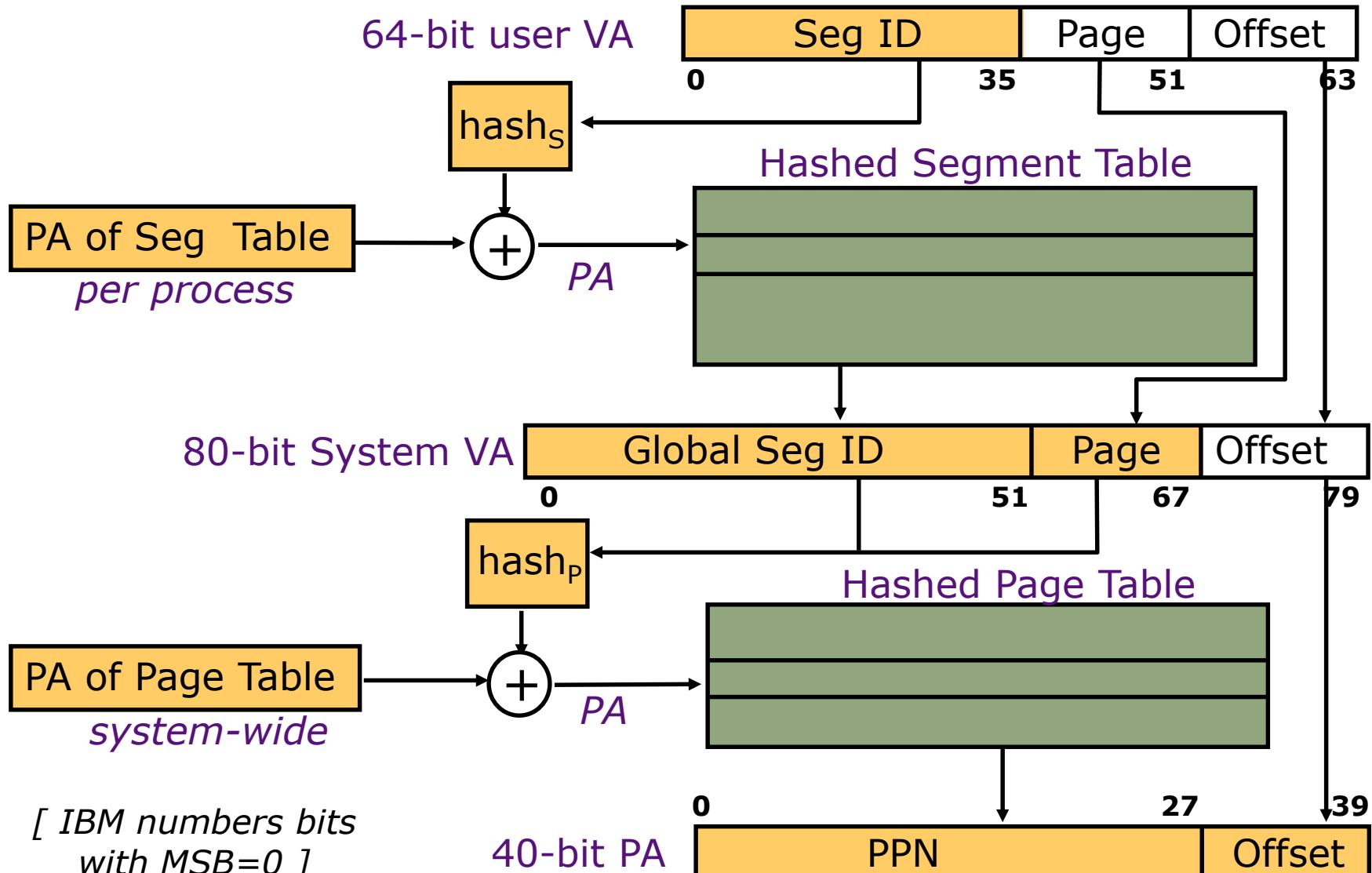
Next lecture: Pipelining!

Global System Address Space

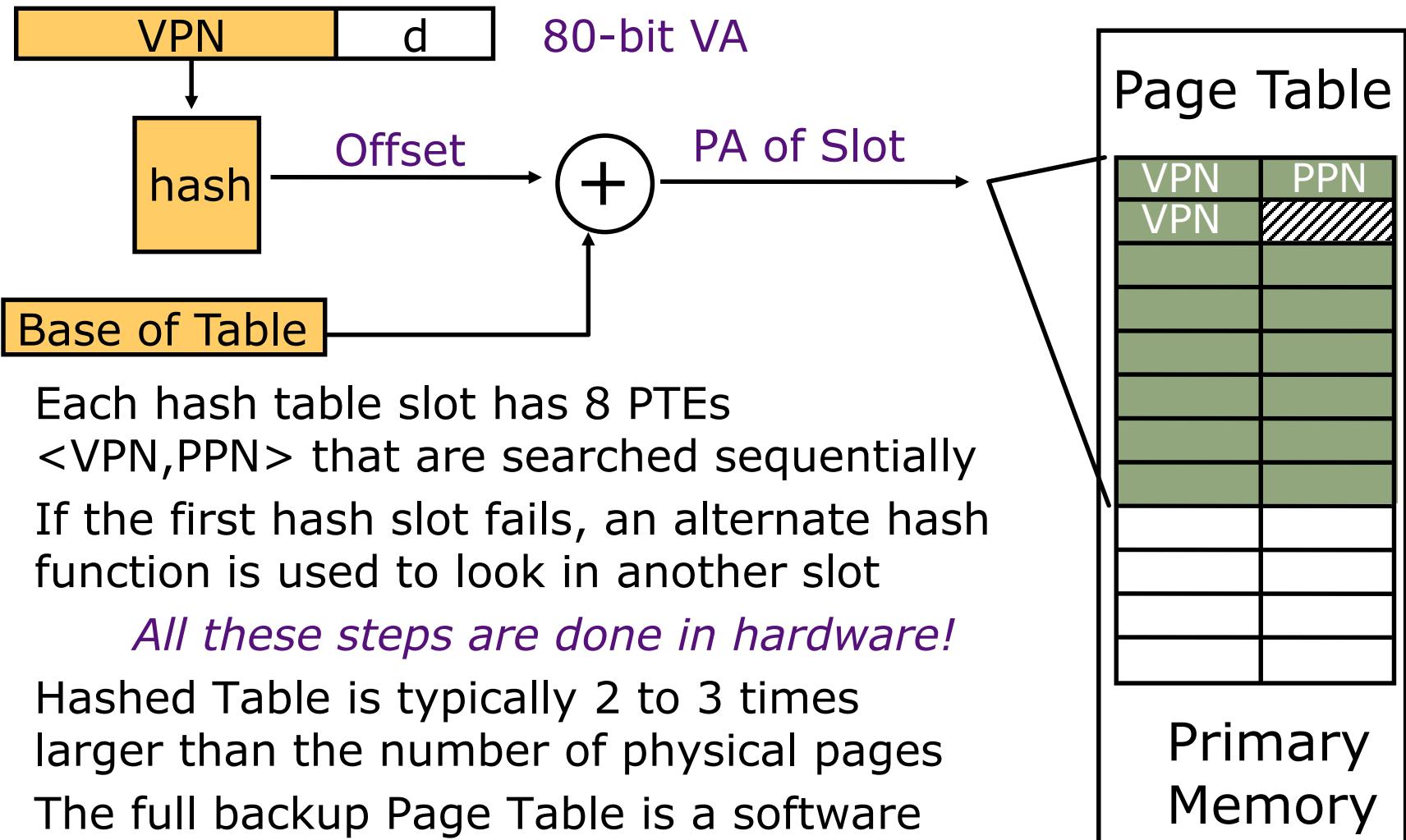


- Level A maps users' address spaces into the global space providing privacy, protection, sharing etc.
- Level B provides demand paging for the large global system address space
- Level A and Level B translations may be kept in separate TLB's

Hashed Page Table Walk: PowerPC Two-level, Segmented Addressing



Power PC: Hashed Page Table

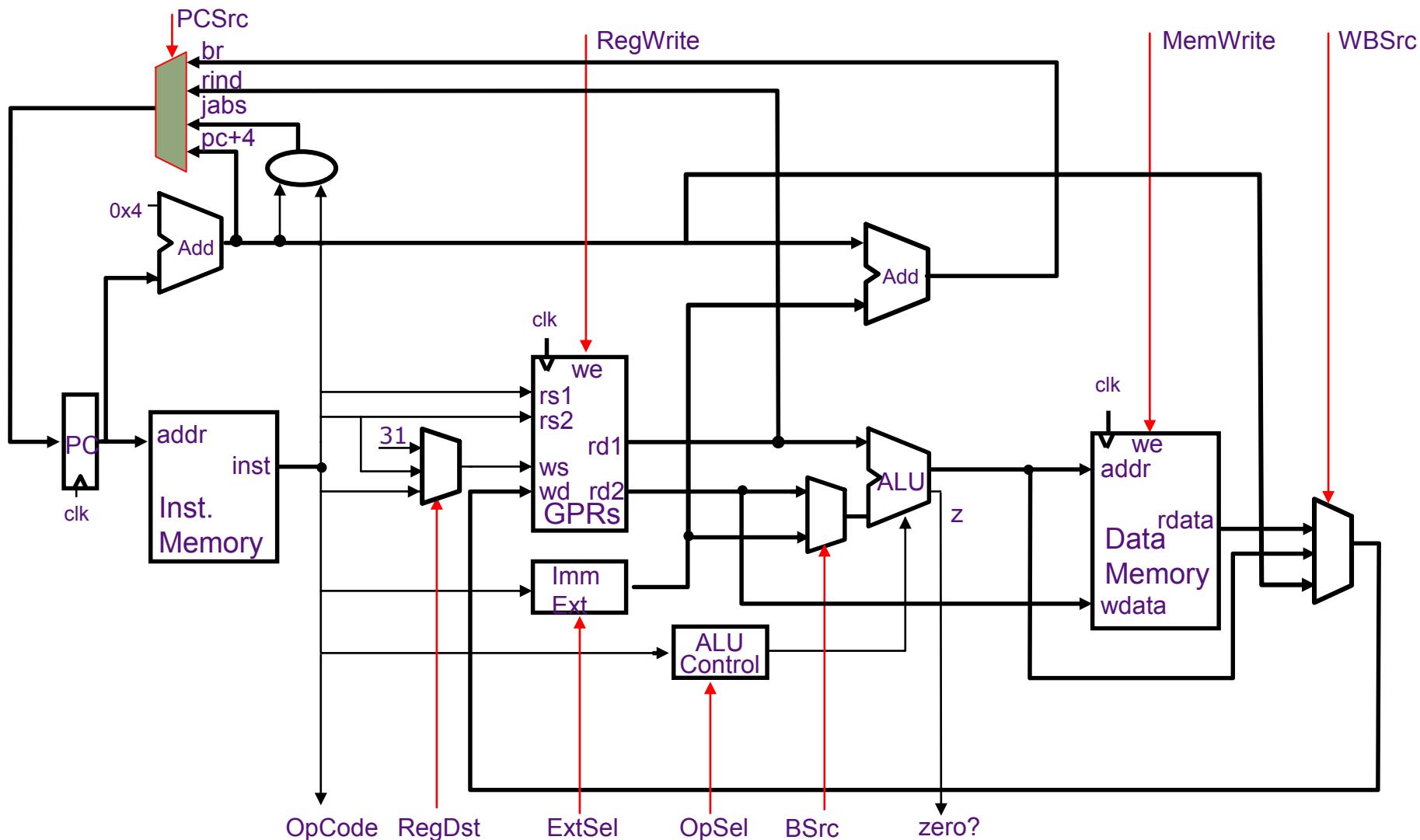


Instruction Pipelining and Hazards

Daniel Sanchez

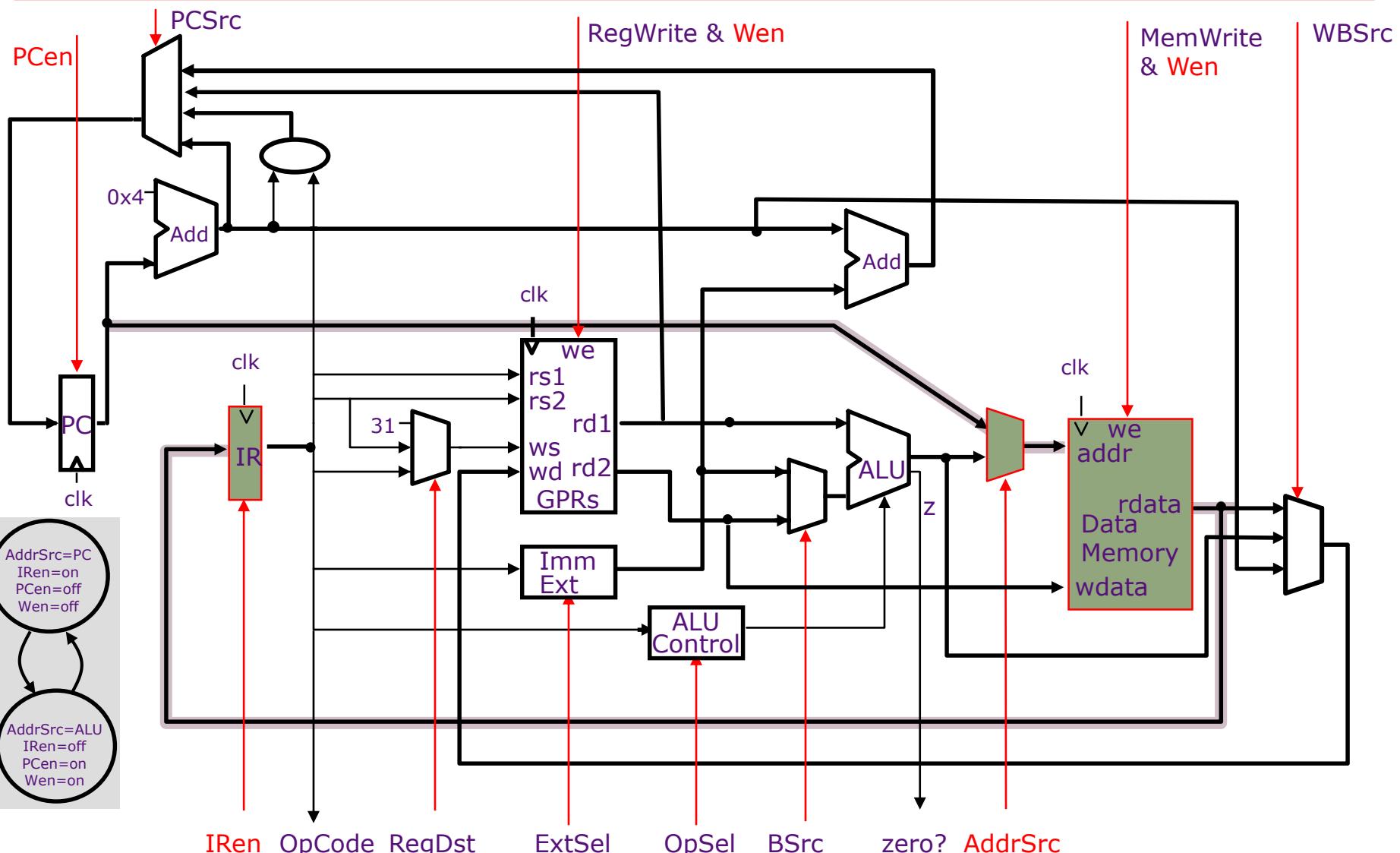
Computer Science and Artificial Intelligence Laboratory
M.I.T.

Reminder: Harvard-Style Single-Cycle Datapath for MIPS

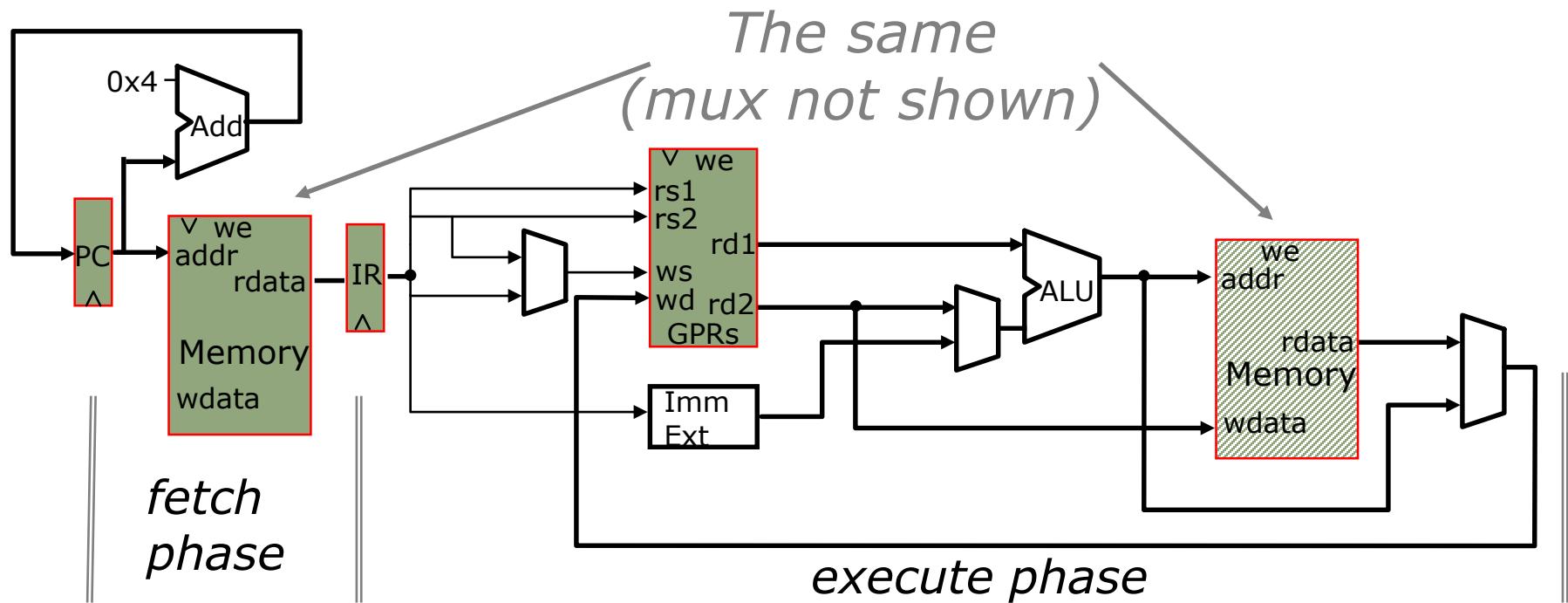


Reminder: Princeton Microarchitecture

Datapath & Control for 2 cycles-per-instruction



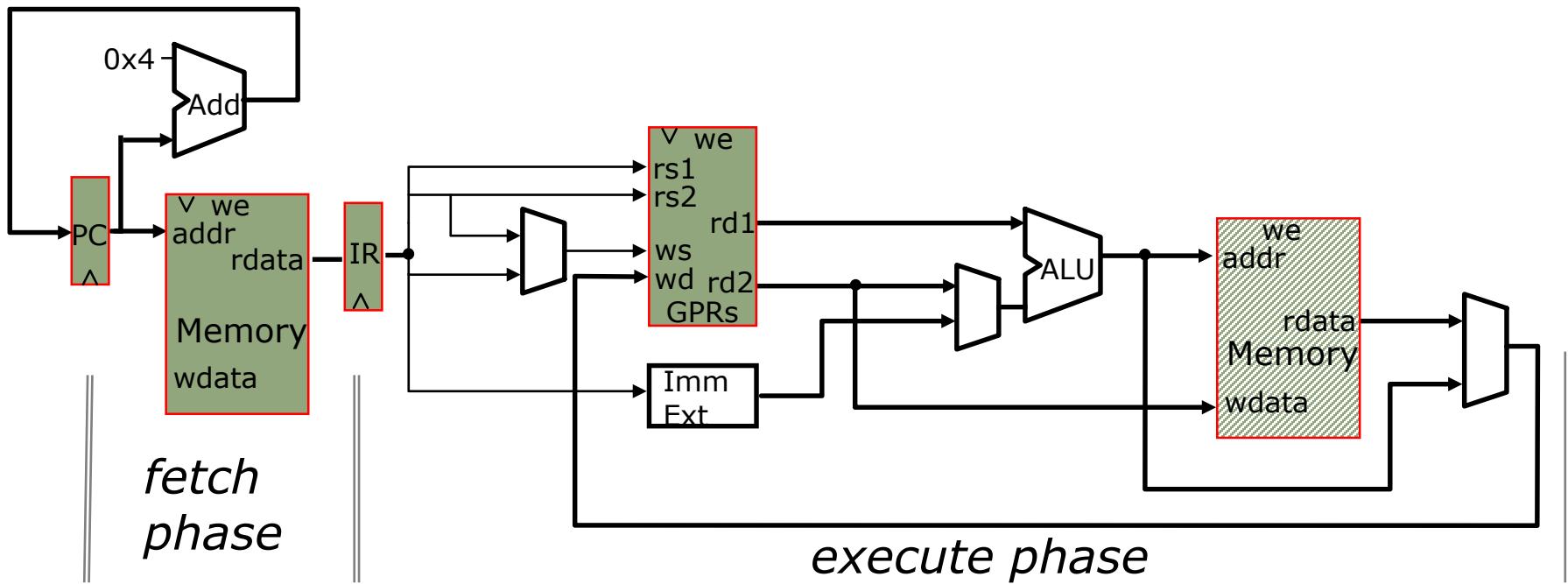
Princeton Microarchitecture (redrawn)



Only one of the phases is active in any cycle
→ a lot of datapath not used at any given time

Princeton Microarchitecture

Overlapped execution



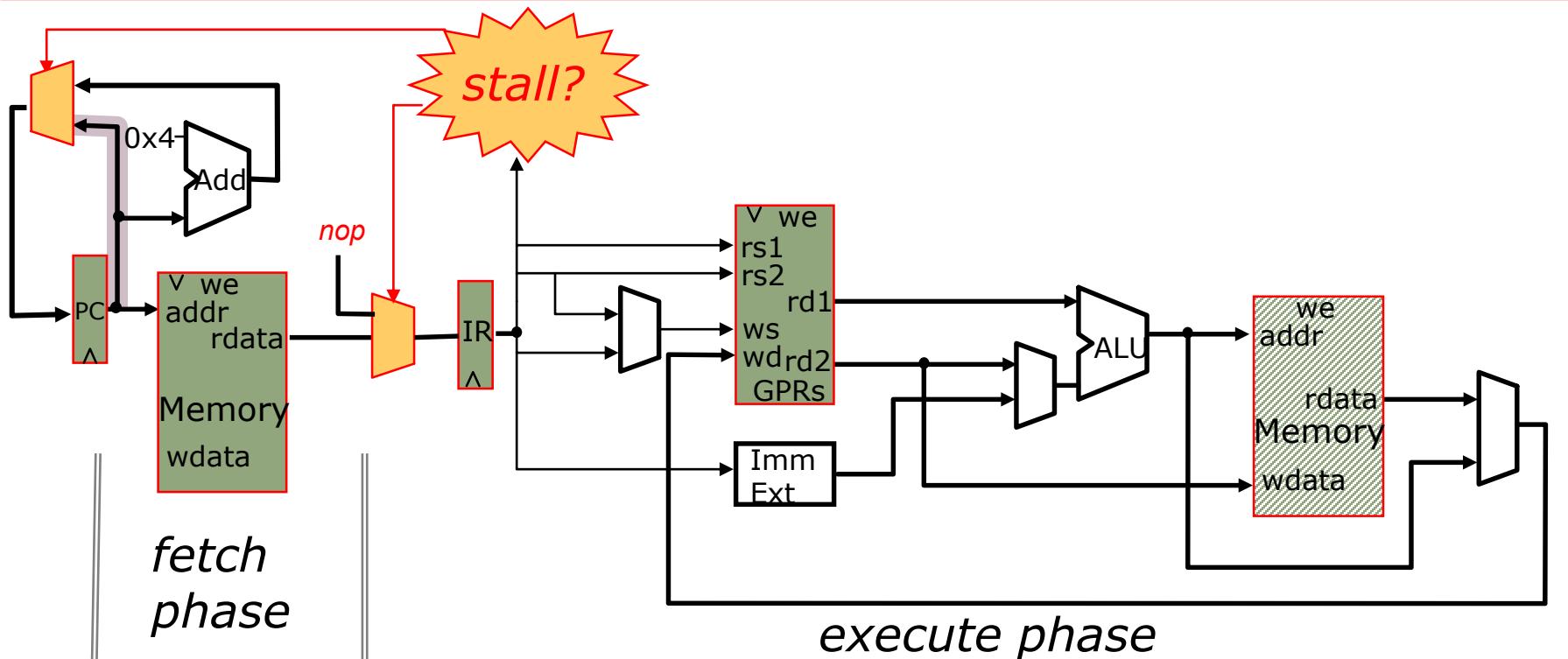
Can we overlap instruction fetch and execute?

Which action should be prioritized?

What do we do with Fetch?

Stalling the instruction fetch

Princeton Microarchitecture



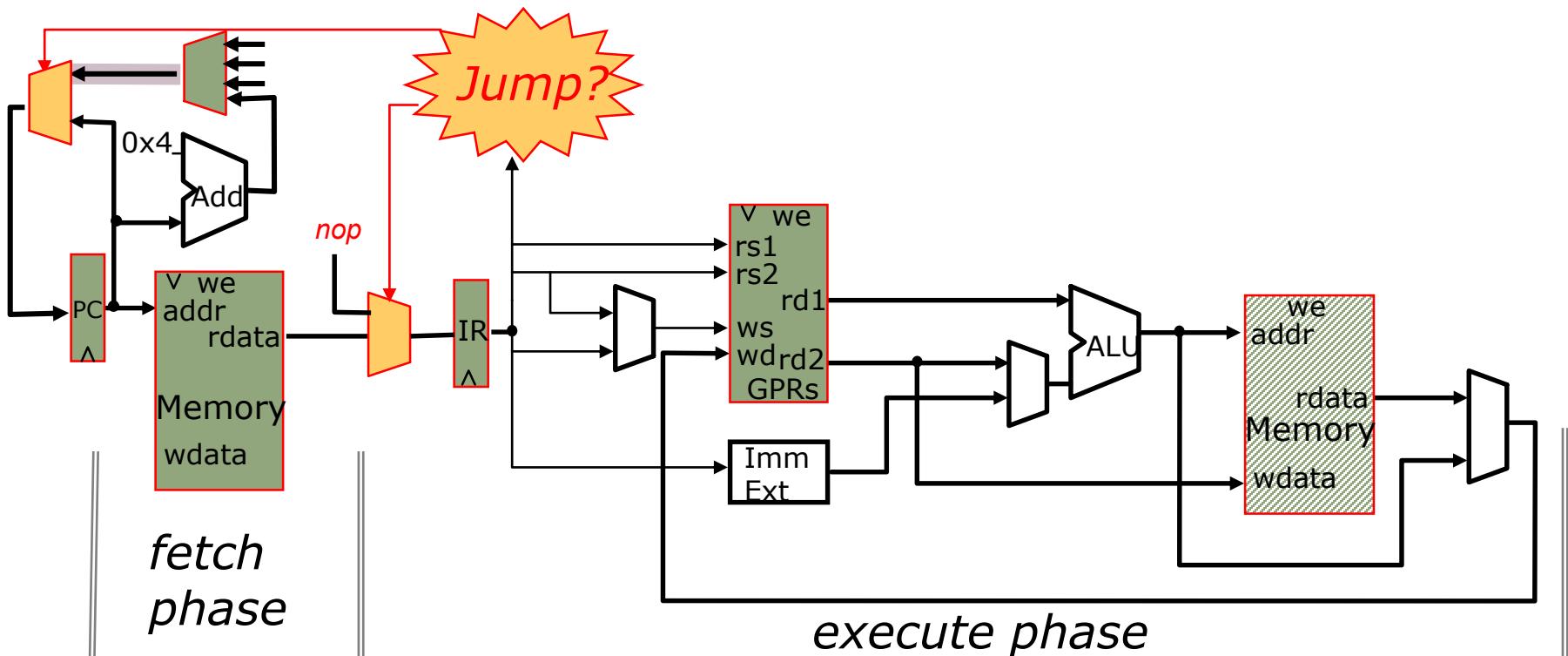
When stall condition is indicated

- *don't fetch a new instruction and don't change the PC*
- *insert a nop in the IR*
- *set the Memory Address mux to ALU (not shown)*

What if IR contains a jump or branch instruction?

Need to stall on branches

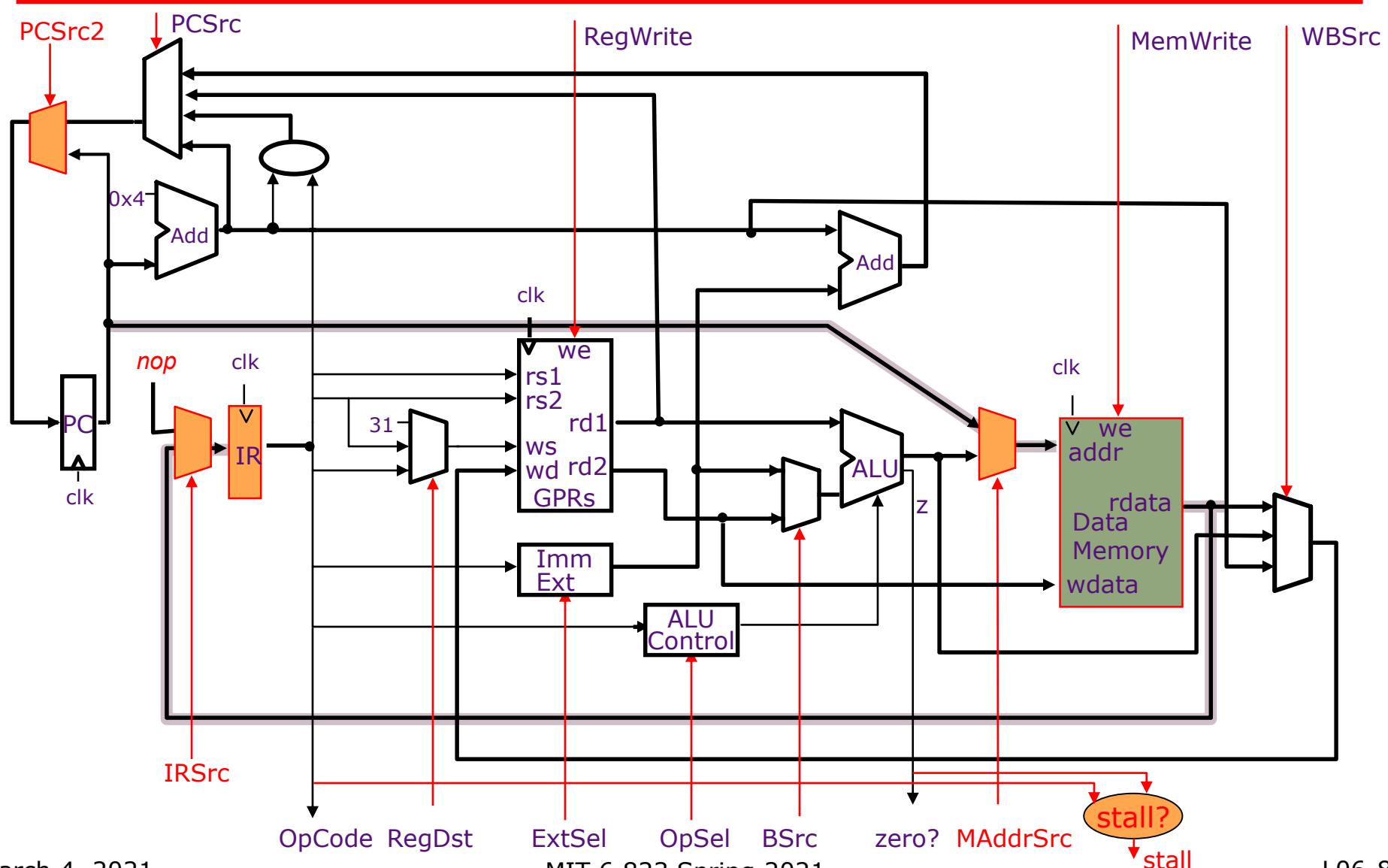
Princeton Microarchitecture



When IR contains a jump or taken branch

- no “*structural conflict*” for the memory
- but we do not have the correct PC value in the PC
- memory cannot be used – Address Mux setting is irrelevant
- insert a *nop* in the IR
- insert the *nextPC* (*branch-target*) address in the PC

Pipelined Princeton Microarchitecture



Pipelined Princeton: Control Table

Opcode	Stall	Ext Sel	B Src	Op Sel	Mem W	Reg W	WB Src	Reg Dst	PC Src1	PC Src2	IR Src	MAddr Src
ALU	no	*	Reg	Func	no	yes	ALU	rd	pc+4	npc	mem	pc
ALUi	no	sE ₁₆	Imm	Op	no	yes	ALU	rt	pc+4	npc	mem	pc
ALUiu	no	uE ₁₆	Imm	Op	no	yes	ALU	rt	pc+4	npc	mem	pc
LW	yes	sE ₁₆	Imm	+	no	yes	Mem	rt	pc+4	pc	nop	ALU
SW	yes	sE ₁₆	Imm	+	yes	no	*	*	pc+4	pc	nop	ALU
BEQZ _{z=1}	yes	sE ₁₆	*	0?	no	no	*	*	br	npc	nop	*
BEQZ _{z=0}	no	sE ₁₆	*	0?	no	no	*	*	pc+4	npc	mem	pc
J	yes	*	*	*	no	no	*	*	jabs	npc	nop	*
JAL	yes	*	*	*	no	yes	PC	R31	jabs	npc	nop	*
JR	yes	*	*	*	no	no	*	*	rind	npc	nop	*
JALR	yes	*	*	*	no	yes	PC	R31	rind	npc	nop	*
NOP	no	*	*	*	no	no	*	*	pc+4	npc	mem	pc

BSrc = Reg / Imm ; WBSrc = ALU / Mem / PC; IRSrc = nop/mem; MAddSrc = pc/ALU

RegDst = rt / rd / R31; PCSrc1 = pc+4 / br / rind / jabs; PCSrc2 = pc/nPC

stall & IRSrc columns are identical

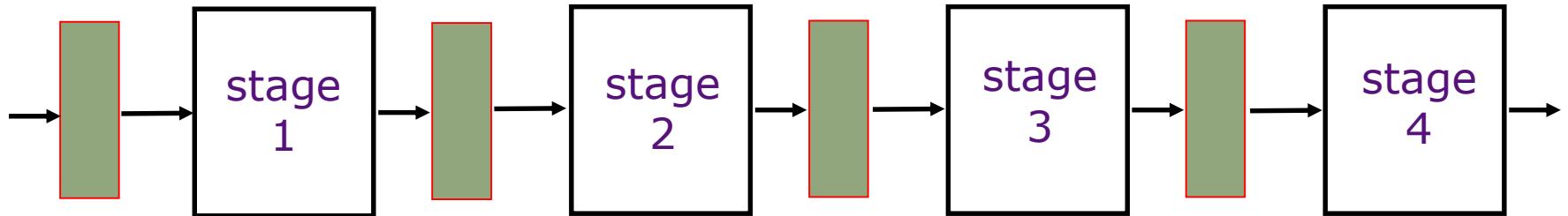
Pipelined Princeton Architecture

Clock: $t_{C-Princeton} > t_{RF} + t_{ALU} + t_M + t_{WB}$

CPI: $(1 - f) + 2f$ cycles per instruction
where f is the fraction of
instructions that cause a stall

What is a likely value of f ?

An Ideal Pipeline

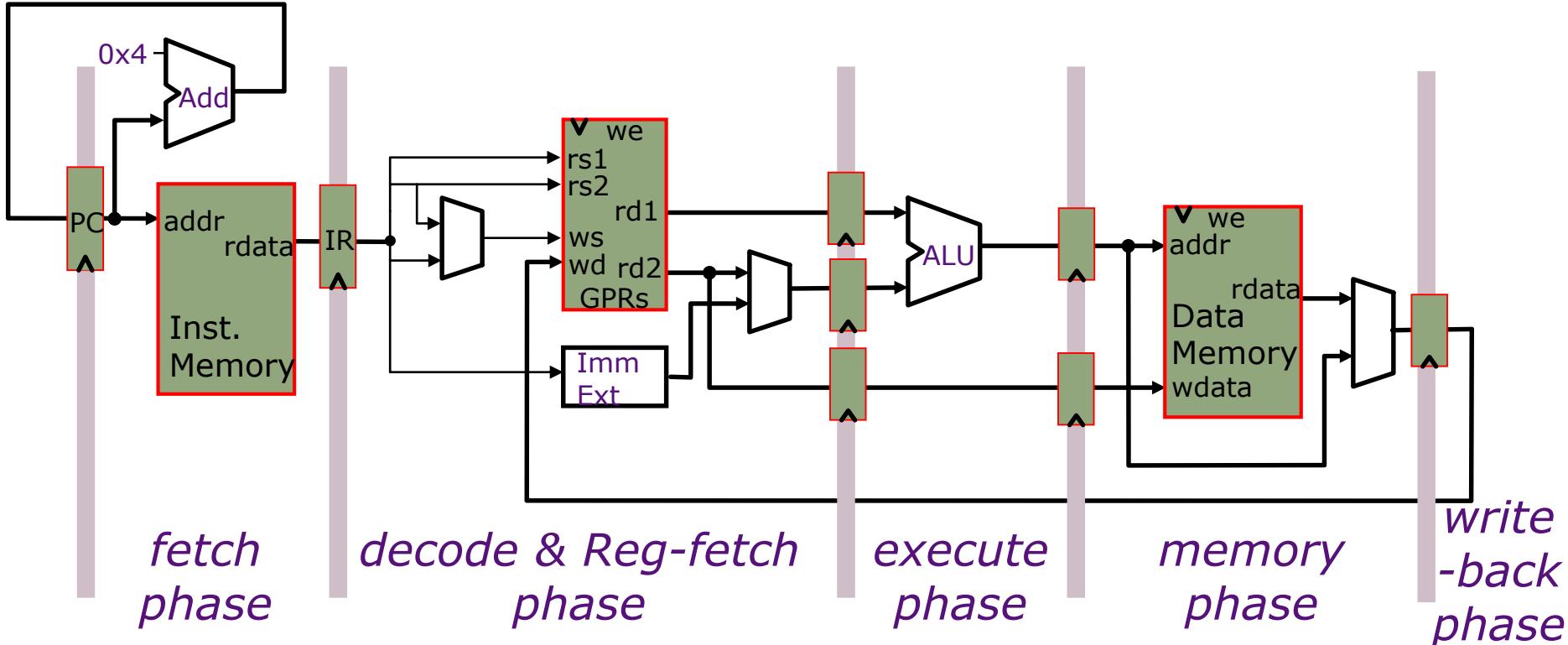


- All objects go through the same stages
- No sharing of resources between any two stages
- Propagation delay through all pipeline stages is equal
- The scheduling of an object entering the pipeline is not affected by the objects in other stages

These conditions generally hold for industrial assembly lines.

But what about an instruction pipeline?

Pipelined Datapath



Clock period can be reduced by dividing the execution of an instruction into multiple cycles

$$t_C > \max \{t_{IM}, t_{RF}, t_{ALU}, t_{DM}, t_{RW}\} \quad (= t_{DM} \text{ probably})$$

However, CPI will increase unless instructions are pipelined

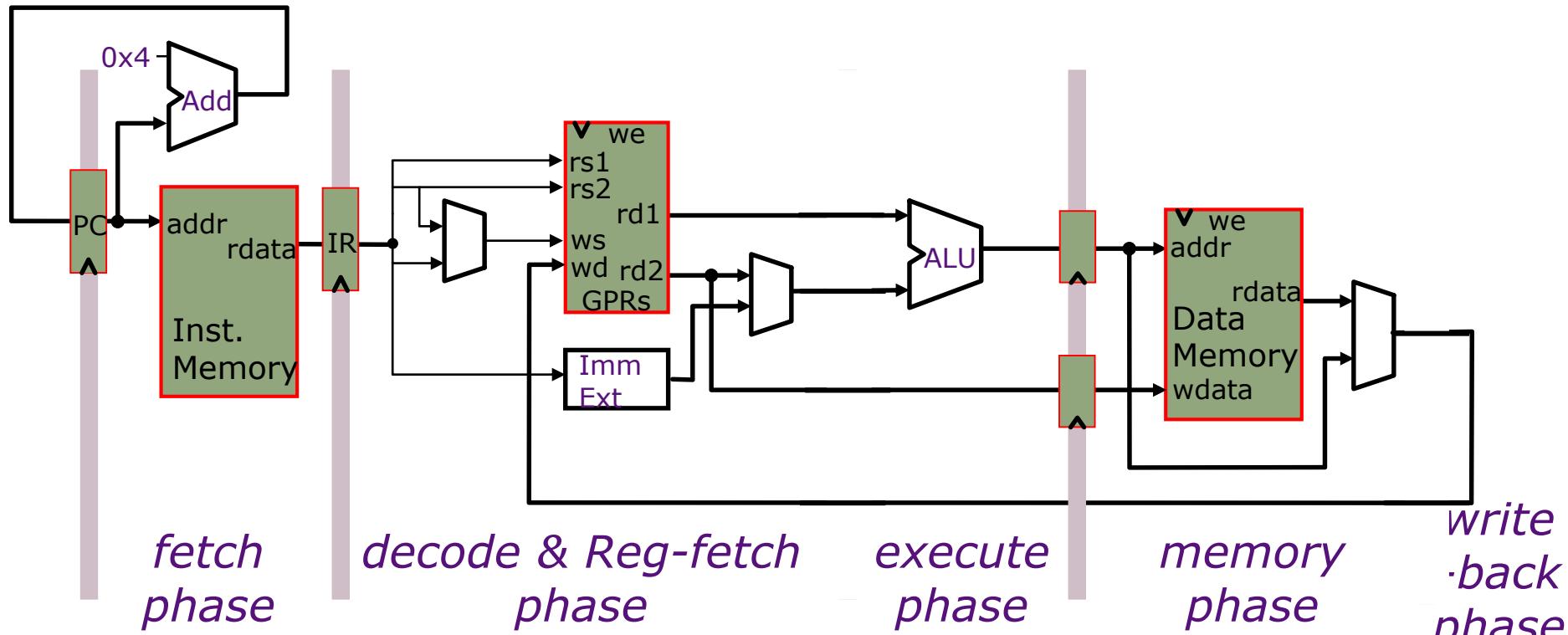
How to divide datapath into stages

Suppose memory is significantly slower than other stages. For example, suppose

$$\begin{aligned}t_{IM} &= 10 \text{ units} \\t_{DM} &= 10 \text{ units} \\t_{ALU} &= 5 \text{ units} \\t_{RF} &= 1 \text{ unit} \\t_{RW} &= 1 \text{ unit}\end{aligned}$$

Since the slowest stage determines the clock, it may be possible to combine some stages without any loss of performance

Alternative Pipelining



$$t_C > \max \{ t_{IM}, t_{RF} + t_{ALU}, t_{DM} + t_{RW} \} = t_{DM} + t_{RW}$$

Write-back stage takes much less time than other stages.
Suppose we combined it with the memory phase

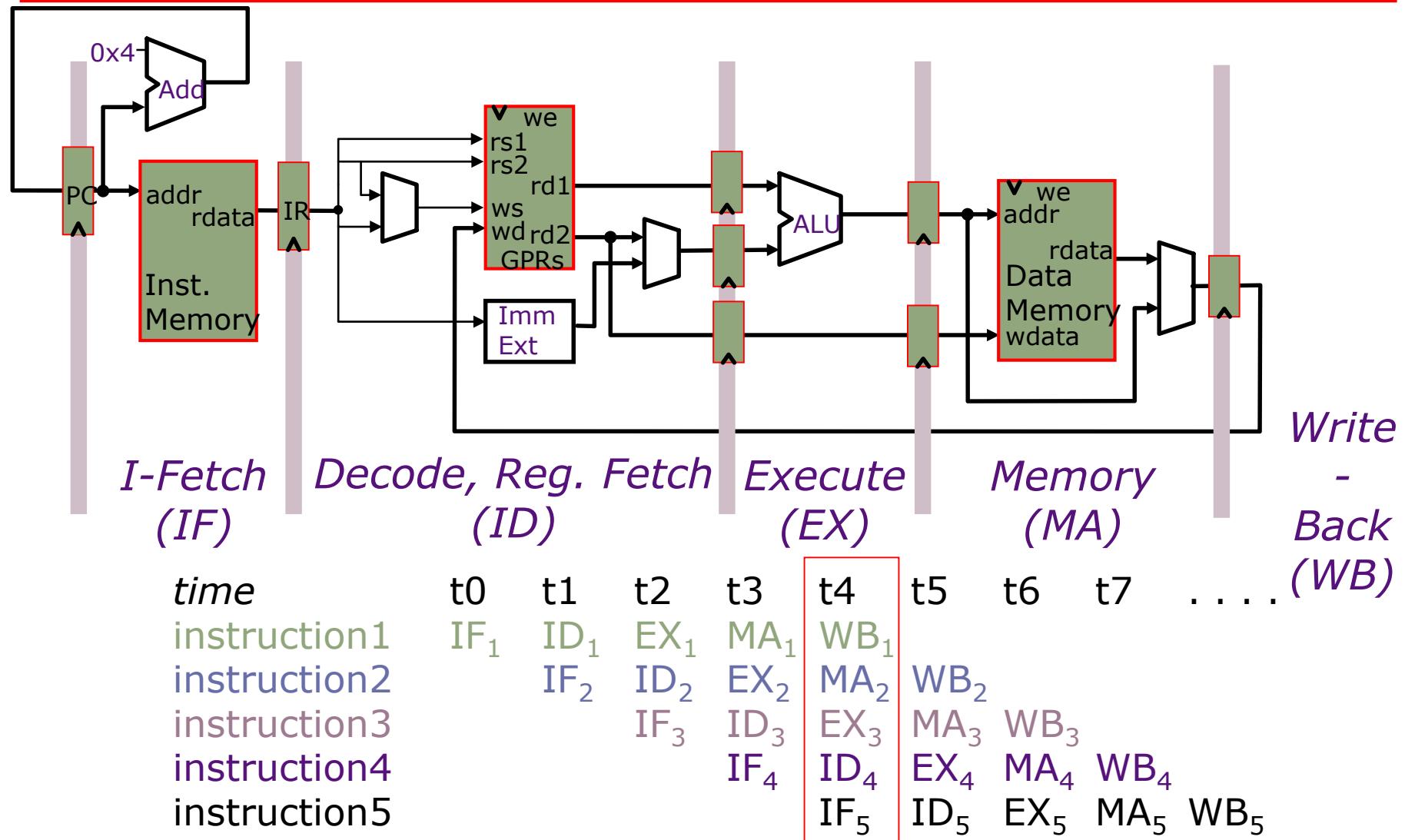
Maximum Speedup by Pipelining

Assumptions	Unpipelined t_C	Pipelined t_C	Speedup
1. $t_{IM} = t_{DM} = 10,$ $t_{ALU} = 5,$ $t_{RF} = t_{RW} = 1$ 4-stage pipeline			
2. $t_{IM} = t_{DM} = t_{ALU} = t_{RF} = t_{RW} = 5$ 4-stage pipeline			
3. $t_{IM} = t_{DM} = t_{ALU} = t_{RF} = t_{RW} = 5$ 5-stage pipeline			

What seems to be the message here?

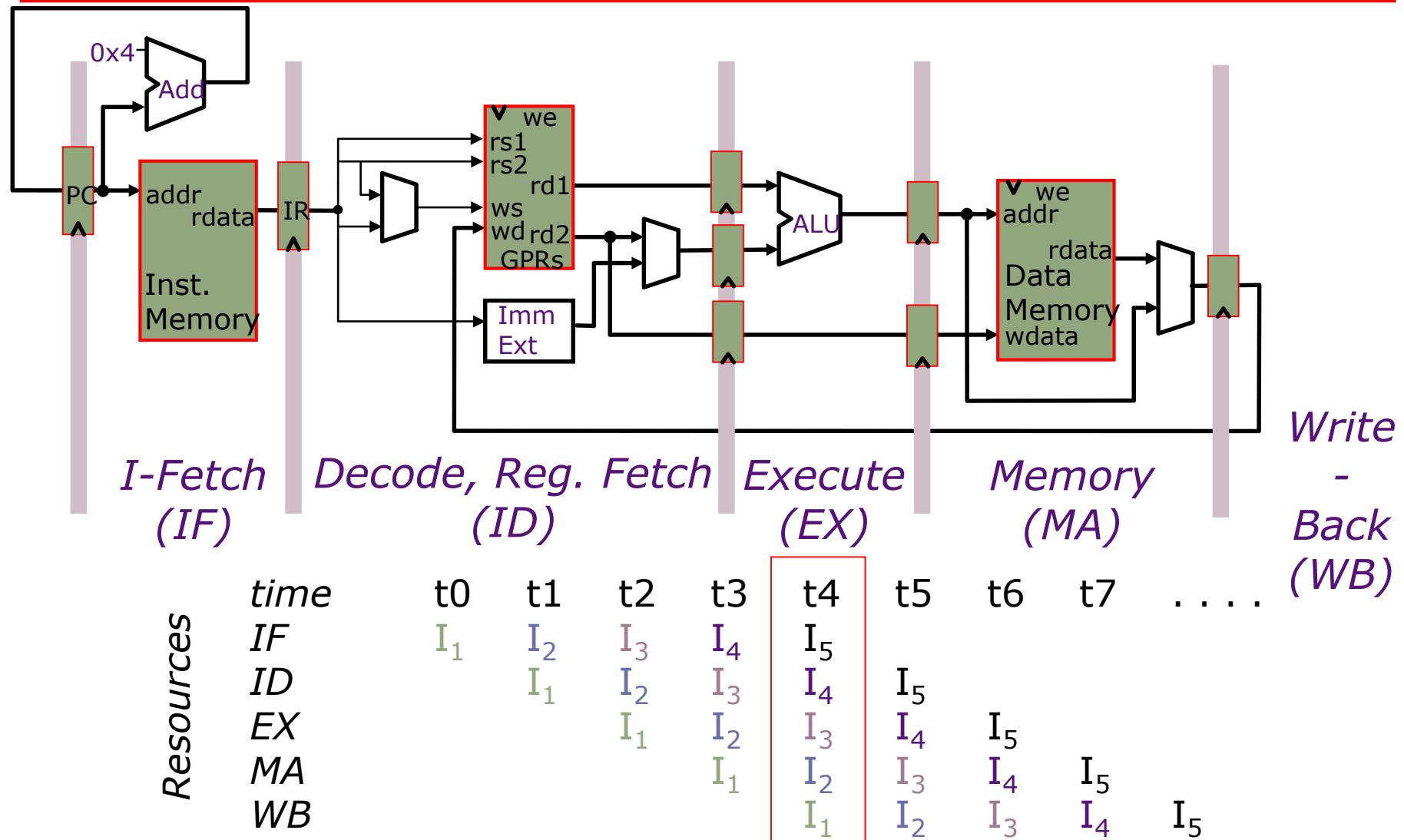
5-Stage Pipelined Execution

Instruction Flow Diagram



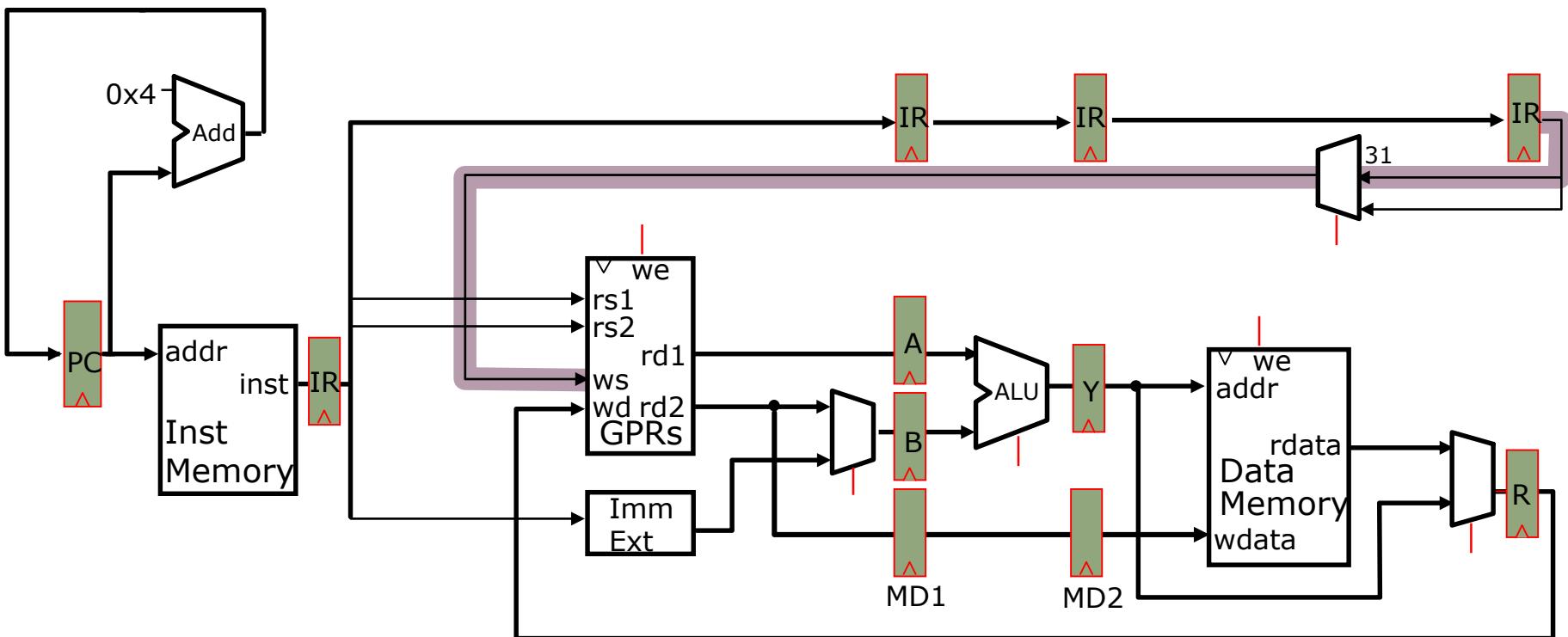
5-Stage Pipelined Execution

Resource Usage Diagram



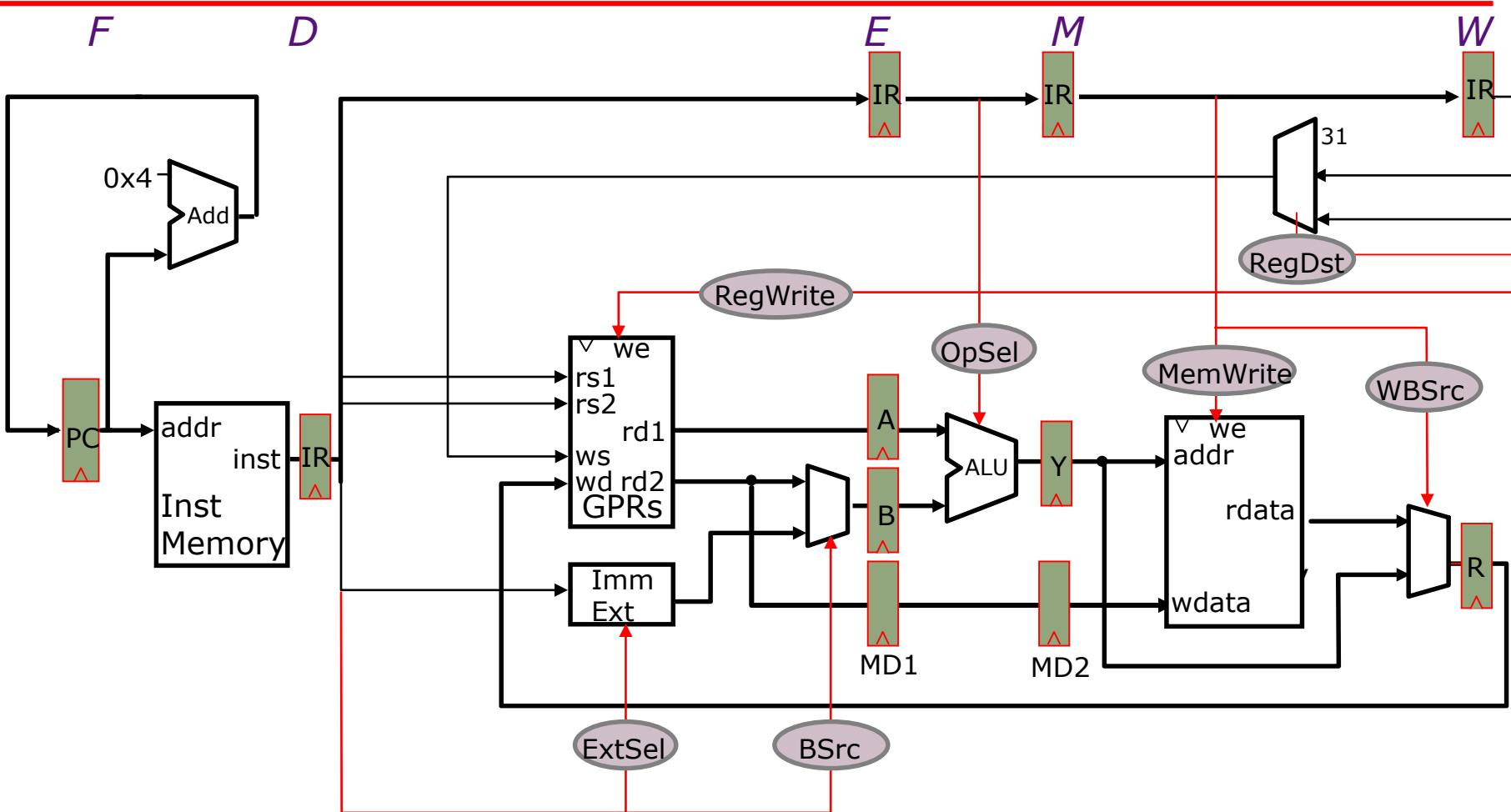
Pipelined Execution

ALU Instructions



Not quite correct!

Pipelined MIPS Datapath *without jumps*

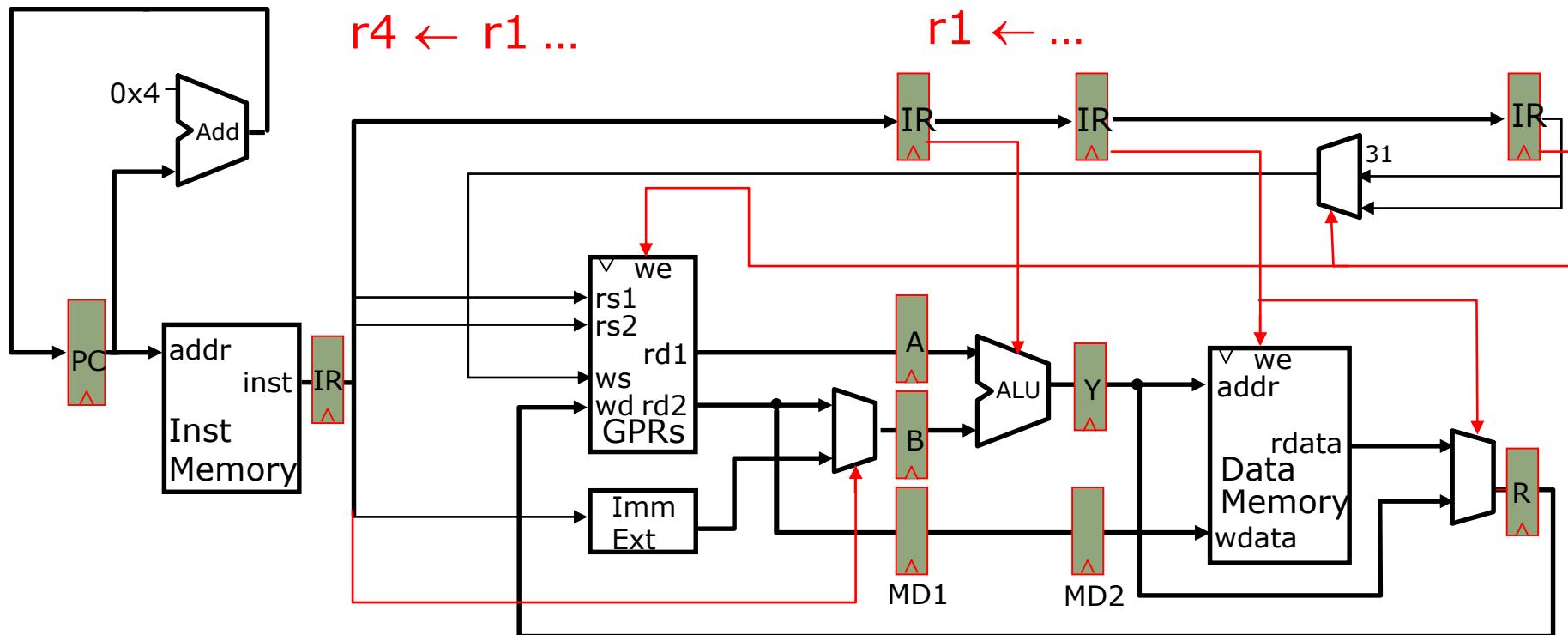


What else is needed?

How instructions can interact with each other in a pipeline

- An instruction in the pipeline may need a resource being used by another instruction in the pipeline
→ *structural hazard*
- An instruction may depend on a value produced by an earlier instruction
 - Dependence may be for a data calculation
→ *data hazard*
 - Dependence may be for calculating the next PC
→ *control hazard (branches, interrupts)*

Data Hazards



...
 $r1 \leftarrow r0 + 10$
 $r4 \leftarrow r1 + 17$
...

r1 is stale. Oops!

Resolving Data Hazards

Strategy 1: *Wait for the result to be available by freezing earlier pipeline stages → stall*

Strategy 2: *Route data as soon as possible after it is calculated to the earlier pipeline stage → bypass*

Strategy 3: *Speculate on the dependence*

Two cases:

Guessed correctly → do nothing

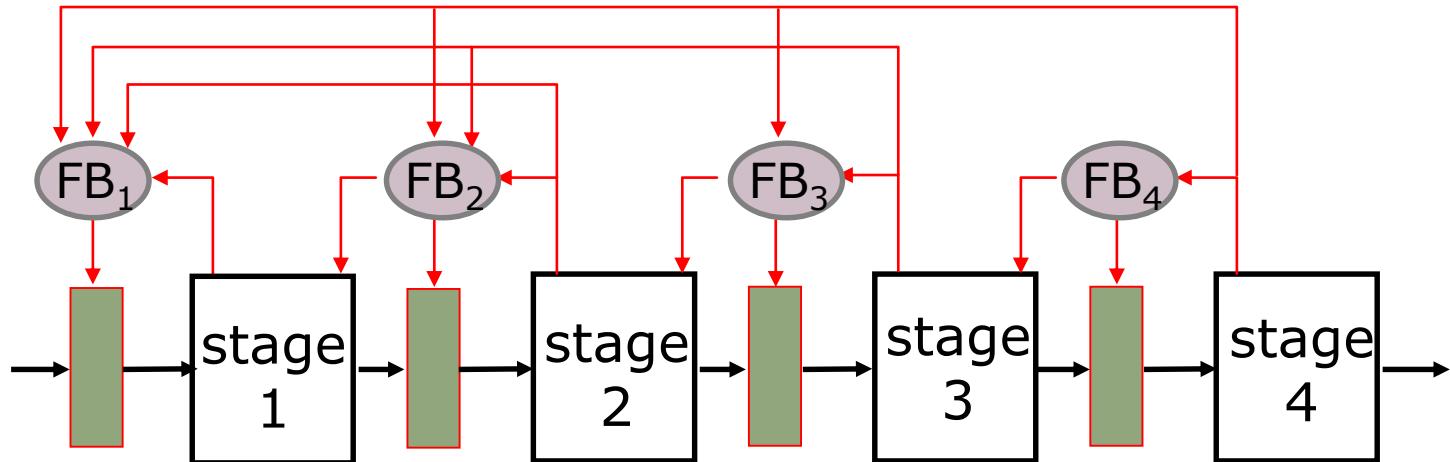
Guessed incorrectly → kill and restart

Resolving Data Hazards (1)

Strategy 1:

*Wait for the result to be available by freezing
earlier pipeline stages → **stall***

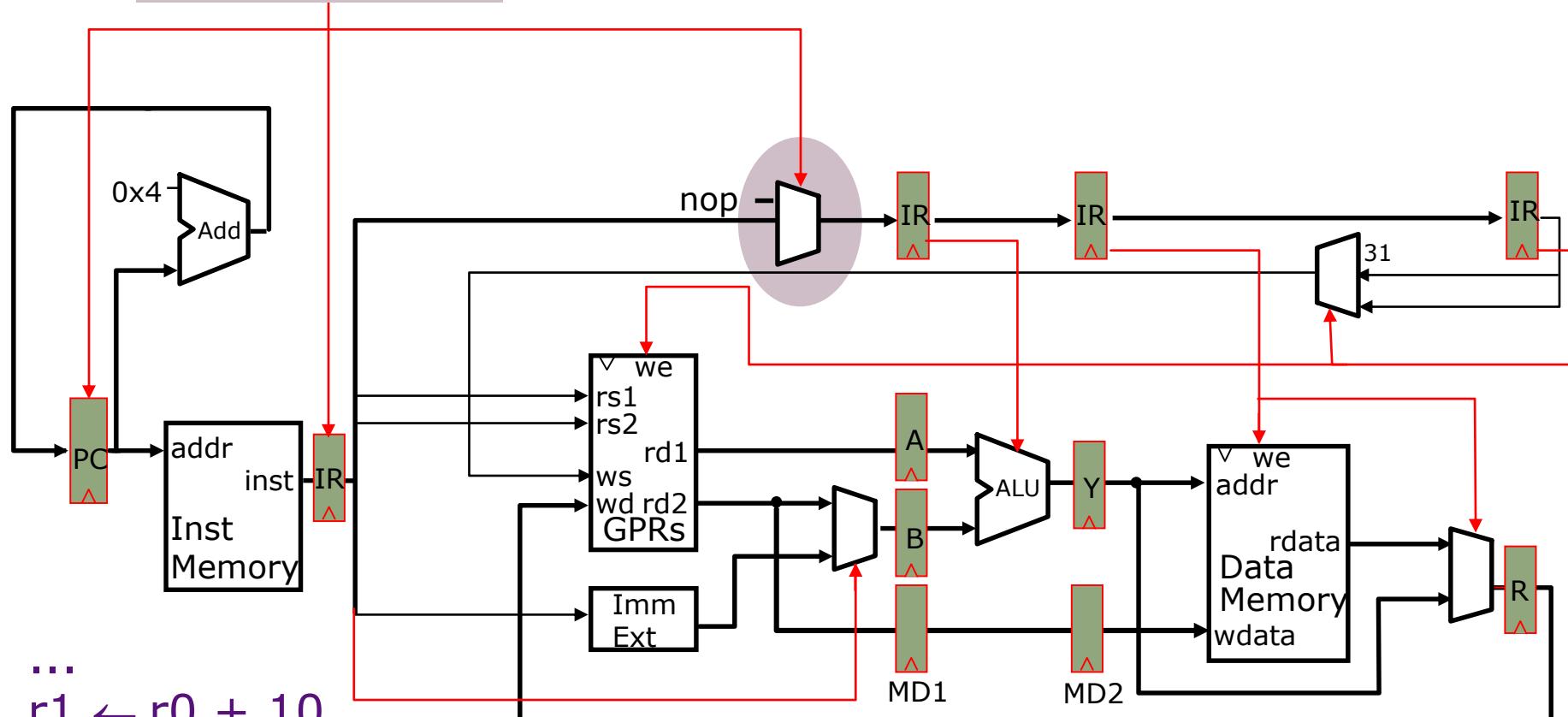
Feedback to Resolve Hazards



- Later stages provide dependence information to earlier stages which can *stall (or kill) instructions*
- Controlling a pipeline in this manner works provided *the instruction at stage $i+1$ can complete without any interference from instructions in stages 1 to i* (otherwise deadlocks may occur)

Resolving Data Hazards by Stalling

Stall Condition



Stalled Stages and Pipeline Bubbles

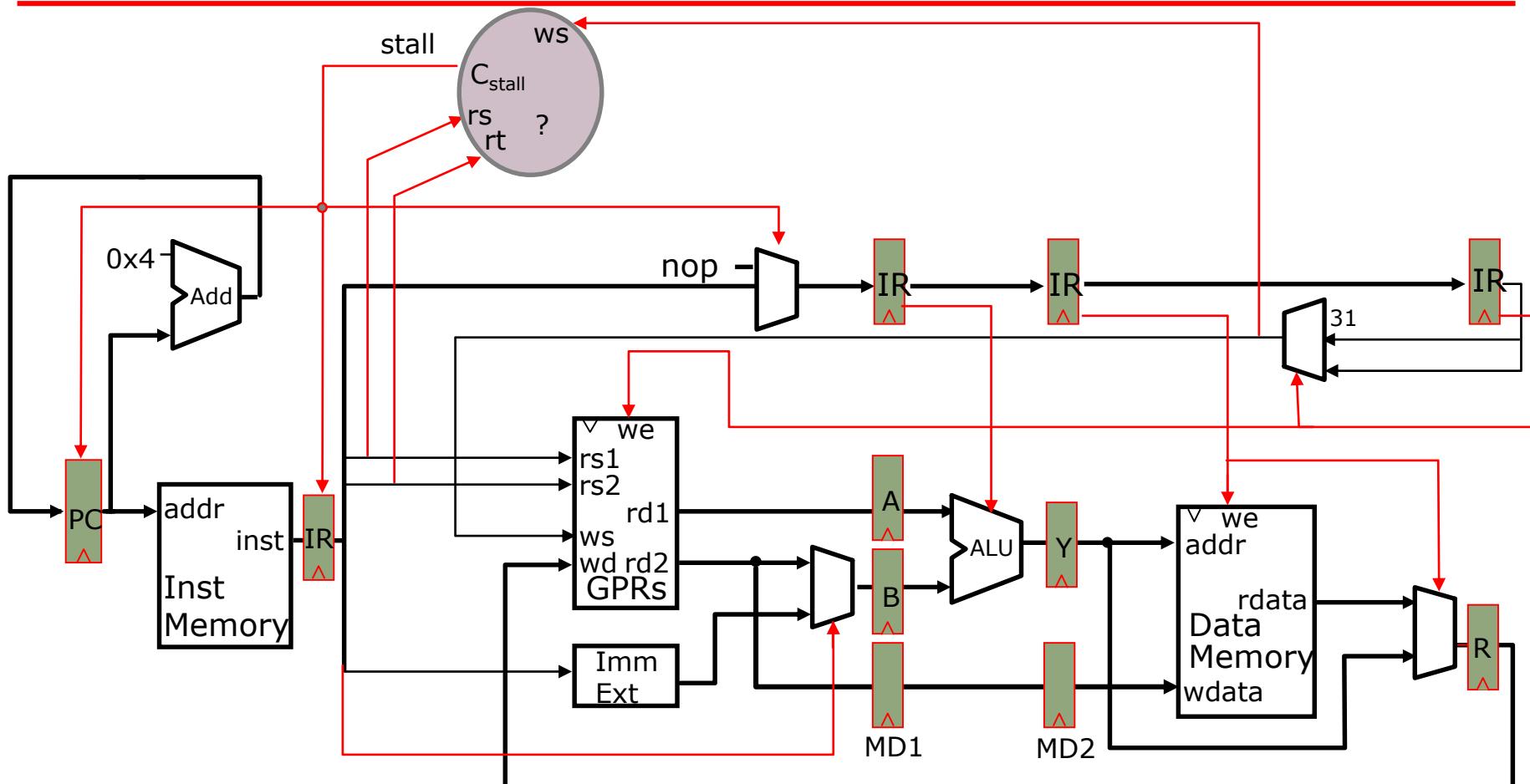
	time												
	t0	t1	t2	t3	t4	t5	t6	t7	...				
(I ₁) $r1 \leftarrow (r0) + 10$		IF ₁	ID ₁	EX ₁	MA ₁	WB ₁							
(I ₂) $r4 \leftarrow (r1) + 17$			IF ₂	ID ₂	ID ₂	ID ₂	ID ₂	EX ₂	MA ₂	WB ₂			
(I ₃)				IF ₃	IF ₃	IF ₃	IF ₃	ID ₃	EX ₃	MA ₃	WB ₃		
(I ₄)								IF ₄	ID ₄	EX ₄	MA ₄	WB ₄	
(I ₅)									IF ₅	ID ₅	EX ₅	MA ₅	WB ₅

stalled stages

	time									
	t0	t1	t2	t3	t4	t5	t6	t7	...	
Resource Usage	IF	I ₁	I ₂	I ₃	I ₃	I ₃	I ₃	I ₄	I ₅	
	ID		I ₁	I ₂	I ₂	I ₂	I ₂	I ₃	I ₄	I ₅
	EX			I ₁	nop	nop	nop	I ₂	I ₃	I ₄
	MA				I ₁	nop	nop	nop	I ₂	I ₃
	WB					I ₁	nop	nop	nop	I ₂

nop \Rightarrow *pipeline bubble*

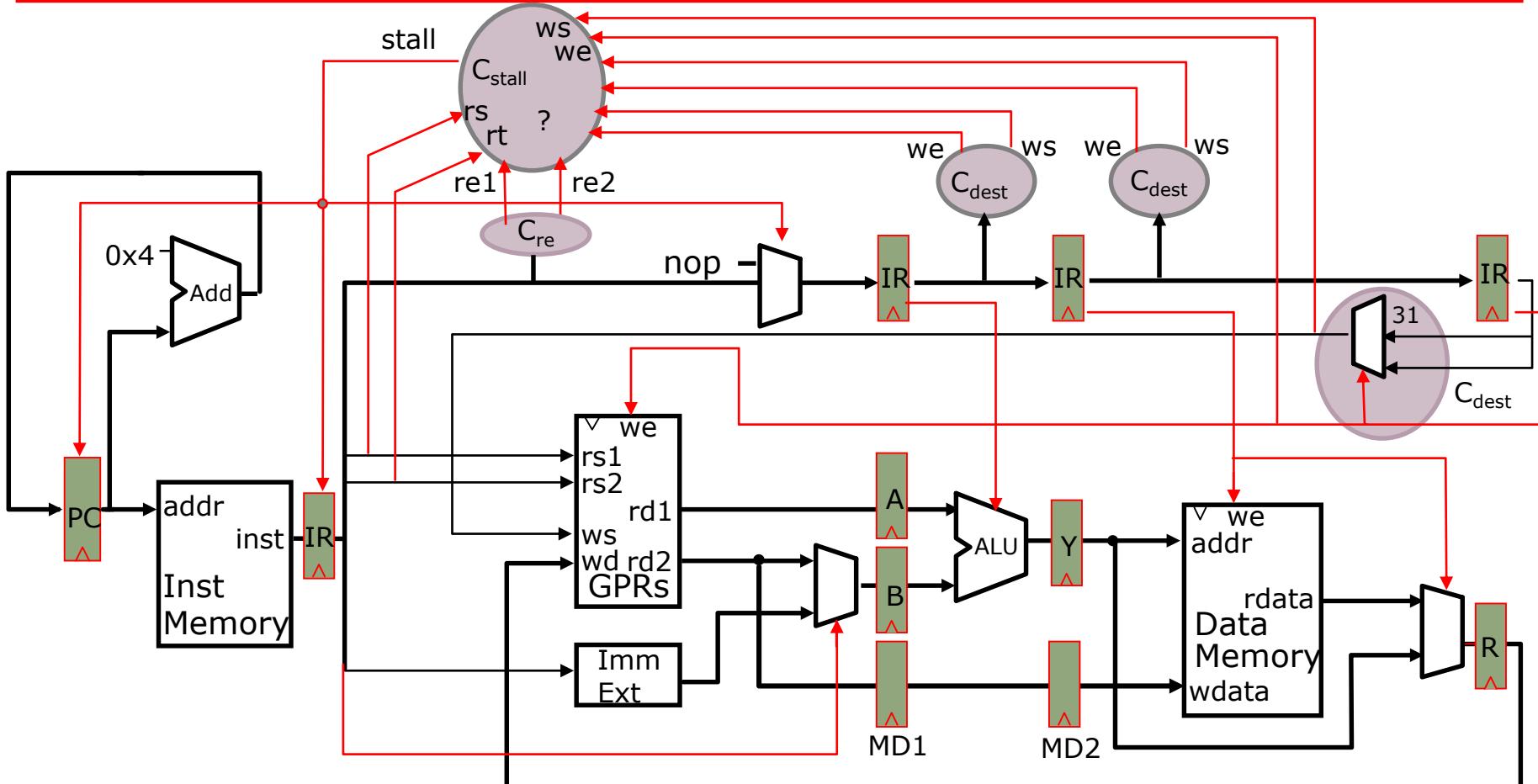
Stall Control Logic



Compare the *source registers* of the instruction in the decode stage with the *destination register* of the *uncommitted instructions*.

Stall Control Logic

ignoring jumps & branches



Should we always stall if the rs field matches some rd?

Source & Destination Registers

R-type:



I-type:



J-type:



		<i>source(s)</i>	<i>destination</i>
ALU	$rd \leftarrow (rs) \text{ func } (rt)$	rs, rt	rd
ALUi	$rt \leftarrow (rs) \text{ op imm}$	rs	rt
LW	$rt \leftarrow M [(rs) + \text{imm}]$	rs	rt
SW	$M [(rs) + \text{imm}] \leftarrow (rt)$	rs, rt	
BZ	<i>cond</i> (rs) <i>true</i> : $PC \leftarrow (PC) + \text{imm}$ <i>false</i> : $PC \leftarrow (PC) + 4$	rs rs	
J	$PC \leftarrow (PC) + \text{imm}$		
JAL	$r31 \leftarrow (PC), PC \leftarrow (PC) + \text{imm}$		31
JR	$PC \leftarrow (rs)$	rs	
JALR	$r31 \leftarrow (PC), PC \leftarrow (rs)$	rs	31

Deriving the Stall Signal

C_{dest}

$ws = Case$ opcode	
ALU	$\Rightarrow rd$
ALUi, LW	$\Rightarrow rt$
JAL, JALR	$\Rightarrow R31$

$we = Case$ opcode

ALU, ALUi, LW	$\Rightarrow (ws \neq 0)$
JAL, JALR	$\Rightarrow on$
...	$\Rightarrow off$

C_{re}

$re1 = Case$ opcode	
ALU, ALUi,	$\Rightarrow on$
LW, SW, BZ,	
JR, JALR	
J, JAL	$\Rightarrow off$

$re2 = Case$ opcode

ALU, SW	$\Rightarrow on$
...	$\Rightarrow off$

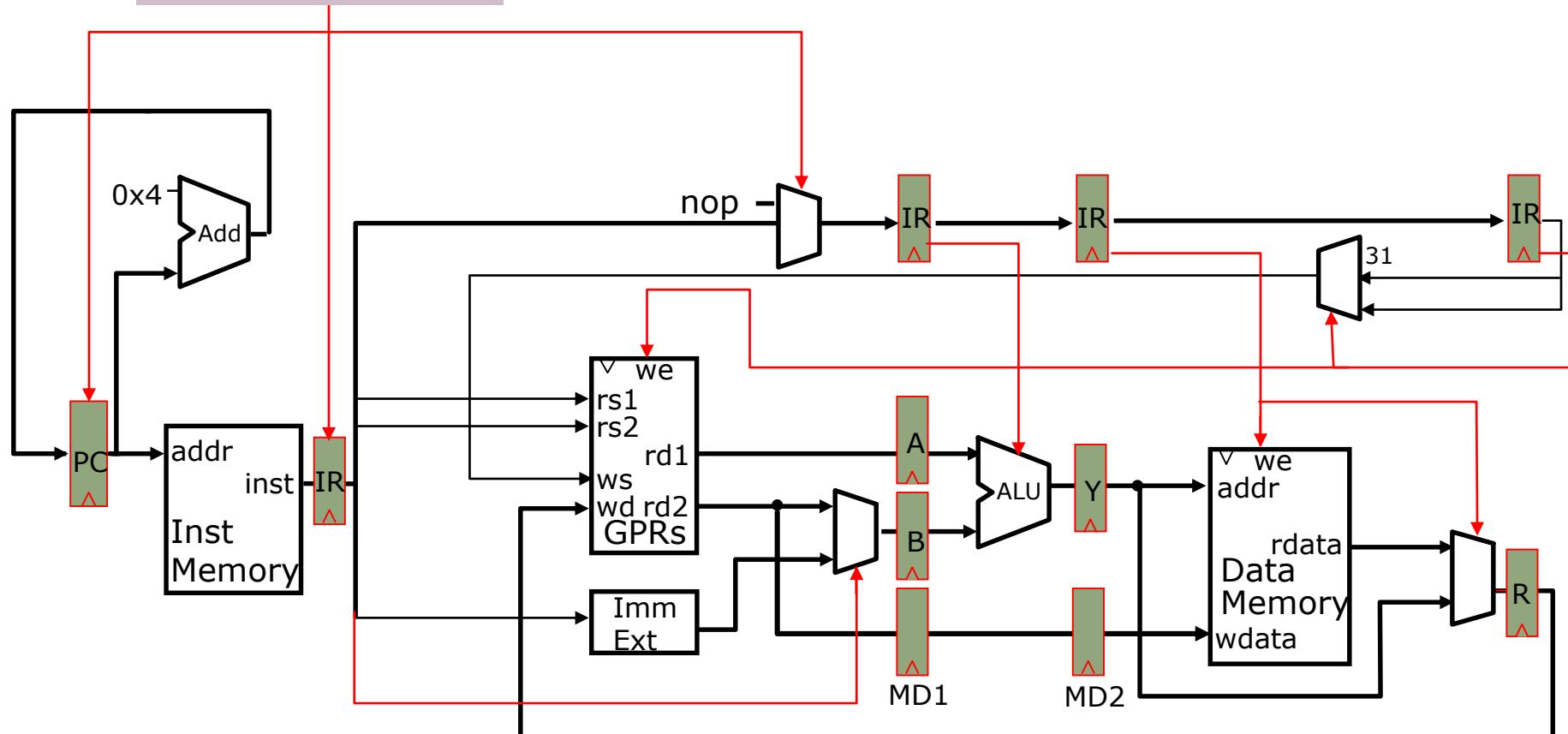
C_{stall}

$$\begin{aligned} stall = & ((rs_D == ws_E) \cdot we_E + \\ & (rs_D == ws_M) \cdot we_M + \\ & (rs_D == ws_W) \cdot we_W) \cdot re1_D + \\ & ((rt_D == ws_E) \cdot we_E + \\ & (rt_D == ws_M) \cdot we_M + \\ & (rt_D == ws_W) \cdot we_W) \cdot re2_D \end{aligned}$$

This is not
the full story !

Hazards due to Loads & Stores

Stall Condition



...
 $M[(r1)+7] \leftarrow (r2)$
 $r4 \leftarrow M[(r3)+5]$

*Is there any possible data hazard
in this instruction sequence?*

Load & Store Hazards

```
...  
M[(r1)+7] ← (r2)  
r4 ← M[(r3)+5]  
...
```

$(r1)+7 = (r3)+5 \Rightarrow \text{data hazard}$

However, the hazard is avoided because *our memory system completes writes in one cycle!*

Load/Store hazards are sometimes resolved in the pipeline and sometimes in the memory system itself.

More on this later in the course.

Next lecture:
Control Hazards,
Bypassing,
and Speculation

Instruction Pipelining: Hazard Resolution, Timing Constraints

Daniel Sanchez

Computer Science and Artificial Intelligence Laboratory
M.I.T.

Resolving Data Hazards

Strategy 1: *Wait for the result to be available by freezing earlier pipeline stages → stall*

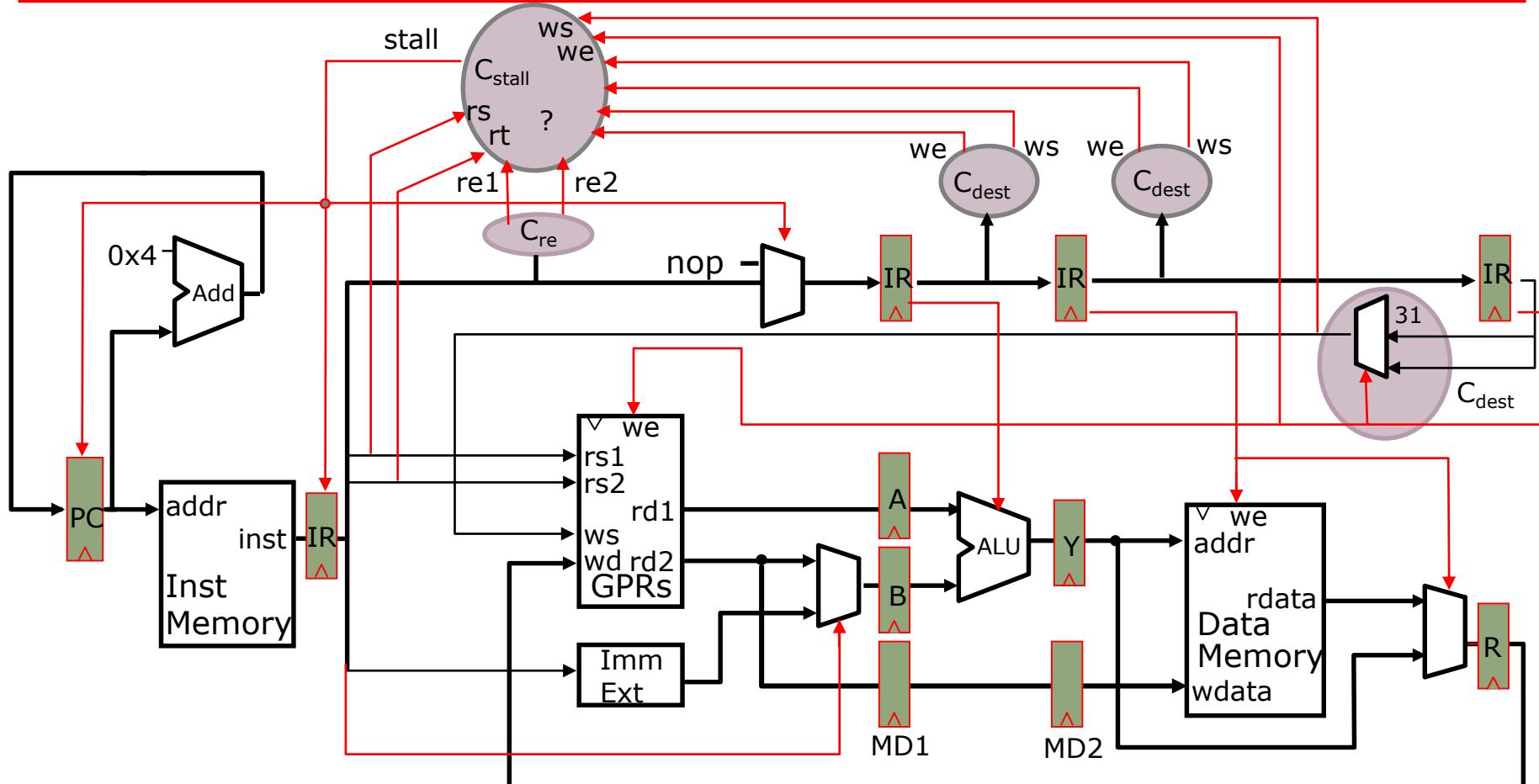
Strategy 2: *Route data as soon as possible after it is calculated to the earlier pipeline stage → bypass*

Strategy 3: *Speculate on the dependence*
Two cases:

Guessed correctly → no special action required
Guessed incorrectly → kill and restart

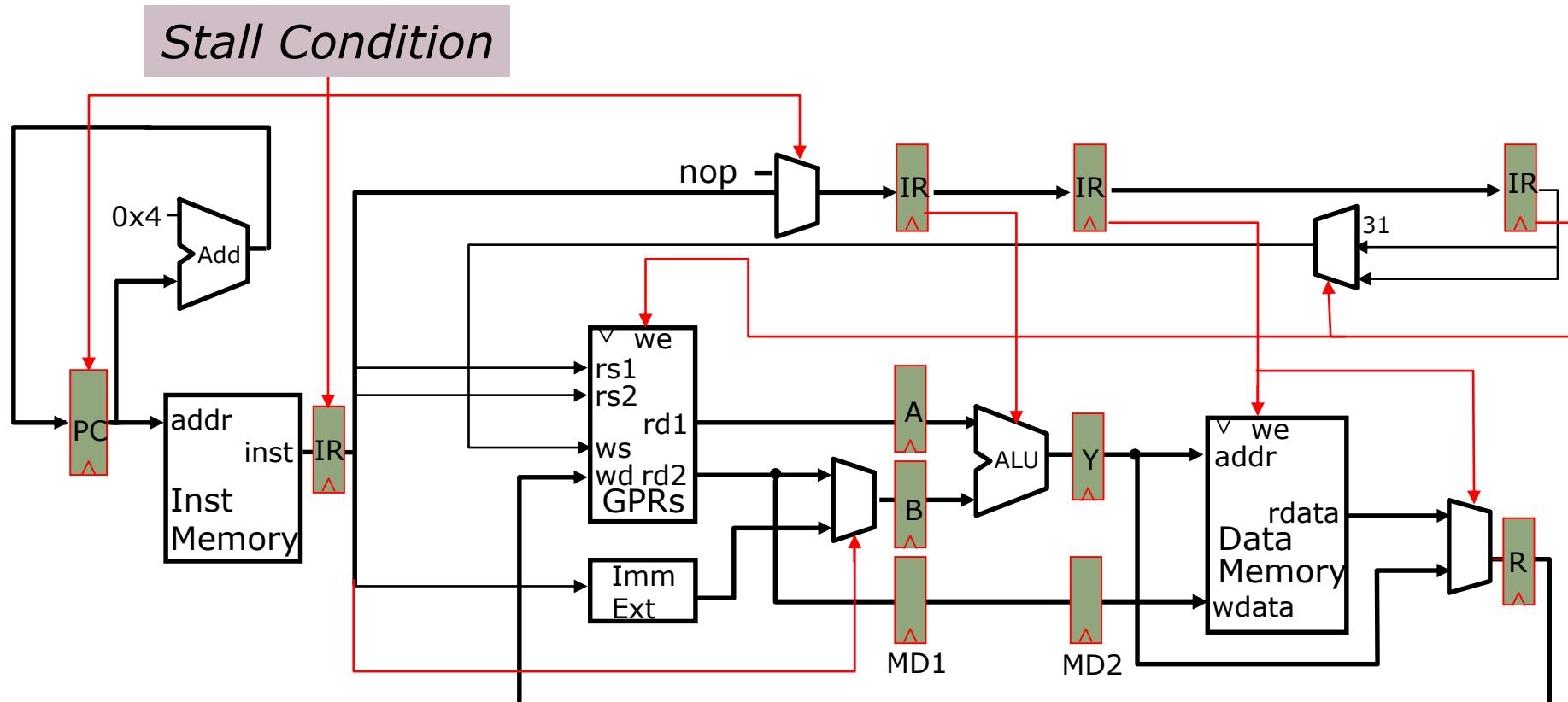
Reminder: Stall Control Logic

ignoring jumps & branches



Stall DEC & IF when instruction in DEC reads a register that is written by any earlier in-flight instruction (in EXE, MEM, or WB)

Reminder: Load & Store Hazards



...
 $M[(r1)+7] \leftarrow (r2)$
 $r4 \leftarrow M[(r3)+5]$
...

$(r1)+7 = (r3)+5 \Rightarrow \text{data hazard}$

These hazards do not need pipeline changes because
our memory system completes writes in one cycle

Resolving Data Hazards (2)

Strategy 2:

Route data as soon as possible after it is calculated to the earlier pipeline stage → *bypass*

Bypassing

	<i>time</i>	t0	t1	t2	t3	t4	t5	t6	t7	...
(I ₁)	$r1 \leftarrow r0 + 10$	IF ₁	ID ₁	EX ₁	MA ₁	WB ₁				
(I ₂)	$r4 \leftarrow r1 + 17$		IF ₂	ID ₂	ID ₂	ID ₂	ID ₂	EX ₂	MA ₂	WB ₂
(I ₃)				IF ₃	IF ₃	IF ₃	IF ₃	ID ₃	EX ₃	MA ₃
(I ₄)								IF ₄	ID ₄	EX ₄
(I ₅)								IF ₅	ID ₅	

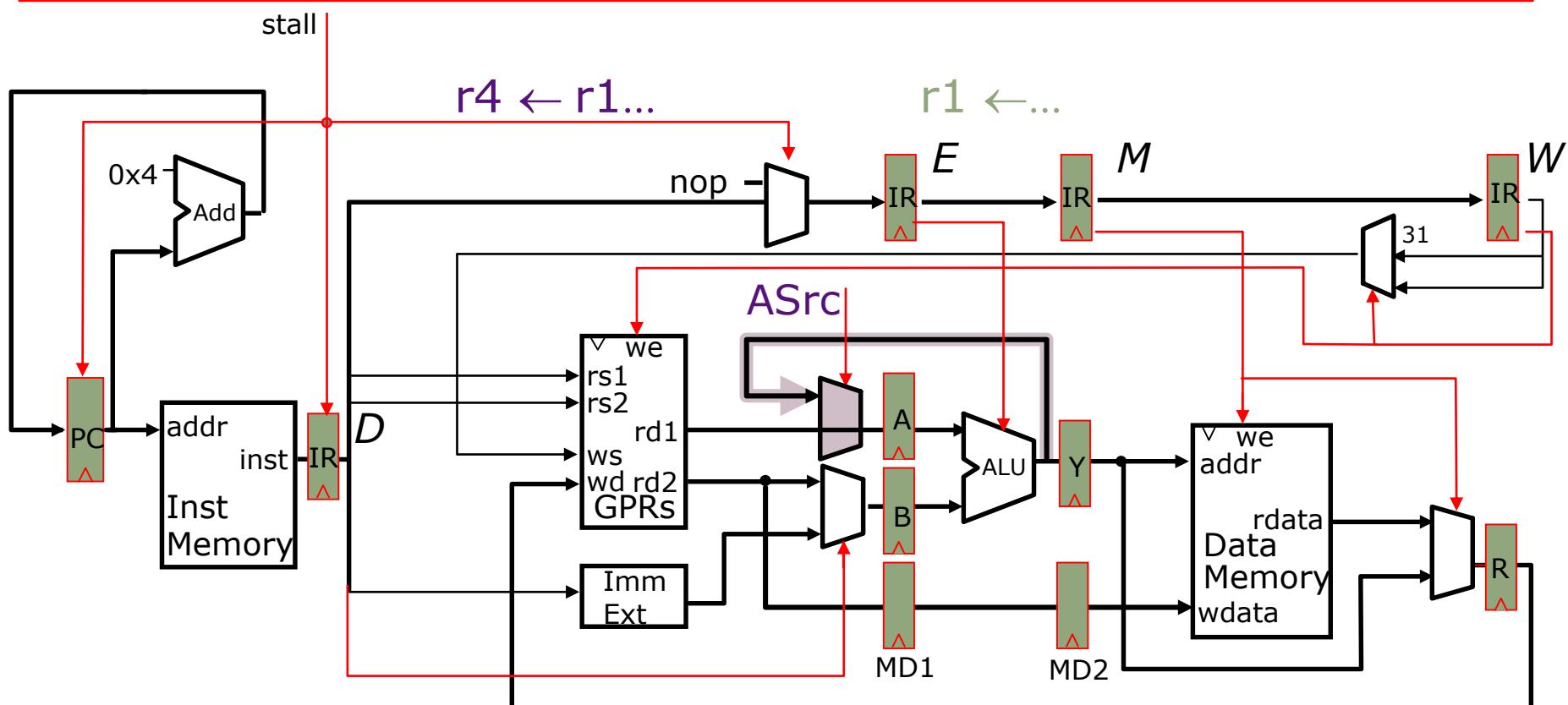
Each *stall* or *kill* introduces a bubble $\Rightarrow CPI > 1$

When is data actually available?

	<i>time</i>	t0	t1	t2	t3	t4	t5	t6	t7	...
(I ₁)	$r1 \leftarrow r0 + 10$	IF ₁	ID ₁	EX ₁	MA ₁	WB ₁				
(I ₂)	$r4 \leftarrow r1 + 17$		IF ₂	ID ₂	EX ₂	MA ₂	WB ₂			
(I ₃)				IF ₃	ID ₃	EX ₃	MA ₃	WB ₃		
(I ₄)					IF ₄	ID ₄	EX ₄	MA ₄	WB ₄	
(I ₅)						IF ₅	ID ₅	EX ₅	MA ₅	WB ₅

A new datapath, i.e., a *bypass*, can get the data from the output of the ALU to its input

Adding a Bypass



When does *this* bypass help?

...
 $r1 \leftarrow r0 + 10$
 $r4 \leftarrow r1 + 17$ *yes*

$r1 \leftarrow M[r0 + 10]$
 $r4 \leftarrow r1 + 17$ *no*

JAL 500
 $r4 \leftarrow r31 + 17$ *no*

The Bypass Signal

Deriving it from the Stall Signal

$$\begin{aligned} \text{stall} = & ((\cancel{\text{rs}_D == \text{ws}_E}) \cdot \text{we}_E + (\text{rs}_D == \text{ws}_M) \cdot \text{we}_M + (\text{rs}_D == \text{ws}_W) \cdot \text{we}_W) \cdot \text{re1}_D \\ & + ((\text{rt}_D == \text{ws}_E) \cdot \text{we}_E + (\text{rt}_D == \text{ws}_M) \cdot \text{we}_M + (\text{rt}_D == \text{ws}_W) \cdot \text{we}_W) \cdot \text{re2}_D \end{aligned}$$

ws = Case opcode

ALU	$\Rightarrow \text{rd}$
ALUi, LW	$\Rightarrow \text{rt}$
JAL, JALR	$\Rightarrow \text{R31}$

we = Case opcode

ALU, ALUi, LW	$\Rightarrow (\text{ws} \neq 0)$
JAL, JALR	$\Rightarrow \text{on}$
...	$\Rightarrow \text{off}$

$$\text{ASrc} = (\text{rs}_D == \text{ws}_E) \cdot \text{we}_E \cdot \text{re1}_D$$

Is this correct?

How might we address this?

Bypass and Stall Signals

Split we_E into two components: we-bypass, we-stall

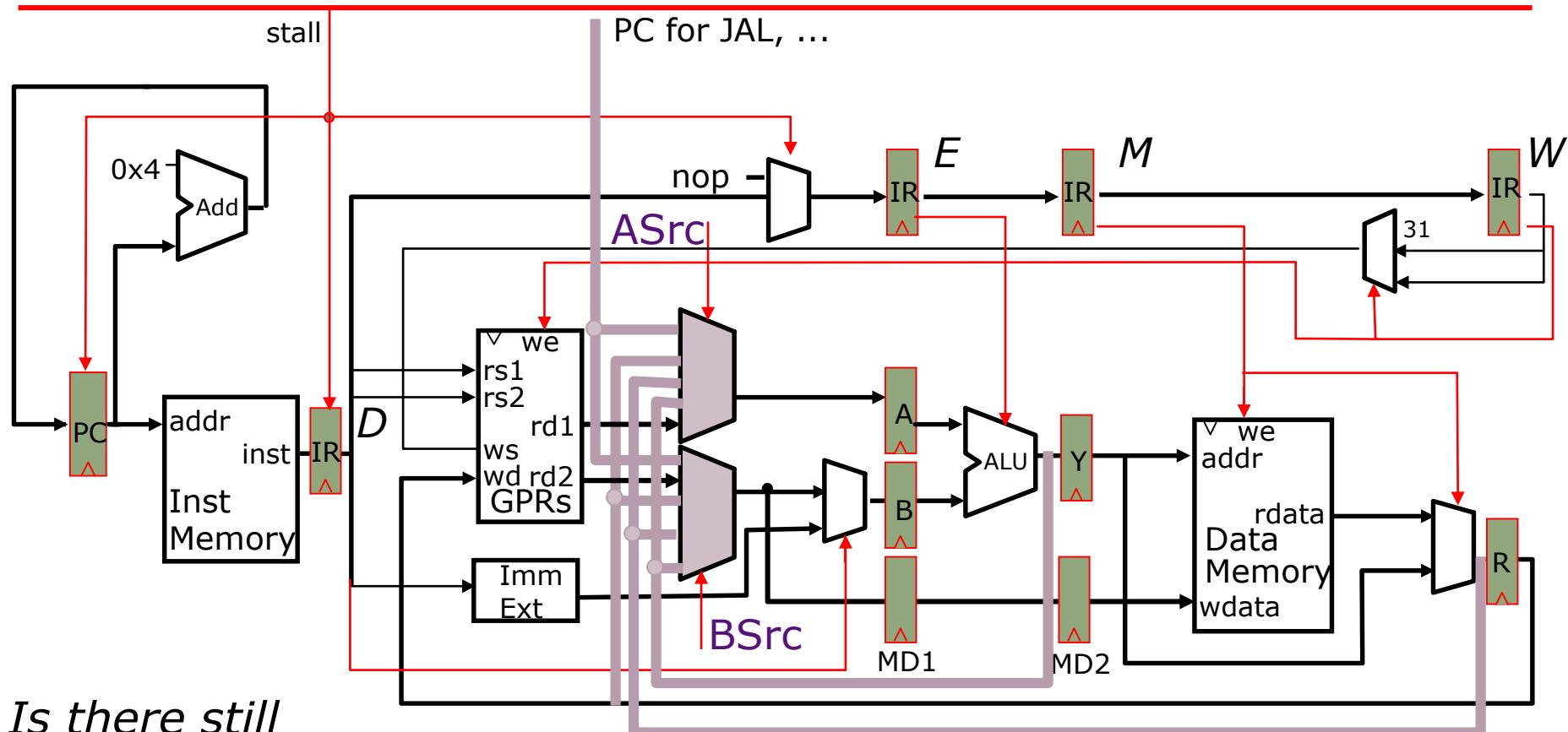
$we\text{-bypass}_E = \begin{cases} \text{Case } opcode_E \\ \text{ALU, ALUi} & \Rightarrow (ws \neq 0) \\ \dots & \Rightarrow \text{off} \end{cases}$

$we\text{-stall}_E = \begin{cases} \text{Case } opcode_E \\ LW & \Rightarrow (ws \neq 0) \\ JAL, JALR & \Rightarrow \text{on} \\ \dots & \Rightarrow \text{off} \end{cases}$

$$ASrc = (rs_D == ws_E) \cdot we\text{-bypass}_E \cdot re1_D$$

$$\begin{aligned} stall = & ((rs_D == ws_E) \cdot we\text{-stall}_E + \\ & (rs_D == ws_M) \cdot we_M + (rs_D == ws_W) \cdot we_W) \cdot re1_D \\ & + ((rt_D == ws_E) \cdot we_E + (rt_D == ws_M) \cdot we_M + (rt_D == ws_W) \cdot we_W) \cdot re2_D \end{aligned}$$

Fully Bypassed Datapath



*Is there still
a need for the
stall signal?*

Resolving Data Hazards (3)

Strategy 3:

Speculate on the dependence. Two cases:

Guessed correctly → no special action required

Guessed incorrectly → kill and restart

Instruction to Instruction Dependence

- What do we need to calculate next PC?
 - For Jumps
 - Opcode, offset, and PC
 - For Jump Register
 - Opcode and register value
 - For Conditional Branches
 - Opcode, offset, PC, and register (for condition)
 - For all others
 - Opcode and PC
- In what stage do we know these?
 - PC → Fetch
 - Opcode, offset → Decode (or Fetch?)
 - Register value → Decode
 - Branch condition ($(rs) == 0$) → Execute (or Decode?)

NextPC Calculation Bubbles

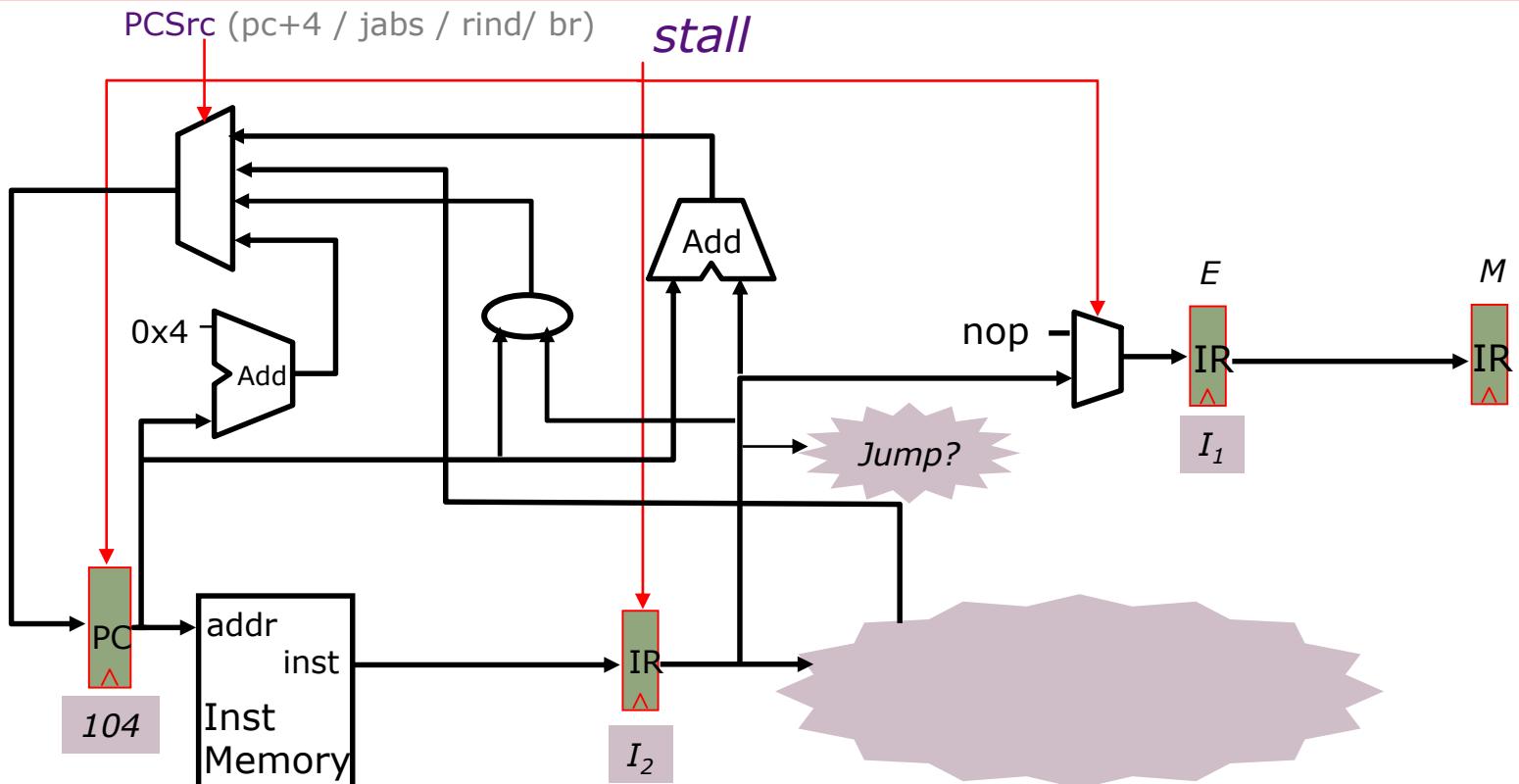
	time									
	t0	t1	t2	t3	t4	t5	t6	t7	...	
(I ₁)	r1 ← (r0) + 10	IF ₁	ID ₁	EX ₁	MA ₁	WB ₁				
(I ₂)	r3 ← (r2) + 17		IF ₂	IF ₂	ID ₂	EX ₂	MA ₂	WB ₂		
(I ₃)				IF ₃	IF ₃	ID ₃	EX ₃	MA ₃	WB ₃	
(I ₄)					IF ₄	IF ₄	ID ₄	EX ₄	MA ₄	WB ₄

	time									
	t0	t1	t2	t3	t4	t5	t6	t7	...	
Resource Usage	IF	I ₁	nop	I ₂	nop	I ₃	nop	I ₄		
	ID		I ₁	nop	I ₂	nop	I ₃	nop	I ₄	
	EX			I ₁	nop	I ₂	nop	I ₃	nop	I ₄
	MA				I ₁	nop	I ₂	nop	I ₃	nop
	WB					I ₁	nop	I ₂	nop	I ₄

nop ⇒ *pipeline bubble*

What's a good guess for next PC?

Speculate NextPC is PC+4

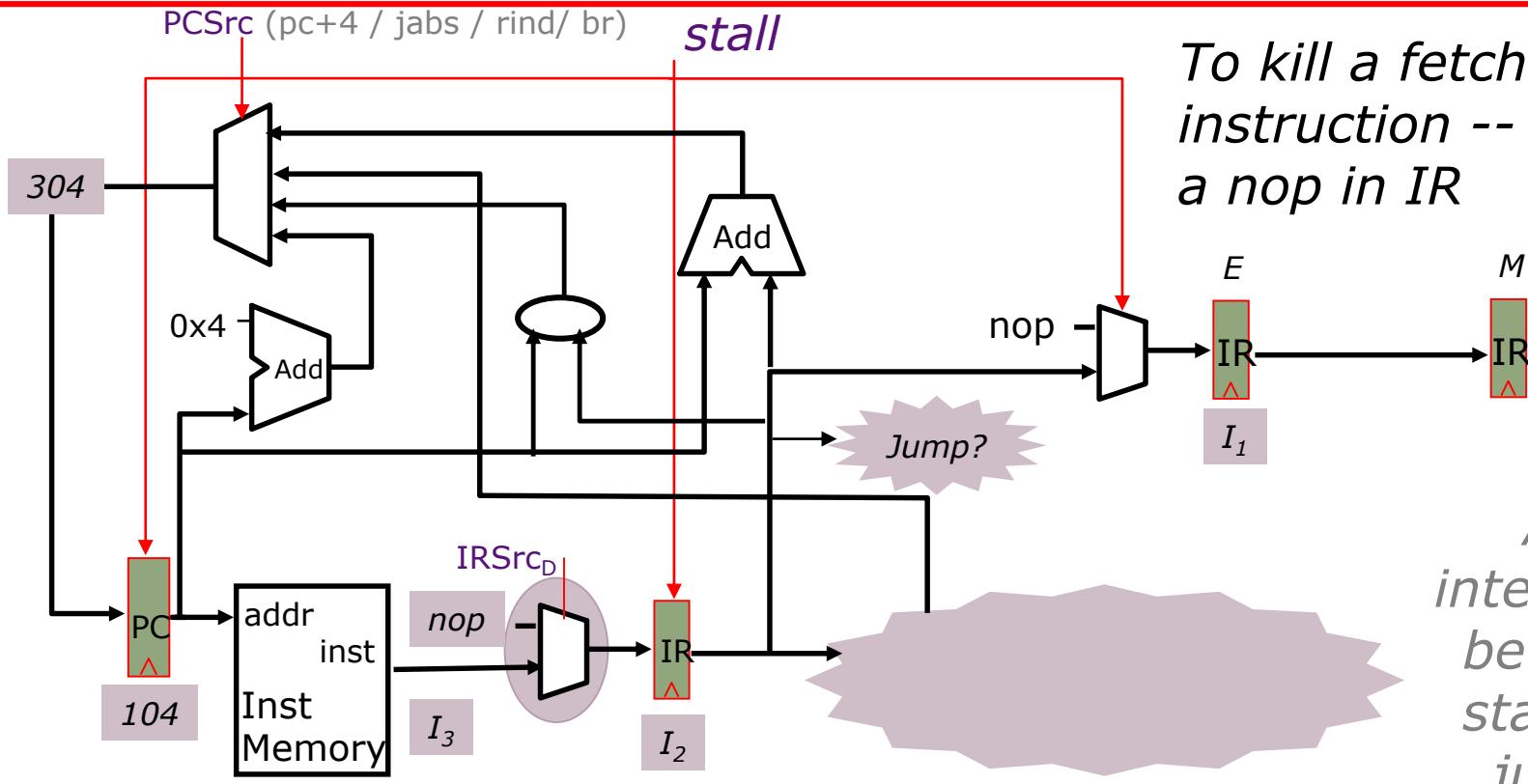


I_1	096	ADD	
I_2	100	J	200
I_3	104	ADD	<i>kill</i>
I_4	304	ADD	

What happens on mis-speculation,
i.e., when next instruction is not $PC+4$?

How?

Pipelining Jumps



To kill a fetched instruction -- Insert a nop in IR

Any interaction between stall and jump?

I_1	096	ADD	
I_2	100	J	200
I_3	104	ADD	<i>kill</i>
I_4	304	ADD	

$IRSrc_D = \begin{cases} \text{Case opcode}_D \\ J, JAL \\ \dots \end{cases} \Rightarrow \begin{cases} \text{nop} \\ \text{IM} \end{cases}$

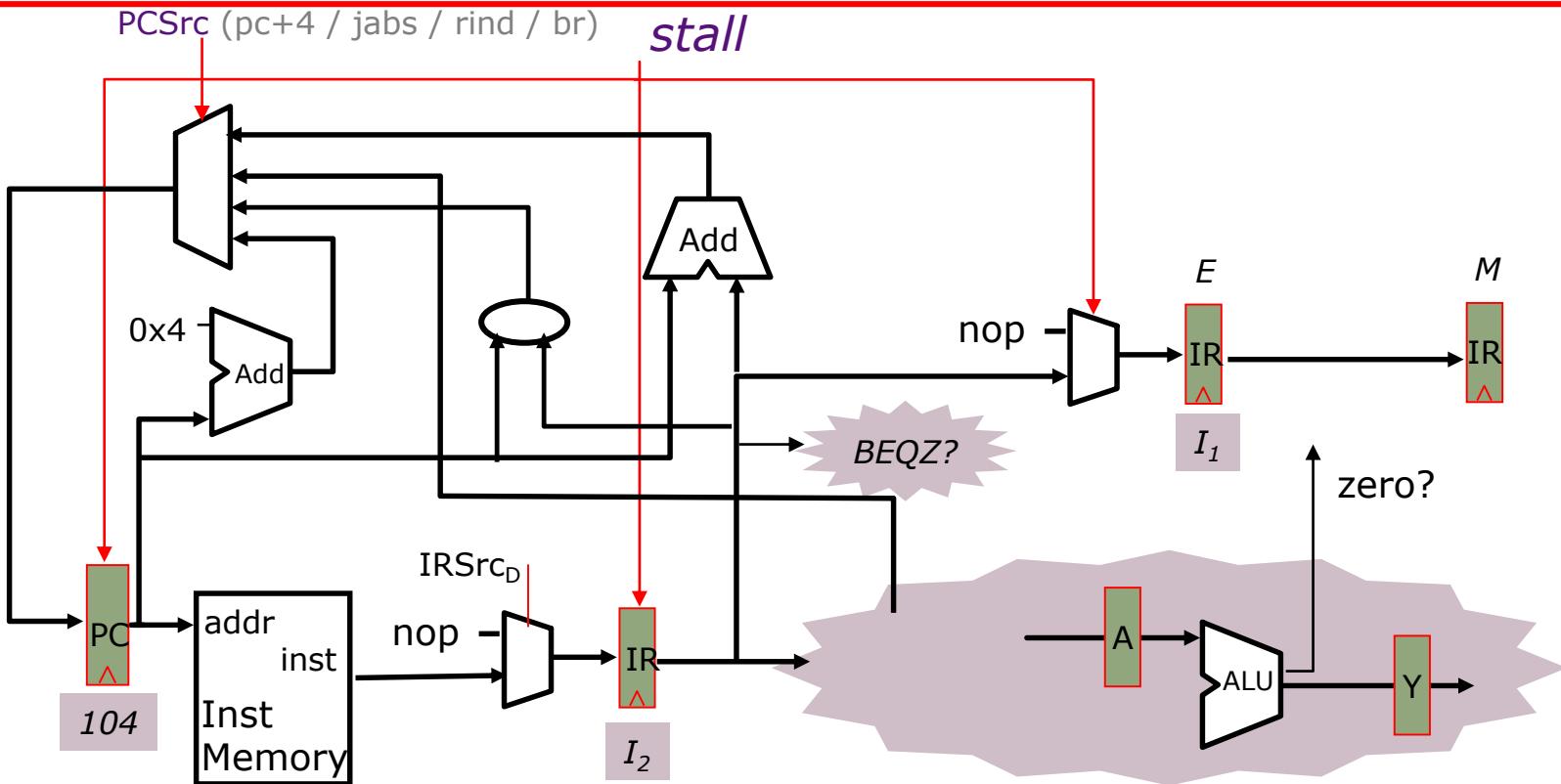
Jump Pipeline Diagrams

	time								
	t0	t1	t2	t3	t4	t5	t6	t7
(I ₁) 096: ADD	IF ₁	ID ₁	EX ₁	MA ₁	WB ₁				
(I ₂) 100: J 200		IF ₂	ID ₂	EX ₂	MA ₂	WB ₂			
(I ₃) 104: ADD		IF ₃	nop	nop	nop	nop			
(I ₄) 304: ADD			IF ₄	ID ₄	EX ₄	MA ₄	WB ₄		

Resource Usage	time								
	t0	t1	t2	t3	t4	t5	t6	t7	
	IF	I ₁	I ₂	I ₃	I ₄	I ₅			
	ID		I ₁	I ₂	nop	I ₄	I ₅		
	EX			I ₁	I ₂	nop	I ₄	I ₅	
	MA				I ₁	I ₂	nop	I ₄	I ₅
	WB					I ₁	I ₂	nop	I ₄

nop \Rightarrow *pipeline bubble*

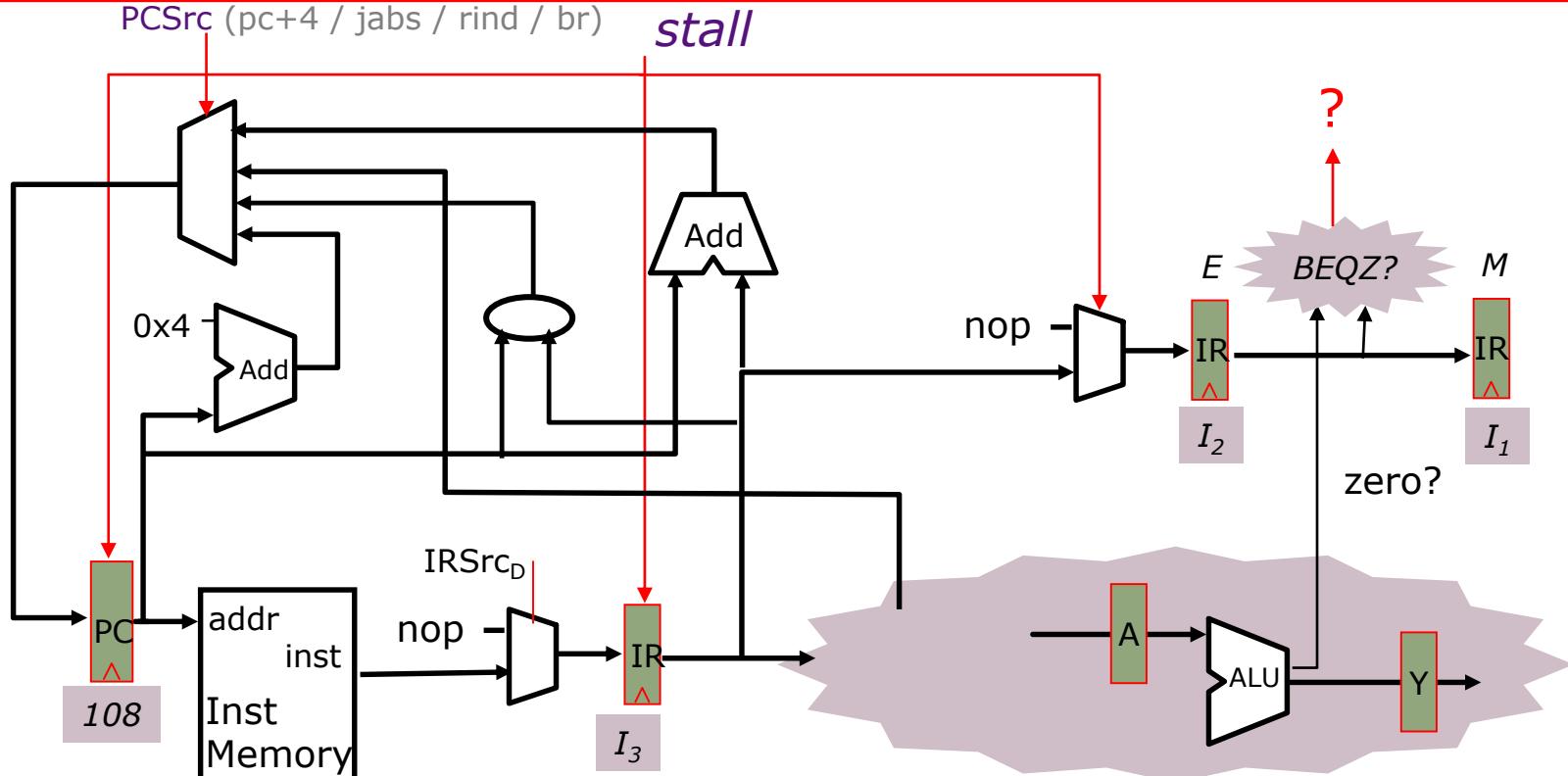
Pipelining Conditional Branches



096	ADD
100	BEQZ r1 200
104	ADD
304	ADD

Branch condition is not known until
the execute stage
*what action should be taken in the
decode stage?*

Pipelining Conditional Branches

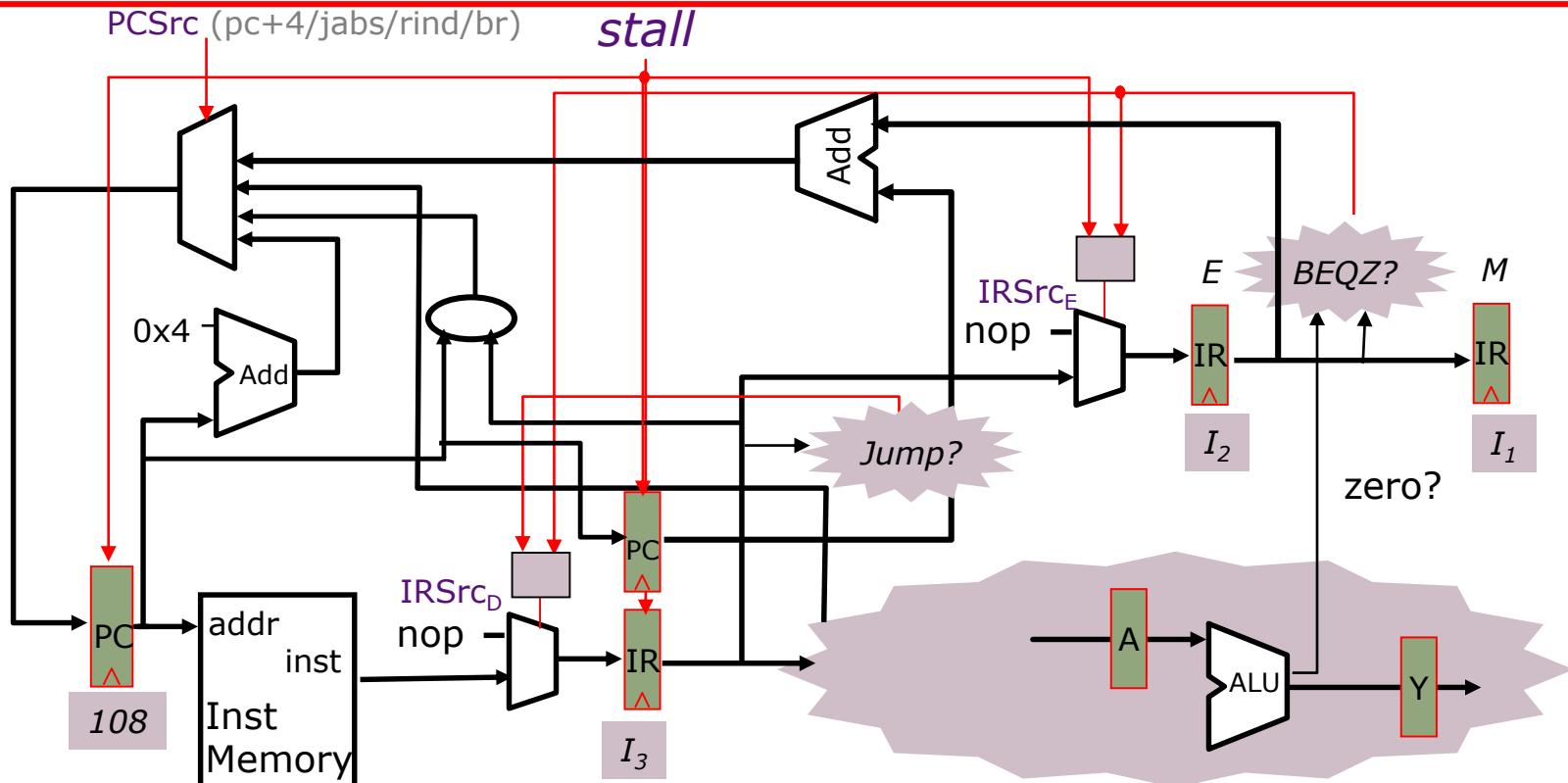


If the branch is taken

- kill the two following instructions
 - the instruction at the decode stage is not valid
- \Rightarrow *stall signal is not valid*

I_1	096	ADD
I_2	100	BEQZ r1 200
I_3	104	ADD
I_4	304	ADD

Pipelining Conditional Branches



If the branch is taken

- kill the two following instructions
 - the instruction at the decode stage is not valid
- \Rightarrow **stall signal is not valid**

I_1	096	ADD
I_2	100	BEQZ r1 200
I_3	104	ADD
I_4	304	ADD

New Stall Signal

```
stall = ( ((rsD==wsE)·weE + (rsD==wsM)·weM + (rsD==wsW)·weW)·re1D
          + ((rtD==wsE)·weE + (rtD==wsM)·weM + (rtD==wsW)·weW)·re2D
      ) · !((opcodeE==BEQZ)·z + (opcodeE==BNEZ)·!z)
```

Don't stall if the branch is taken. Why?

Control Equations for PC and IR Muxes

$\text{IRSrc}_D = \text{Case opcode}_E$

$\text{BEQZ}\cdot z, \text{BNEZ}\cdot !z \Rightarrow \text{nop}$

...

\Rightarrow

Case opcode_D

$J, \text{JAL}, \text{JR}, \text{JALR} \Rightarrow \text{nop}$

...

$\Rightarrow \text{IM}$

Give priority to the older instruction, i.e., execute stage instruction over decode stage instruction

$\text{IRSrc}_E = \text{Case opcode}_E$

$\text{BEQZ}\cdot z, \text{BNEZ}\cdot !z \Rightarrow \text{nop}$

...

$\Rightarrow \text{stall}\cdot \text{nop} + \text{!stall}\cdot \text{IR}_D$

$\text{PCSsrc} = \text{Case opcode}_E$

$\text{BEQZ}\cdot z, \text{BNEZ}\cdot !z \Rightarrow \text{br}$

...

\Rightarrow

Case opcode_D

$J, \text{JAL} \Rightarrow \text{jabs}$

$\text{JR}, \text{JALR} \Rightarrow \text{rind}$

... $\Rightarrow \text{pc+4}$

pc+4 is a speculative guess

$\text{nop} \Rightarrow \text{Kill}$

$\text{br/jabs/rind} \Rightarrow \text{Restart}$

$\text{pc+4} \Rightarrow \text{Speculate}$

Branch Pipeline Diagrams (resolved in execute stage)

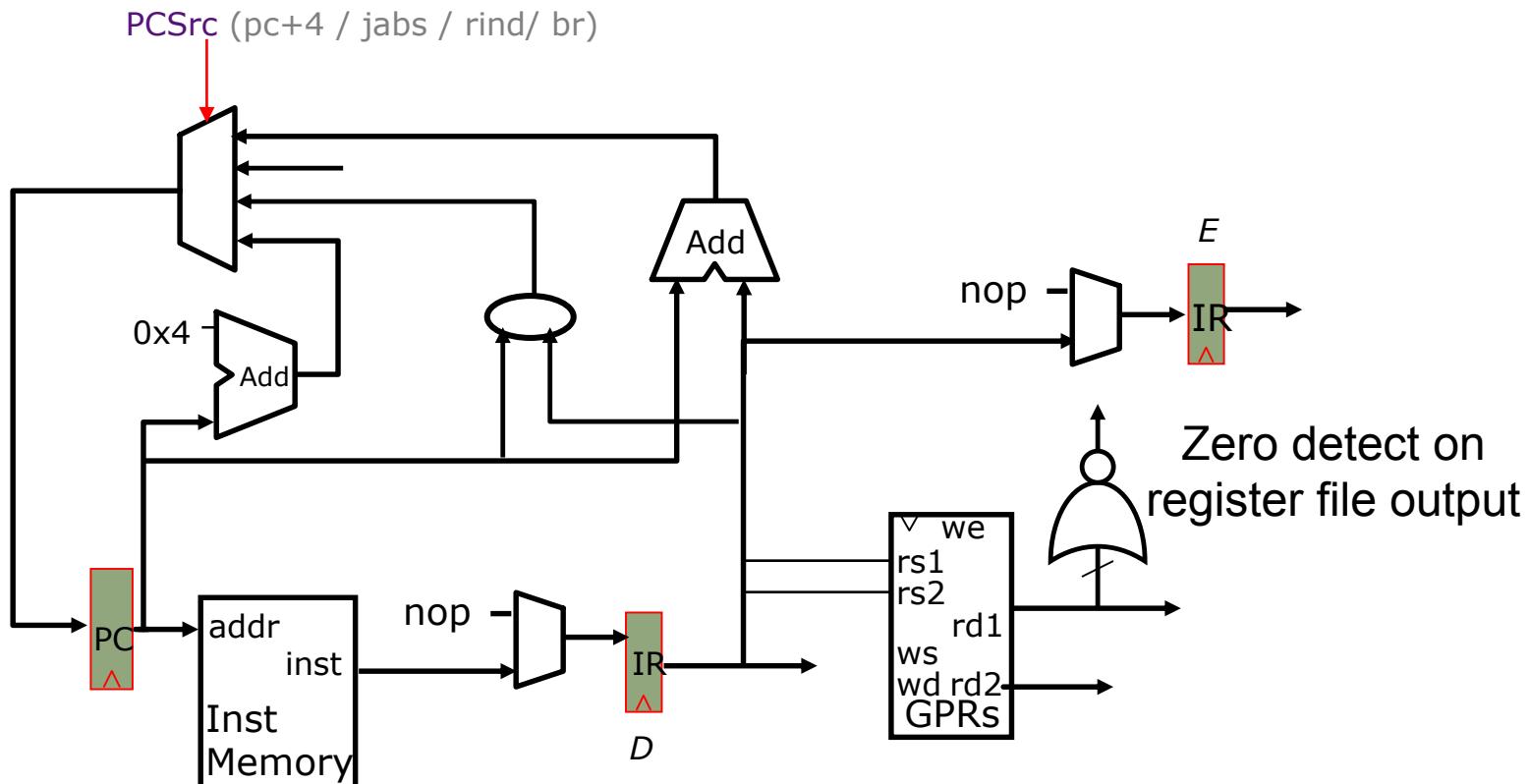
	time									
	t0	t1	t2	t3	t4	t5	t6	t7	...	
(I ₁) 096: ADD	IF ₁	ID ₁	EX ₁	MA ₁	WB ₁					
(I ₂) 100: BEQZ 200		IF ₂	ID ₂	EX ₂	MA ₂	WB ₂				
(I ₃) 104: ADD			IF ₃	ID ₃	nop	nop	nop			
(I ₄) 108:				IF ₄	nop	nop	nop	nop		
(I ₅) 304: ADD					IF ₅	ID ₅	EX ₅	MA ₅	WB ₅	

Resource Usage	time									
	t0	t1	t2	t3	t4	t5	t6	t7	...	
	IF	I ₁	I ₂	I ₃	I ₄	I ₅				
	ID		I ₁	I ₂	I ₃	nop	I ₅			
	EX			I ₁	I ₂	nop	nop	I ₅		
	MA				I ₁	I ₂	nop	nop	I ₅	
	WB					I ₁	I ₂	nop	nop	I ₅

nop \Rightarrow *pipeline bubble*

Reducing Branch Penalty (resolve in decode stage)

- One pipeline bubble can be removed if an extra comparator is used in the Decode stage



Pipeline diagram now same as for jumps

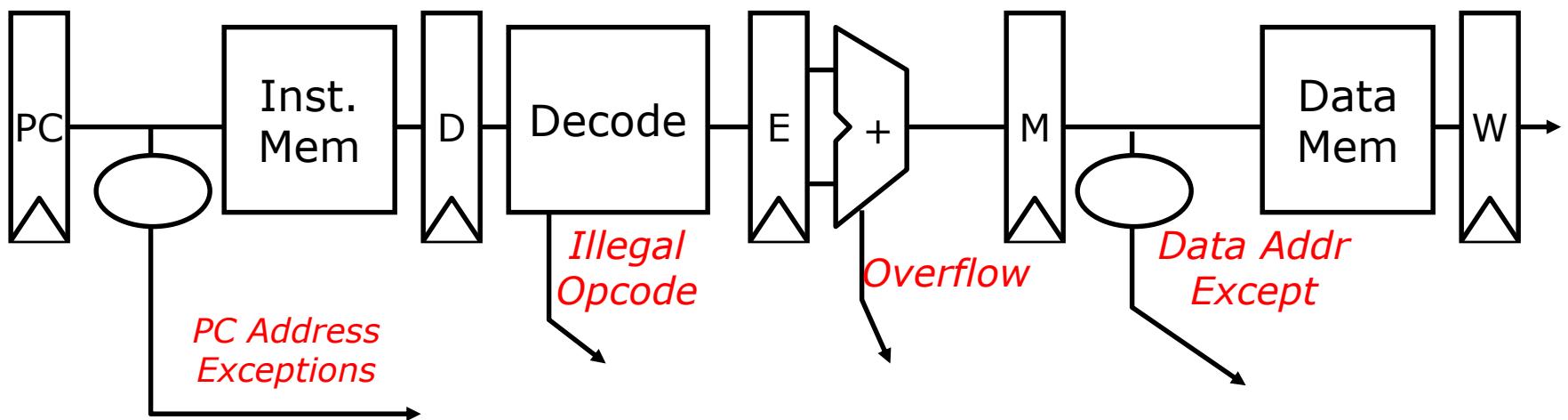
Branch Delay Slots (expose control hazard to software)

- Change the ISA semantics so that the instruction that follows a jump or branch is always executed
 - gives compiler the flexibility to put in a useful instruction where normally a pipeline bubble would have resulted.

I ₁	096	ADD	
I ₂	100	BEQZ r1 200	<i>Delay slot instruction</i>
I ₃	104	ADD	<i>executed regardless of branch outcome</i>
I ₄	304	ADD	

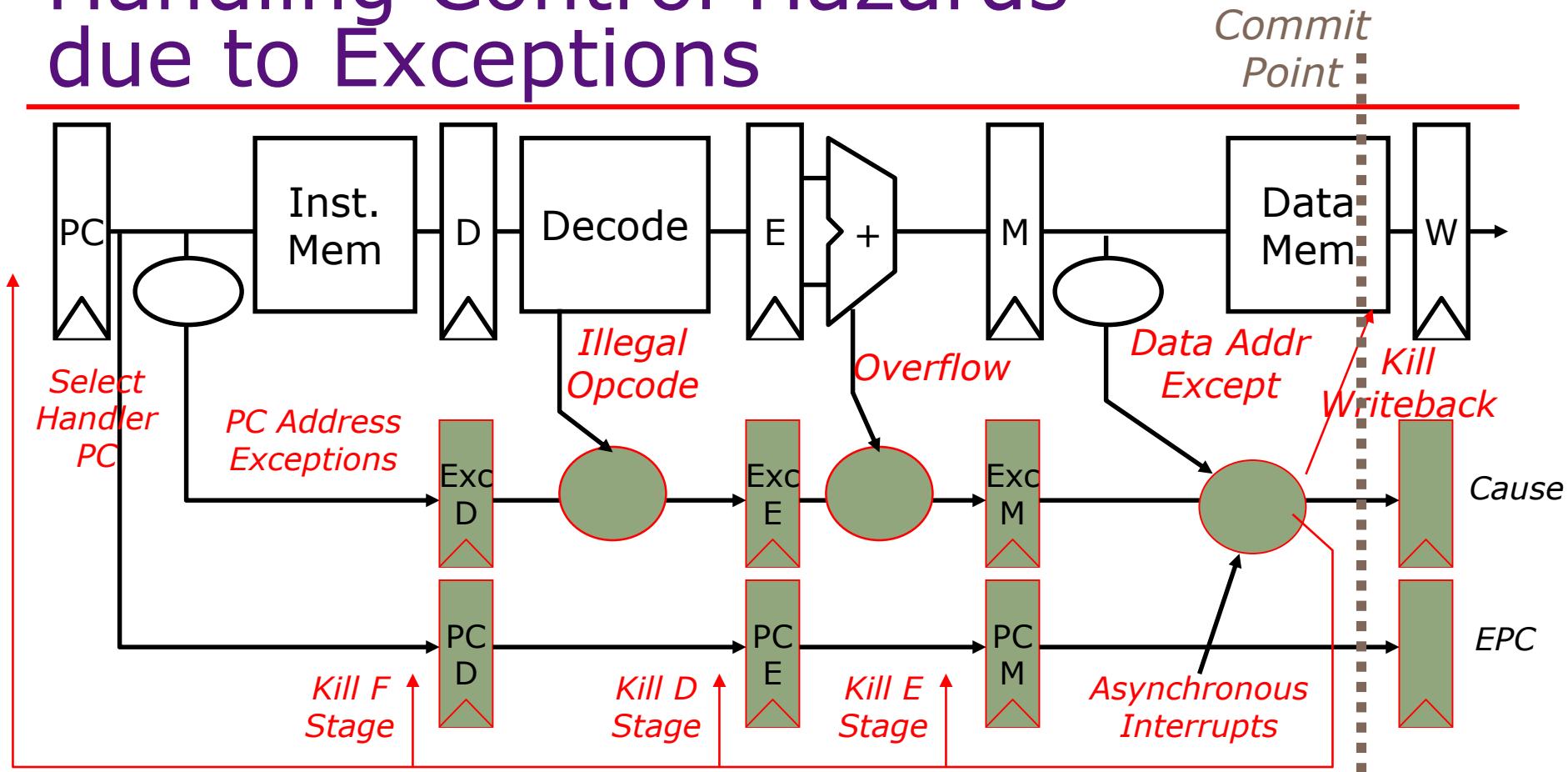
- Other techniques include branch prediction, which can dramatically reduce the branch penalty... *to come later*

Handling Control Hazards due to Exceptions



- Instructions may suffer exceptions in different pipeline stages
- Must prioritize exceptions from earlier instructions

Handling Control Hazards due to Exceptions



- Typical strategy: Record exceptions, process the first one to reach commit point (i.e., the point where architectural state is modified)
 - Pros/cons vs handling exceptions eagerly, like branches?

Why an instruction may not be dispatched every cycle (CPI>1)

- Full bypassing may be too expensive to implement
 - Typically all frequently used paths are provided
 - Some infrequently used bypass paths may increase cycle time and counteract the benefit of reducing CPI
- Loads have two-cycle latency
 - Instruction after load cannot use load result
 - MIPS-I ISA defined *load delay slots*, a software-visible pipeline hazard (compiler schedules independent instruction or inserts NOP to avoid hazard). Removed in MIPS-II.
- Conditional branches, jumps, and exceptions may cause bubbles
 - Kill instruction(s) following branch if no delay slots

Machines with software-visible delay slots may execute significant number of NOP instructions inserted by the compiler.

Next lecture: Superscalar & Scoreboarded Pipelines

Complex Pipelining

Daniel Sanchez

Computer Science and Artificial Intelligence Laboratory
M.I.T.

Complex Pipelining: Motivation

Instruction pipelining becomes complex when we want high performance in the presence of

- Multi-cycle operations, for example:
 - Full or partially pipelined floating-point units, or
 - Long-latency operations, e.g., divides
- Variable-latency operations, for example:
 - Memory systems with variable access time
- Replicated function units, for example:
 - Multiple floating-point or memory units

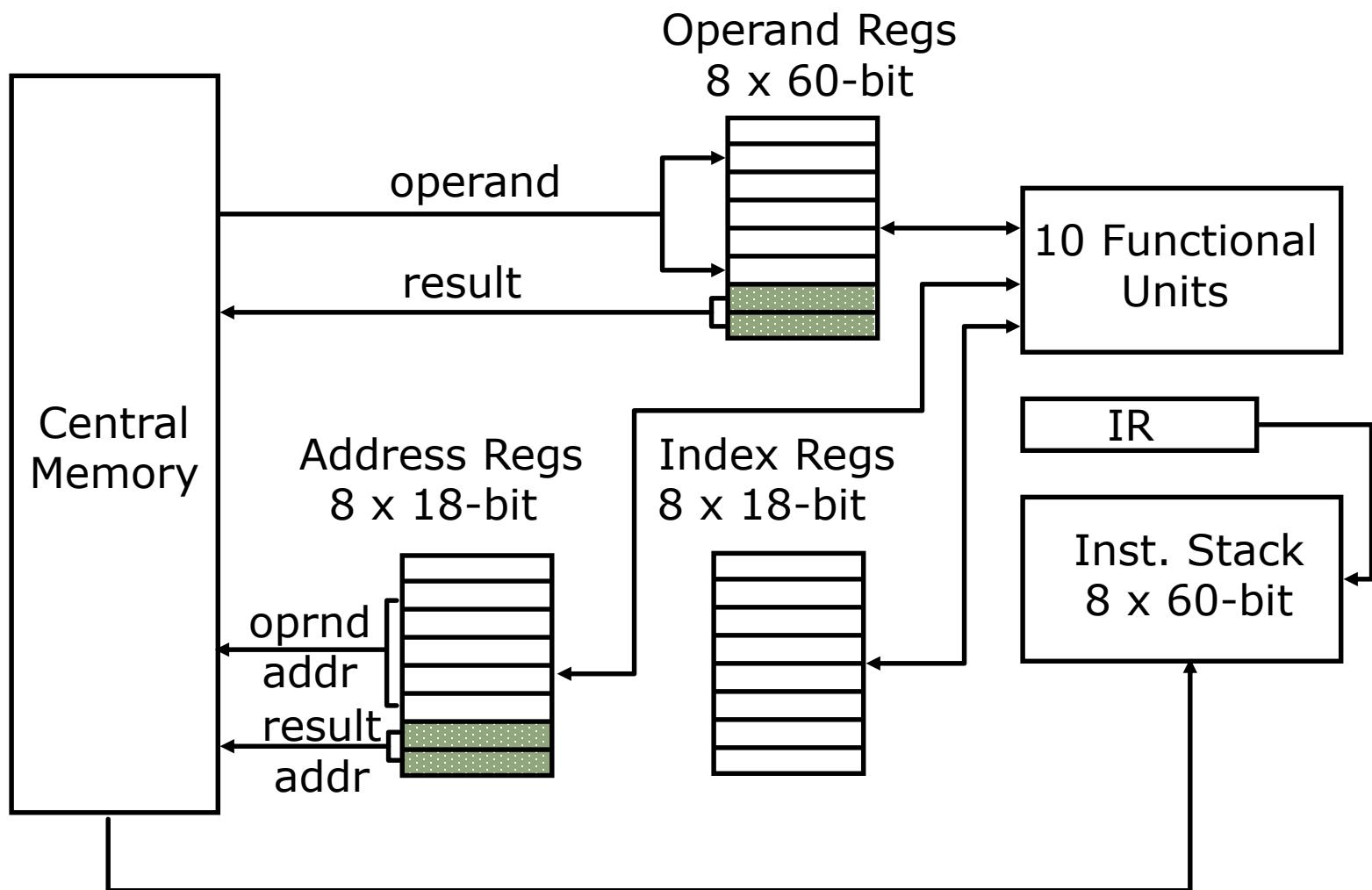
CDC 6600

Seymour Cray, 1963



- A fast pipelined machine with 60-bit words
 - 128 Kword main memory capacity, 32 banks
- Ten functional units (parallel, unpipelined)
 - Floating Point: adder, 2 multipliers, divider
 - Integer: adder, 2 incrementers, ...
- Hardwired control
- **Dynamic scheduling of instructions using a scoreboard**
- Ten Peripheral Processors for Input/Output
 - A fast multi-threaded 12-bit integer ALU
- Very fast clock, 10 MHz (FP add in 4 clocks)
- >400,000 transistors, 750 sq. ft., 5 tons, 150 kW, new freon-based cooling technology
- Fastest machine in world for 5 years (until CDC 7600)
 - Over 100 sold (\$7-10M each)

CDC 6600: Datapath

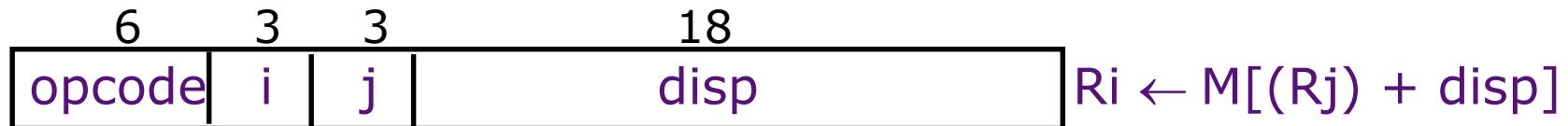


CDC 6600: A Load/Store Architecture

- Separate instructions to manipulate three types of reg.
 - 8 60-bit data registers (X)
 - 8 18-bit address registers (A)
 - 8 18-bit index registers (B)
- All arithmetic and logic instructions are reg-to-reg



- Only Load and Store instructions refer to memory!



- Touching address registers 1 to 5 initiates a load
6 to 7 initiates a store
 - *very useful for vector operations*

CDC6600: Vector Addition

```
B1 ← - n
loop: JZE B1, exit
      A1 ← B1 + a1      load into X1
      A2 ← B1 + b1      load into X2
      X6 ← X1 + X2
      A6 ← B1 + c1      store X6
      B1 ← B1 + 1
      jump loop
```

A_i = address register

B_i = index register

X_i = data register

more on vector processing later...

We will present complex
pipelining issues more
abstractly ...

Floating Point ISA

Interaction between the Floating point datapath and the Integer datapath is determined largely by the ISA

MIPS ISA

- separate register files for FP and Integer instructions
the only interaction is via a set of move instructions (some ISAs don't even permit this)
- separate load/store for FPR's and GPR's but both use GPR's for address calculation
- separate conditions for branches
FP branches are defined in terms of condition codes

Floating Point Unit

Much more hardware than an integer unit

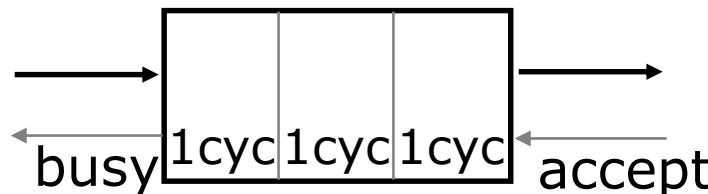
Single-cycle floating point unit is a bad idea - *why?*

- it is common to have several floating point units
- it is common to have different types of FPUs
Fadd, Fmul, Fdiv, ...
- an FPU may be pipelined, partially pipelined or not pipelined

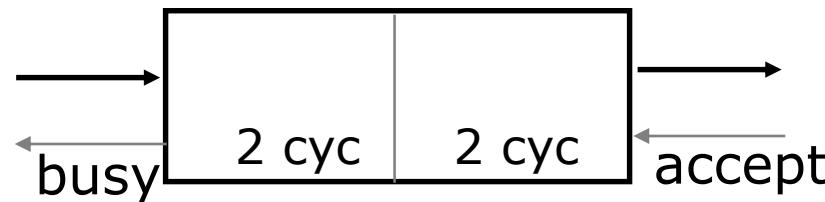
To operate several FPUs concurrently the register file needs to have more read and write ports

Functional Unit Characteristics

*fully
pipelined*



*partially
pipelined*



Functional units have internal pipeline registers

- ⇒ operands are latched when an instruction enters a functional unit
- ⇒ inputs to a functional unit (e.g., register file) can change during a long latency operation

Realistic Memory Systems

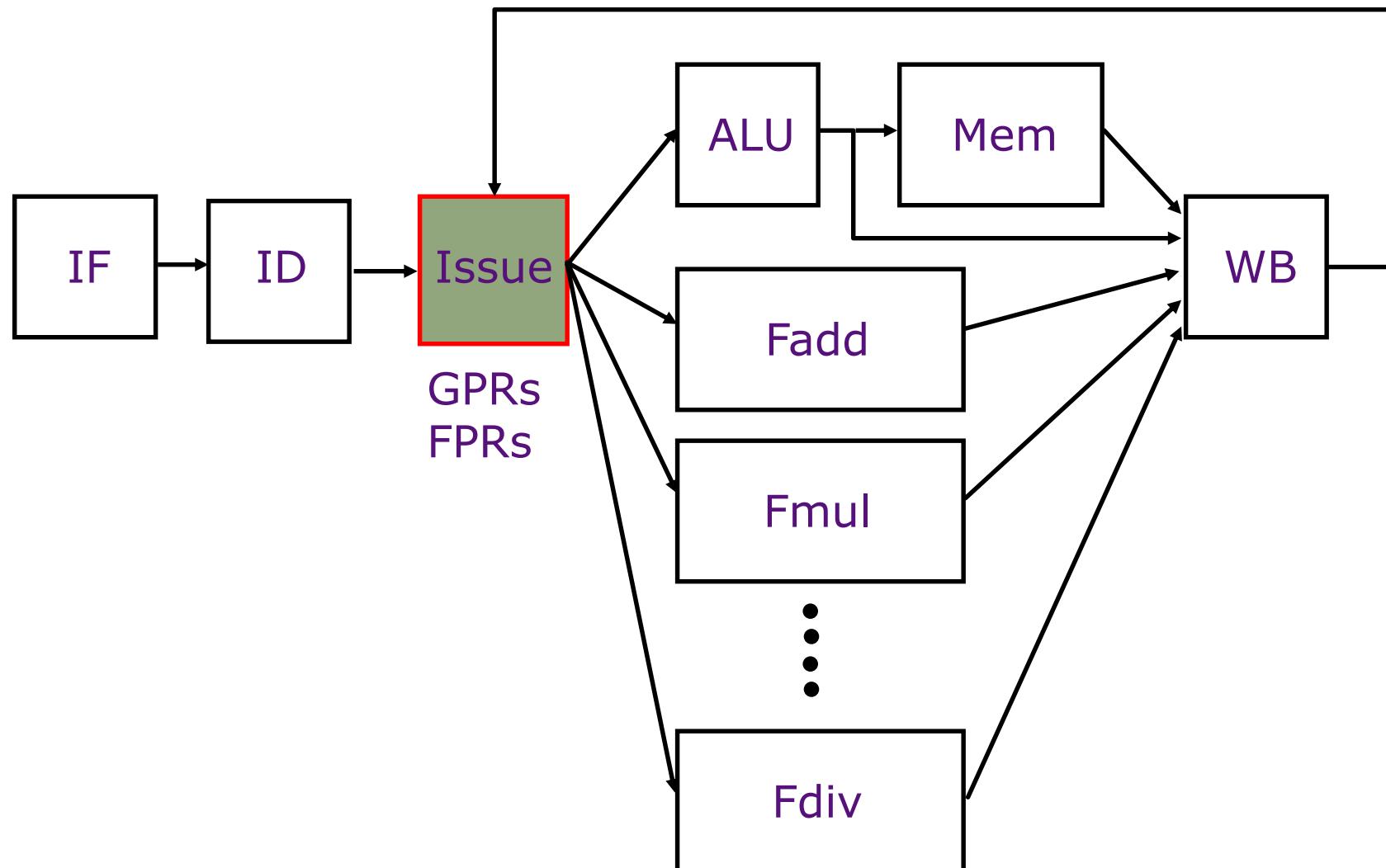
Latency of access to the main memory is usually much higher than one cycle and often unpredictable

Solving this problem is a central issue in computer architecture

Common approaches to improving memory performance

- separate instruction and data memory ports
 \Rightarrow *no self-modifying code*
- caches
single cycle except in case of a miss \Rightarrow *stall*
- interleaved memory
multiple memory accesses \Rightarrow *bank conflicts*
- split-phase memory operations
 \Rightarrow *out-of-order responses*

Complex Pipeline Structure



Complex Pipeline Control Issues

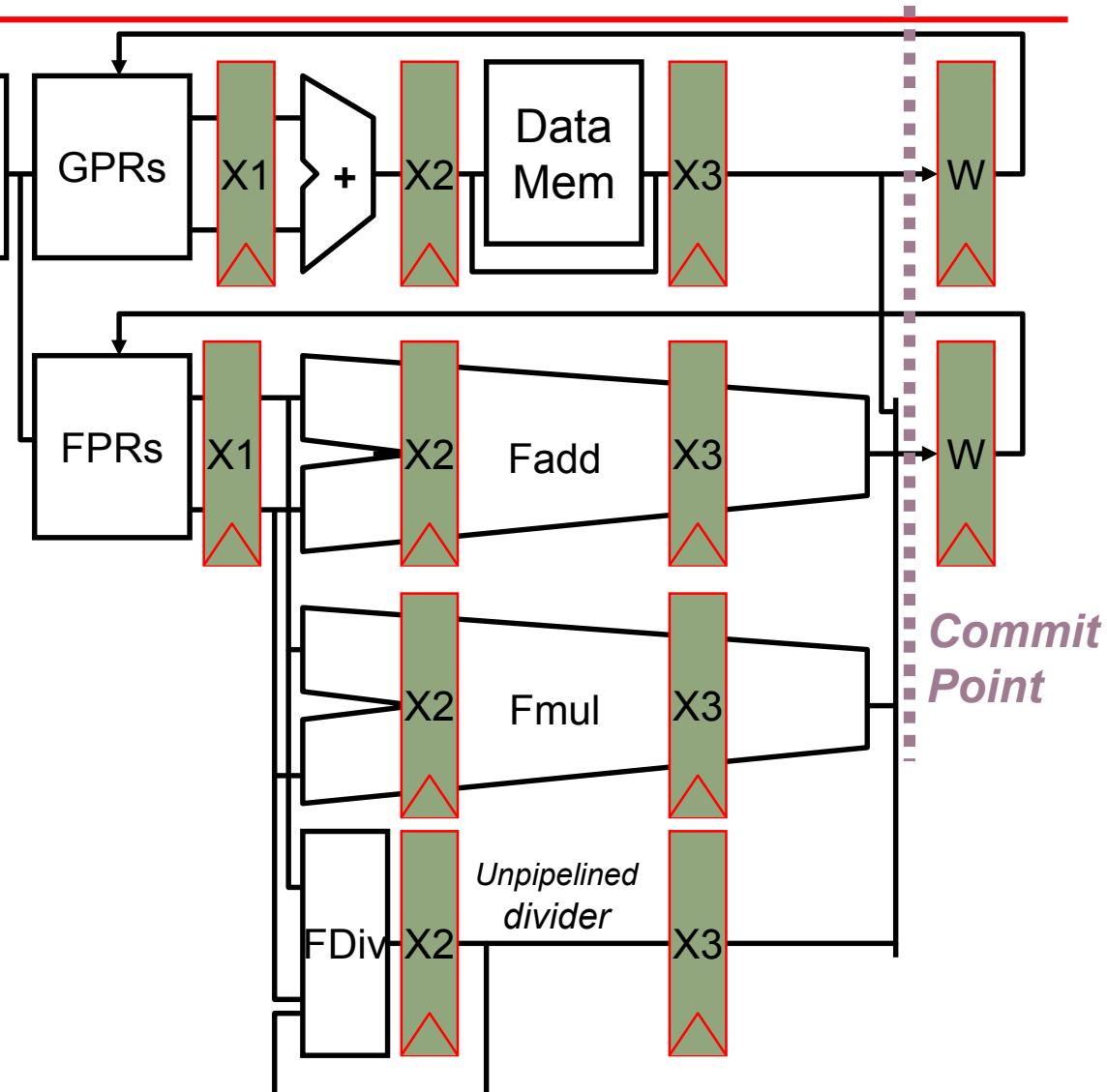
- Structural hazards at the execution stage if some FPU or memory unit is not pipelined and takes more than one cycle
- Structural hazards at the write-back stage due to variable latencies of different function units
- Out-of-order write hazards due to variable latencies of different function units
- How to handle exceptions?

Complex In-Order Pipeline

- Delay writeback so all operations have same latency to W stage

- Write ports never oversubscribed (one inst. in & one inst. out every cycle)

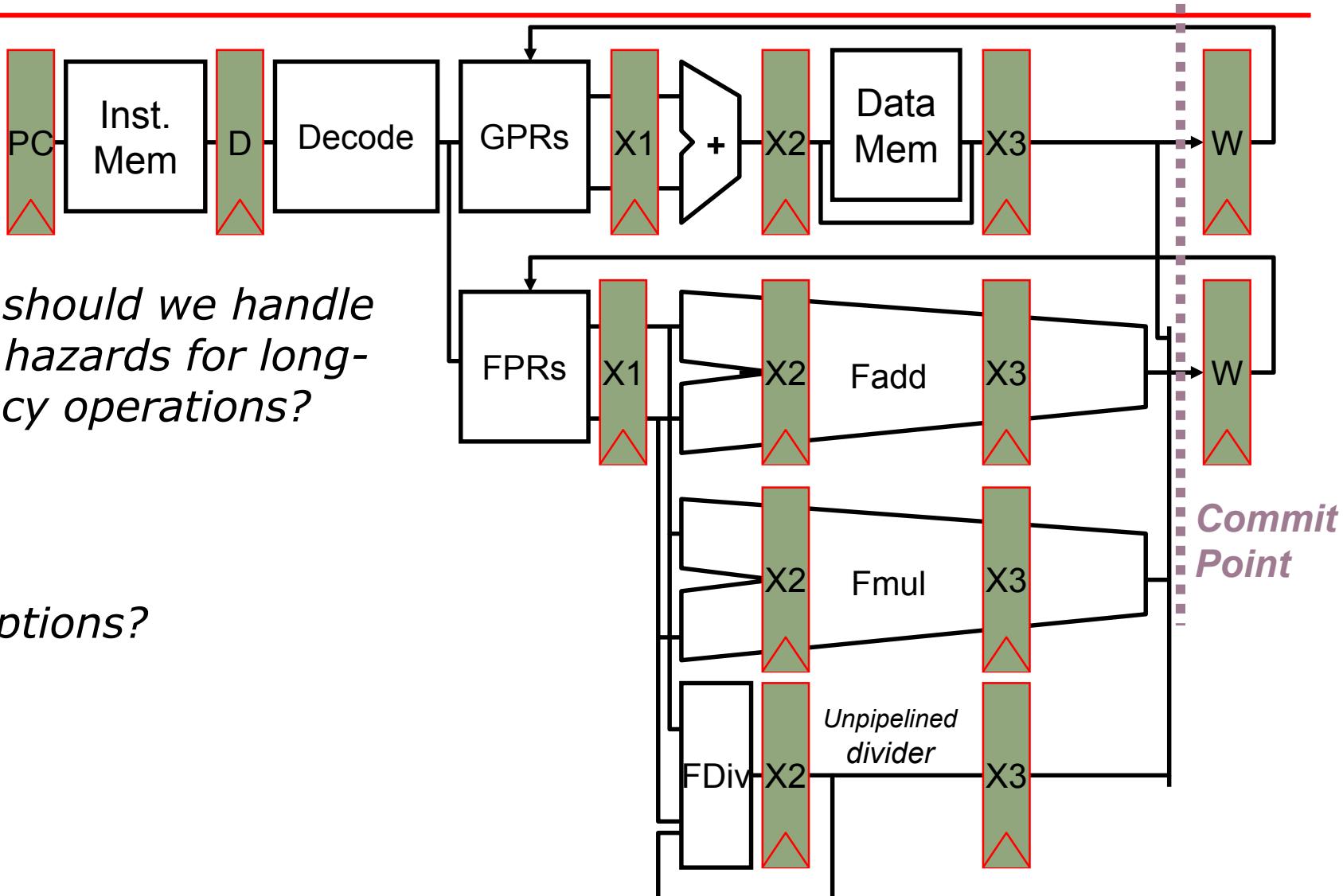
How to prevent increased writeback latency from slowing down single-cycle integer operations?



Complex In-Order Pipeline

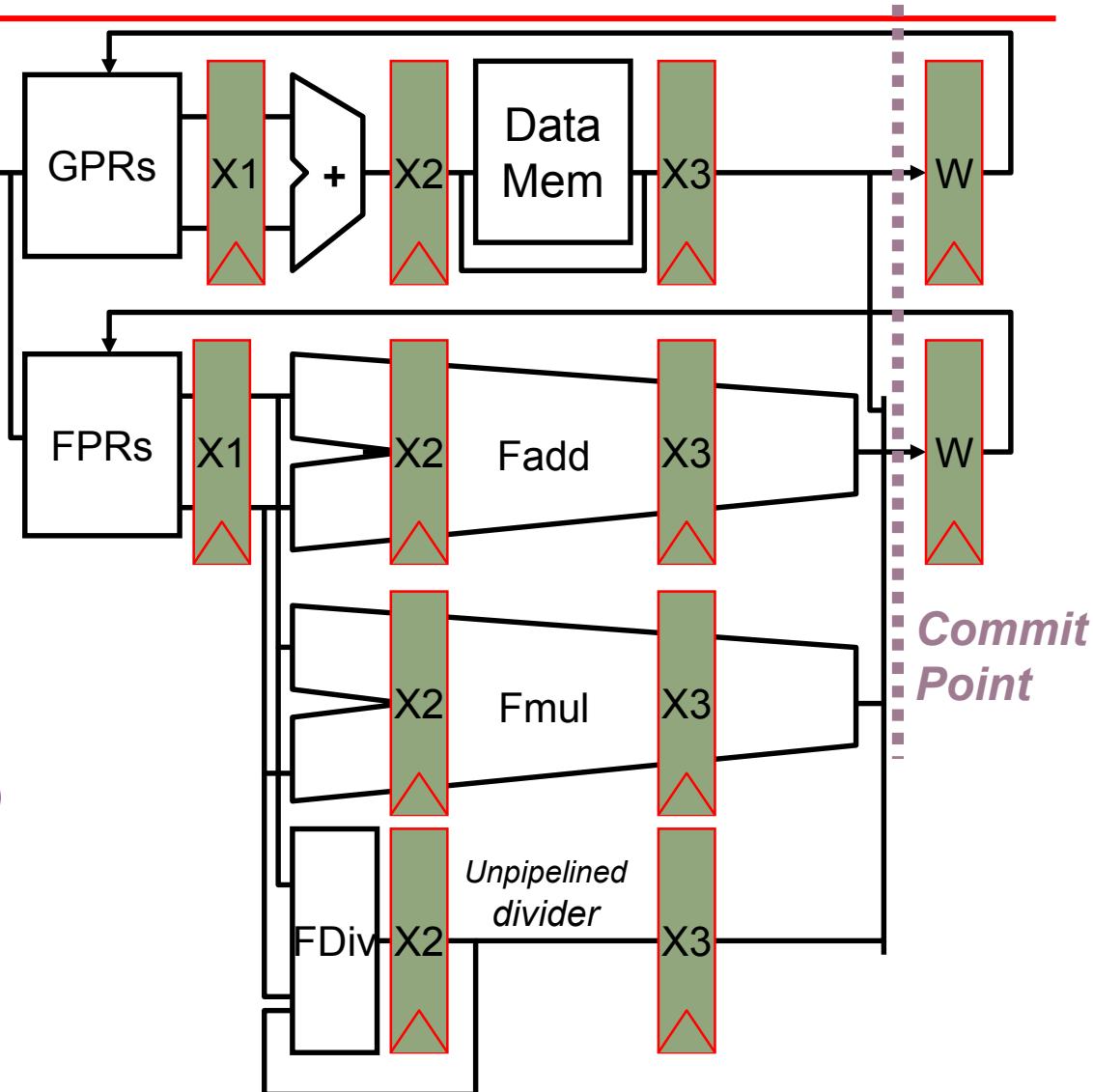
How should we handle data hazards for long-latency operations?

Exceptions?



Superscalar In-Order Pipeline

- Fetch two instructions per cycle; issue both simultaneously *if* one is integer/memory and other is floating-point
- Inexpensive way of increasing throughput
 - Examples:
Alpha 21064 (1992)
MIPS R5000 series (1996)
- Can be extended to wider issue but register file ports and bypassing costs grow quickly
 - E.g., 4-issue UltraSPARC



Dependence Analysis

Needed to Exploit Instruction-level Parallelism

Types of Data Hazards

Consider executing a sequence of

$$r_k \leftarrow (r_i) \text{ op } (r_j)$$

type of instructions

Data-dependence

$$\begin{array}{l} r_3 \leftarrow (r_1) \text{ op } (r_2) \\ r_5 \leftarrow (r_3) \text{ op } (r_4) \end{array}$$

Read-after-Write
(RAW) hazard

Anti-dependence

$$\begin{array}{l} r_3 \leftarrow (r_1) \text{ op } (r_2) \\ r_1 \leftarrow (r_4) \text{ op } (r_5) \end{array}$$

Write-after-Read
(WAR) hazard

Output-dependence

$$\begin{array}{l} r_3 \leftarrow (r_1) \text{ op } (r_2) \\ r_3 \leftarrow (r_6) \text{ op } (r_7) \end{array}$$

Write-after-Write
(WAW) hazard

Detecting Data Hazards

Range and Domain of instruction i

$R(i)$ = Registers (or other storage) modified by instruction i

$D(i)$ = Registers (or other storage) read by instruction i

Suppose instruction j follows instruction i in the program order. Executing instruction j before the effect of instruction i has taken place can cause a

RAW hazard if $R(i) \cap D(j) \neq \emptyset$

WAR hazard if $D(i) \cap R(j) \neq \emptyset$

WAW hazard if $R(i) \cap R(j) \neq \emptyset$

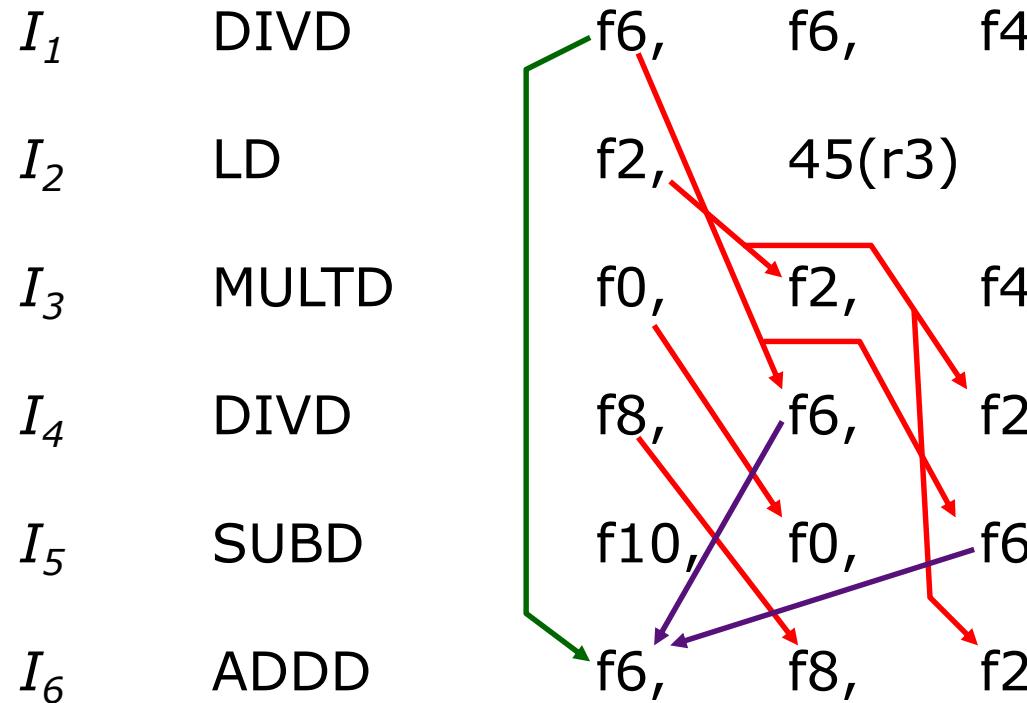
Register vs. Memory Data Dependencies

- Data hazards due to register operands can be determined at the decode stage *but*
- Data hazards due to memory operands can be determined only after computing the effective address

<i>store</i>	$M[(r1) + \text{disp1}] \leftarrow (r2)$
<i>load</i>	$r3 \leftarrow M[(r4) + \text{disp2}]$

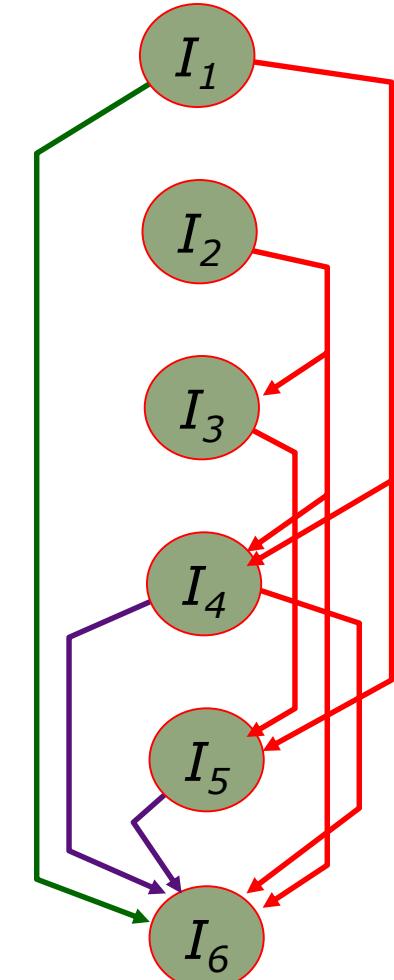
Does $(r1) + \text{disp1} == (r4) + \text{disp2}$?

Data Hazards: An Example



RAW Hazards
WAR Hazards
WAW Hazards

Instruction Scheduling



Valid orderings:

in-order I_1 I_2 I_3 I_4 I_5

out-of-order I_2 I_1 I_3 I_4 I_5

out-of-order I_1 I_2 I_3 I_5 I_4

Out-of-order Completion

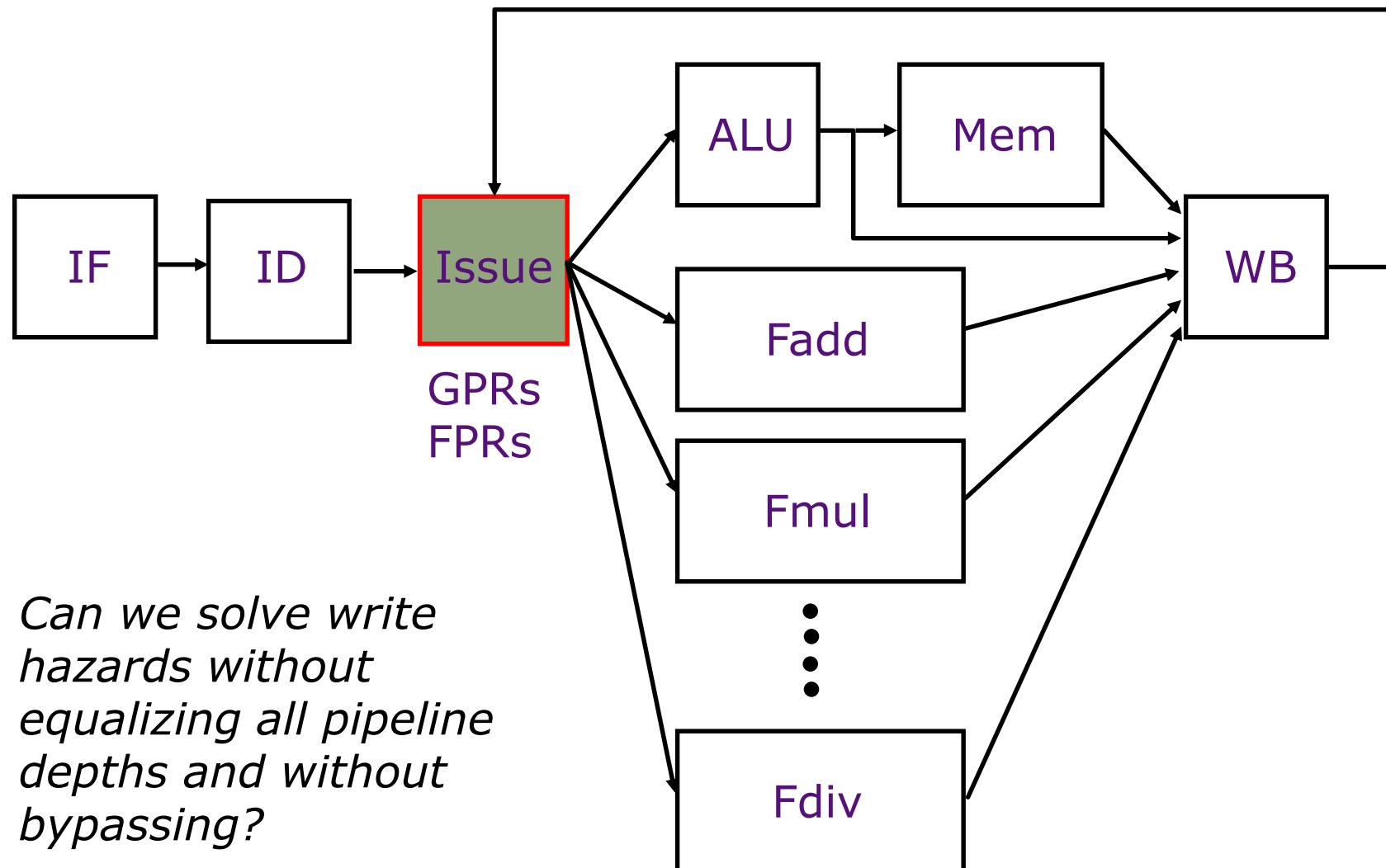
In-order Issue

														<i>Latency</i>		
I_1	DIVD		f6,		f6,		f4							4		
I_2	LD		f2,		45(r3)									1		
I_3	MULTD		f0,		f2,		f4							3		
I_4	DIVD		f8,		f6,		f2							4		
I_5	SUBD		f10,		f0,		f6							1		
I_6	ADDD		f6,		f8,		f2							1		
<i>cycle</i>		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
<i>in-order comp</i>		1	2			<u>1</u>	<u>2</u>	3	4		<u>3</u>	5	<u>4</u>	6	<u>5</u>	<u>6</u>
<i>out-of-order comp</i>		1	2	<u>2</u>	3	<u>1</u>	4	<u>3</u>	5	<u>5</u>	<u>4</u>	6	<u>6</u>			

What problems can out-of-order comp cause?

Scoreboard: A Hardware Data Structure to Detect Hazards Dynamically

Complex Pipeline



When is it Safe to Issue an Instruction?

- Approach: Stall issue until sure that issuing will cause no dependence problems...
- Suppose a data structure keeps track of all the instructions in all the functional units
- The following checks need to be made before the Issue stage can dispatch an instruction
 - Is the required function unit available?
 - Is the input data available? \Rightarrow RAW?
 - Is it safe to write the destination? \Rightarrow WAR? WAW?
 - Is there a structural conflict at the WB stage?

A Data Structure for Correct Issues

Keeps track of the status of Functional Units

Name	Busy	Op	Dest	Src1	Src2
Int					
Mem					
Add1					
Add2					
Add3					
Mult1					
Mult2					
Div					

The instruction i at the Issue stage consults this table

FU available?

RAW?

WAR?

WAW?

*An entry is added to the table if no hazard is detected;
An entry is removed from the table after Write-Back*

Simplifying the Data Structure Assuming In-order Issue

- Suppose the instruction is not dispatched by the Issue stage
 - If a RAW hazard exists
 - or if the required FU is busy
- Suppose operands are latched by the functional unit on issue

Can the dispatched instruction cause a
WAR hazard?

WAW hazard?

Simplifying the Data Structure

- No WAR hazard
 - ⇒ no need to keep $src1$ and $src2$
- The Issue stage does not dispatch an instruction in case of a WAW hazard
 - ⇒ a register name can occur at most once in the $dest$ column
- WP[reg#]: a bit-vector to record the registers for which writes are pending
 - *These bits are set to true by the Issue stage and set to false by the WB stage*
 - ⇒ Each pipeline stage in the FU's must carry the $dest$ field and a flag to indicate if it is valid “*the (we, ws) pair*”

Scoreboard for In-order Issues

Busy[FU#] : a bit-vector to indicate FU's availability.
(FU = Int, Add, Mult, Div)

These bits are hardwired to FU's.

WP[reg#] : a bit-vector to record the registers for which writes are pending.

These bits are set to true by the Issue stage and set to false by the WB stage

Issue checks the instruction (opcode dest src1 src2) against the scoreboard (Busy & WP) to dispatch

FU available?

RAW?

WAR?

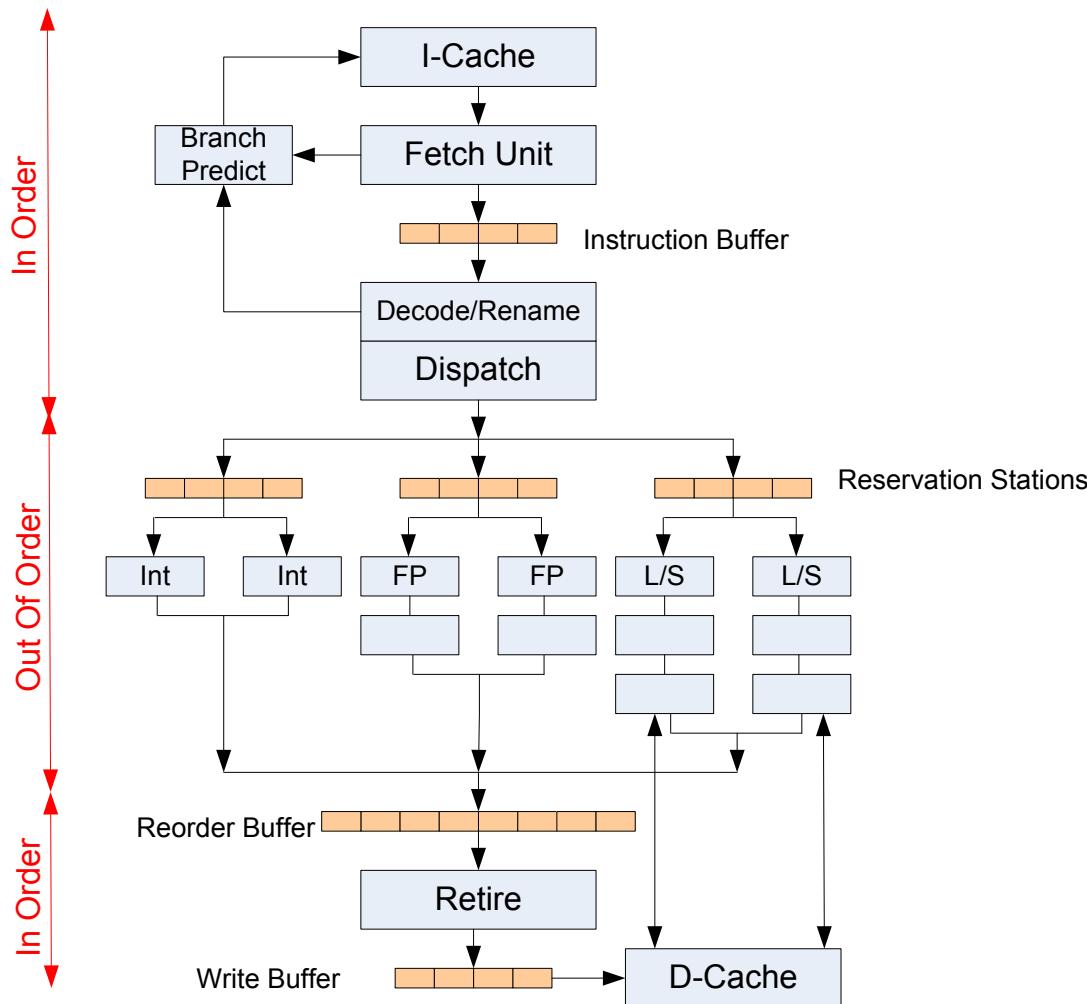
WAW?

Scoreboard Dynamics

	Functional Unit Status					Registers Reserved for Writes	
	Int(1)	Add(1)	Mult(3)	Div(4)	WB		
t0	I_1			f6		f6	
t1	I_2	f2		f6		f6, f2	
t2				f6	f2	f6, f2	I_2
t3	I_3		f0		f6	f6, f0	
t4			f0		f6	f6, f0	I_1
t5	I_4			f0 f8		f0, f8	
t6				f8	f0	f0, f8	I_3
t7	I_5		f10		f8	f8, f10	
t8					f8 f10	f8, f10	I_5
t9					f8	f8	I_4
t10	I_6		f6			f6	
t11					f6	f6	I_6

I_1	DIVD	f6,	f6,	f4
I_2	LD	f2,	45(r3)	
I_3	MULTD	f0,	f2,	f4
I_4	DIVD	f8,	f6,	f2
I_5	SUBD	f10,	f0,	f6
I_6	ADDD	f6,	f8,	f2

Preview: Anatomy of a Modern Out-of-Order Superscalar Core



- L08 (Today): Complex pipes w/ in-order issue
- L09: Out-of-order exec & renaming
- L10: Branch prediction
- L11: Speculative execution and recovery
- L12: Advanced Memory Ops

Complex Pipelining: Out-of-Order Execution, Register Renaming, and Exceptions

Daniel Sanchez

Computer Science and Artificial Intelligence Laboratory
M.I.T.

CDC 6600-style Scoreboard

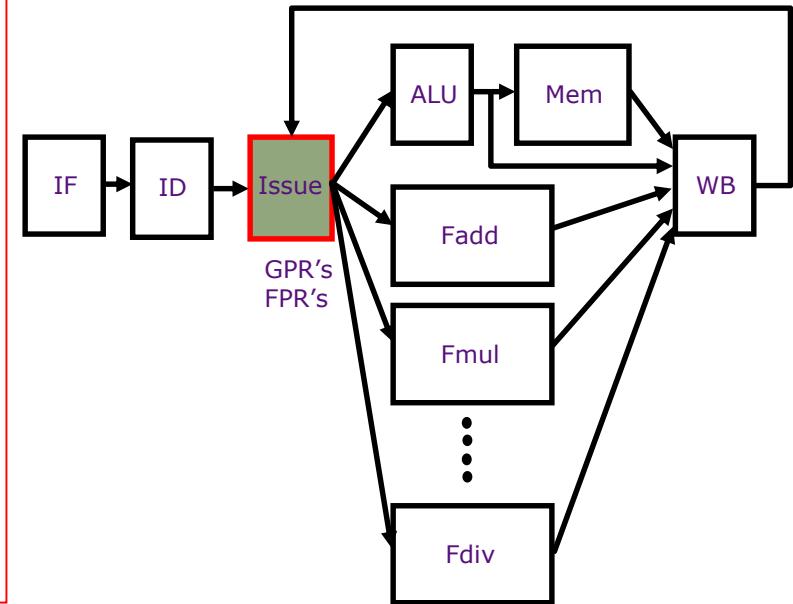
Instructions are issued in order.

An instruction is issued only if

- It cannot cause a RAW hazard
- It cannot cause a WAW hazard
⇒ *There can be at most one instruction in the execute phase that can write in a particular register*

WAR hazards are not possible

- Due to in-order issue +
operands read immediately



Scoreboard:
Two bit-vectors

Busy[FU#]: Indicates FU's availability
These bits are hardwired to FU's.

WP[reg#]: Records if a write is pending
for a register
Set to true by the Issue stage and
set to false by the WB stage

Reminder: Scoreboard Dynamics

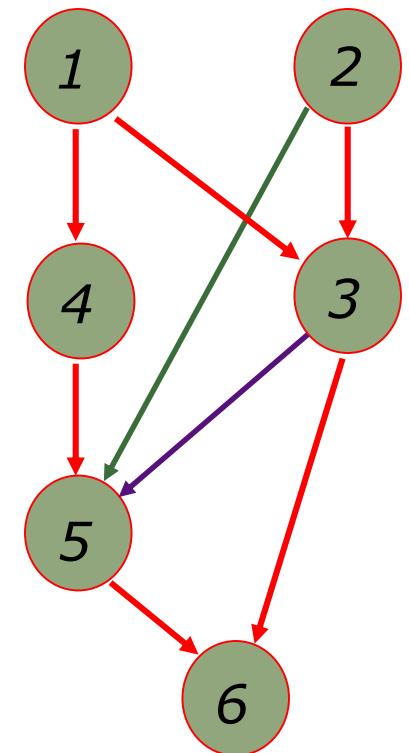
Issue time	Functional Unit Status					WP	WB time
	Int(1)	Add(1)	Mult(3)	Div(4)	WB		
t0 I_1				f6		f6	
t1 I_2	f2			f6		f6, f2	
t2				f6	f2	f6, f2	I_2
t3 I_3		f0		f6		f6, f0	
t4		f0			f6	f6, f0	I_1
t5 I_4			f0 f8			f0, f8	
t6				f8	f0	f0, f8	I_3
t7 I_5		f10		f8		f8, f10	
t8				f8 f10		f8, f10	I_5
t9					f8	f8	I_4
t10 I_6		f6				f6	
t11					f6	f6	I_6

I_1	DIVD	f6,	f6,	f4
I_2	LD	f2,	45(r3)	
I_3	MULTD	f0,	f2,	f4
I_4	DIVD	f8,	f6,	f2
I_5	SUBD	f10,	f0,	f6
I_6	ADDD	f6,	f8,	f2

In-Order Issue Limitations

An example

					<i>latency</i>
1	LD	F2,	34(R2)		1
2	LD	F4,	45(R3)		<i>long</i>
3	MULTD	F6,	F4,	F2	3
4	SUBD	F8,	F2,	F2	1
5	DIVD	F4,	F2,	F8	4
6	ADDD	F10,	F6,	F4	1



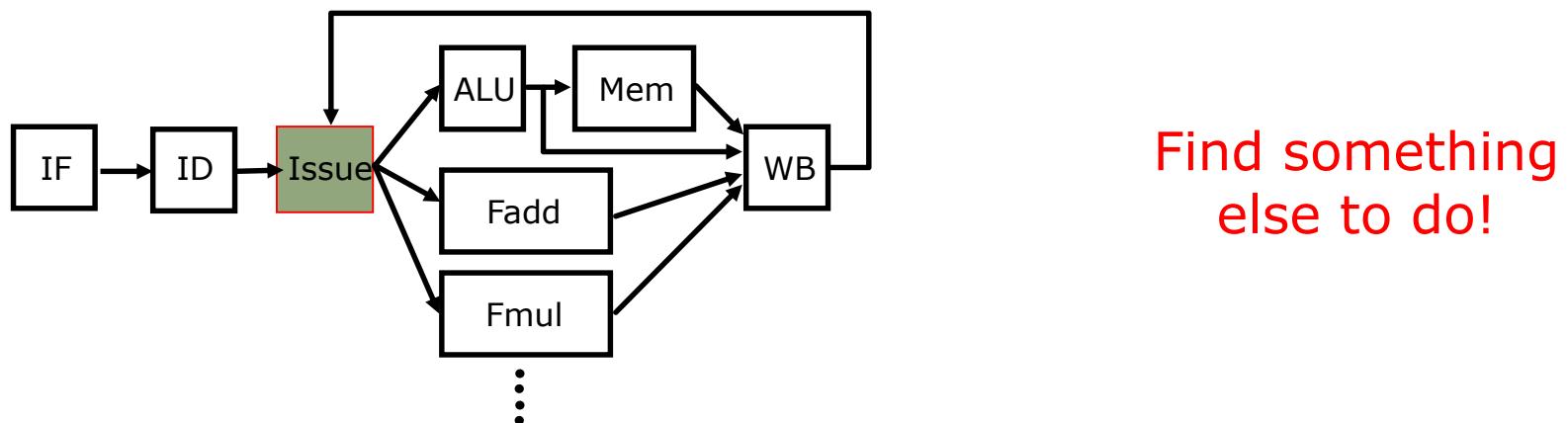
In-order:

1 (2,1) 2 3 4 4 3 5 . . 5 6 6

In-order restriction prevents instruction 4 from being dispatched

Out-of-Order Issue

How can we address the delay caused by a RAW dependence associated with the next in-order instruction?

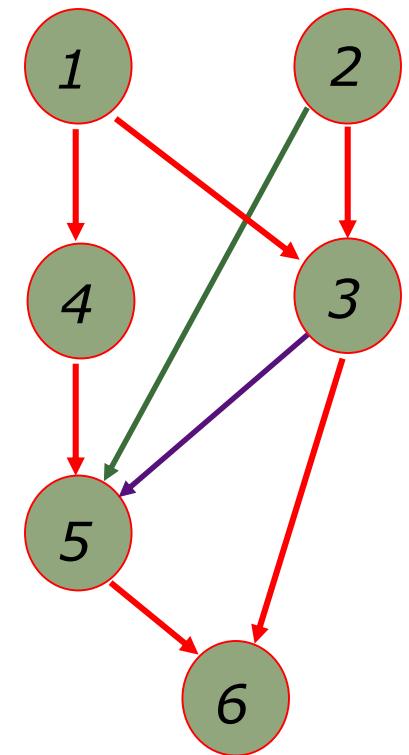


- Issue stage buffer holds multiple instructions waiting to issue.
- Decode adds next instruction to buffer if there is space and the instruction does not cause a WAR or WAW hazard.
- Can issue any instruction in buffer whose RAW hazards are satisfied (*for now at most one dispatch per cycle*).
Note: A writeback (WB) may enable more instructions.

In-Order Issue Limitations

An example

					latency
1	LD	F2,	34(R2)		1
2	LD	F4,	45(R3)		long
3	MULTD	F6,	F4, F2		3
4	SUBD	F8,	F2, F2		1
5	DIVD	F4,	F2, F8		4
6	ADDD	F10,	F6, F4		1



In-order: 1 (2,1) 2 3 4 4 3 5 . . . 5 6 6

Out-of-order: 1 (2,1) 4 4 . . . 2 3 5 . 3 . 5 6 6

WAR/WAW hazards prevent instruction 5 from being dispatched

Out-of-order execution did not produce a significant improvement!

How many Instructions can be in the pipeline

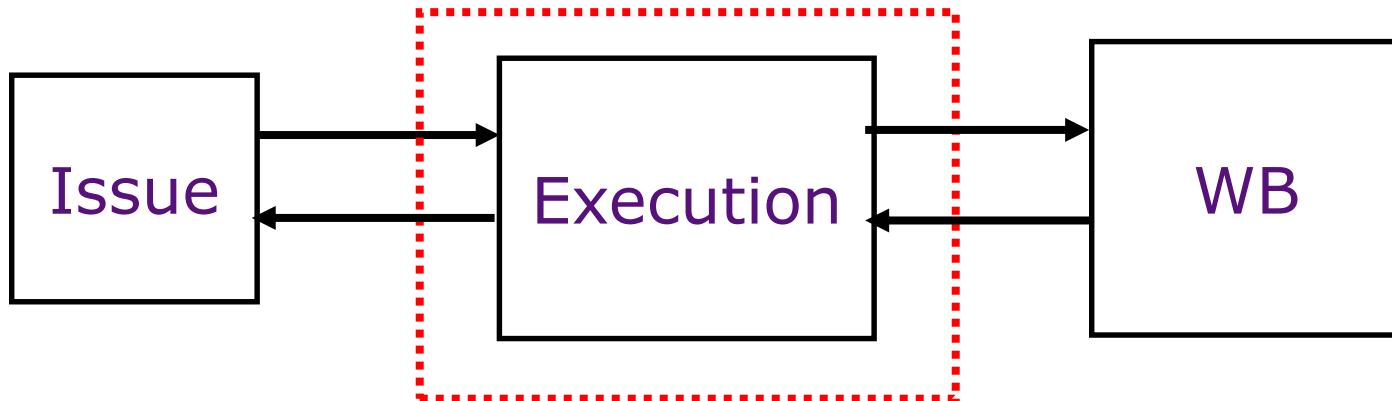
Throughput is limited by number of instructions in flight, but which feature of an ISA limits the number of instructions in the pipeline?

Out-of-order dispatch by itself does not provide a significant performance improvement!

How can we better understand the impact of number of registers on throughput?

Little's Law

$$\text{Throughput } (\bar{T}) = \text{Number in Flight } (\bar{N}) / \text{Latency } (\bar{L})$$



Example:

4 floating point registers

8 cycles per floating point operation

⇒

Overcoming the Lack of Register Names

Floating Point pipelines often cannot be kept filled with small number of registers.

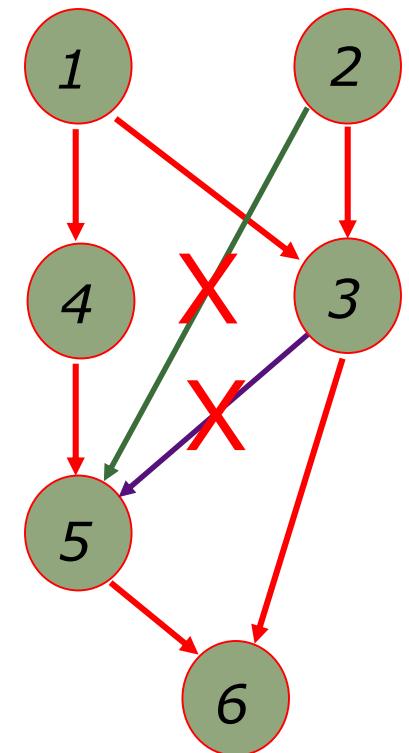
IBM 360 had only 4 Floating Point Registers

Can a microarchitecture use more registers than specified by the ISA without loss of ISA compatibility ?

Yes, Robert Tomasulo of IBM suggested an ingenious solution in 1967 based on on-the-fly *register renaming*

Instruction-level Parallelism via Renaming

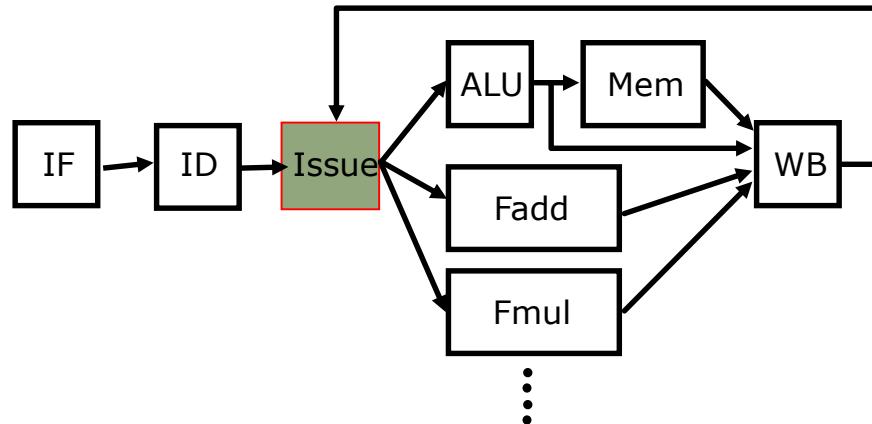
					latency
1	LD	F2,	34(R2)		1
2	LD	F4,	45(R3)		long
3	MULTD	F6,	F4, F2		3
4	SUBD	F8,	F2, F2		1
5	DIVD	F4' ,	F2, F8		4
6	ADDD	F10,	F6, F4'		1



In-order: 1 (2,1) 2 3 4 4 3 5 . . . 5 6 6
Out-of-order: 1 (2,1) 4 4 5 . . . (2,5) 3 . . 3 6 6

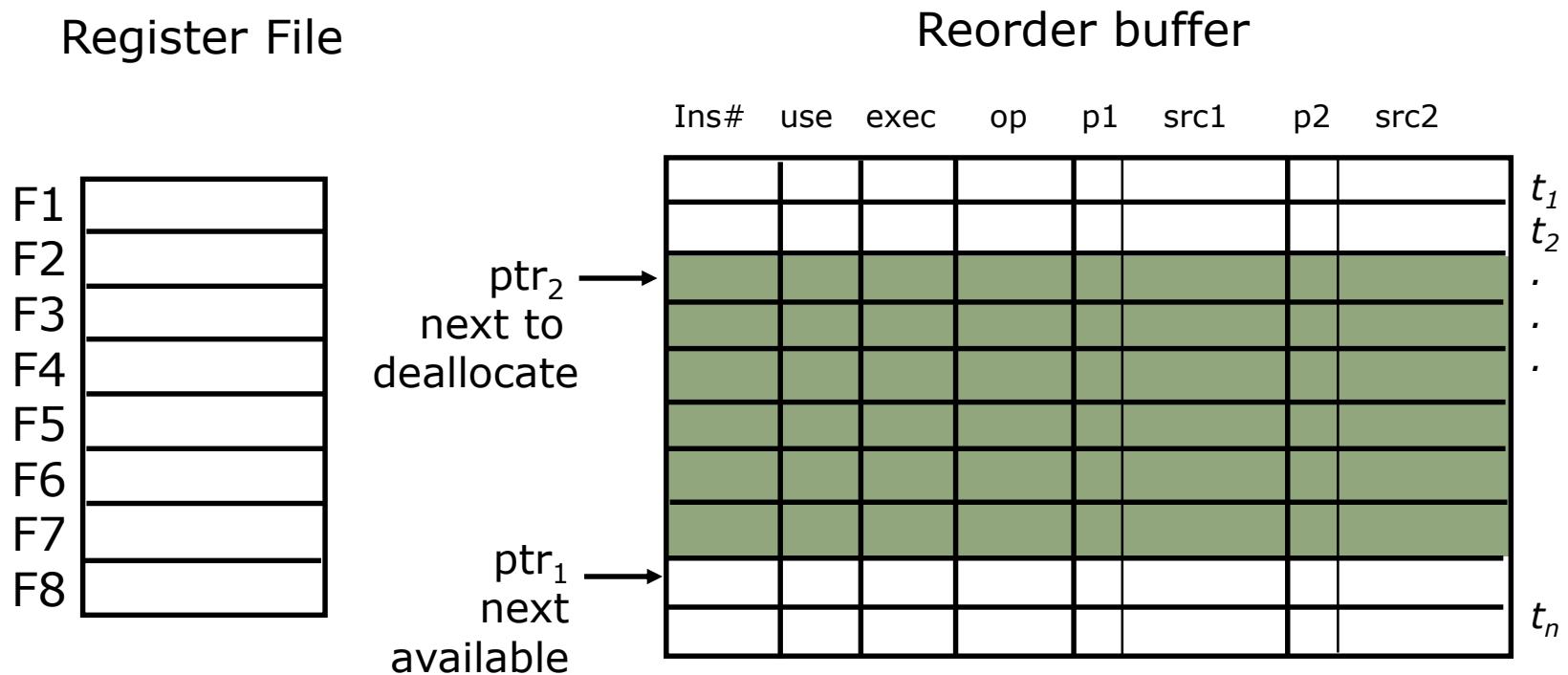
*Renaming eliminates WAR and WAW hazards
(renaming \Rightarrow additional storage)*

Handling register dependencies



- Decode does register renaming, providing a new spot for each register write
 - Renaming eliminates WAR and WAW hazards by allowing use of more storage space
- Renamed instructions added to an issue stage structure, called the **reorder buffer (ROB)**. Any instruction in the ROB whose RAW hazards have been satisfied can be dispatched
 - Out-of-order or dataflow execution handles RAW hazards

Reorder Buffer



Instruction slot is candidate for execution when:

- It holds a valid instruction ("use" bit is set)
- It has not already started execution ("exec" bit is clear)
- Both operands are available ("present" bits p1 and p2 are set)

Is it obvious where an architectural register value is? **No**

Renaming & Out-of-order Issue

Renaming table & reg file

A plot showing a signal p versus data. The y-axis is labeled with F1 through F8. The x-axis is labeled 'p' and 'data'. A gray line starts at F8, goes up to F7, then down to F6, then up to F5, then down to F4, then up to F3, then down to F2, then up to F1, then down to F8. It then turns right and goes up to a peak at approximately (data=10, $p=10$), then down to (data=12, $p=8$), then up to (data=14, $p=10$), then down to (data=16, $p=8$), then up to (data=18, $p=10$), then down to (data=20, $p=8$).

Holds data (v_i)
or tag(t_i)

Reorder buffer

- *When are names in sources replaced by data?*
 - *When can a name be reused?*

Renaming & Out-of-order Issue

An example

Renaming table & reg file

	p	data
F1		
F2	Ø	t1
F3		
F4	Ø	t2
F5		
F6	Ø	t3
F7		
F8	Ø	t4

data (v_i) / tag(t_i)

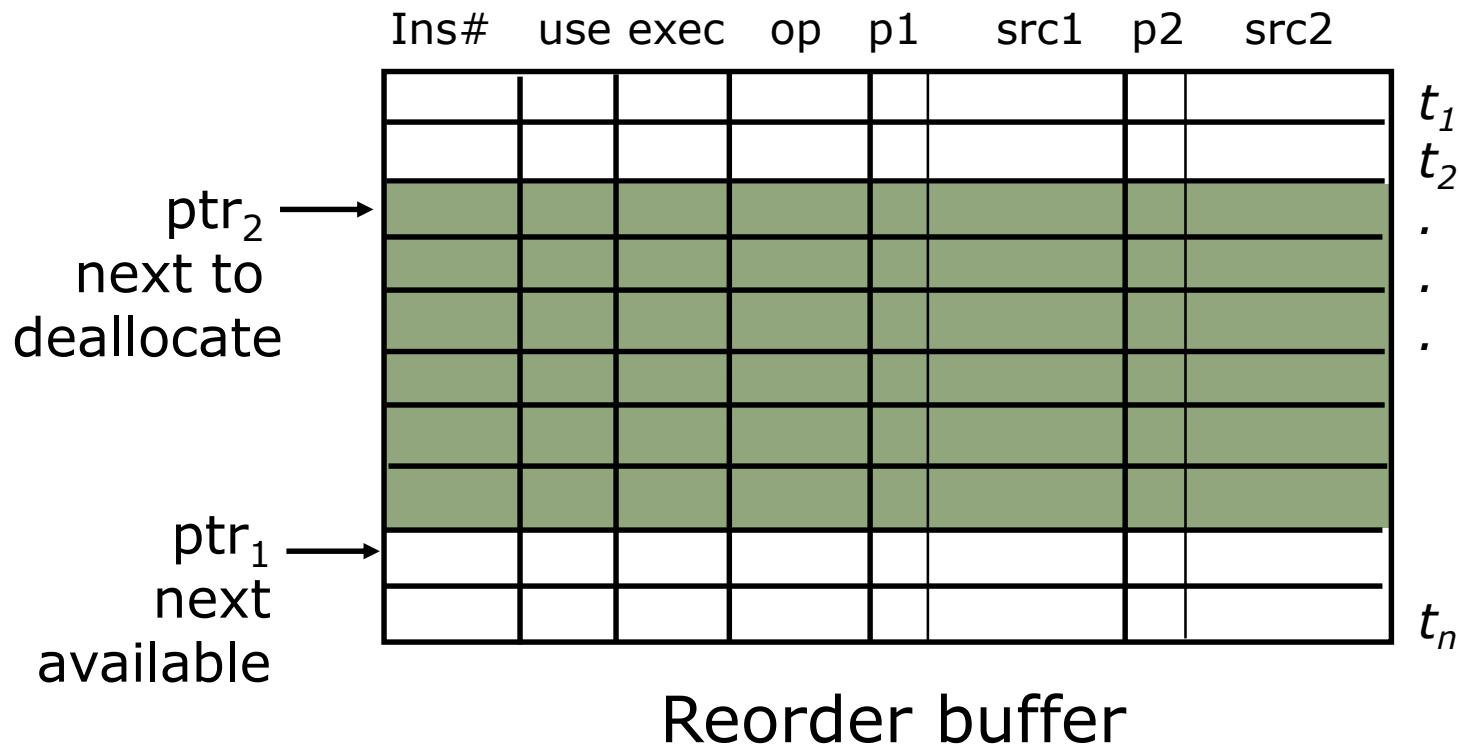
Reorder buffer

t_1
 t_2
 t_3
 t_4
 t_5
-

- *Insert instruction in ROB*
 - *Issue instruction from ROB*
 - *Complete instruction*
 - *Empty ROB entry*

1	LD	F2,	34(R2)
2	LD	F4,	45(R3)
3	MULTD	F6,	F4,
4	SUBD	F8,	F2,
5	DIVD	F4,	F2,
6	ADDD	F10,	F4

Simplifying Allocation/Deallocation



Instruction buffer is managed circularly

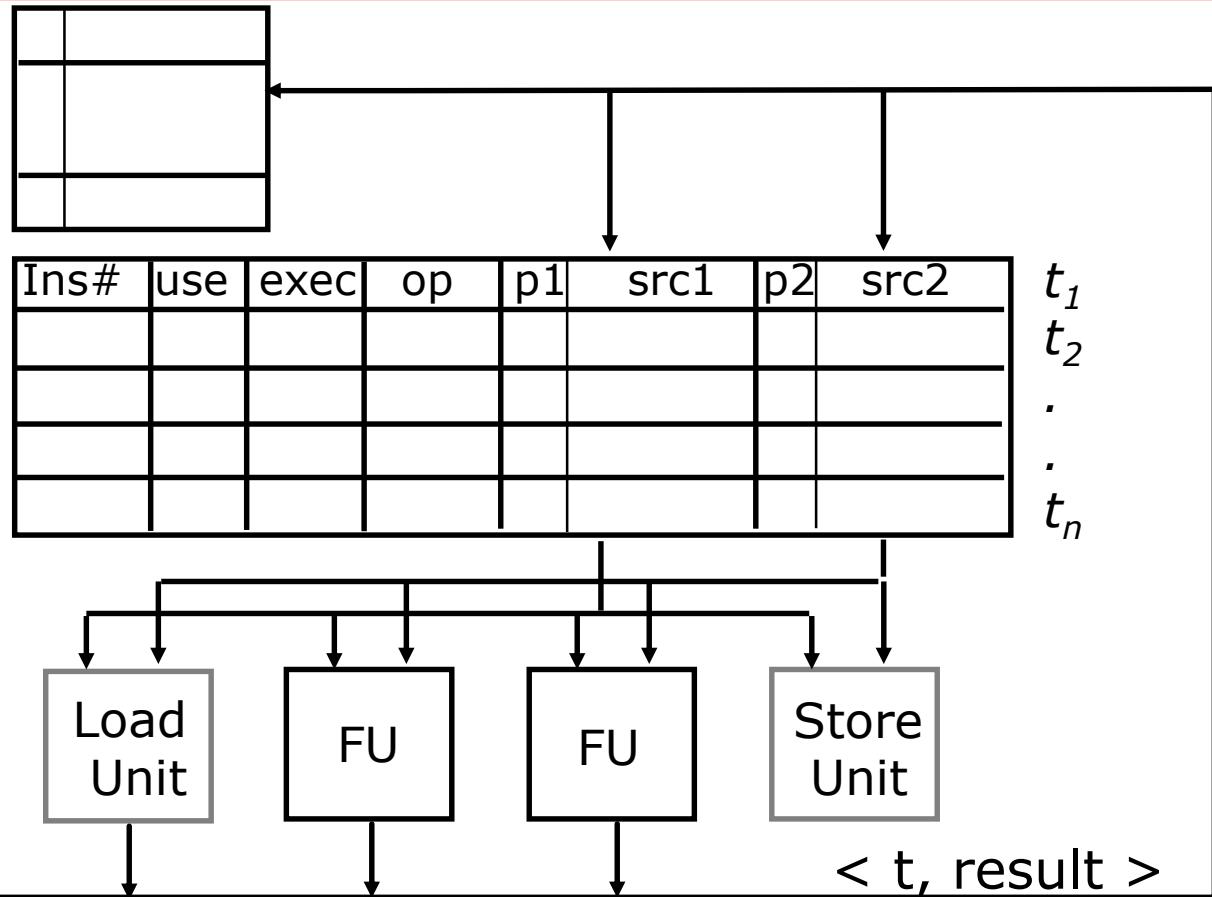
- Set “exec” bit when instruction begins execution
 - When an instruction completes its “use” bit is marked free
 - Increment ptr, only if the “use” bit is marked free

Data-Driven Execution

Renaming table & reg file

Reorder buffer

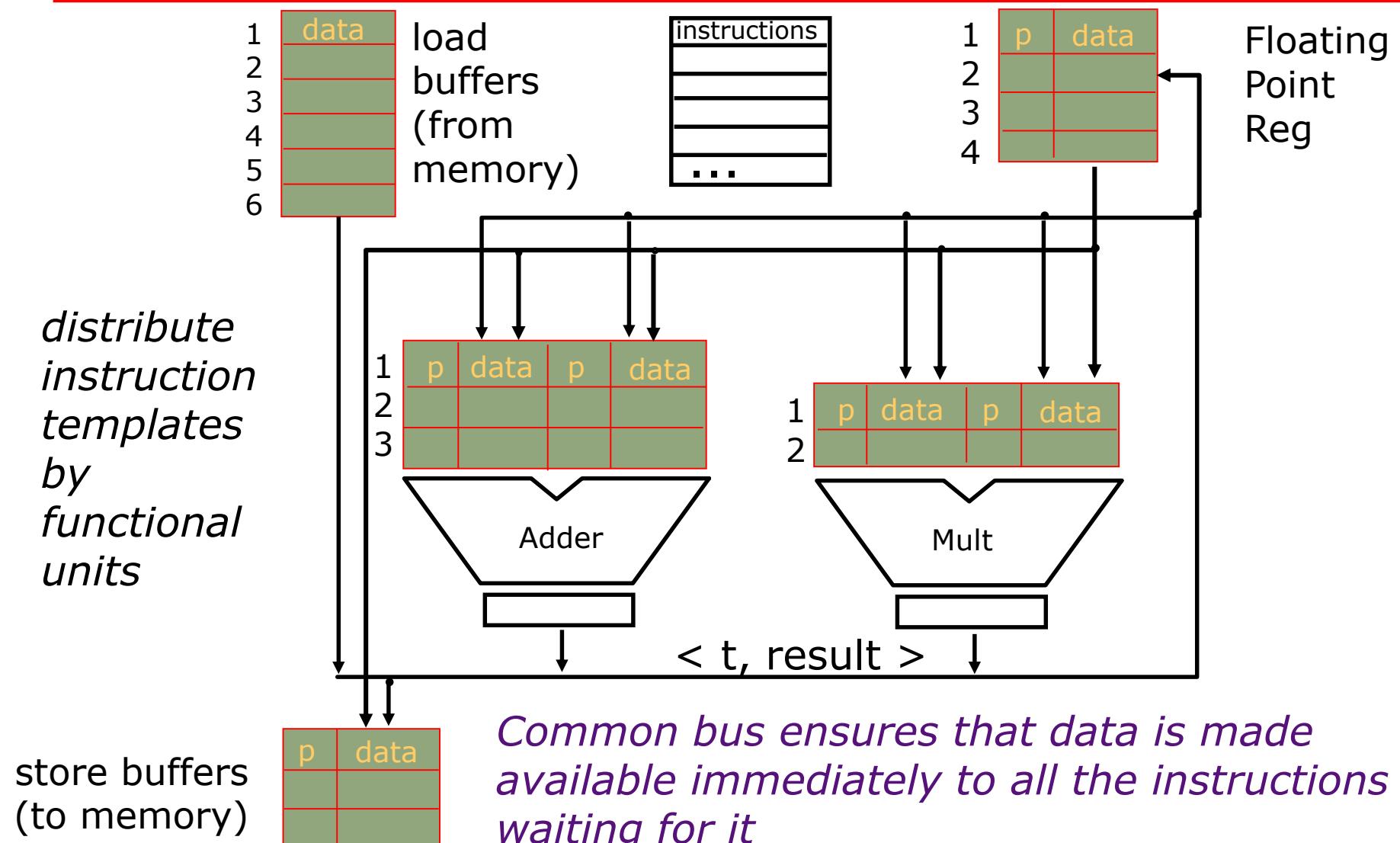
Replacing the tag by its value is an expensive operation



- Instruction template (i.e., tag t) is allocated by the Decode stage, which also stores the tag in the reg file
- When an instruction completes, its tag is deallocated

IBM 360/91 Floating Point Unit

R. M. Tomasulo, 1967



Effectiveness?

Renaming and Out-of-order execution was first implemented in 1969 in IBM 360/91 but was effective only on a very small class of problems and thus did not show up in the subsequent models until mid-nineties.

Why?

1. Did not address the memory latency problem which turned out to be a much bigger issue than FU latency
2. Made exceptions imprecise

One more problem needed to be solved

Reminder: Precise Exceptions

Exceptions are relatively unlikely events that need special processing, but where adding explicit control flow instructions is not desired, e.g., divide by 0, page fault

Exceptions can be viewed as an implicit conditional subroutine call that is inserted between two instructions.

Therefore, it must appear as if the exception is taken between two instructions (say I_i and I_{i+1})

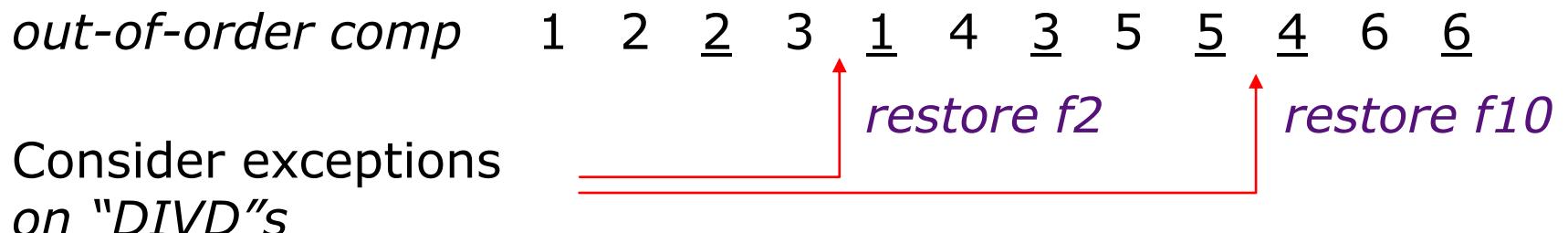
- the effect of all instructions up to and including I_i is complete
- no effect of any instruction after I_i has taken place

The handler either aborts the program or restarts it at I_{i+1} .

Effect on Exceptions

Out-of-order Completion

I_1	DIVD	f6,	f6,	f4
I_2	LD	f2,	45(r3)	
I_3	MULTD	f0,	f2,	f4
I_4	DIVD	f8,	f6,	f2
I_5	SUBD	f10,	f0,	f6
I_6	ADDD	f6,	f8,	f2



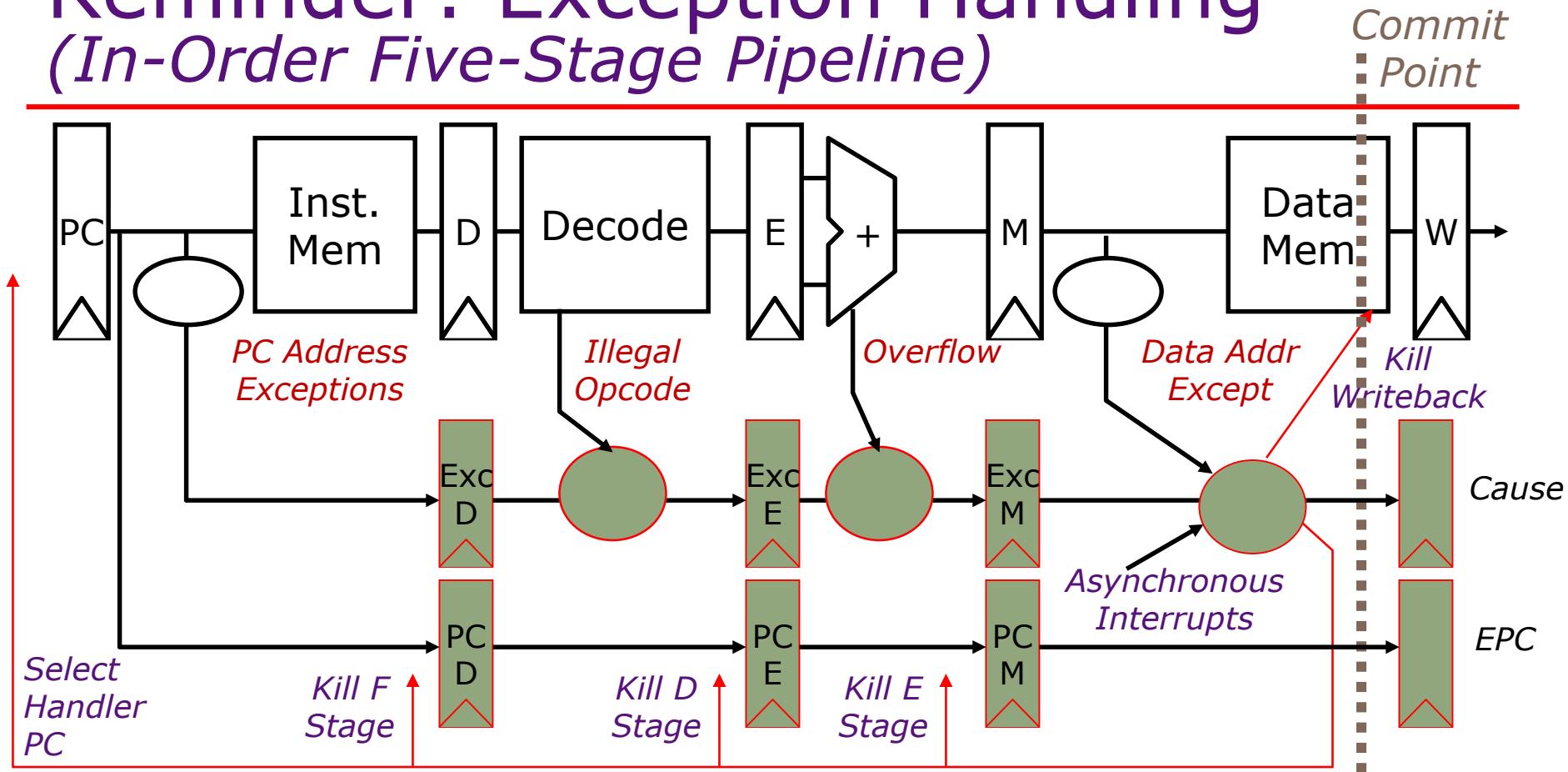
Precise exceptions are difficult to implement at high speed

- want to start execution of later instructions before exception checks finished on earlier instructions

Exceptions

- Exceptions create a dependence on the value of the next PC
- *Options for handling this dependence:*
 - Stall
 - Bypass
 - Find something else to do
 - Change the architecture
 - Speculate!
- *How can we handle rollback on mis-speculation?*
- Note: earlier exceptions must override later ones

Reminder: Exception Handling (In-Order Five-Stage Pipeline)

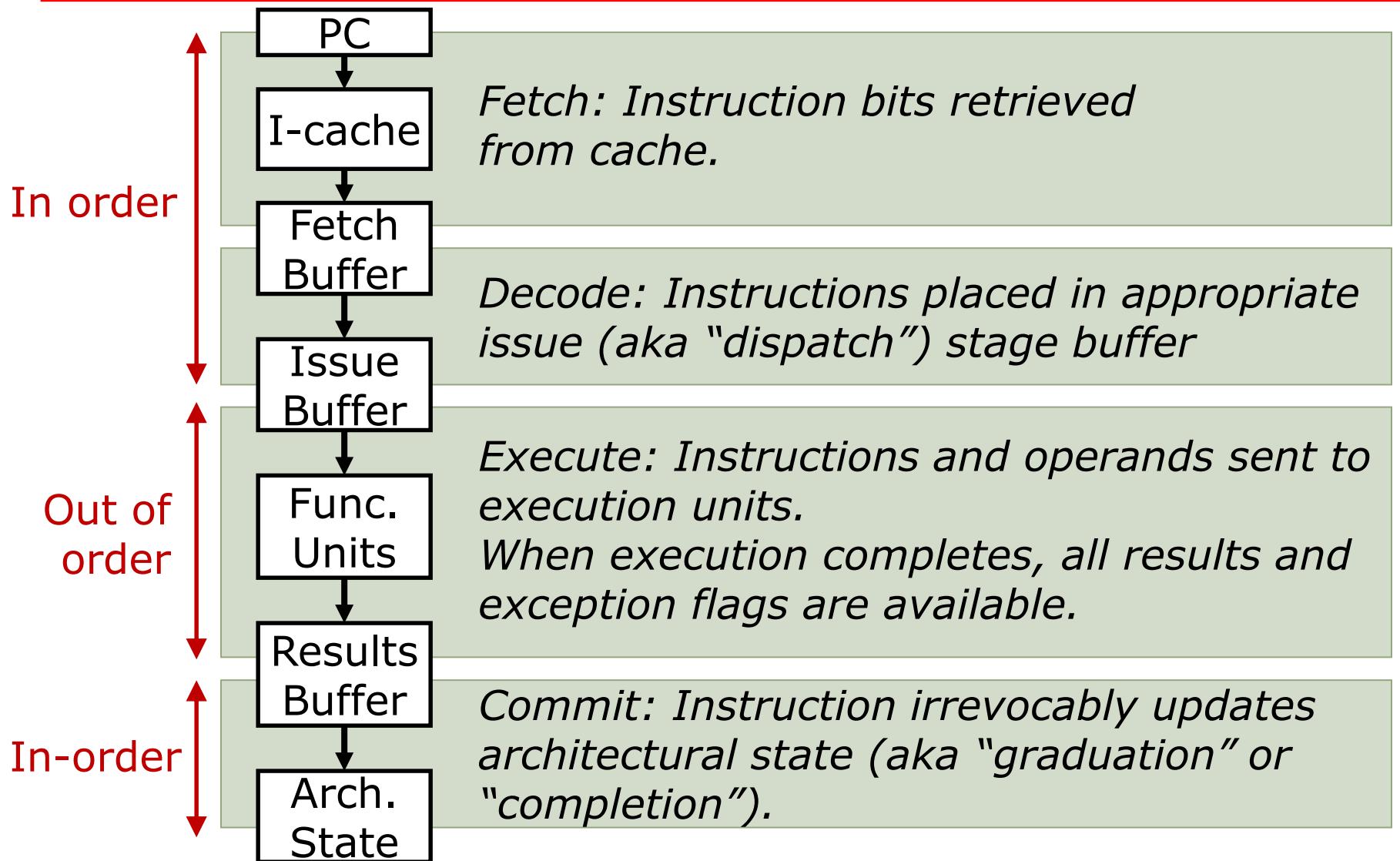


Hold exception flags in pipeline until commit point (M stage)

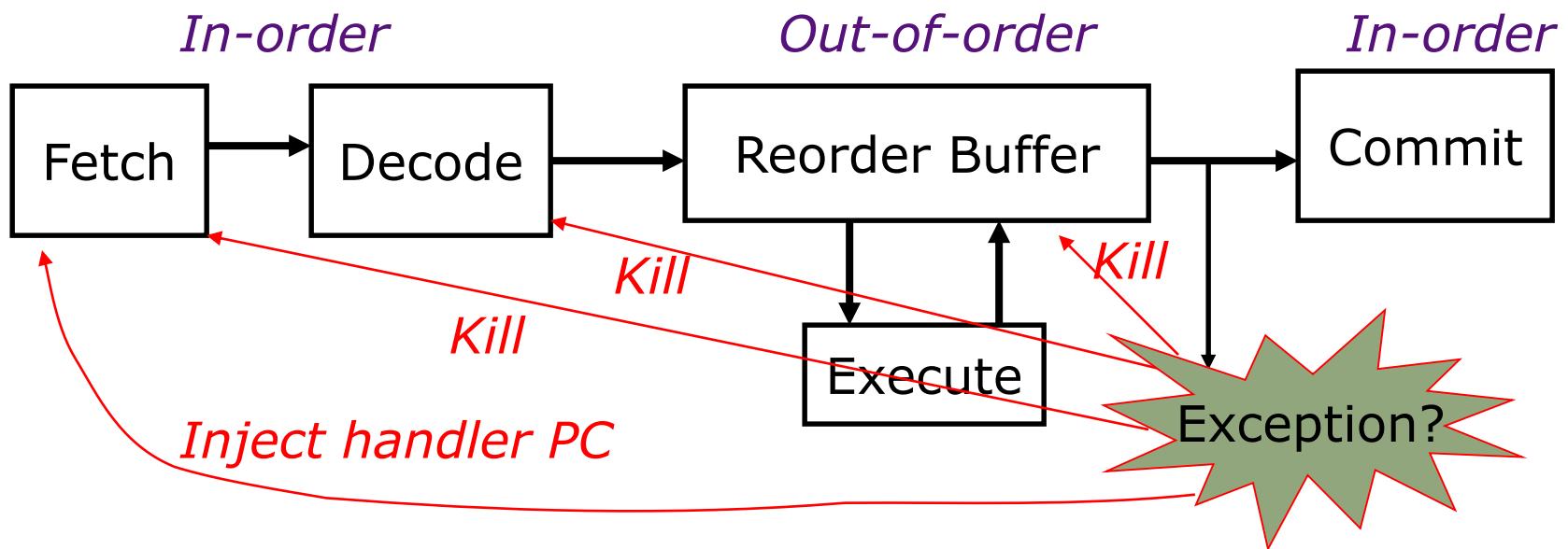
- If exception at commit:
 - update Cause/EPC registers
 - kill all stages
 - fetch at handler PC

Inject external interrupts at commit point

Phases of Instruction Execution



In-Order Commit for Precise Exceptions



- Instructions fetched and decoded into instruction reorder buffer in-order
- Execution is out-of-order (\Rightarrow out-of-order completion)
- *Commit* (write-back to architectural state, i.e., regfile & memory) is in-order

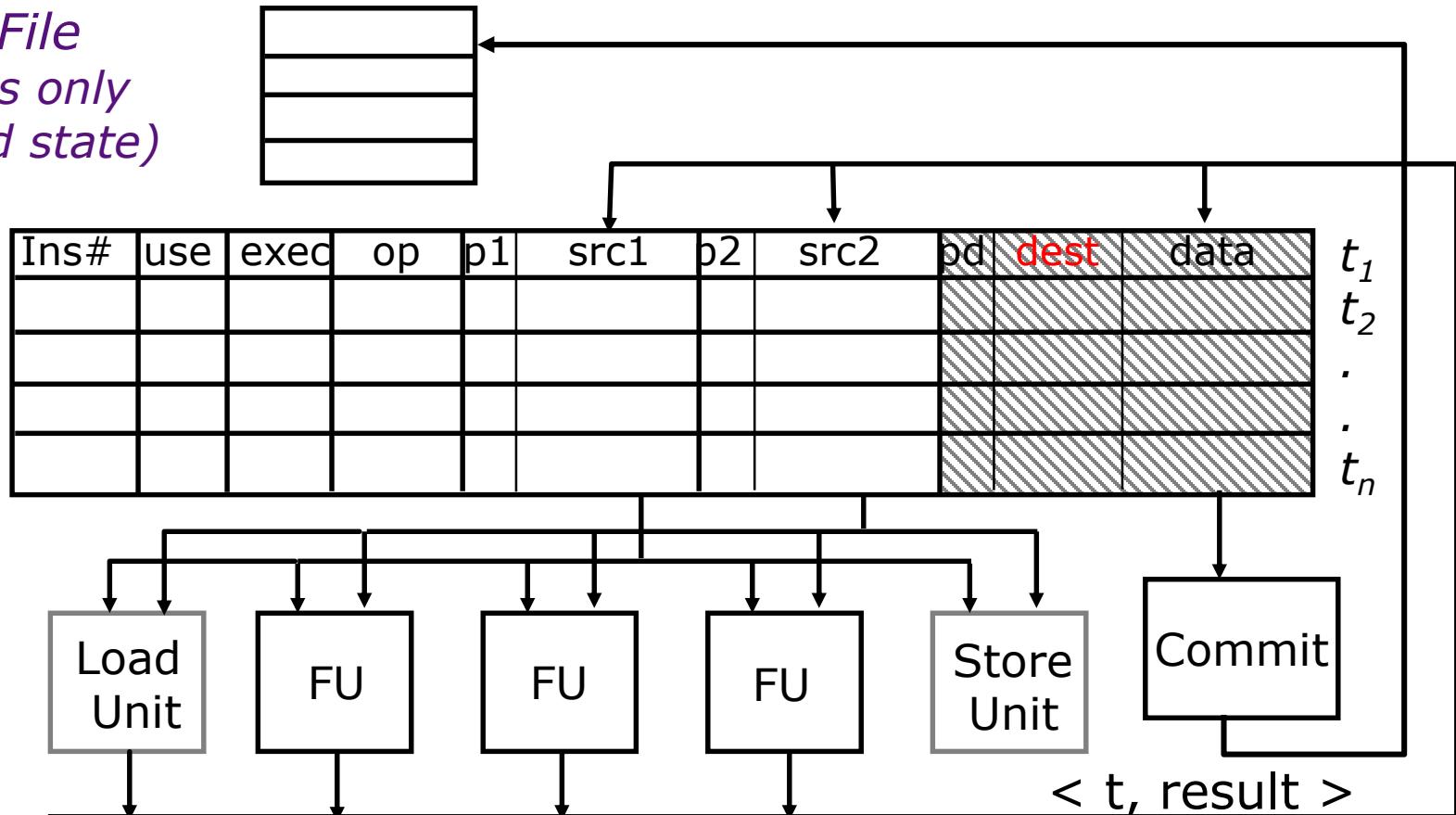
Temporary storage needed to hold results before commit (shadow registers and store buffers)

Extensions for Precise Exceptions

- add $\langle pd, dest, data, cause \rangle$ fields in the instruction template
 - commit instructions to reg file and memory in program order \Rightarrow buffers can be maintained circularly
 - on exception, clear reorder buffer by resetting $ptr_1 = ptr_2$
(stores must wait for commit before updating memory)

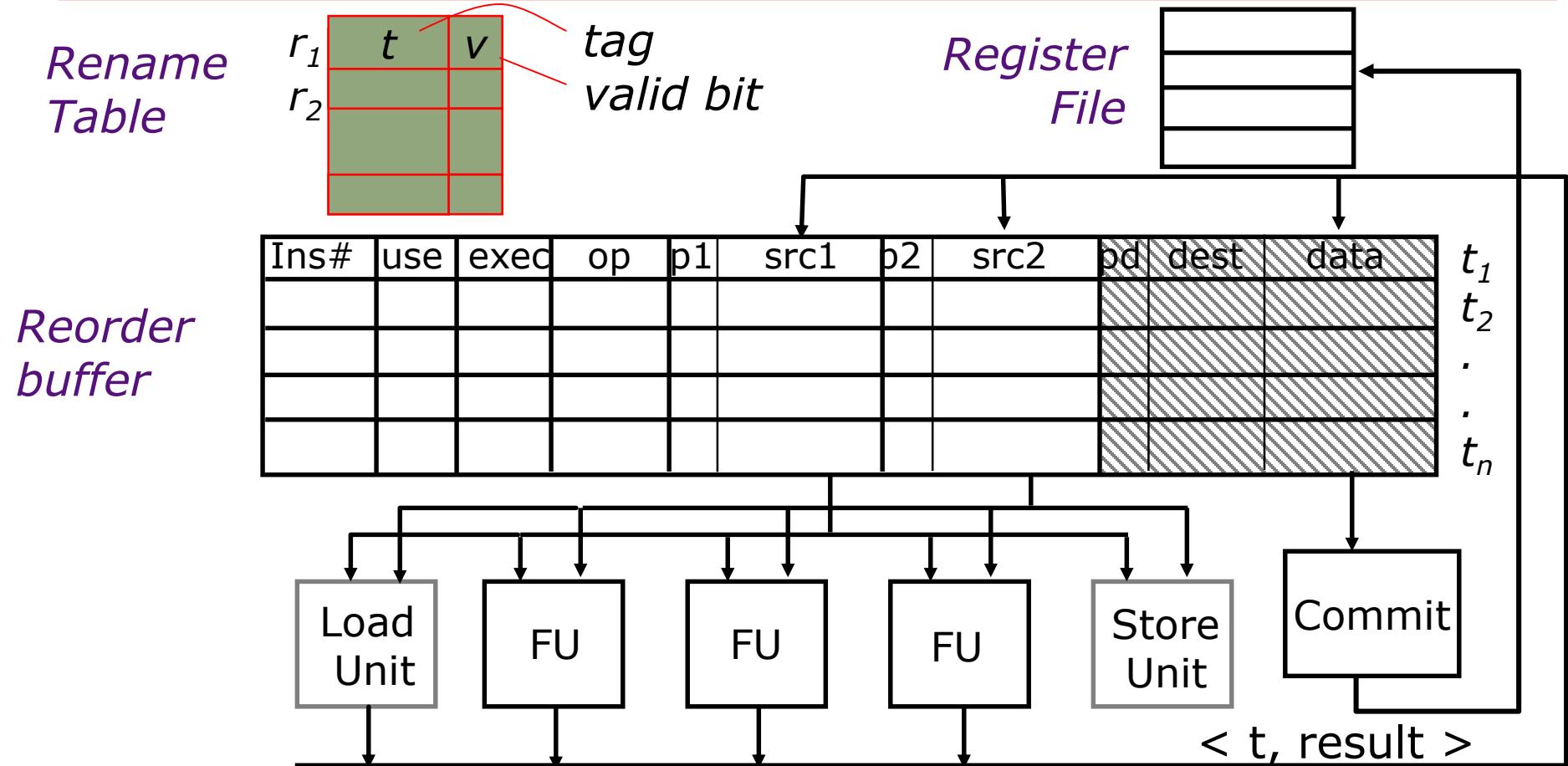
Rollback and Renaming

*Register File
(now holds only
committed state)*



Register file does not contain renaming tags any more.
How does the decode stage find the tag of a source register?

Renaming Table



Renaming table is a cache to speed up register name lookup.
It needs to be cleared after each exception taken.
When else are valid bits cleared?

Physical Register Files

- Reorder buffers are space inefficient – a data value may be stored in multiple places in the reorder buffer
- Idea: Keep all data values in a physical register file
 - Tag represents the name of the data value and name of the physical register that holds it
 - Reorder buffer contains only tags

Thus, 64-bit data values may be replaced by 8-bit tags for a 256-element physical register file

More on this in later lectures ...

Branch Penalty

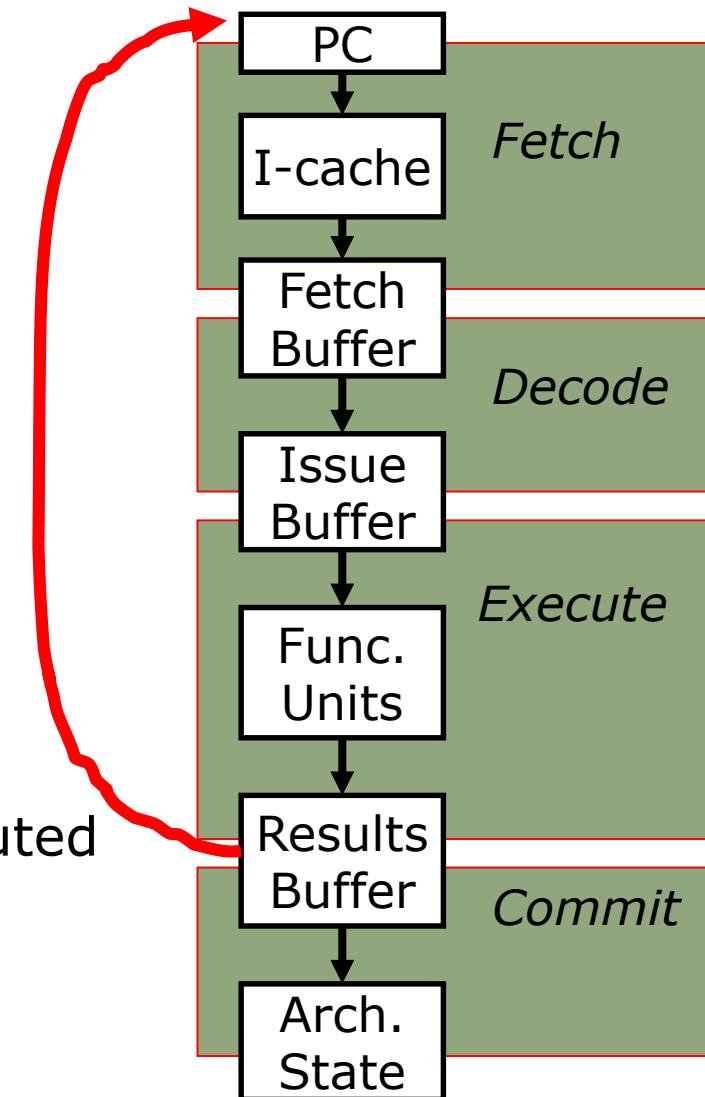
*How many instructions
need to be killed on a
misprediction?*

Modern processors may
have > 10 pipeline stages
between nextPC calculation
and branch resolution !

Next fetch
started

Branch executed

Next lecture:
Branch prediction &
Speculative execution

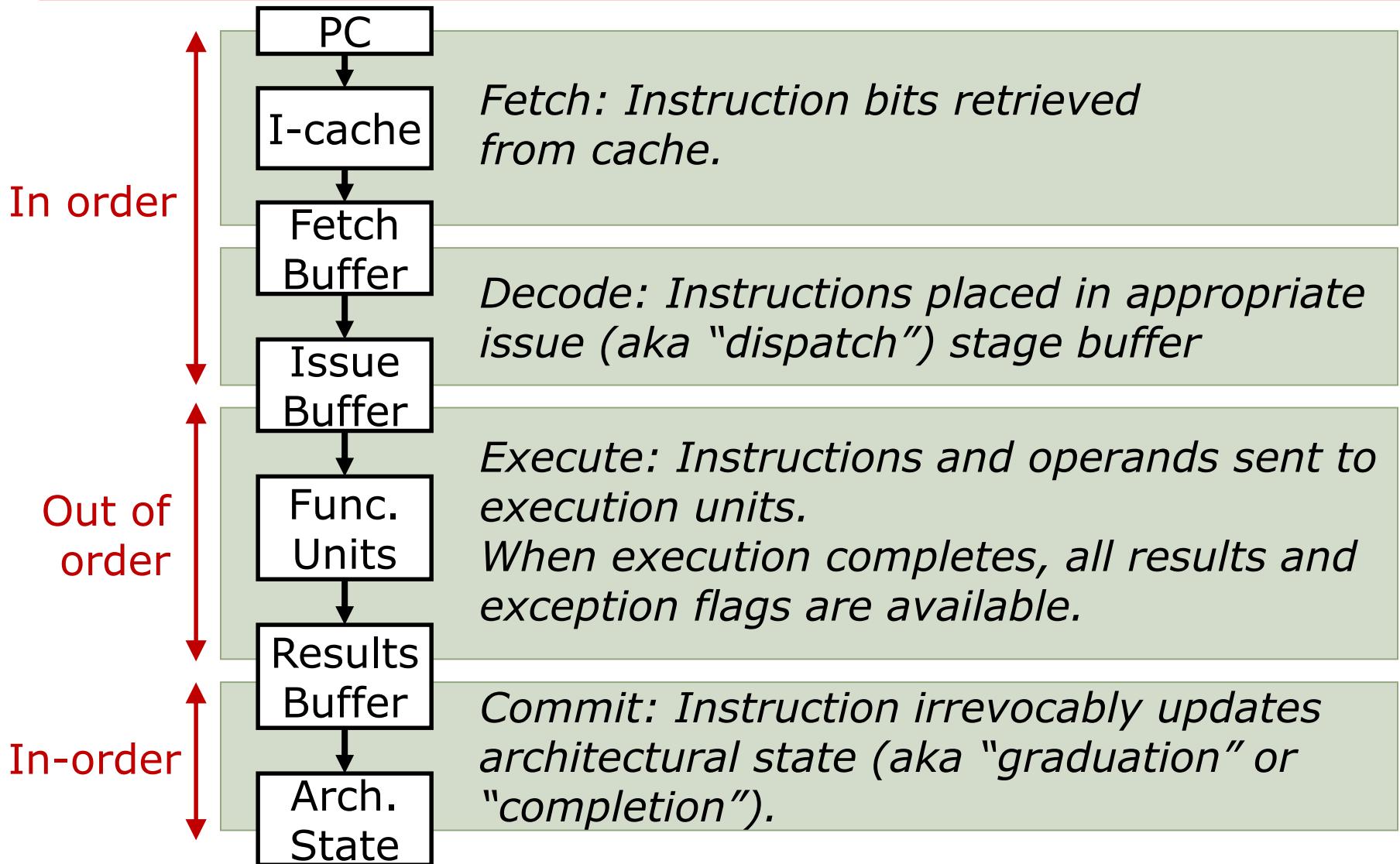


Branch Prediction

Daniel Sanchez

Computer Science and Artificial Intelligence Laboratory
M.I.T.

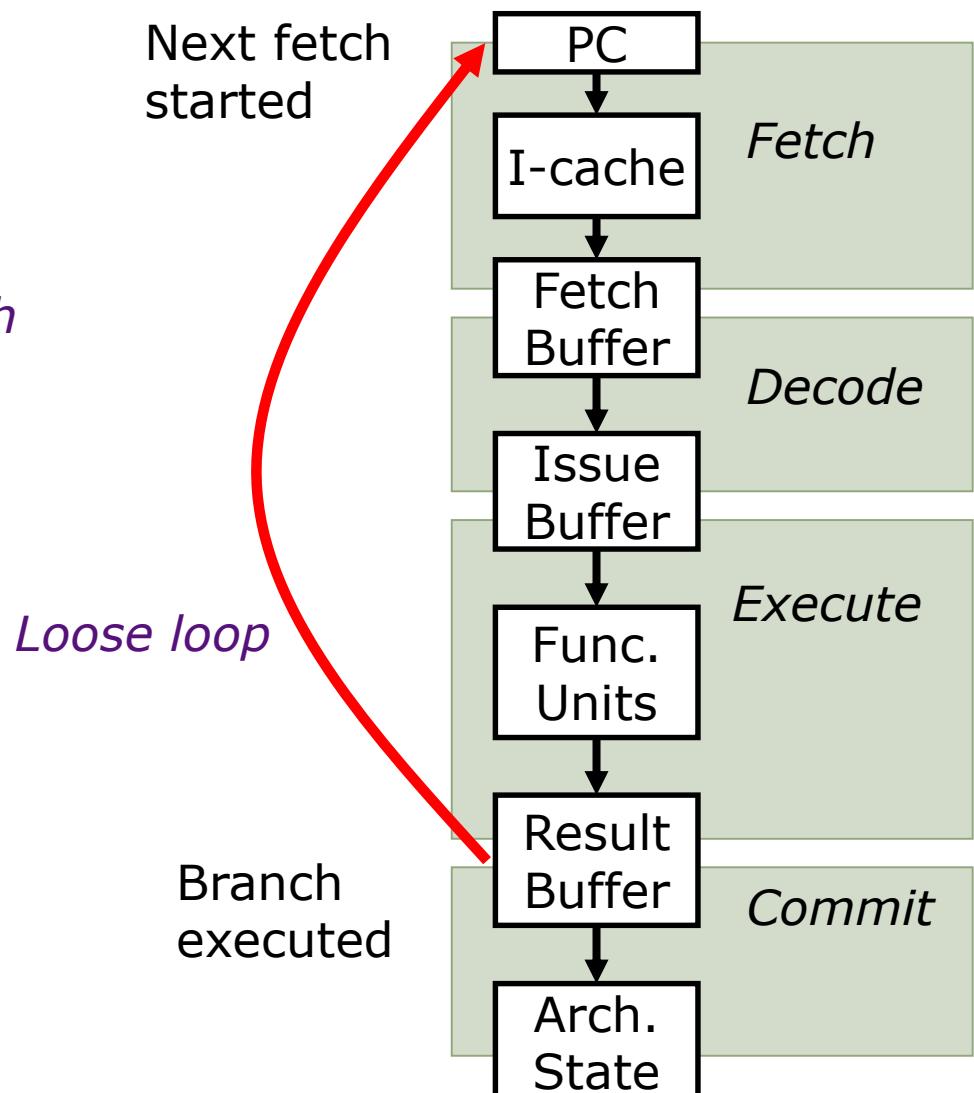
Reminder: Phases of Instruction Execution



Control Flow Penalty

*Modern processors may have
> 10 pipeline stages between
next PC calculation and branch
resolution!*

*How much work is lost if
pipeline doesn't follow
correct instruction flow?*



Average Run-Length between Branches

Average dynamic instruction mix of SPEC CPU 2017
[Limaye and Adegbiya, ISPASS'18]:

	SPECint	SPECfp
Branches	19 %	11 %
Loads	24 %	26 %
Stores	10 %	7 %
Other	47 %	56 %

SPECint17: *perlbench, gcc, mcf, omnetpp, xalancbmk, x264, deepsjeng, leela, exchange2, xz*

SPECfp17: *bwaves, cactus, lbm, wrf, pop2, imagick, nab, fotonik3d, roms*

What is the average run length between branches?

MIPS Branches and Jumps

Each instruction fetch depends on one or two pieces of information from the preceding instruction:

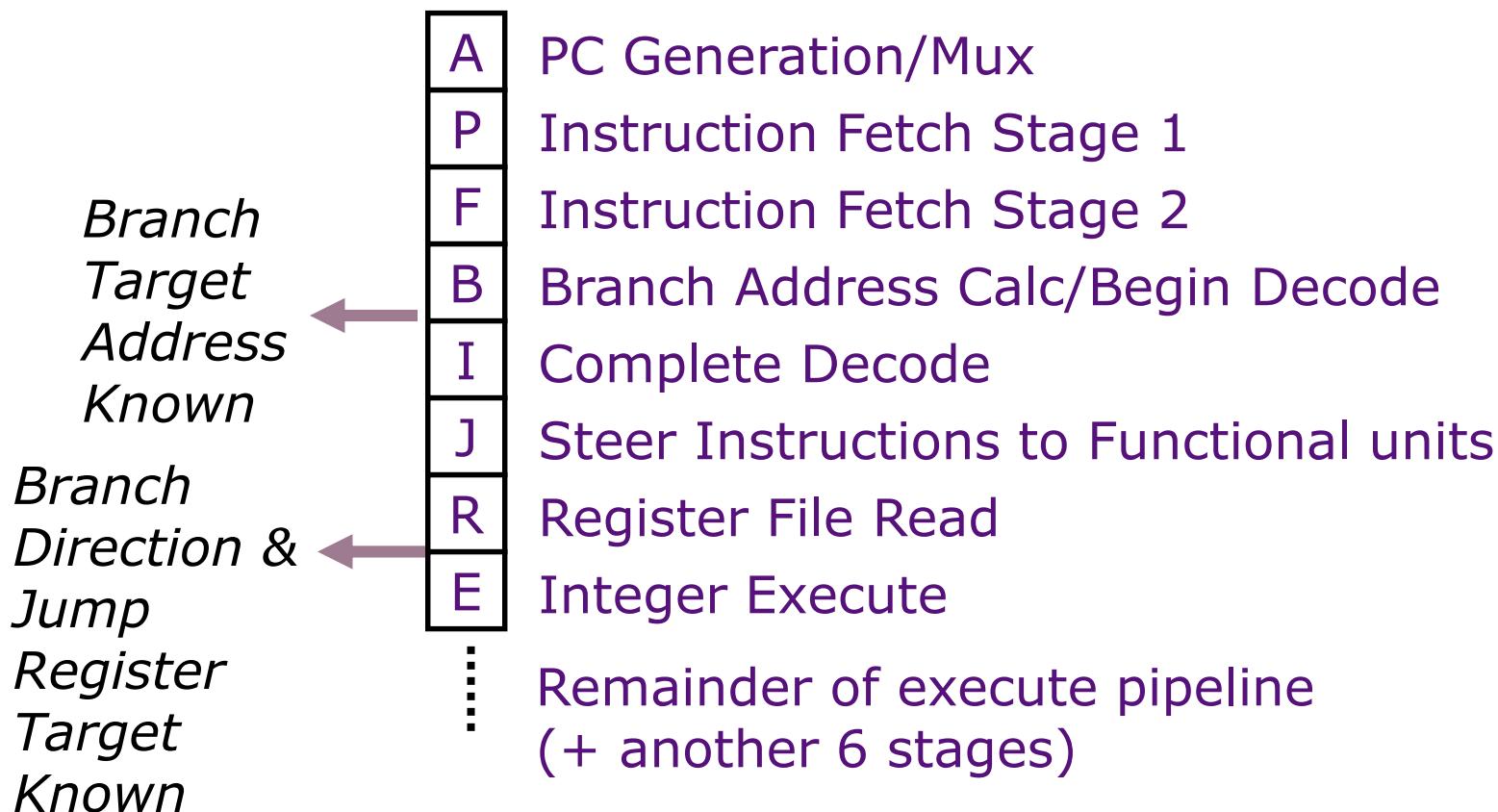
- 1) Is the preceding instruction a taken branch?
- 2) If so, what is the target address?

<i>Instruction</i>	<i>Taken known?</i>	<i>Target known?</i>
J		
JR		
BEQZ/BNEZ		

*Assuming zero detect on register read

Example Branch Penalties

UltraSPARC-III instruction fetch pipeline stages
(in-order issue, 4-way superscalar, 750MHz, 2000)



Reducing Control Flow Penalty

- Software solutions
 - *Eliminate branches* – *loop unrolling*
Increases run length between branches
 - *Reduce resolution time* – *instruction scheduling*
Compute the branch condition as early as possible
(of limited value)
- Hardware solutions
 - Bypass – usually results are used immediately
 - Change architecture – find something else to do
Delay slots – replace pipeline bubbles with useful work
(requires software cooperation)
 - *Speculate* – *branch prediction*
Speculative execution of instructions beyond the branch

Branch Prediction

Motivation:

Branch penalties limit performance of deeply pipelined processors

Modern branch predictors have high accuracy (>95%) and can reduce branch penalties significantly

Required hardware support:

Prediction structures:

- Branch history tables, branch target buffers, etc.

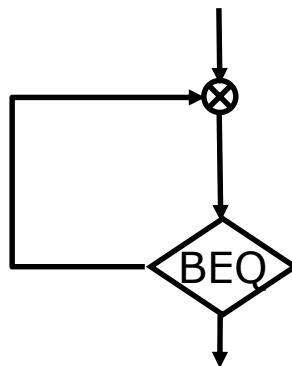
Mispredict recovery mechanisms:

- Keep result computation separate from commit
- Kill instructions following branch in pipeline
- Restore state to state following branch

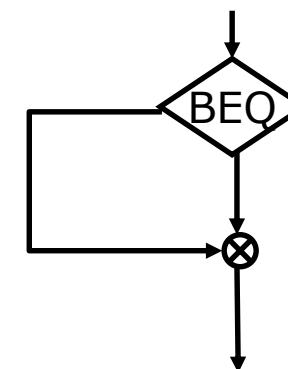
Static Branch Prediction

Overall probability a branch is taken is ~60-70% but:

*backward
90%*



*forward
50%*



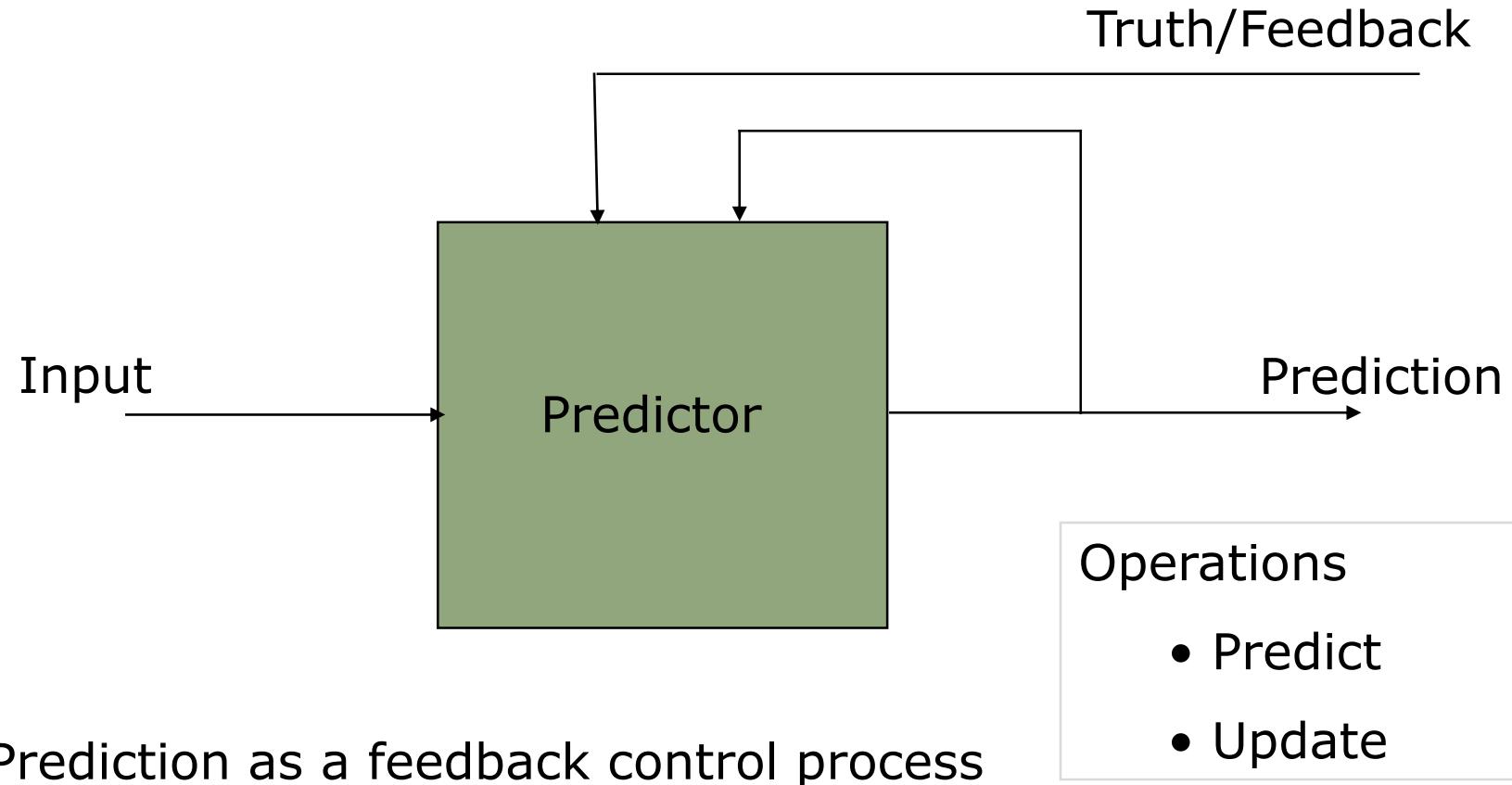
ISA can attach preferred direction semantics to branches,
e.g., Motorola MC88110

bne0 (*preferred taken*) beq0 (*not taken*)

ISA can allow arbitrary choice of statically predicted direction,
e.g., HP PA-RISC, Intel IA-64

typically reported as ~80% accurate

Dynamic Prediction



Dynamic Branch Prediction

Learning based on past behavior

Temporal correlation

The way a branch resolves may be a good predictor of the way it will resolve at the next execution

Spatial correlation

Several branches may resolve in a highly correlated manner (*a preferred path of execution*)

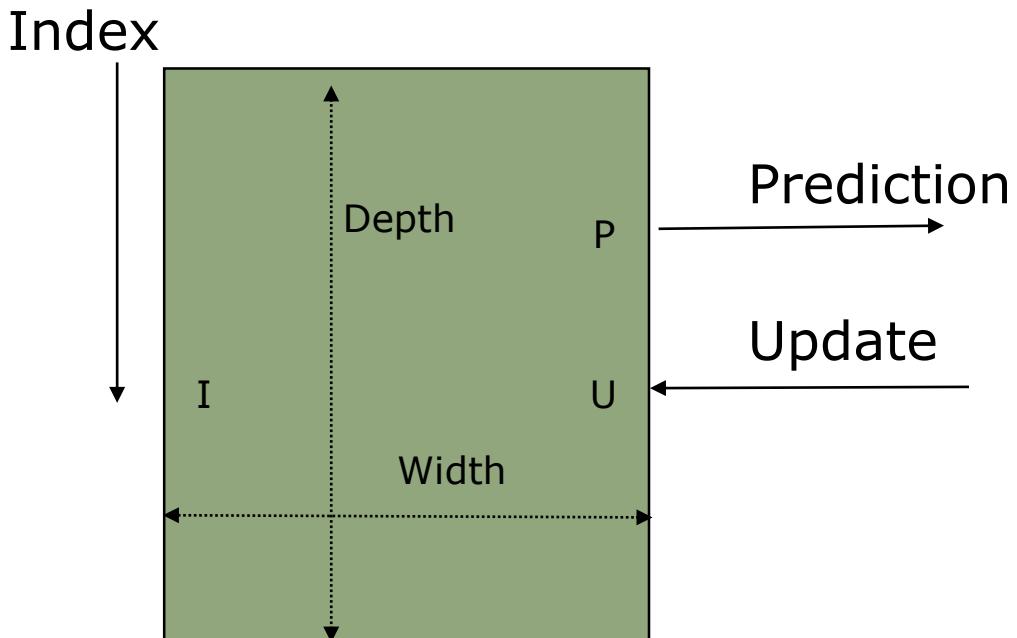
Predictor Primitive

Emer & Gloy, 1997

- Indexed table holding values

- Operations
 - Predict
 - Update

- Algebraic notation

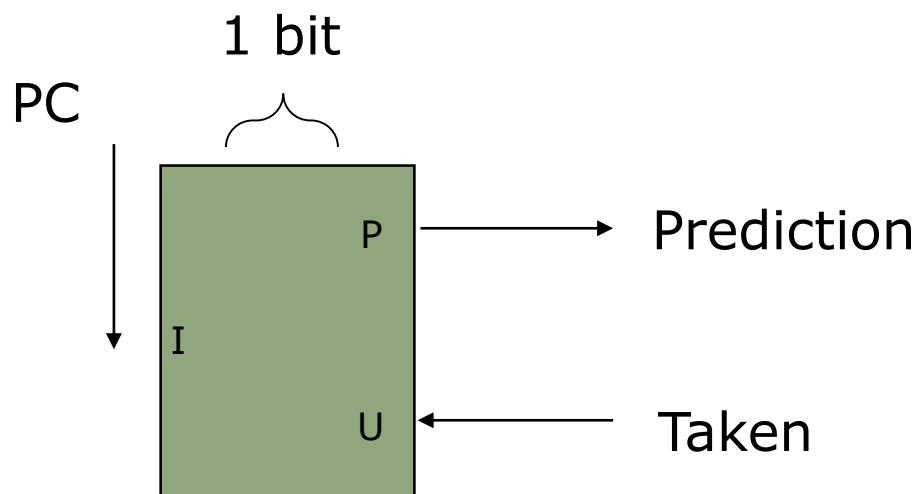


$$\text{Prediction} = P[\text{Width}, \text{Depth}](\text{Index}; \text{Update})$$

One-bit Predictor

aka Branch History Table (BHT)

Simple temporal prediction



$$A21064(PC; T) = P[1, 2K](PC; T)$$

What happens on loop branches?

Two-bit Predictor

Smith, 1981

- Use two bits per entry instead of one bit
- Manage them as a saturating counter:

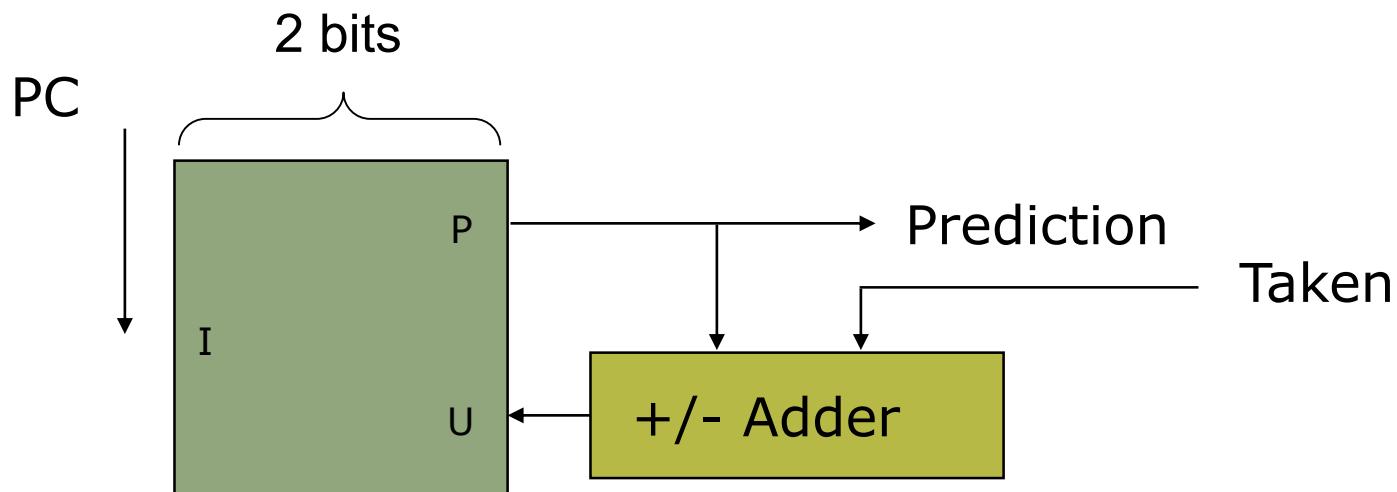
On not-taken ↓	↑ On taken	1	1	Strongly taken
		1	0	Weakly taken
		0	1	Weakly not-taken
		0	0	Strongly not-taken

- Direction prediction changes only after two wrong predictions

How many mispredictions per loop? _____

Two-bit Predictor

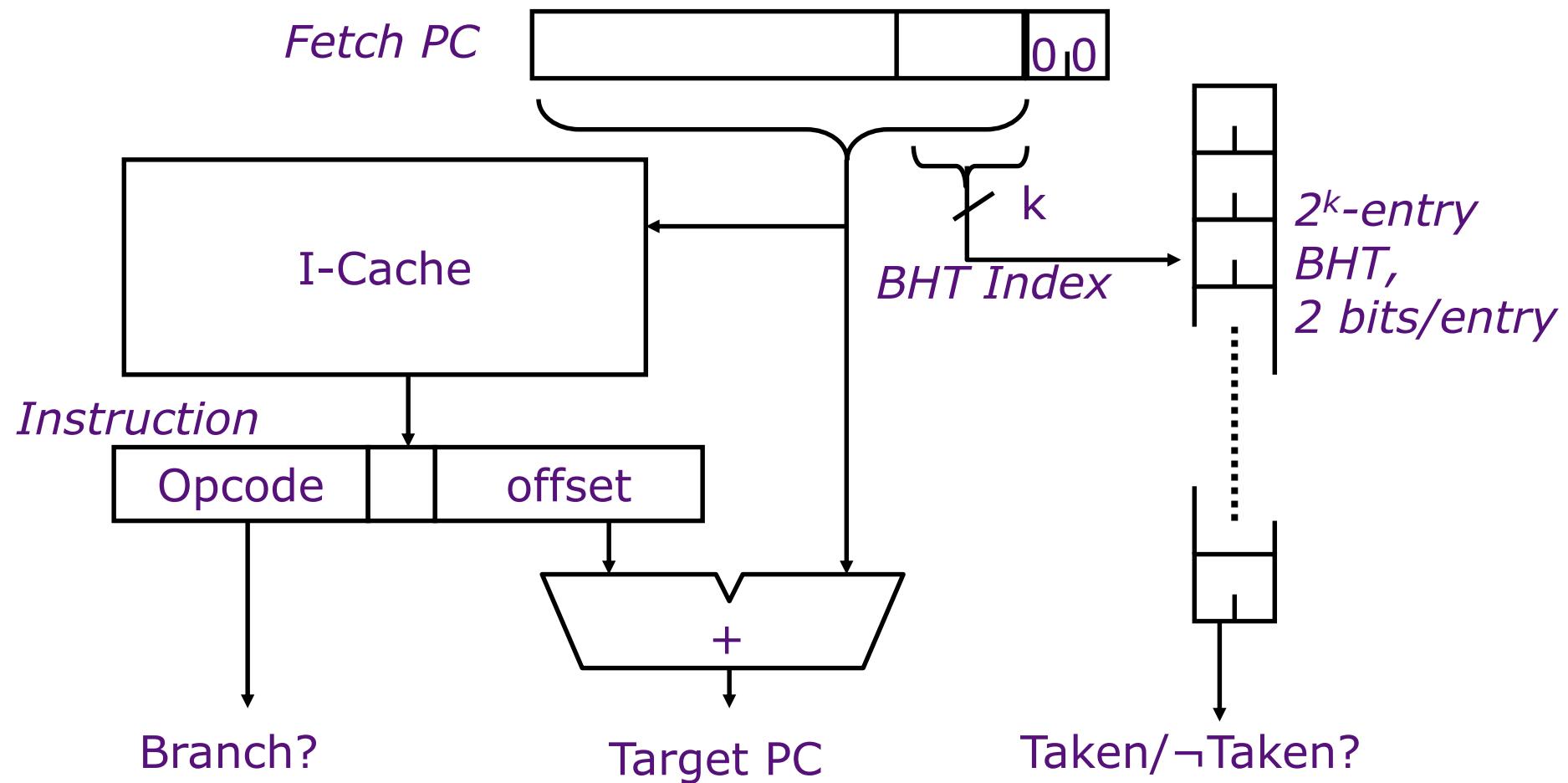
Smith, 1981



$$\text{Counter}[W,D](I; T) = P[W, D](I; \text{ if } T \text{ then } P+1 \text{ else } P-1)$$

$$A21164(\text{PC}; T) = \text{MSB}(\text{Counter}[2, 2K](\text{PC}; T))$$

Branch History Table



4K-entry BHT, 2 bits/entry, ~80-90% correct predictions

Exploiting Spatial Correlation

Yeh and Patt, 1992

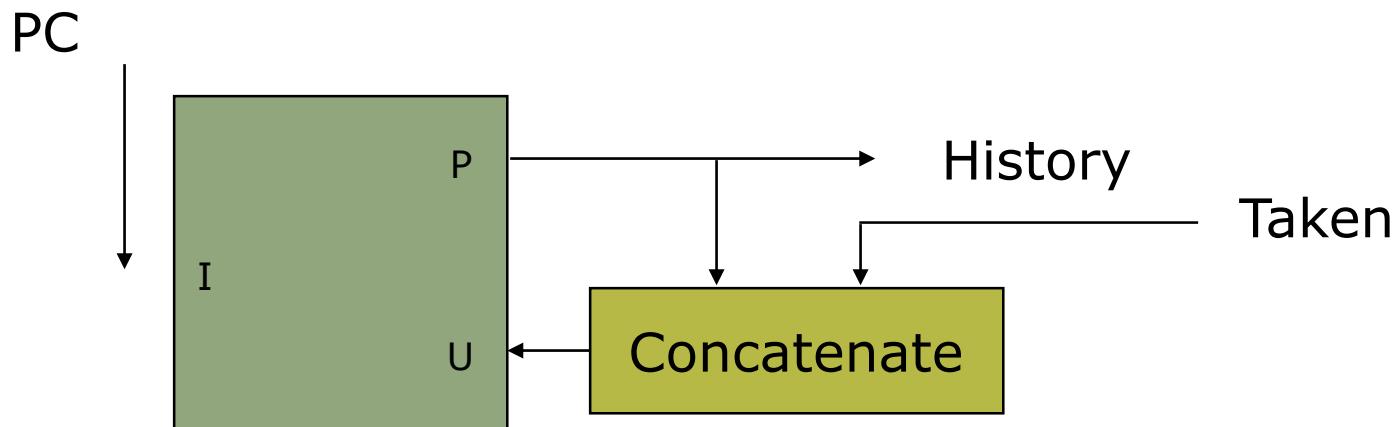
```
if (x[i] < 7) then  
    y += 1;  
if (x[i] < 5) then  
    c -= 4;
```

If first condition false, second condition also false

History register records the direction of the last N branches executed by the processor

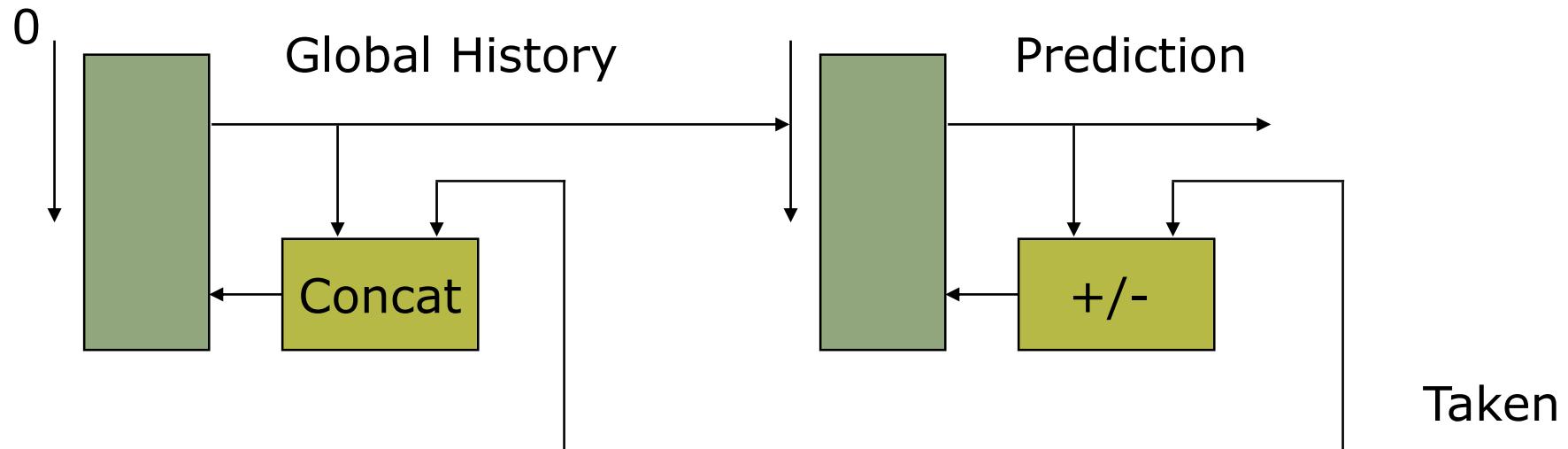
History Registers

aka *Pattern History Table (PHT)*



$$\text{History}(\text{PC}; \text{T}) = \text{P}(\text{PC}; \text{P} \parallel \text{T})$$

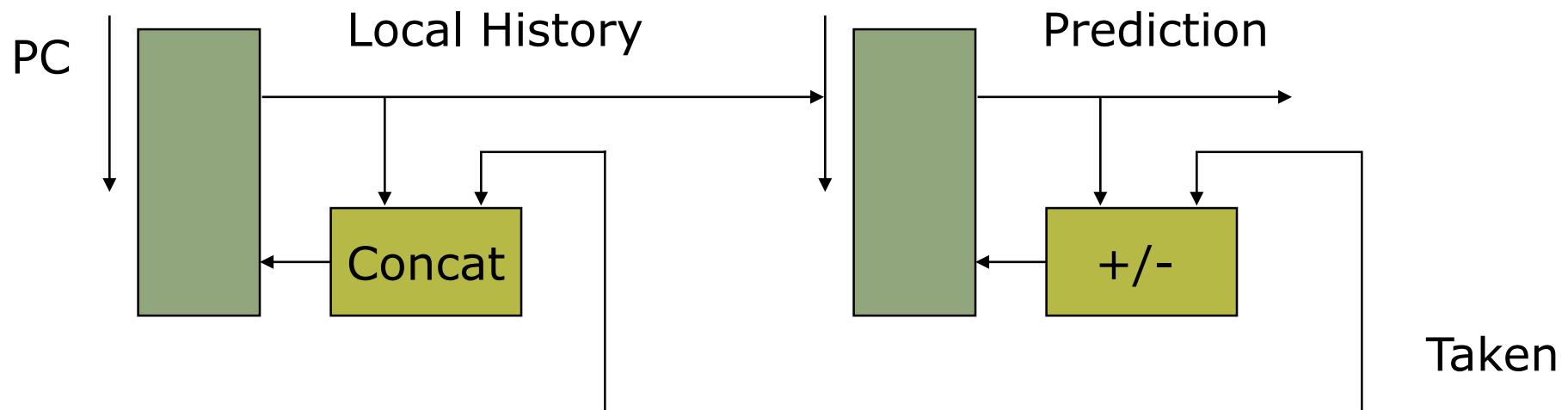
Global-History Predictor



$$GHist(;T) = MSB(Counter(History(0, T); T))$$

Can we take advantage of a pattern at a particular PC?

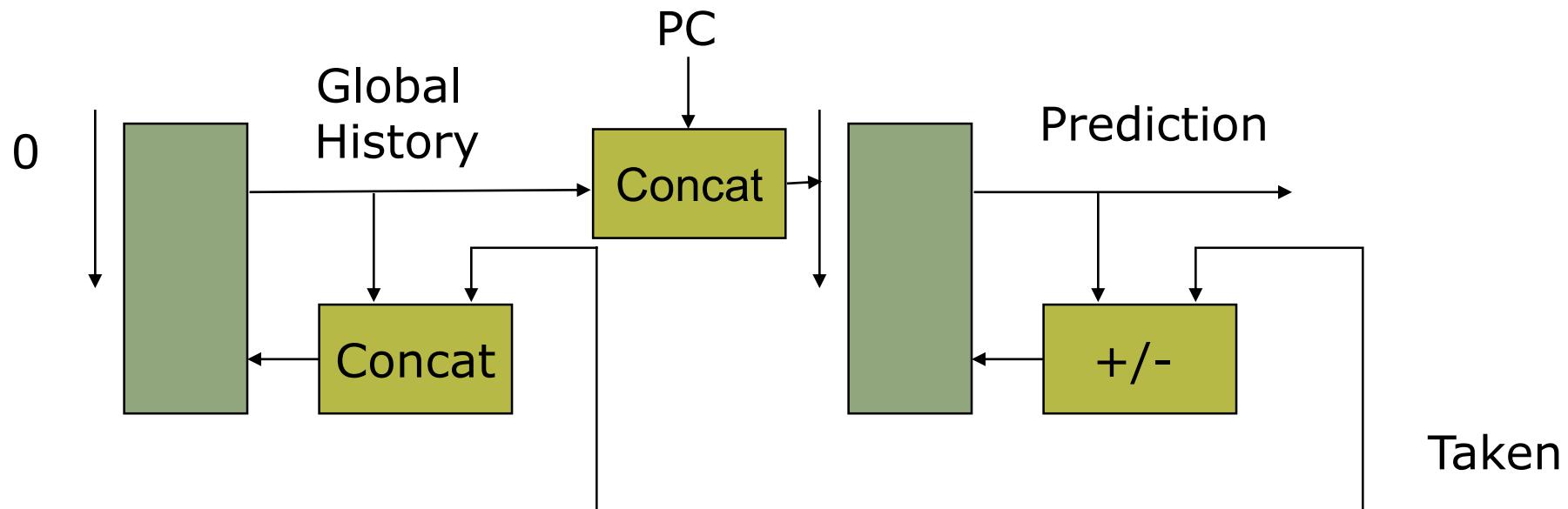
Local-History Predictor



$$\text{LHist}(\text{PC}; T) = \text{MSB}(\text{Counter}(\text{History}(\text{PC}; T); T))$$

Can we take advantage of the global pattern at a particular PC?

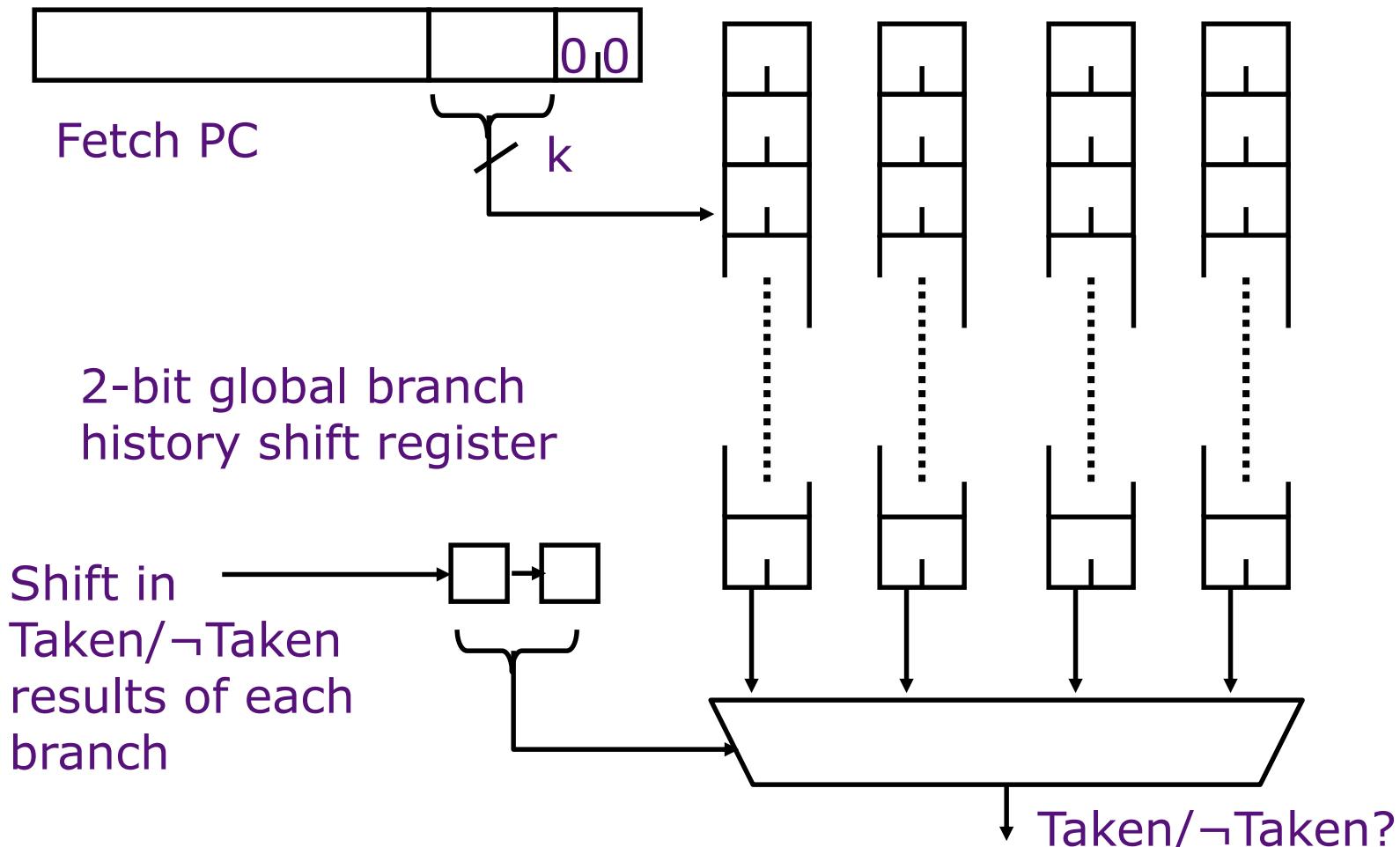
Global-History Predictor with Per-PC Counters



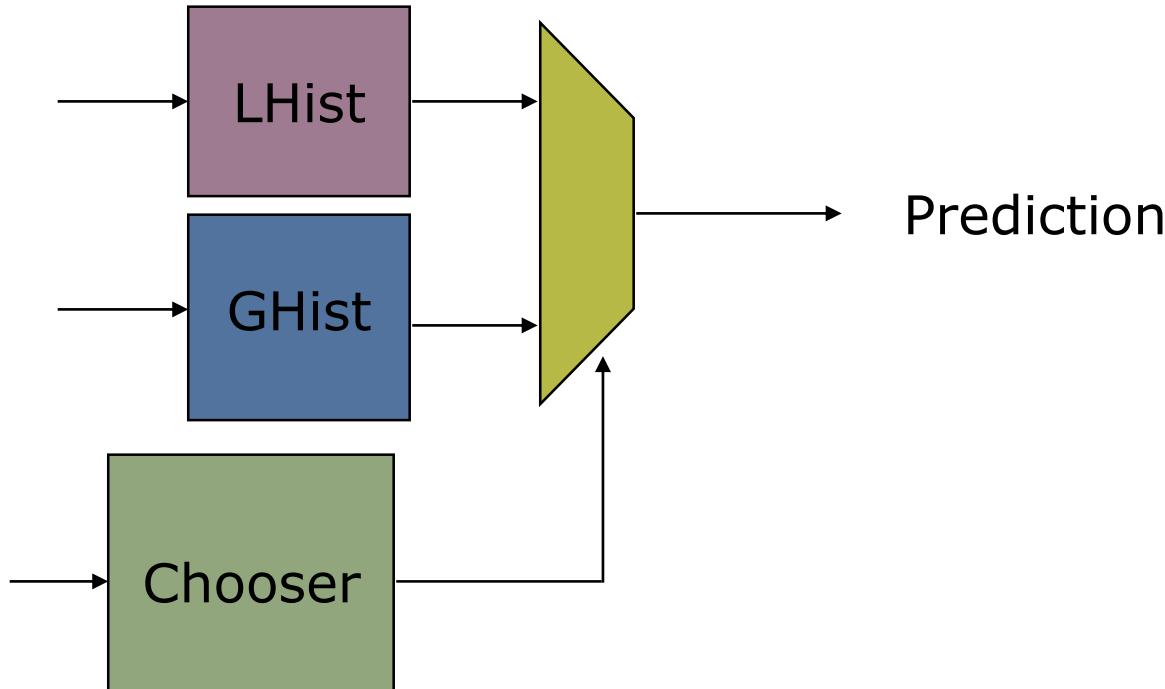
$$\text{GHistPA}(\text{PC}; T) = \text{MSB}(\text{Counter}(\text{History}(0; T) || \text{PC}; T))$$

Two-Level Branch Predictor (Pentium Pro, 1995)

Pentium Pro uses the result from the last two branches to select one of the four sets of BHT bits (~95% correct)



Choosing Predictors

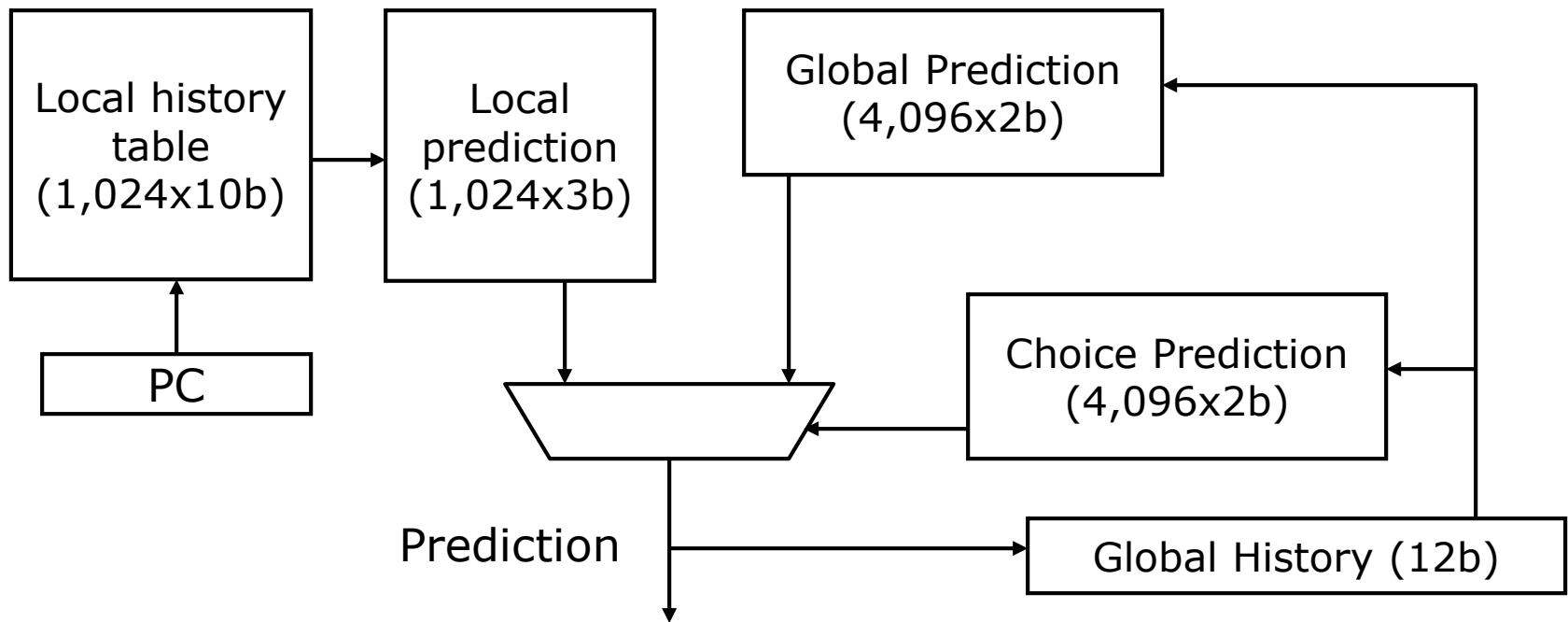


$$\text{Chooser} = \text{MSB}(P(\text{PC}; P + (A == T) - (B == T)))$$

or

$$\text{Chooser} = \text{MSB}(P(\text{GHist}(\text{PC}; T); P + (A == T) - (B == T)))$$

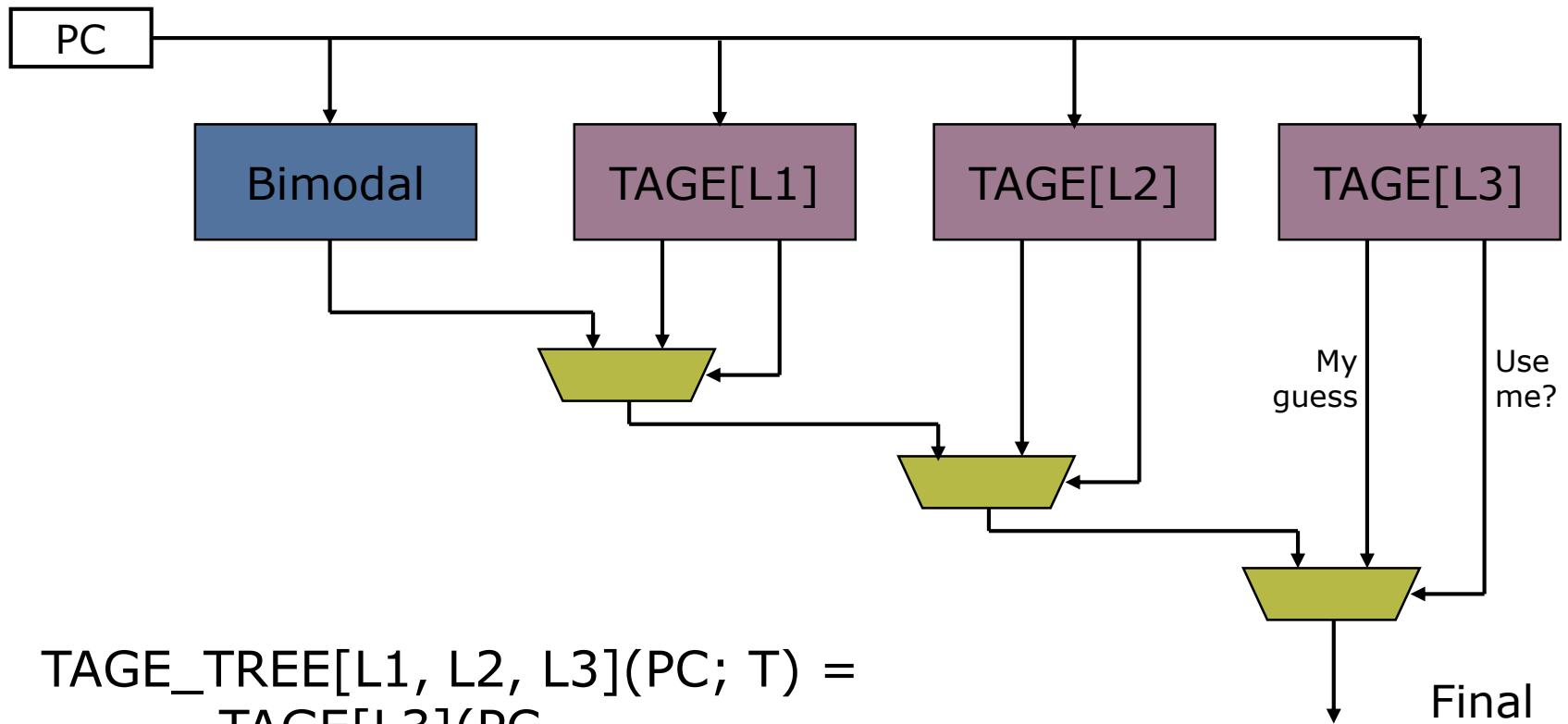
Tournament Branch Predictor *(Alpha 21264, 1996)*



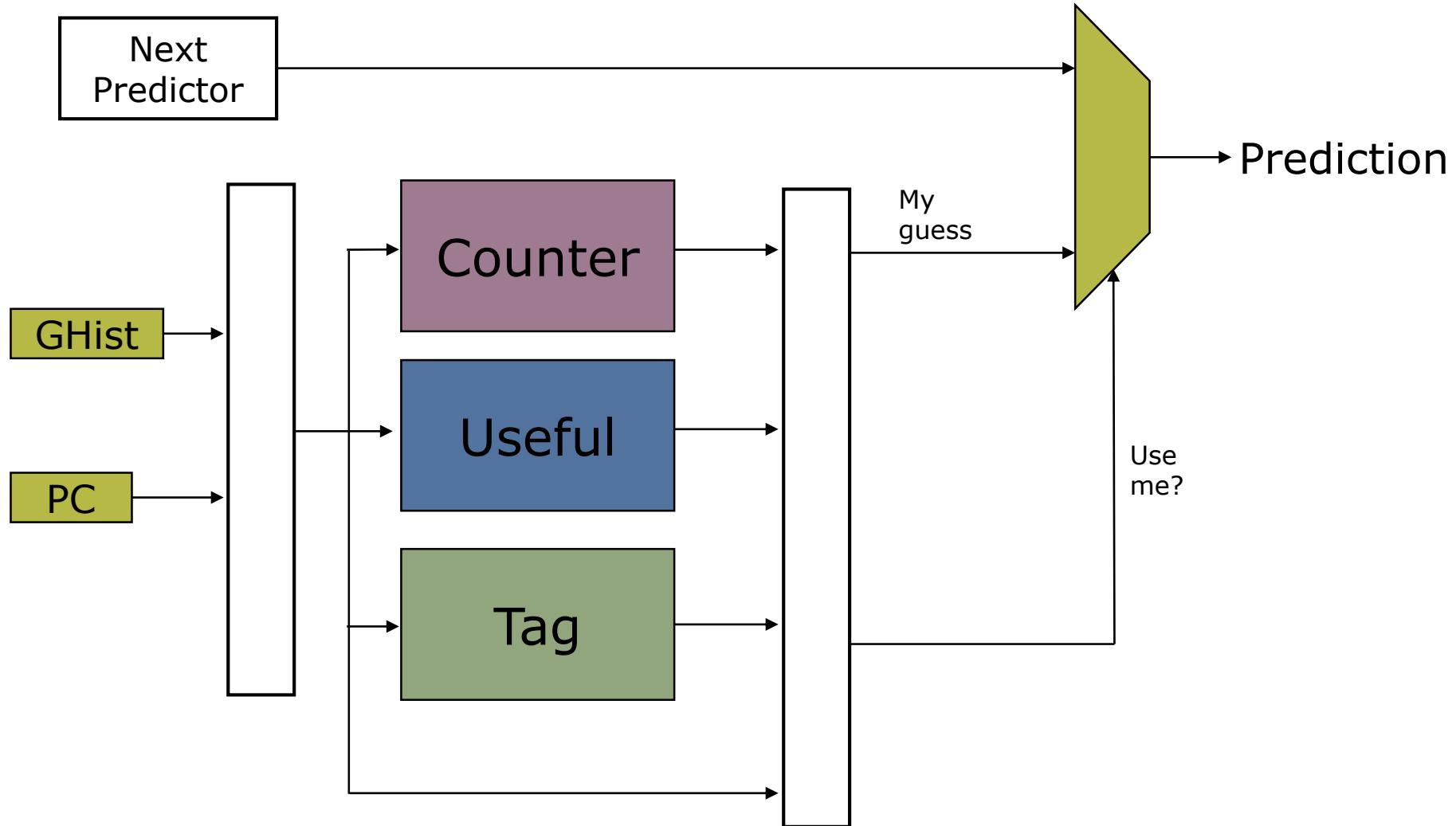
- Choice predictor learns whether best to use local or global branch history in predicting next branch
- Global history is speculatively updated but restored on mispredict
- Claim 90-100% success on range of applications

TAGE predictor

Seznec & Michaud, 2006


$$\begin{aligned} \text{TAGE_TREE[L1, L2, L3](PC; T)} = \\ & \text{TAGE[L3](PC,} \\ & \quad \text{TAGE[L2](PC,} \\ & \quad \quad \text{TAGE[L1](PC, Bimodal(PC; T)} \\ & \quad \quad \quad ; T) \quad ; T \quad ; T) \end{aligned}$$

TAGE component



TAGE predictor component

$\text{TAGE}[L](PC, \text{NEXT}; T) =$

$\text{idx} = \text{hash}(PC, \text{GHIST}[L](;T))$
 $\text{tag} = \text{hash}'(PC, \text{GHIST}[L](;T))$

$\text{TAGE.U} = \text{SA}(\text{idx}, \text{tag}; ((\text{TAGE} == T) \&\& (\text{NEXT} != T)) ? 1 : \text{SA})$
 $\text{TAGE.Counter} = \text{SA}(\text{idx}, \text{tag}; T ? \text{SA} + 1 : \text{SA} - 1)$

$\text{use_me} = \text{TAGE.U} \&\& \text{isStrong}(\text{TAGE.Counter})$
 $\text{TAGE} = \text{use_me} ? \text{MSB}(\text{TAGE.Counter}) : \text{NEXT}$

Notes:

SA is a set-associative structure

SA allocation occurs on mispredict (not shown)

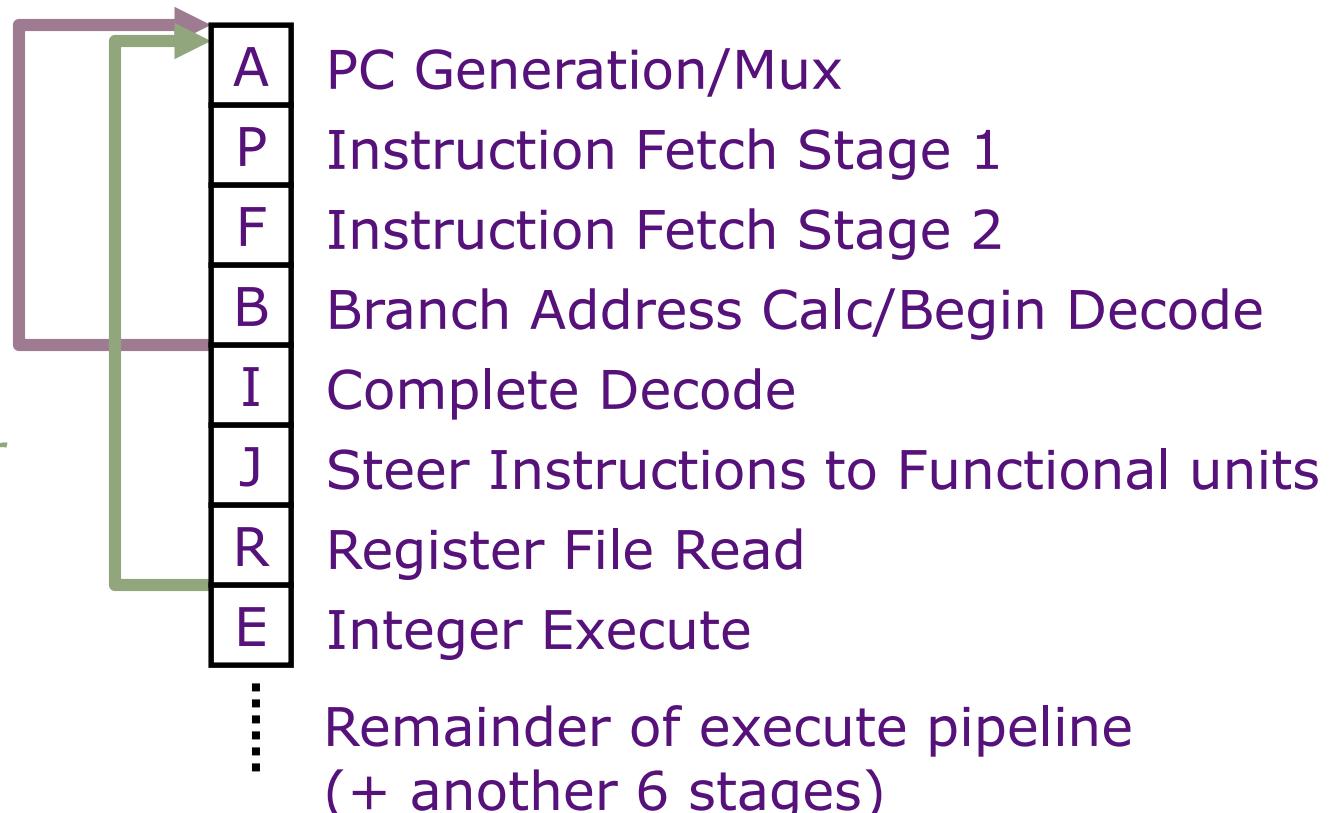
TAGE.U cleared on global counter saturation

Limitations of branch predictors

Only predicts branch direction. Therefore, cannot redirect fetch stream until after branch target is determined.

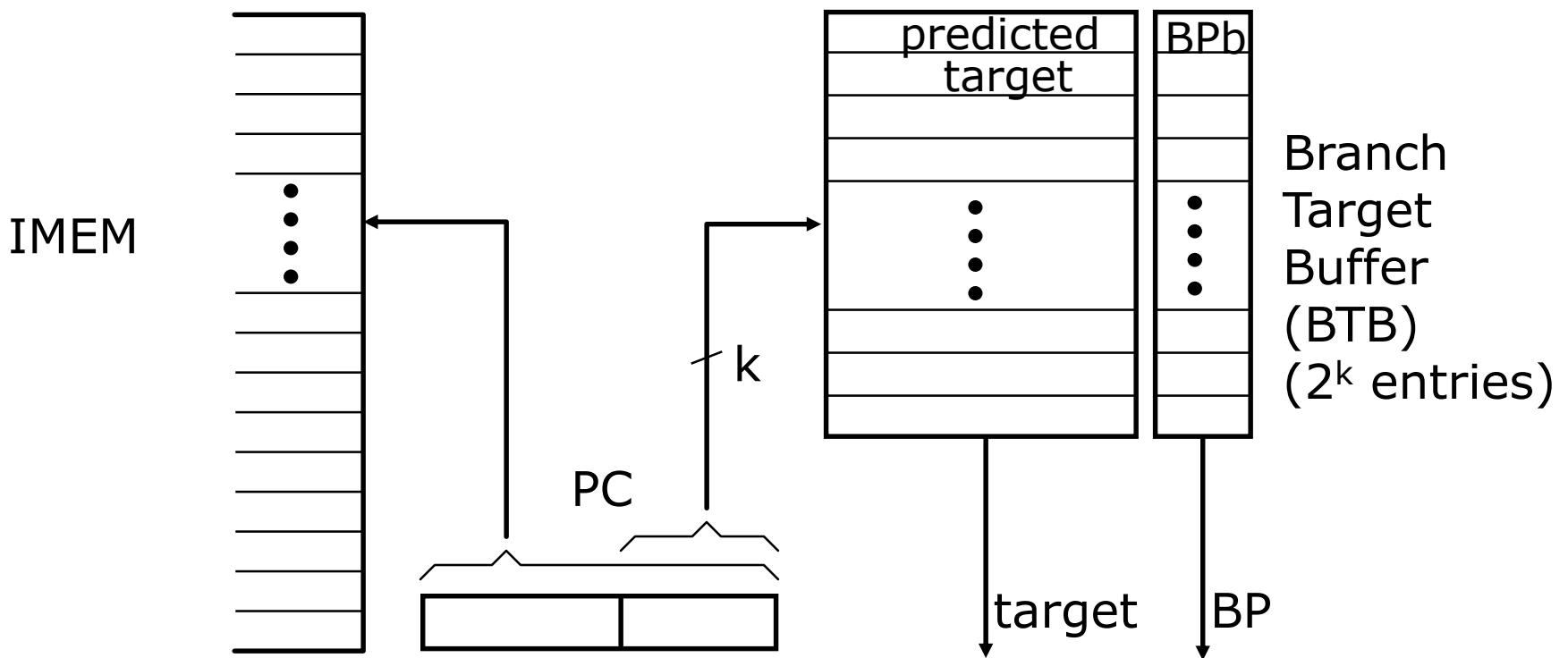
Correctly predicted taken branch penalty

Jump Register penalty



UltraSPARC-III fetch pipeline

Branch Target Buffer (un>tagged)

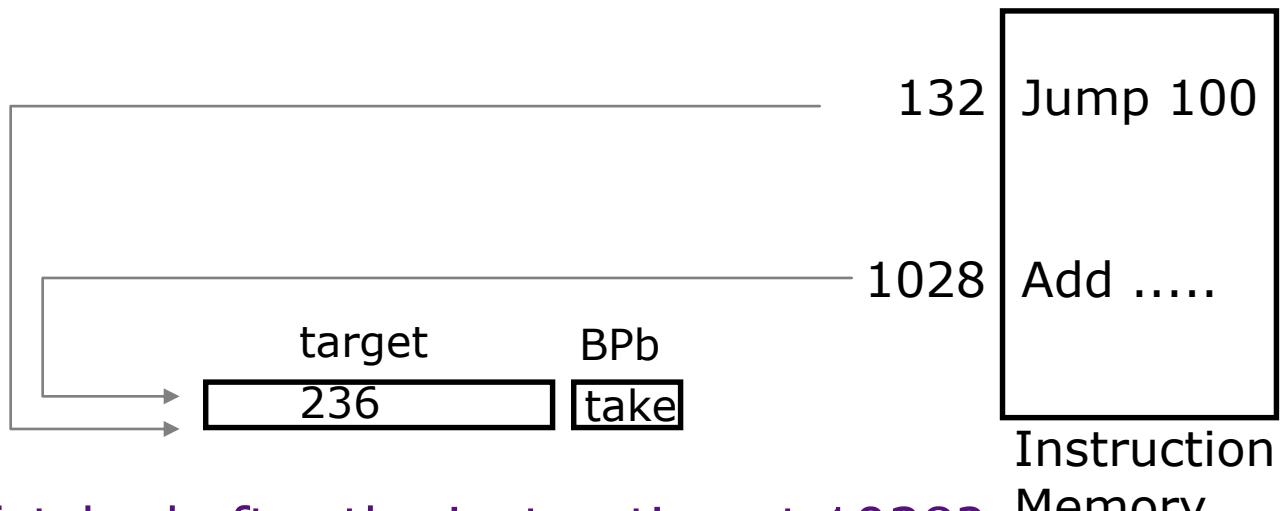


BP bits are stored with the predicted target address.

IF stage: *If (BP=taken) then nPC=target else nPC=PC+4*
later: *check prediction, if wrong then kill the instruction and update BTB & BPb, else update BPb*

Address Collisions

Assume a
128-entry
BTB



What will be fetched after the instruction at 1028?

BTB prediction =
Correct target =

⇒

*Is this a common occurrence?
Can we avoid these mispredictions?*

BTB is only for Control Instructions

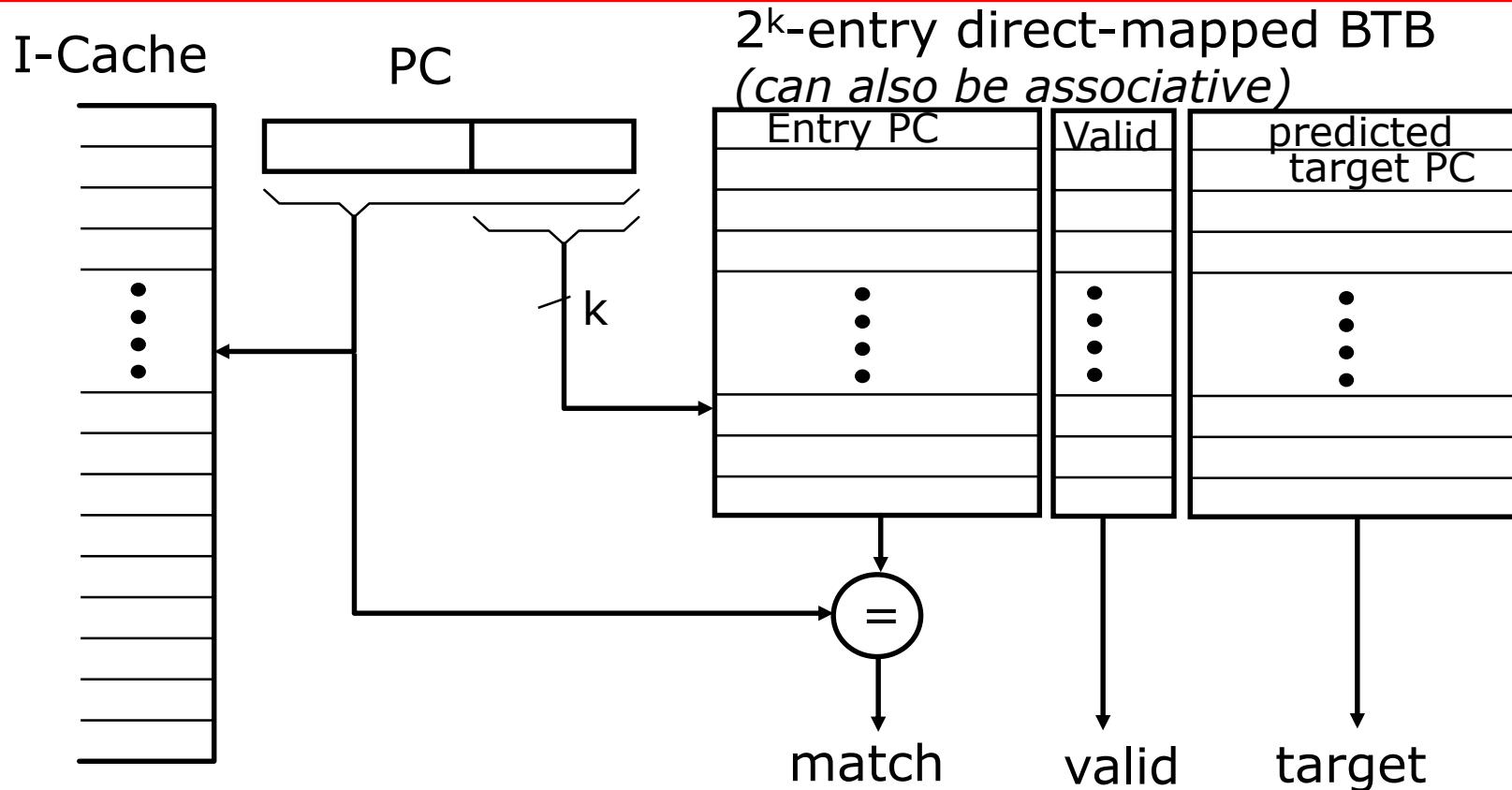
BTB contains useful information for branch and jump instructions only

⇒ Do not update it for other instructions

For all other instructions the next PC is $(PC)+4$!

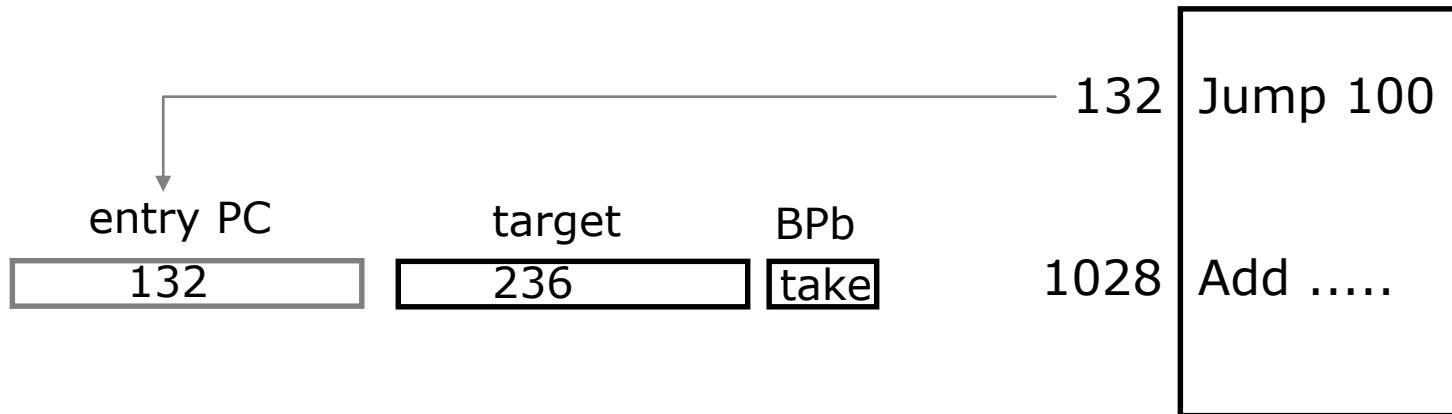
How to achieve this effect without decoding the instruction?

Branch Target Buffer (tagged)



- Keep both the branch PC and target PC in the BTB
- PC+4 is fetched if match fails
- Only *taken* branches and jumps held in BTB
- Next PC determined *before* branch fetched and decoded

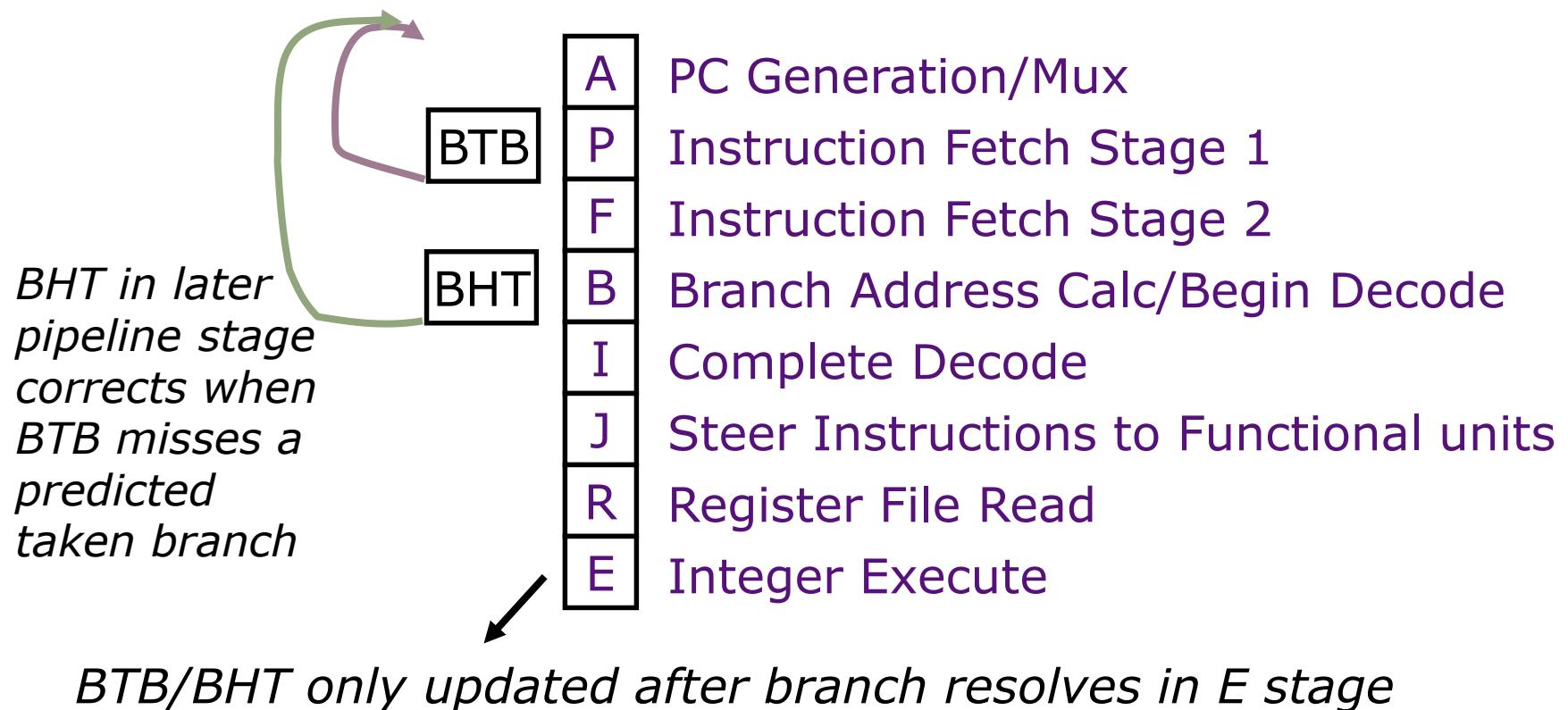
Consulting BTB Before Decoding



- The match for $PC=1028$ fails and $1028+4$ is fetched
 \Rightarrow *eliminates false predictions after ALU instructions*
- BTB contains entries only for control transfer instructions
 \Rightarrow *more room to store branch targets*

Combining BTB and BHT

- BTB entries are considerably more expensive than BHT, but can redirect fetches at earlier stage in pipeline and can accelerate indirect branches (JR)
- BHT can hold many more entries and is more accurate



Uses of Jump Register (JR)

- Switch statements (jump to address of matching case)
- Dynamic function call (jump to run-time function address)
- Subroutine returns (jump to return address)

How well does BTB work for each of these cases?

Subroutine Return Stack

Small structure to accelerate JR for subroutine returns, typically much more accurate than BTBs.

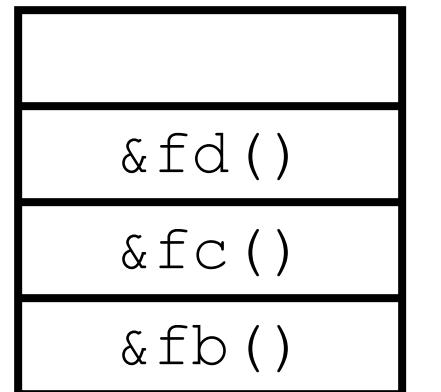
```
fa() { fb(); }
```

```
fb() { fc(); }
```

```
fc() { fd(); }
```

Push call address when function call executed

Pop return address when subroutine return decoded

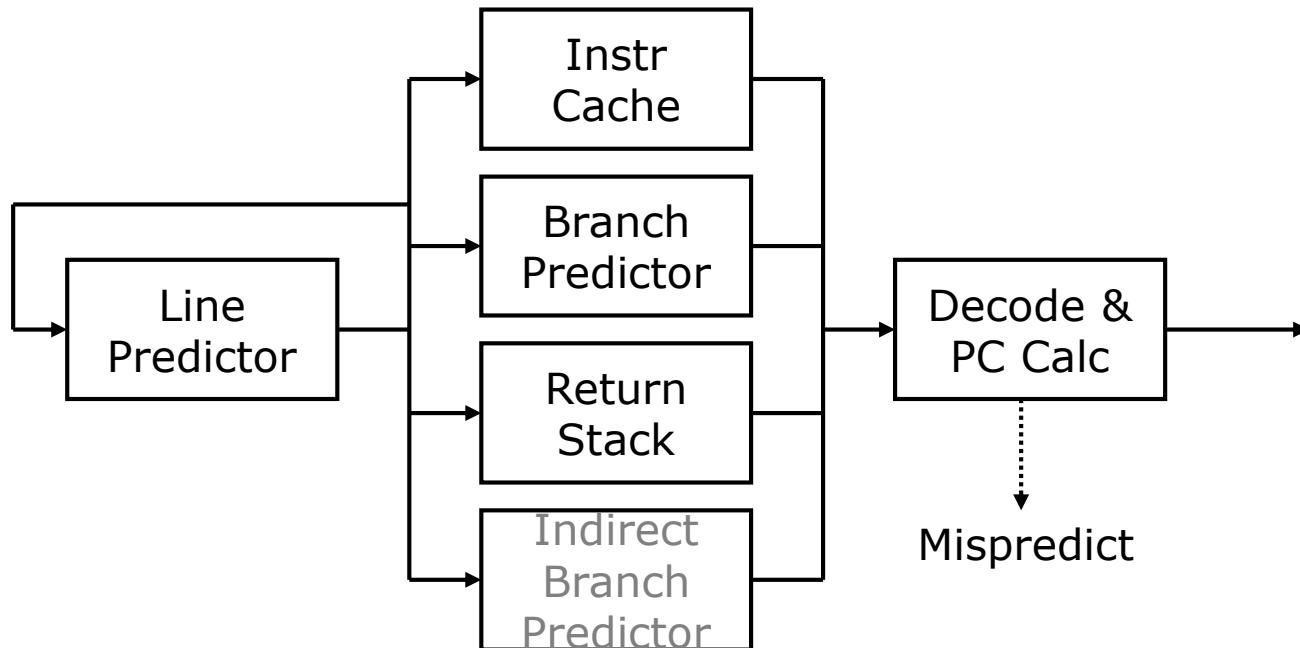


*k entries
(typically k=8-16)*

Line Prediction

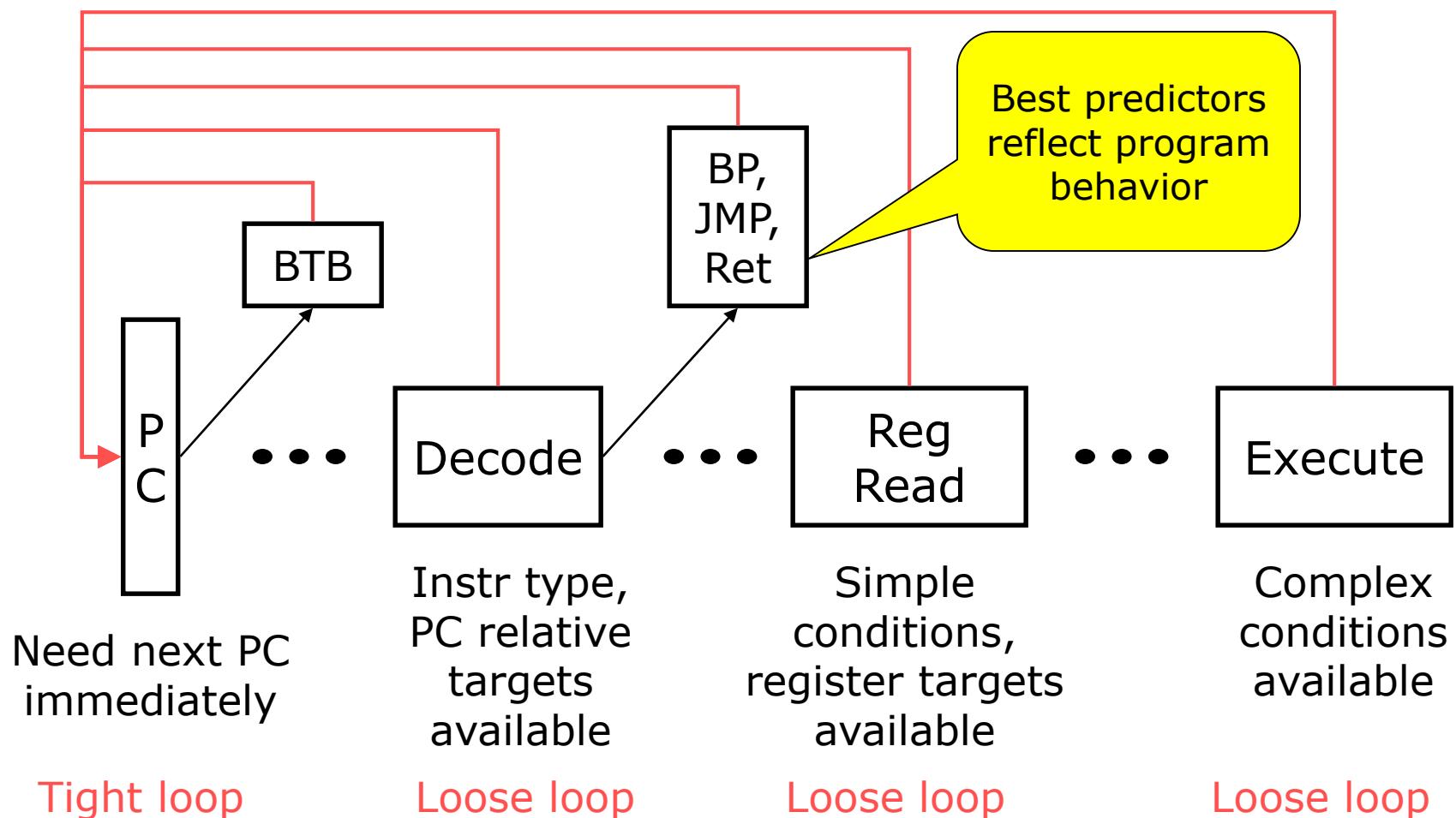
(Alpha 21[234]64)

- For superscalar, useful to predict next cache line(s) to fetch



- Line Predictor predicts line to fetch each cycle (tight loop)
 - Untagged BTB structure – Why?
 - 21464 was to predict 2 lines per cycle
- Icache fetches block, and predictors improve target prediction
- PC Calc checks accuracy of line prediction(s)

Overview of Branch Prediction



Must speculation check always be correct?

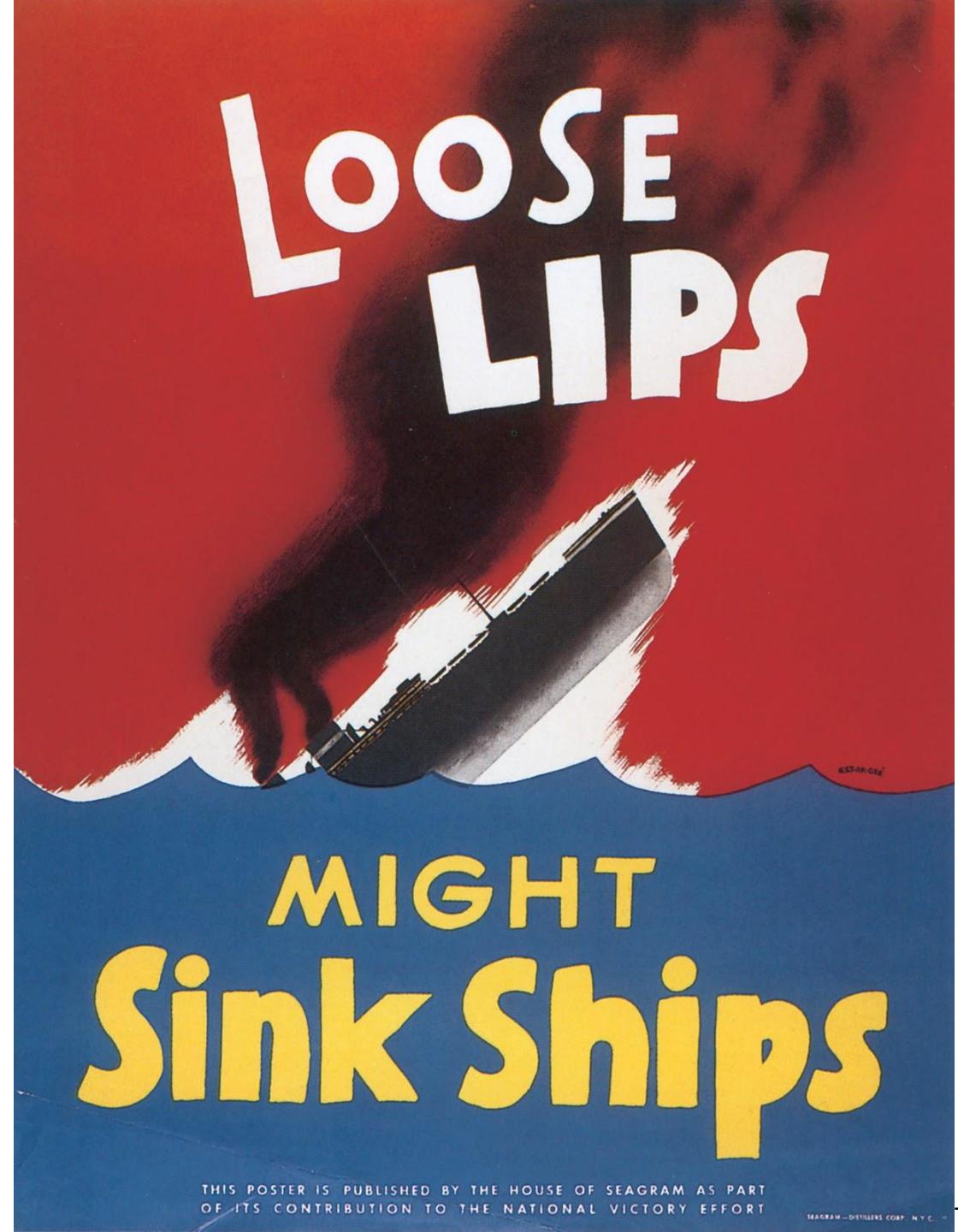
Next Lecture:
Speculative Execution
& Value Management

Speculative Execution

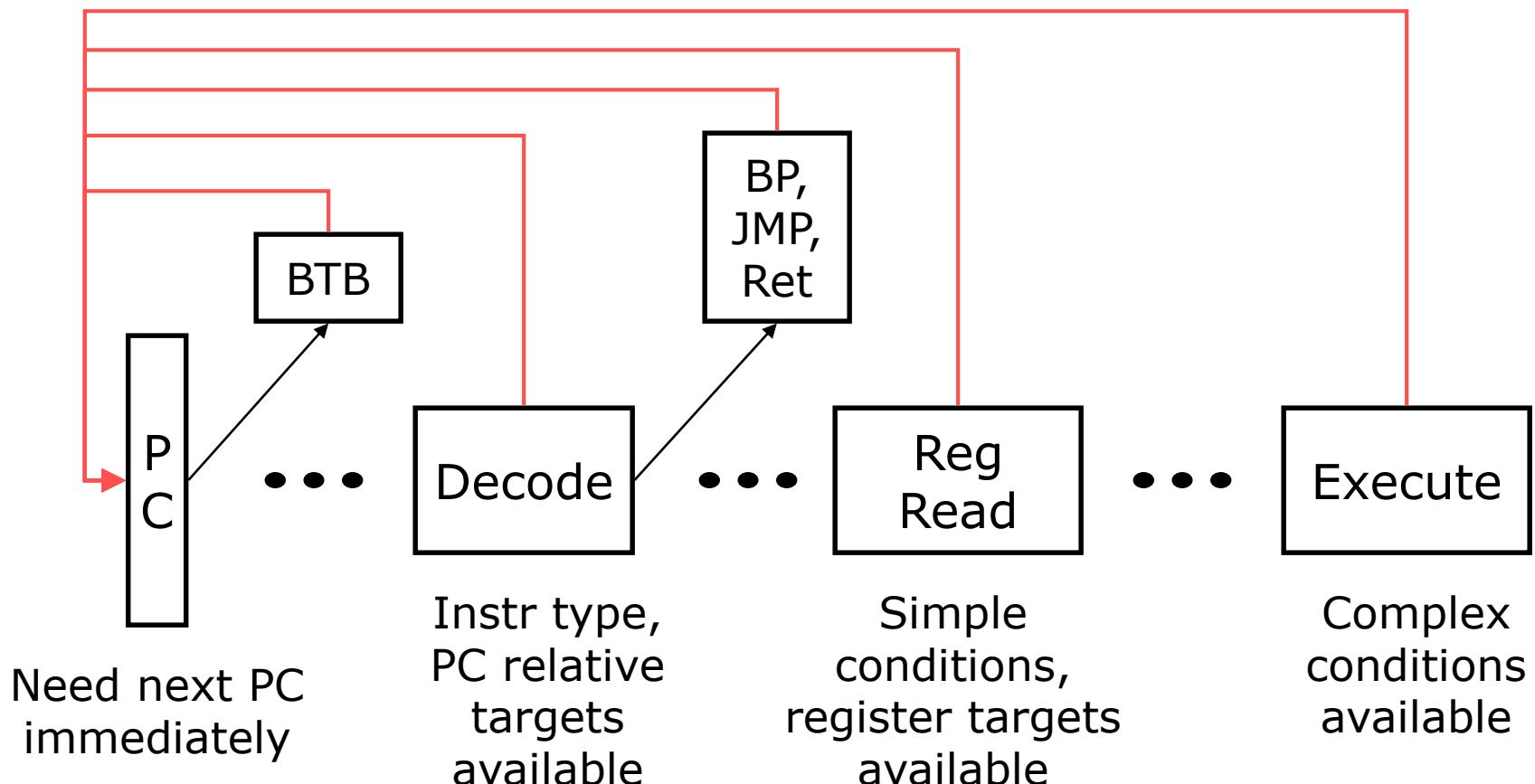
Daniel Sanchez

Computer Science and Artificial Intelligence Laboratory
M.I.T.

What does this
WW2 poster
have in common
with pipelined
processors?



Overview of branch prediction



Must speculation check always be correct?

Speculative Execution Recipe

1. Proceed ahead despite unresolved dependencies using a prediction for an architectural or micro-architectural value
 2. Maintain both old and new values on updates to architectural (and often micro-architectural) state
 3. After sure that there was no mis-speculation and there will be no more uses of the old values, discard old values and just use new values
- OR
3. In event of mis-speculation, dispose of all new values, restore old values, and re-execute from point before mis-speculation

Why might one use old values?

O-O-O WAR hazards

Value Management Strategies

Greedy (or Eager) Update:

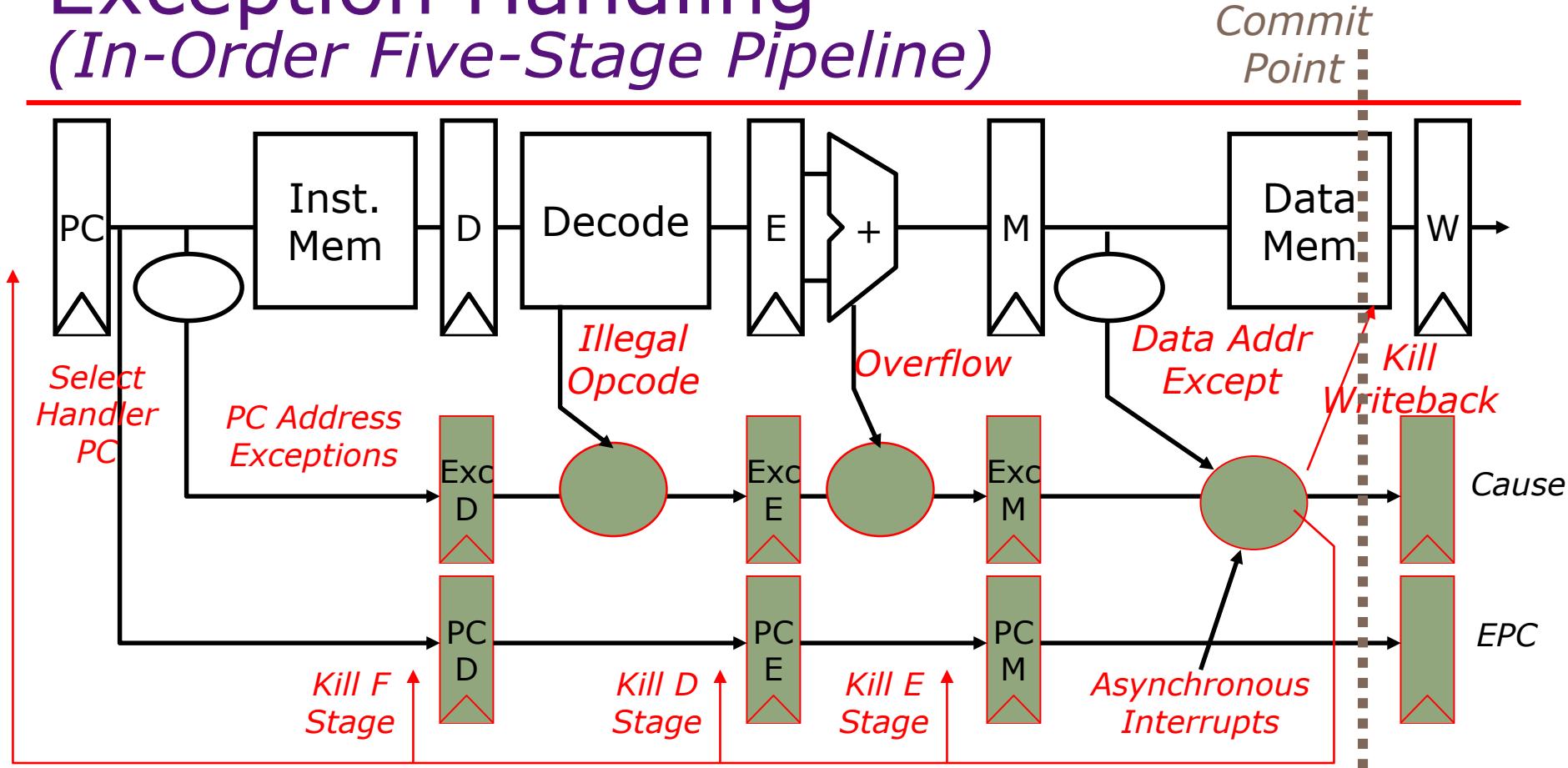
- Update value in place, and
- Provide means to reconstruct old values for recovery
 - often this is a log of old values

Lazy Update:

- Buffer new value, leaving old value in place
- Replace old value only at 'commit' time

Why leave an old value in place?

Exception Handling (In-Order Five-Stage Pipeline)



Strategy for Registers?

Strategy for PC?

Misprediction Recovery

In-order execution machines:

- Guarantee no instruction issued after branch can write-back before branch resolves by keeping values in the pipeline
- Kill all values from all instructions in pipeline behind mispredicted branch

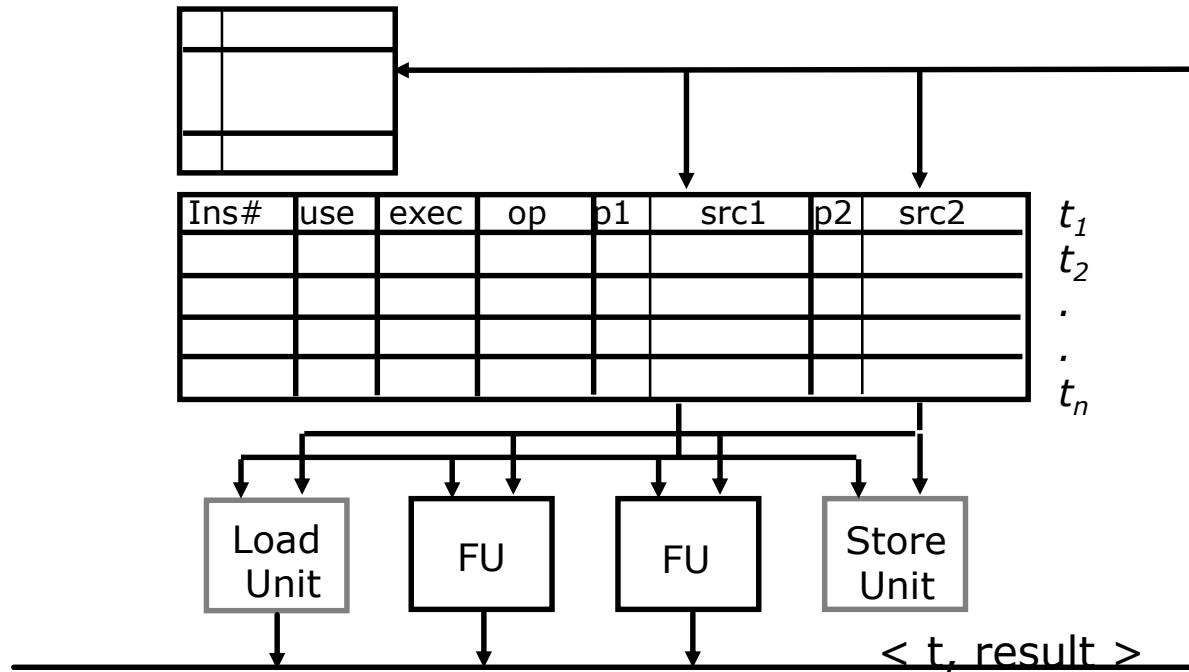
Out-of-order execution?

- Multiple instructions following branch in program order can generate new values before branch resolves

Data-Driven Execution

*Renaming
table &
reg file*

*Reorder
buffer*



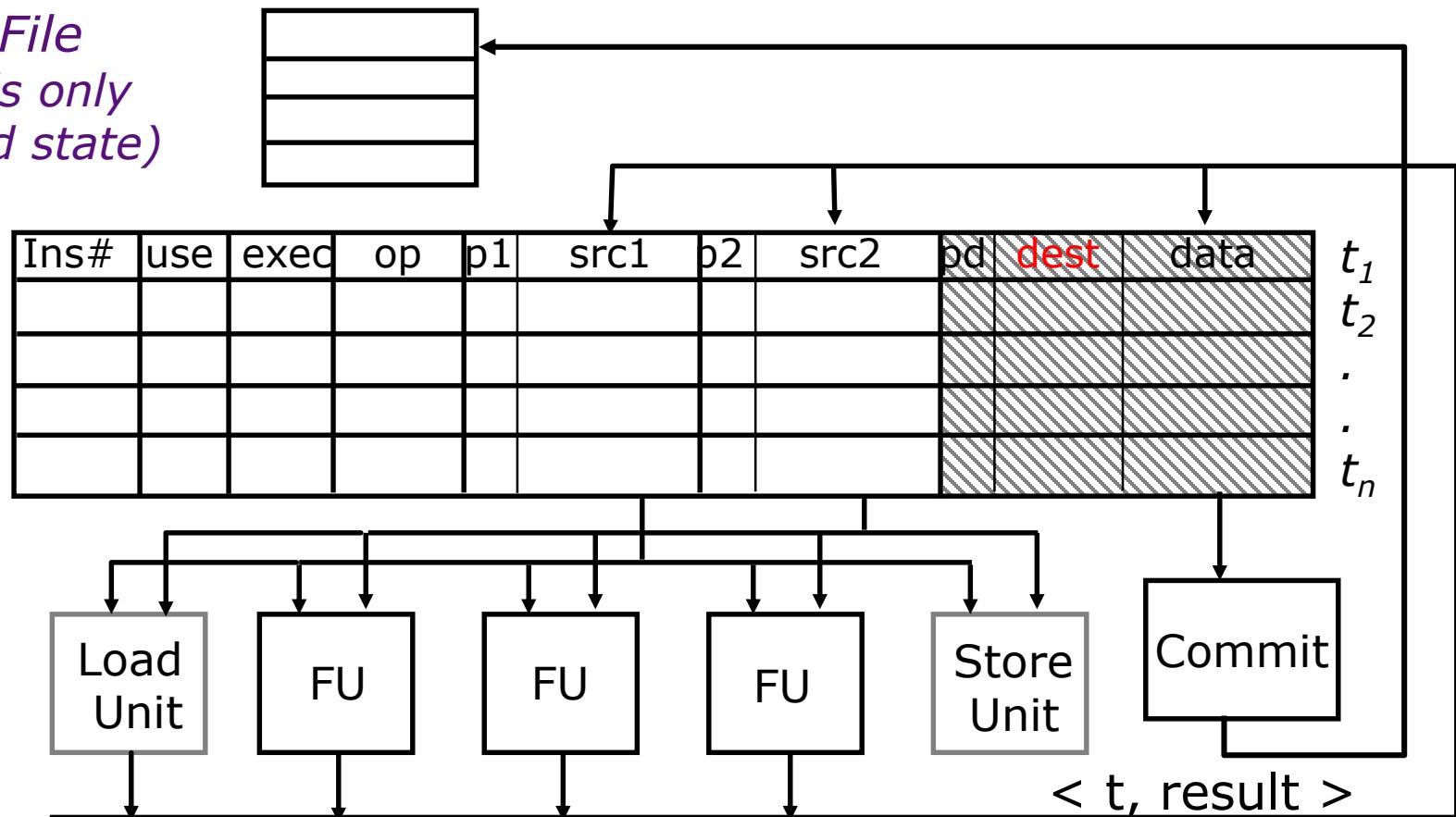
Basic Operation:

- Enter op and tag or data (if known) for each source
- Replace tag with data as it becomes available
- Issue instruction when all sources are available
- Save dest data when operation finishes

Update strategy?

Rollback and Renaming

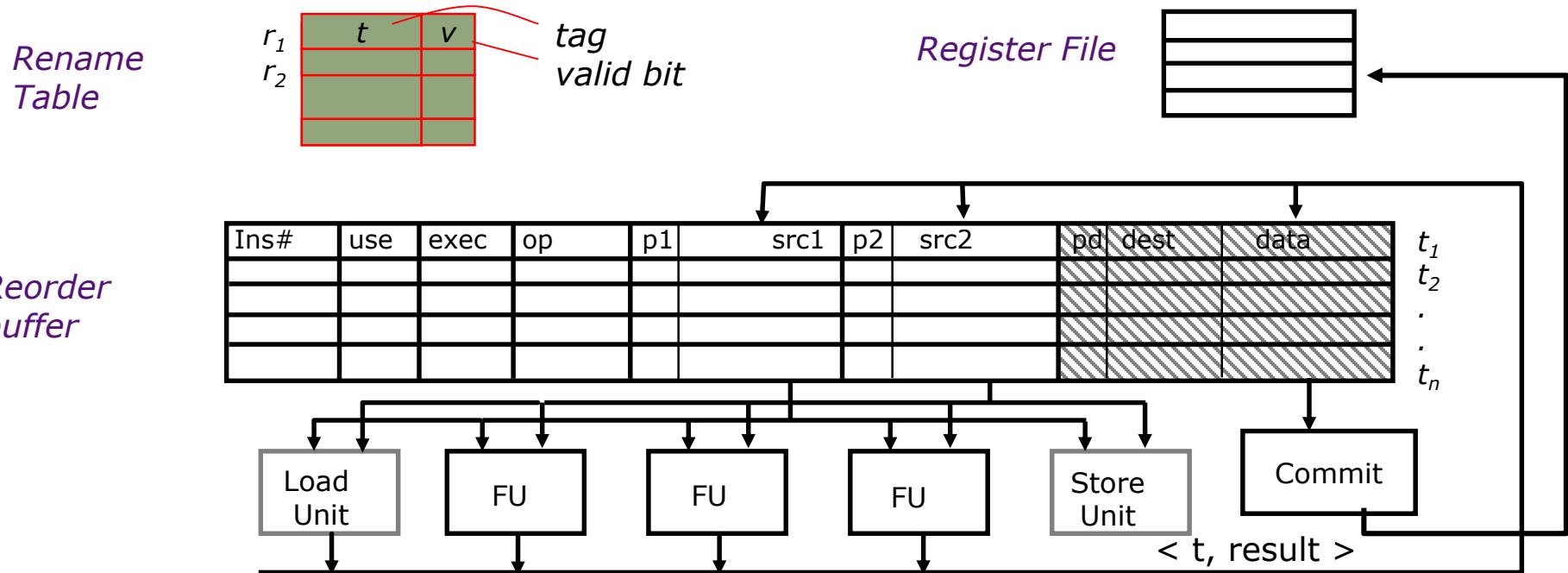
*Register File
(now holds only
committed state)*



Convert to lazy by holding data in ROB.
But how do we find values before they are committed?

Renaming Table

Micro-architectural speculative cache to speed up tag look up.



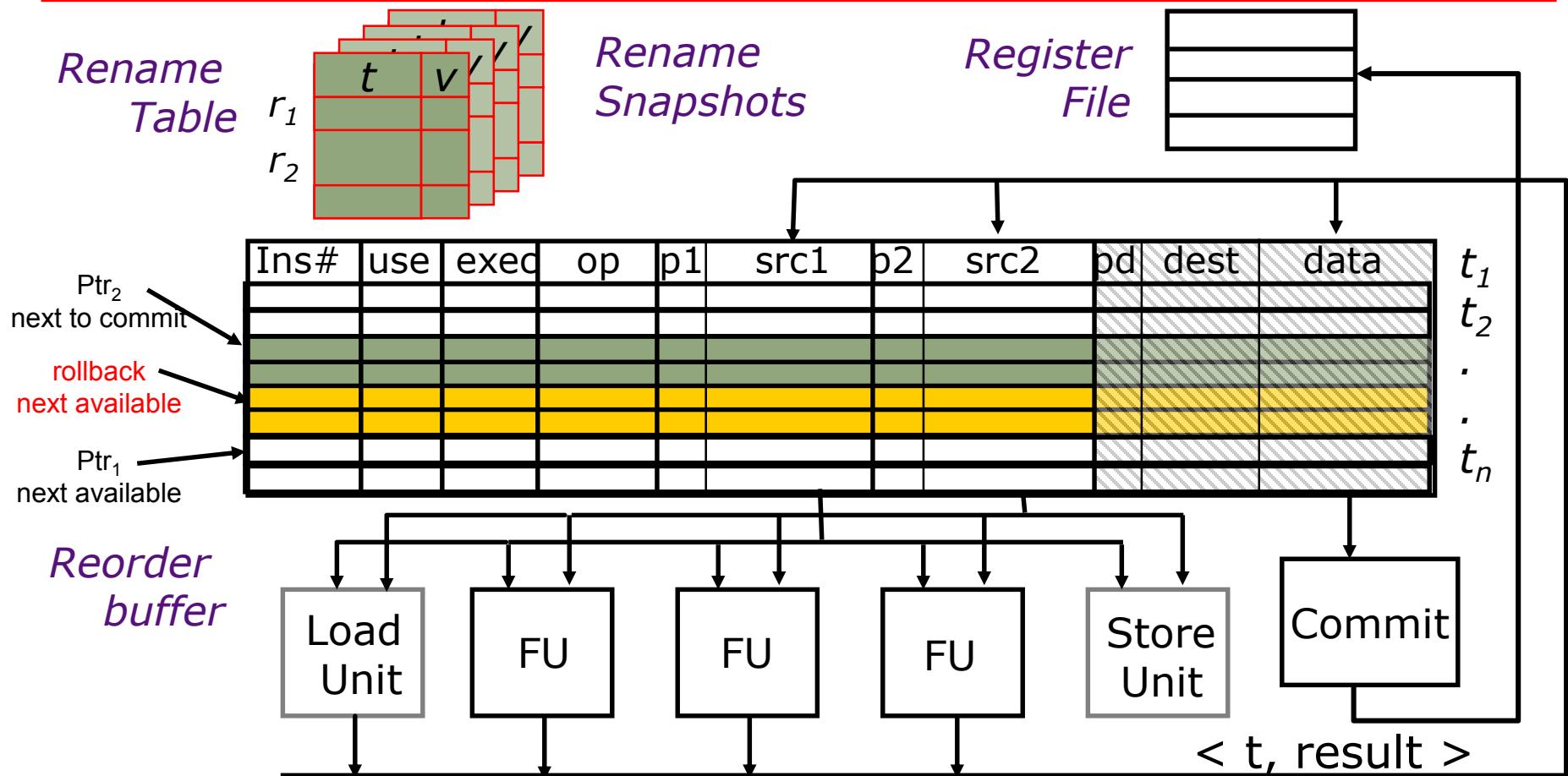
What is the update policy of rename table?

What events cause mis-speculation?

How can we respond to mis-speculation?

After being cleared, when can instructions be added to ROB?

Recovering ROB/Renaming Table



Take snapshot of register rename table at each predicted branch, recover earlier snapshot if branch mispredicted

Map Table Recovery - Snapshots

Speculative value management of microarchitectural state

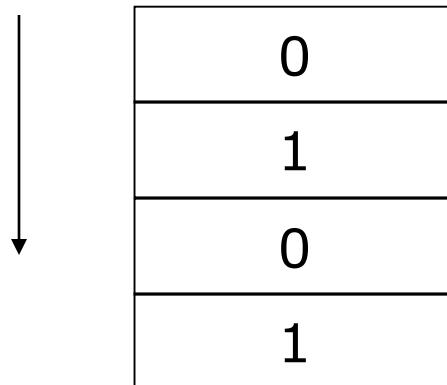
	Reg Map	V	Snap Map V	Snap Map V
R0	T20	X	T20	X
R1	T73	X	T73	X
R2	T45	X	T45	X
R3	T128		T128	
	⋮		⋮	⋮
R30	T54		T54	
R31	T88	X	T88	X

What kind of value management is this?

Branch Predictor Recovery

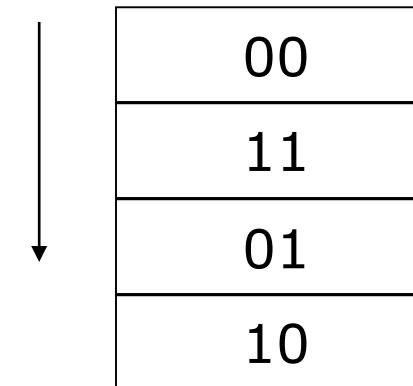
- 1-Bit Counter Recovery

PC

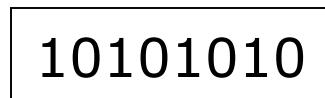


- 2-Bit Counter Recovery

PC

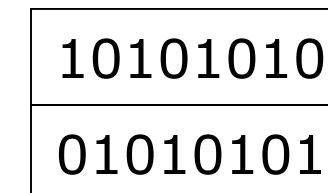


- Global History Recovery



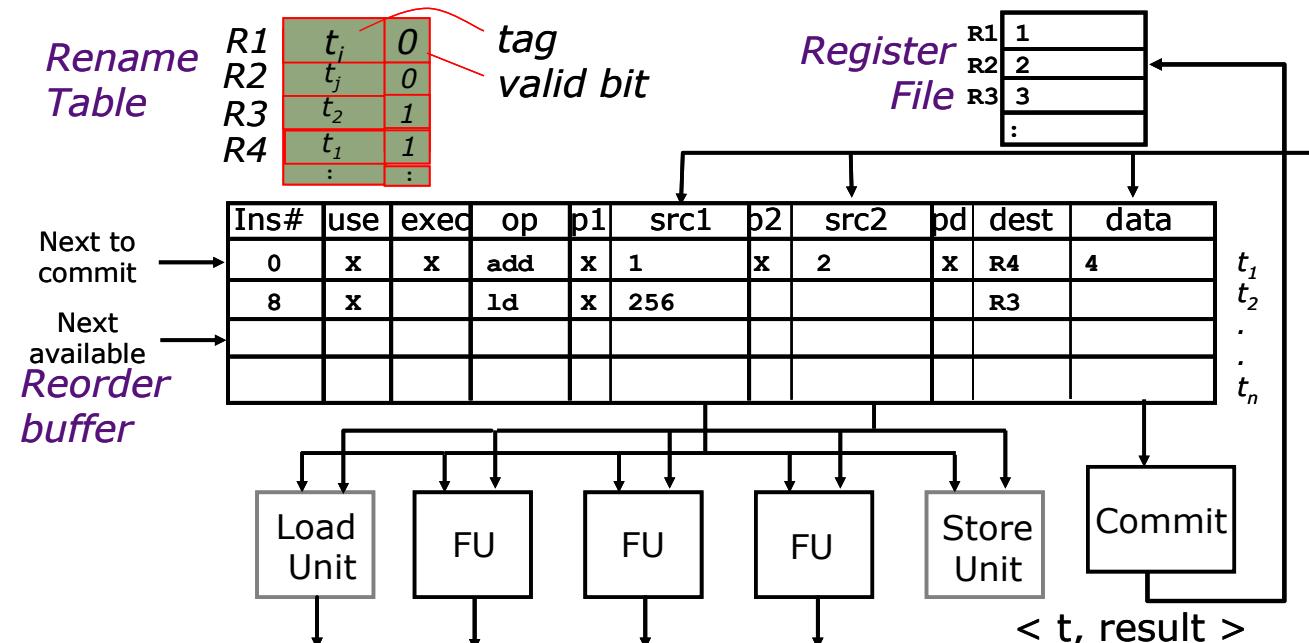
- Local History Recovery

PC



O-o-O Execution with ROB

Data-in-ROB design

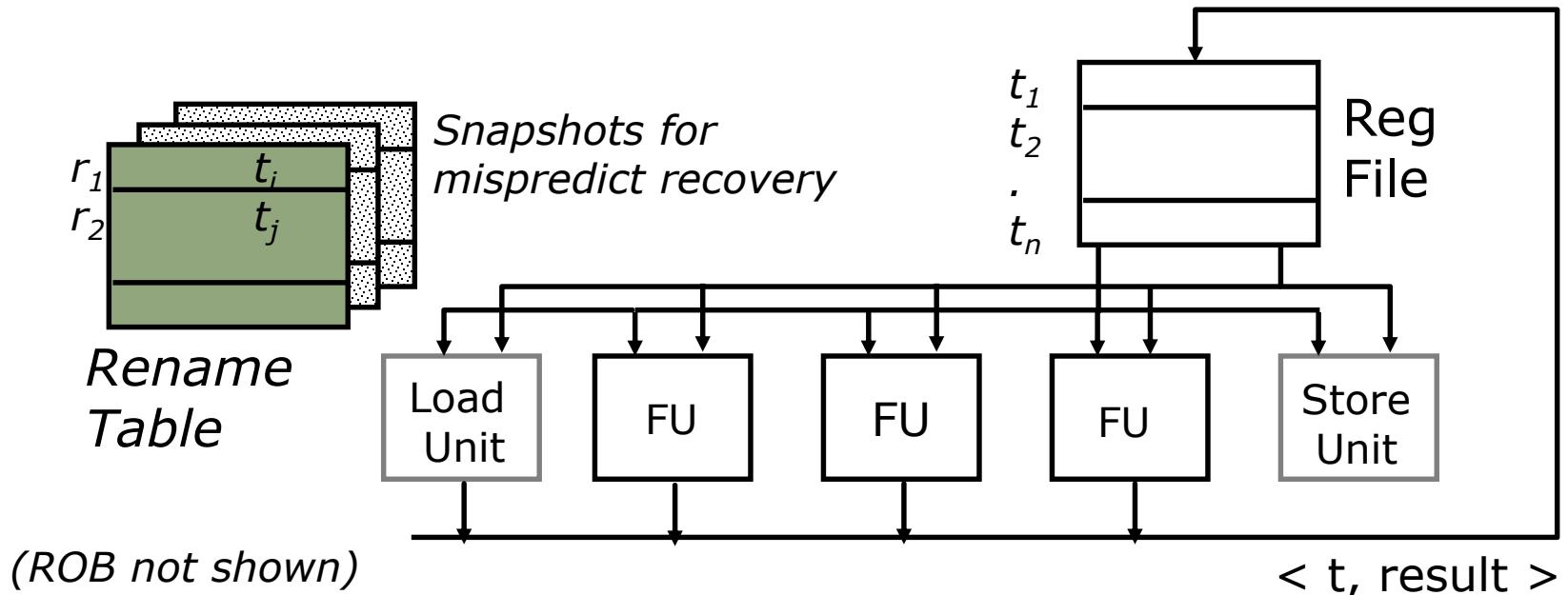


Basic Operation:

- Enter op and tag or data (if known) for each source
- Replace tag with data as it becomes available
- Issue instruction when all sources are available
- Save dest data when operation finishes
- Commit saved dest data when instruction commits

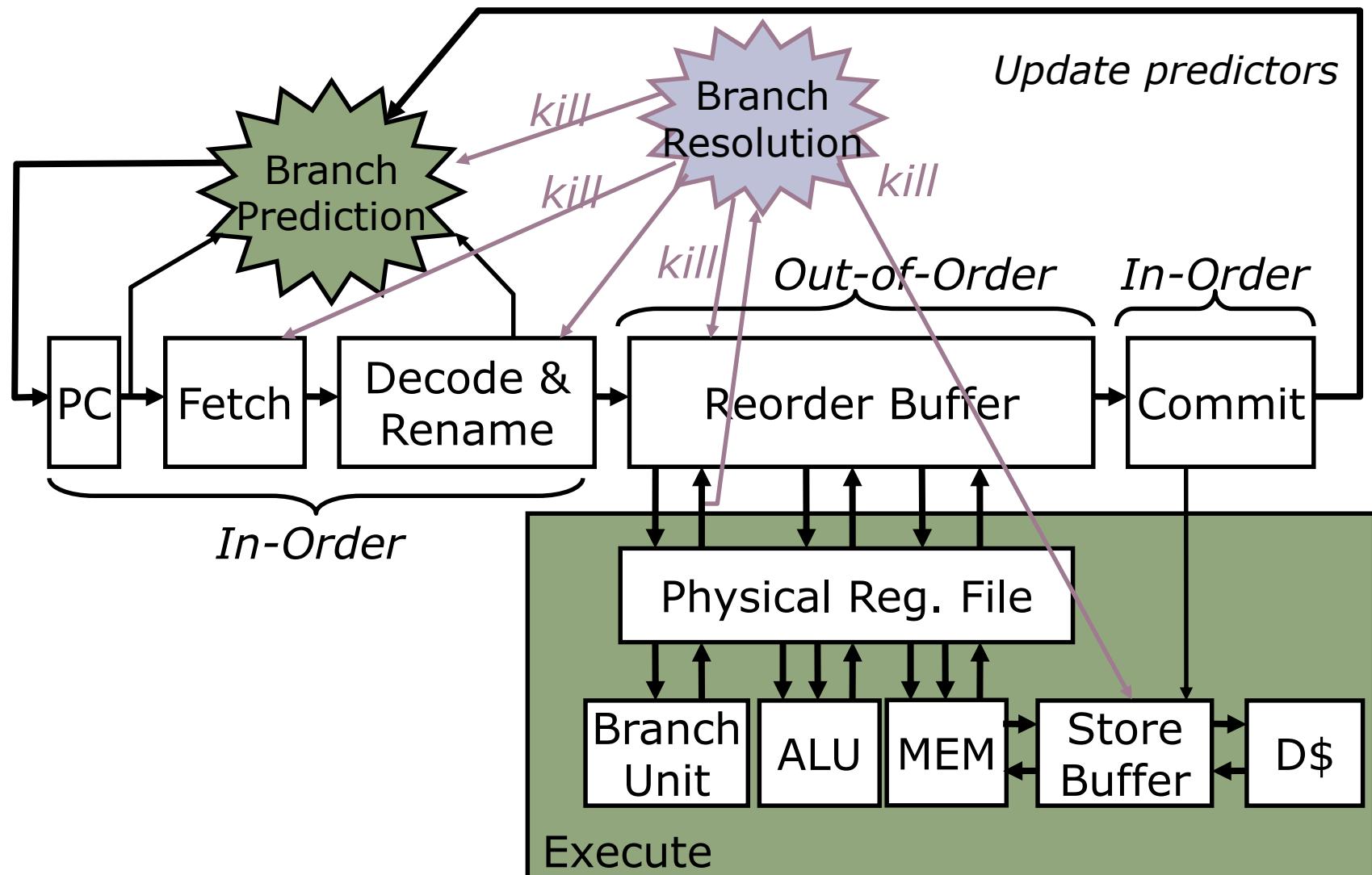
Unified Physical Register File

(MIPS R10K, Alpha 21264, Pentium 4)



- One regfile for both *committed* and *speculative* values (no data in ROB)
- During decode, instruction result allocated new physical register, source regs translated to physical regs through rename table
- Instruction reads data from regfile at start of execute (not in decode)
- Write-back updates reg. busy bits on instructions in ROB (assoc. search)
- Snapshots of rename table taken at every branch to recover mispredicts
- On exception, renaming undone in reverse order of issue (*MIPS R10000*)

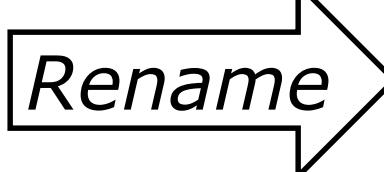
Speculative & Out-of-Order Execution



Lifetime of Physical Registers

- Physical regfile holds committed and speculative values
- Physical registers decoupled from ROB entries (*no data in ROB*)

a)	ld r1 , (r3)	Id P1, (Px)
b)	add r3, r1, #4	add P2, P1, #4
c)	sub r1 , r3, r9	sub P3, P2, Py
d)	add r3 , r1, r7	add P4, P3, Pz
e)	ld r6, (r1)	Id P5, (P3)
f)	add r8, r6, r3	add P6, P5, P4
g)	st r8, (r1)	st P6, (P3)
h)	ld r3 , (r11)	Id P7, (Pw)



When can we reuse a physical register?

Physical Register Management

Rename Table	
R0	
R1	P8
R2	
R3	P7
R4	
R5	
R6	P5
R7	P6

ROB

use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd

Physical Regs

P0		
P1		
P2		
P3		
P4		
P5	<R6>	p
P6	<R7>	p
P7	<R3>	p
P8	<R1>	p
Pn		

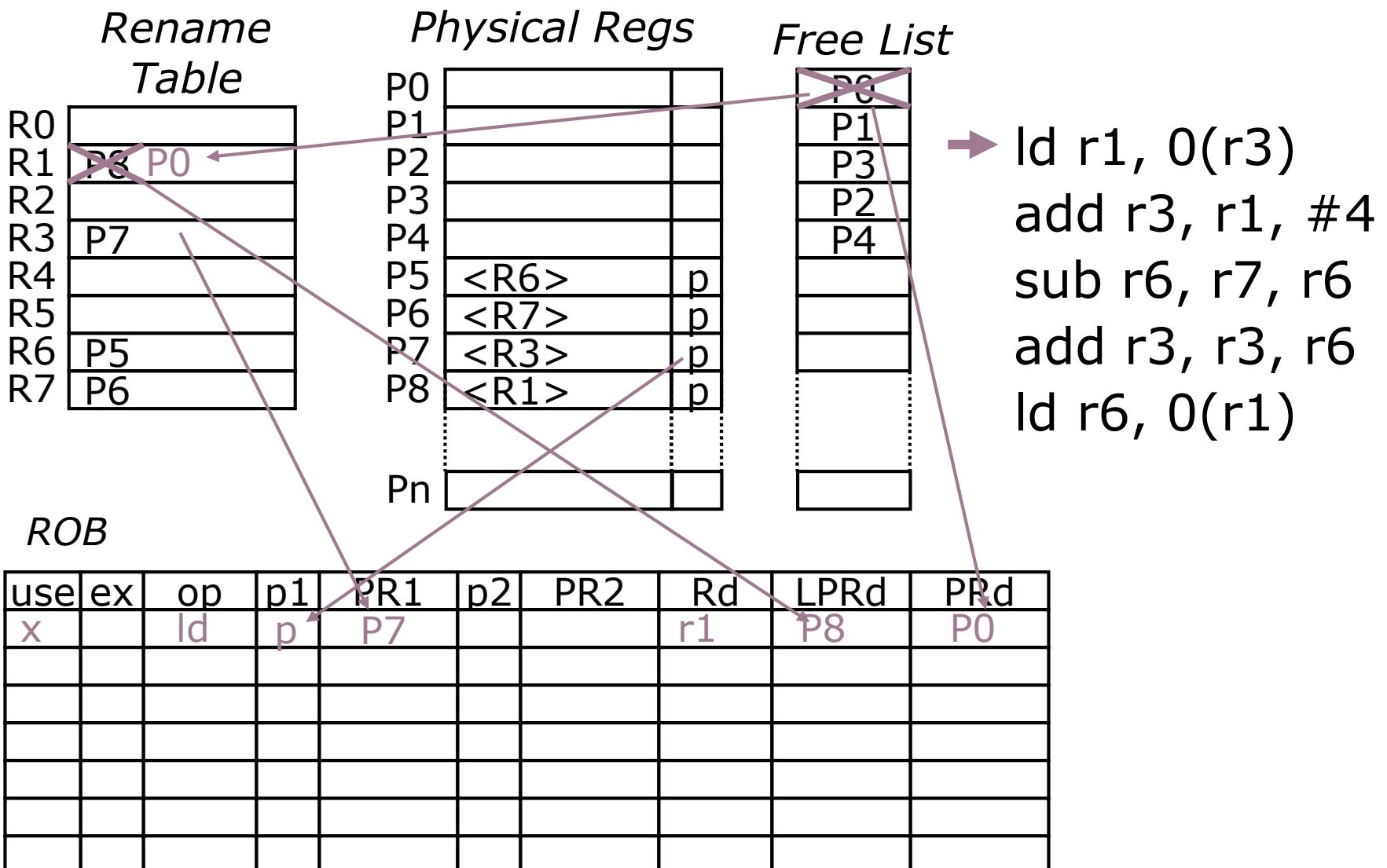
Free List

P0
P1
P3
P2
P4

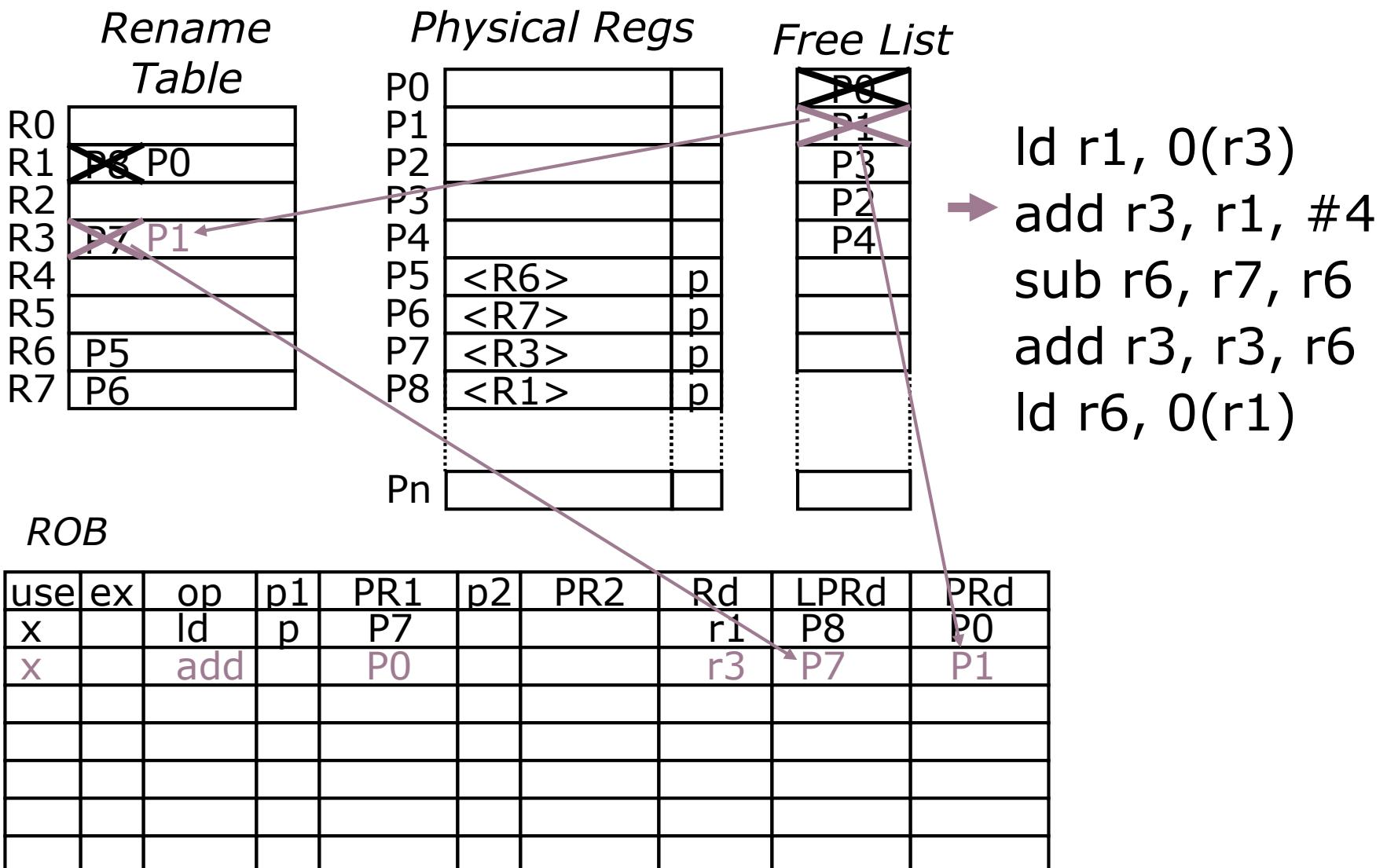
ld r1, 0(r3)
add r3, r1, #4
sub r6, r7, r6
add r3, r3, r6
ld r6, 0(r1)

(LPRd requires third read port on Rename Table for each instruction)

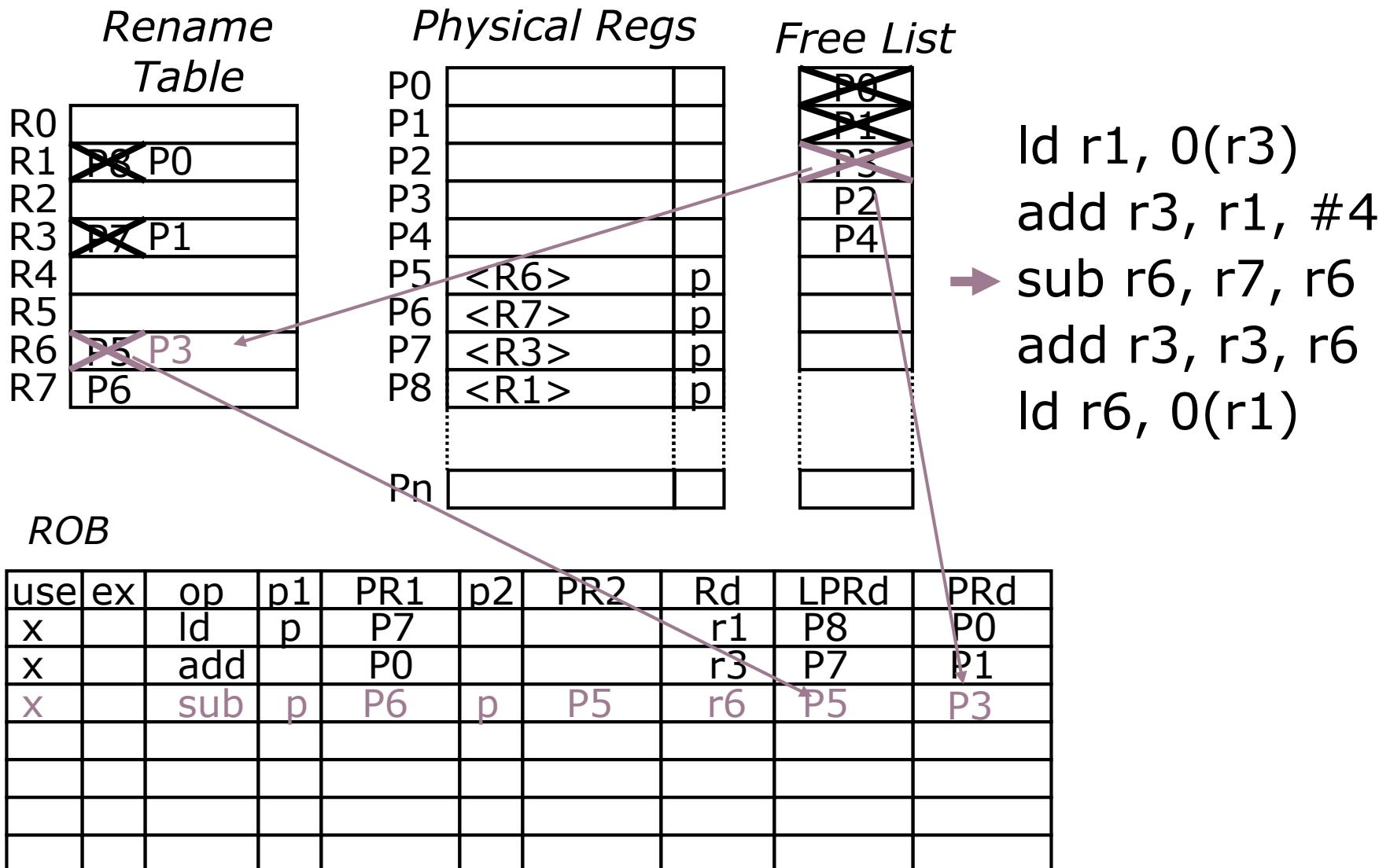
Physical Register Management



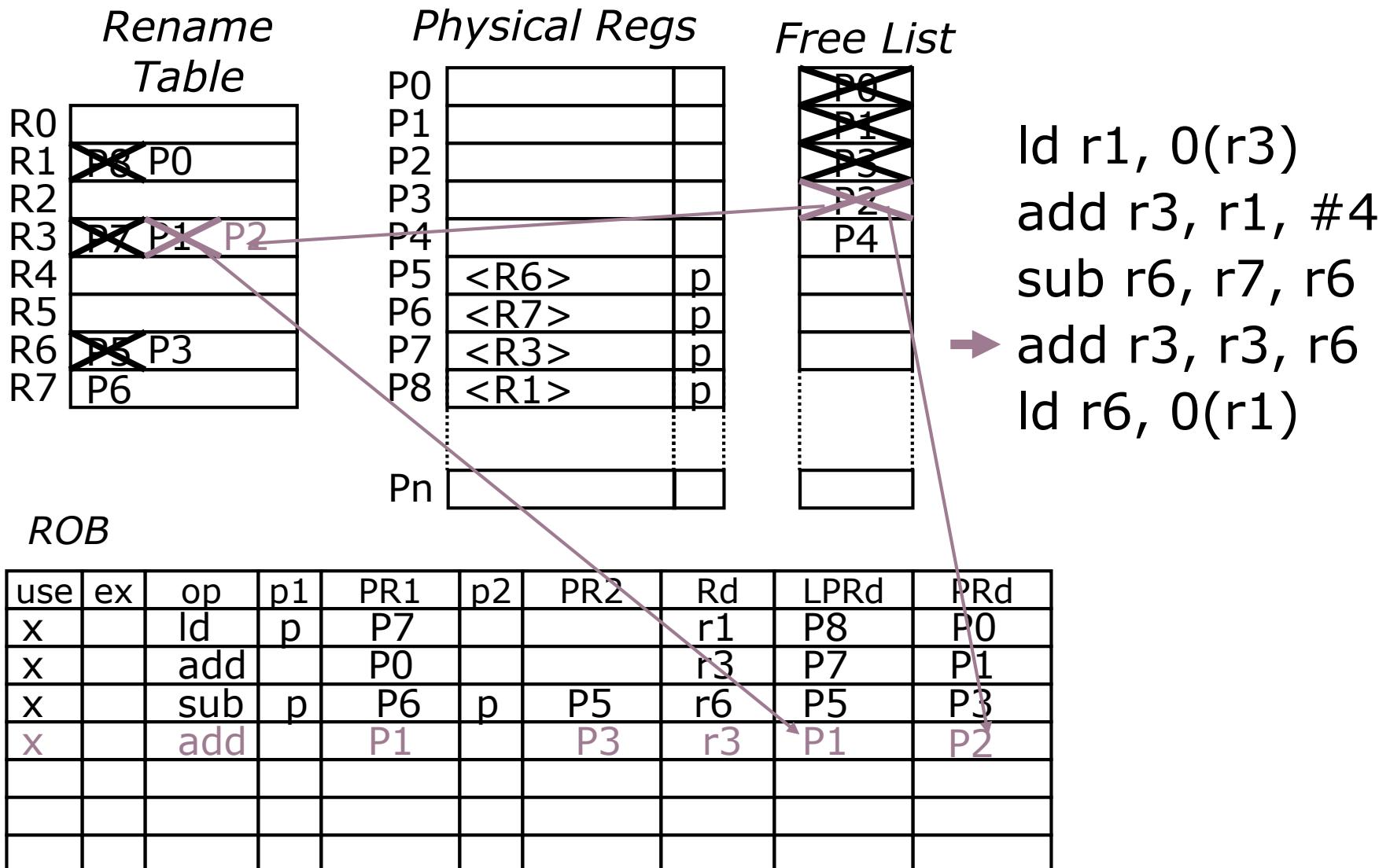
Physical Register Management



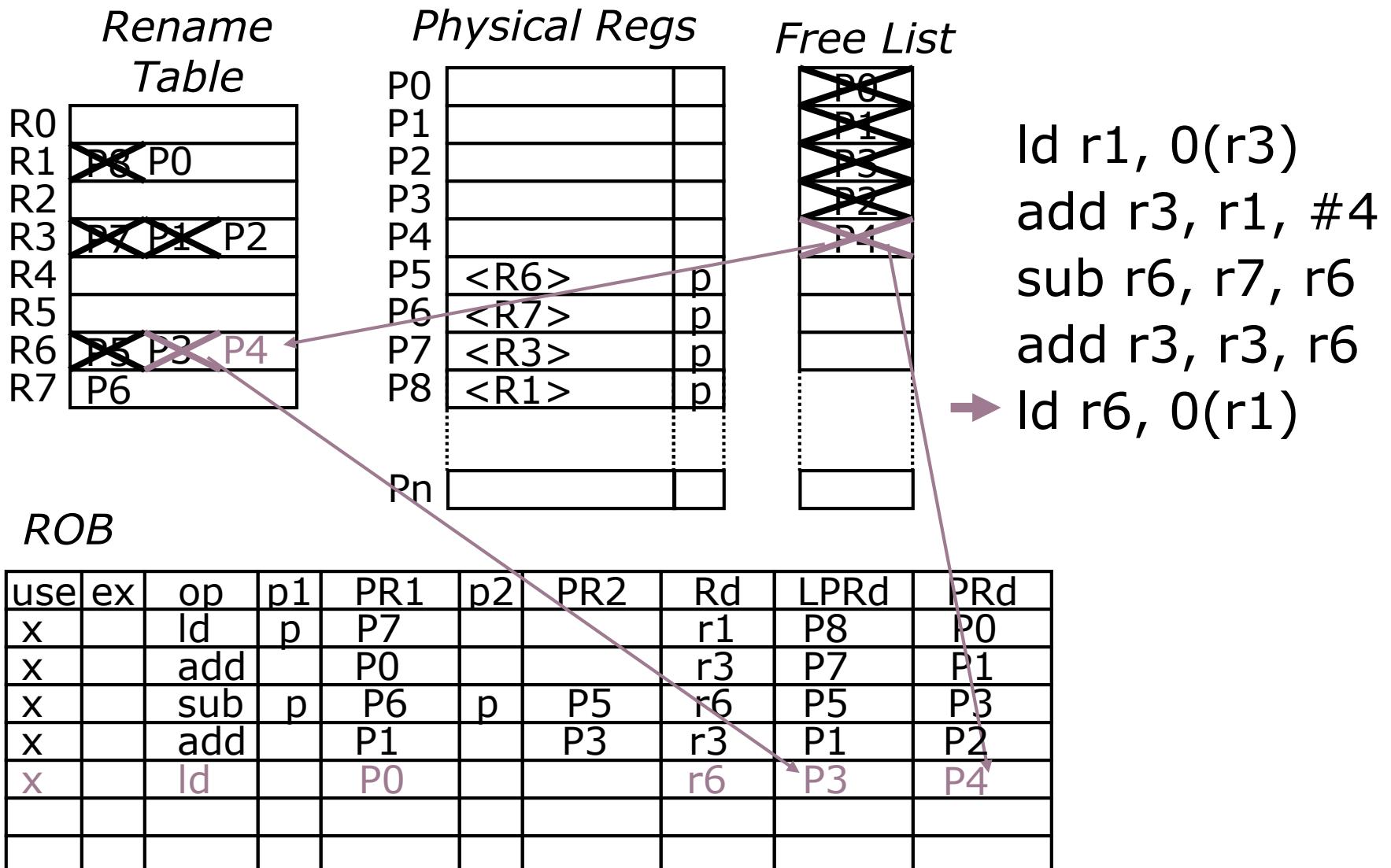
Physical Register Management



Physical Register Management



Physical Register Management



Physical Register Management

Rename Table

R0	
R1	P0
R2	
R3	P1 P2
R4	
R5	
R6	P5 P3 P4
R7	P6

Physical Regs

P0	<R1>	p
P1		
P2		
P3		
P4		
P5	<R6>	p
P6	<R7>	p
P7	<R3>	p
P8	<R1>	p
Pn		

Free List

P6	
P1	
P3	
P2	
P4	
P8	

ROB

use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd
x	x	ld	p	P7			r1	P8	P0
x		add	p	P0			r3	P7	P1
x		sub	p	P6	p	P5	r6	P5	P3
x		add		P1		P3	r3	P1	P2
x		ld	p	P0			r6	P3	P4

ld r1, 0(r3)
 add r3, r1, #4
 sub r6, r7, r6
 add r3, r3, r6
 ld r6, 0(r1)

Execute & Commit

Physical Register Management

Rename Table

R0	
R1	P0
R2	
R3	P1 P2
R4	
R5	
R6	P5 P3 P4
R7	P6

Physical Regs

P0	<R1>	p
P1	<R3>	p
P2		
P3		
P4		
P5	<R6>	p
P6	<R7>	p
P7	<R3>	p
P8		
Pn		

Free List

P6
P1
P3
P2
P4
P8
P7

ROB

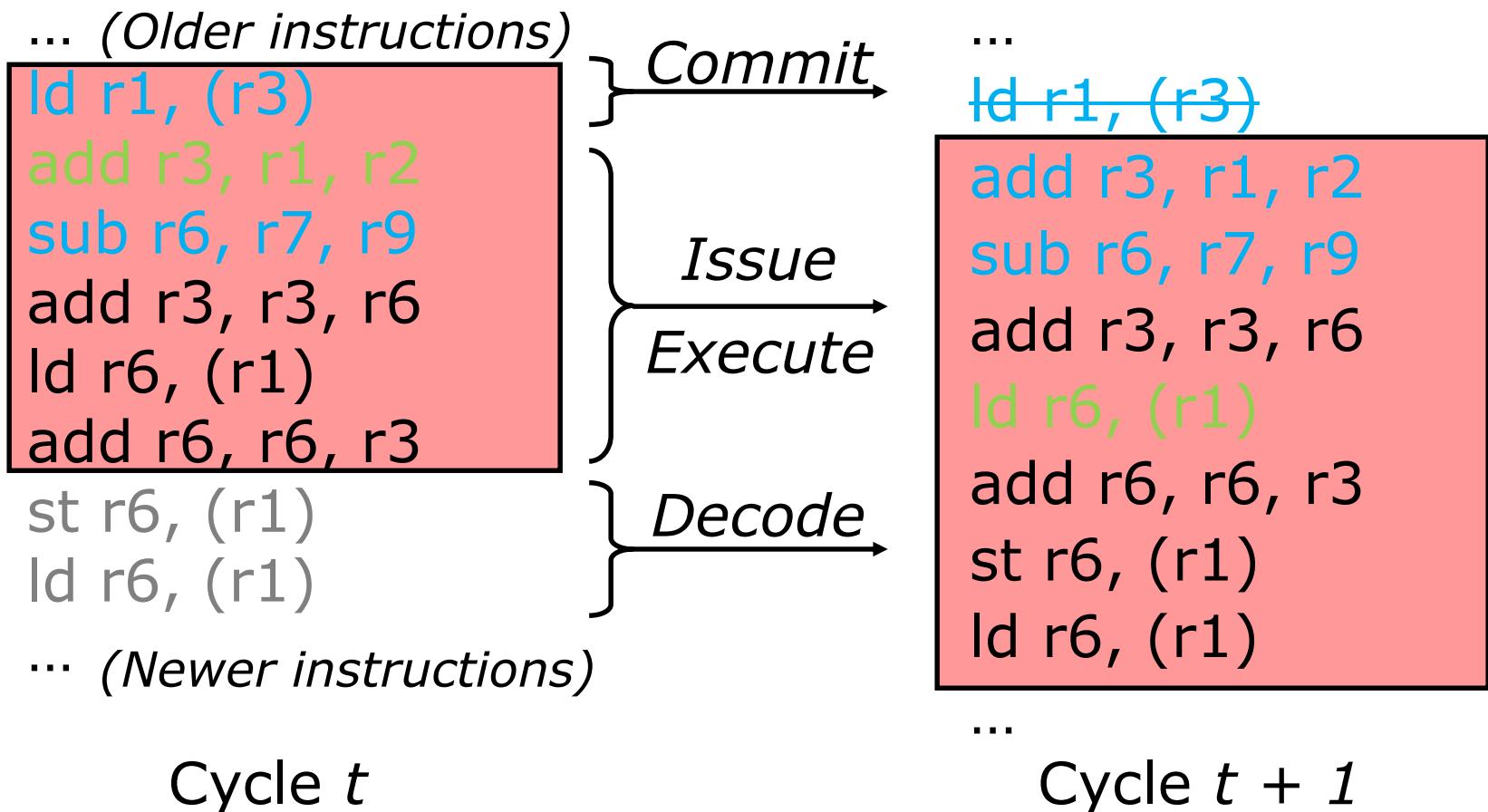
use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd
x	x	Id	p	P7			r1	P8	P0
x	x	add	p	P0			r3	P7	P1
x		sub	p	P6	p	P5	r6	P5	P3
x		add	p	P1		P3	r3	P1	P2
x		Id	p	P0			r6	P3	P4

ld r1, 0(r3)
 add r3, r1, #4
 sub r6, r7, r6
 add r3, r3, r6
 ld r6, 0(r1)

Execute & Commit



Reorder Buffer Holds Active Instruction Window



Key: predecode, decoded, issued, executed, committed

Issue Timing

i1	Add R1,R1,#1	Issue ₁	Execute ₁		
i2	Sub R1,R1,#1			Issue ₂	Execute ₂

How can we issue earlier?

Using knowledge of execution latency (bypass)

i1	LD R1, (R3)	Issue ₁	Execute ₁		
i2	Sub R1,R1,#1		Issue ₂	Execute ₂	

What might make this schedule fail?

If execution latency wasn't as expected

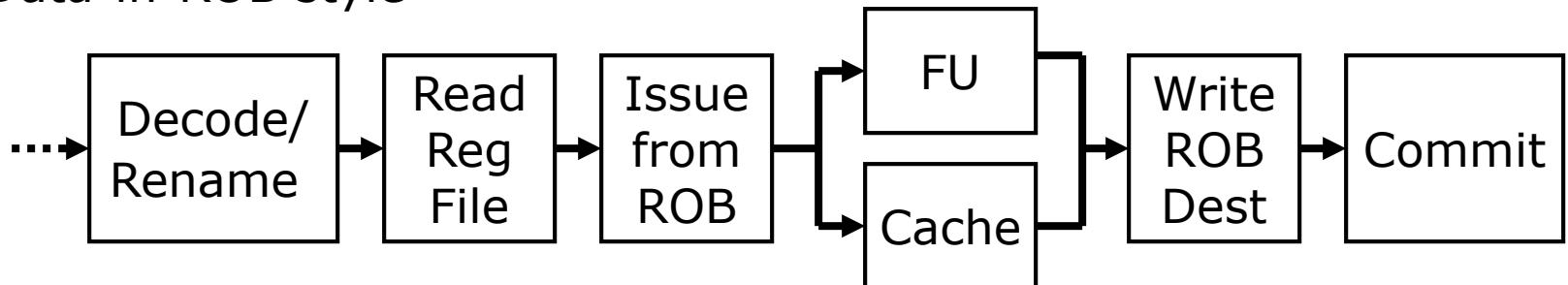
Issue Queue with latency prediction

Issue Queue (Reorder buffer)

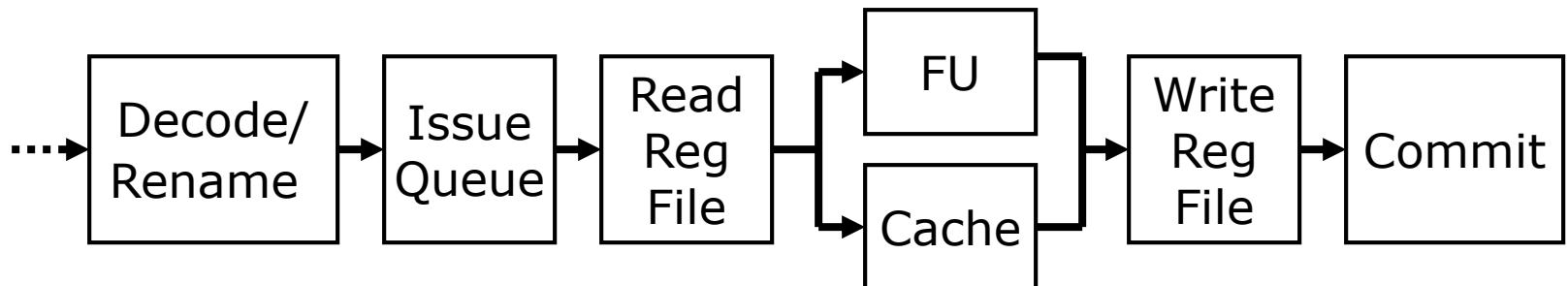
- Fixed latency: latency included in queue entry ('bypassed')
 - Predicted latency: latency included in queue entry (speculated)
 - Variable latency: wait for completion signal (stall)

Data-in-ROB vs. Unified RegFile

Data-in-ROB style



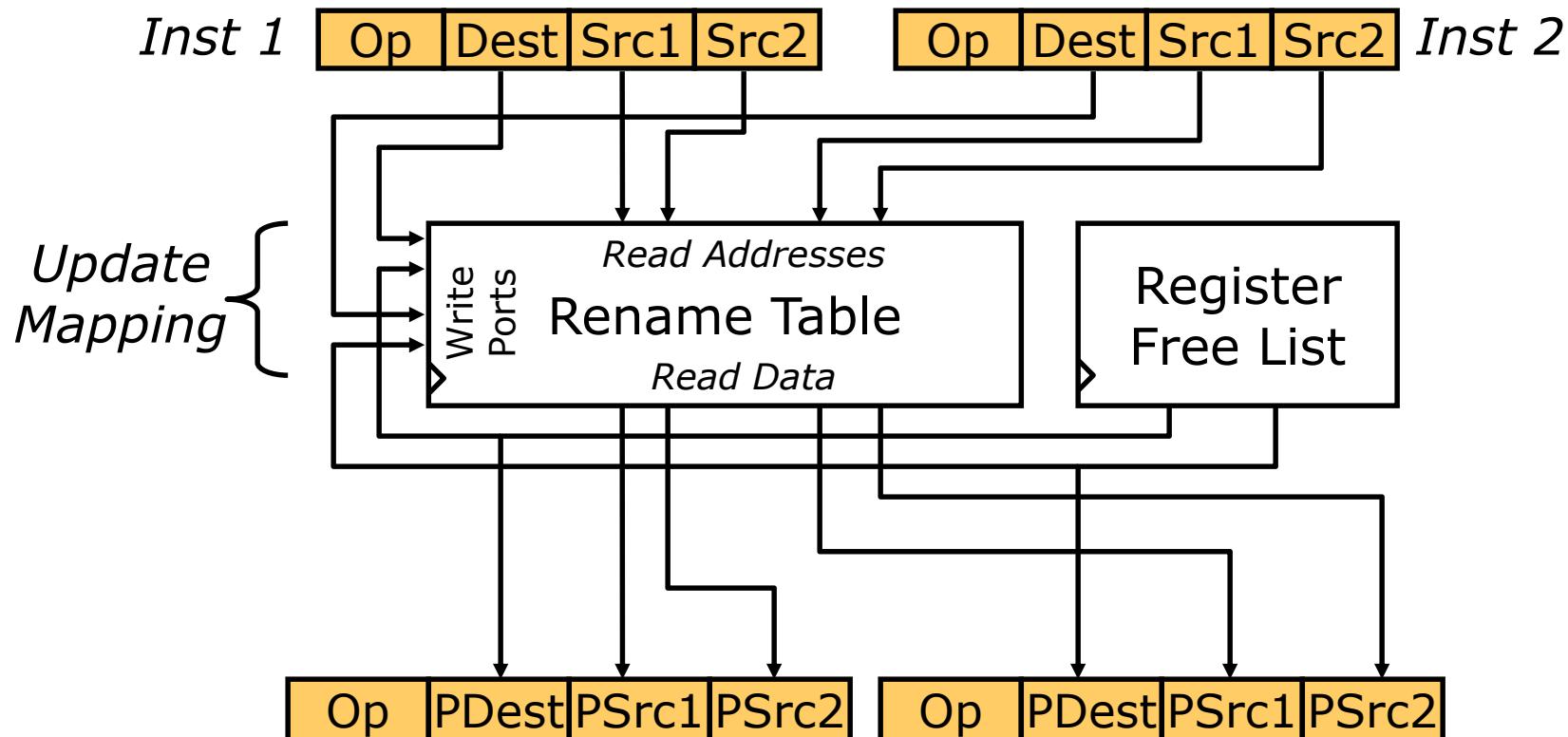
Unified-register-file style



How does issue speculation differ, e.g., on cache miss?

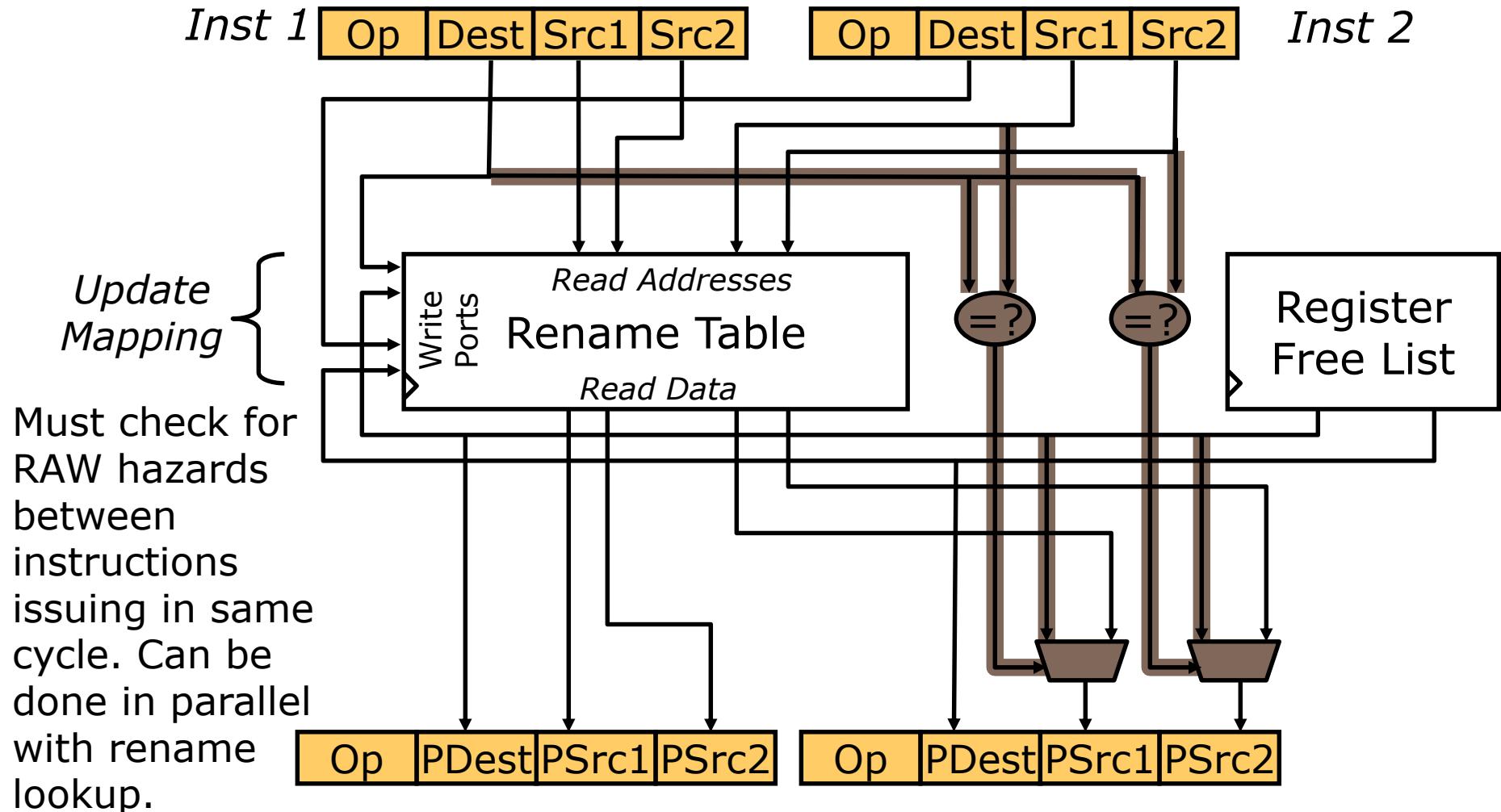
Superscalar Register Renaming

- During decode, instructions allocated new physical destination register
- Source operands renamed to physical register with newest value
- Execution unit only sees physical register numbers



Does this work?

Superscalar Register Renaming



(MIPS R10K renames 4 serially-RAW-dependent insts/cycle)

Split Issue and Commit Queues

- How large should the ROB be?
 - Think Little's Law...
- Can split ROB into issue and commit queues

Issue Queue

use	op	p1	PR1	p2	PR2	tag

Commit Queue

ex	Rd	LPRd	PRd

- Commit queue: Allocate on decode, free on commit
- Issue queue: Allocate on decode, free on dispatch
- Pros: Smaller issue queue → simpler dispatch logic
- Cons: More complex mis-speculation recovery

Speculating Both Directions?

An alternative to branch prediction is to execute both directions of a branch *speculatively*

- Resource requirement is proportional to the number of concurrent speculative executions
- Only half the resources engage in useful work when both directions of a branch are executed speculatively
- Branch prediction takes less resources than speculative execution of both paths

With accurate branch prediction, it is more cost effective to dedicate all resources to the predicted direction

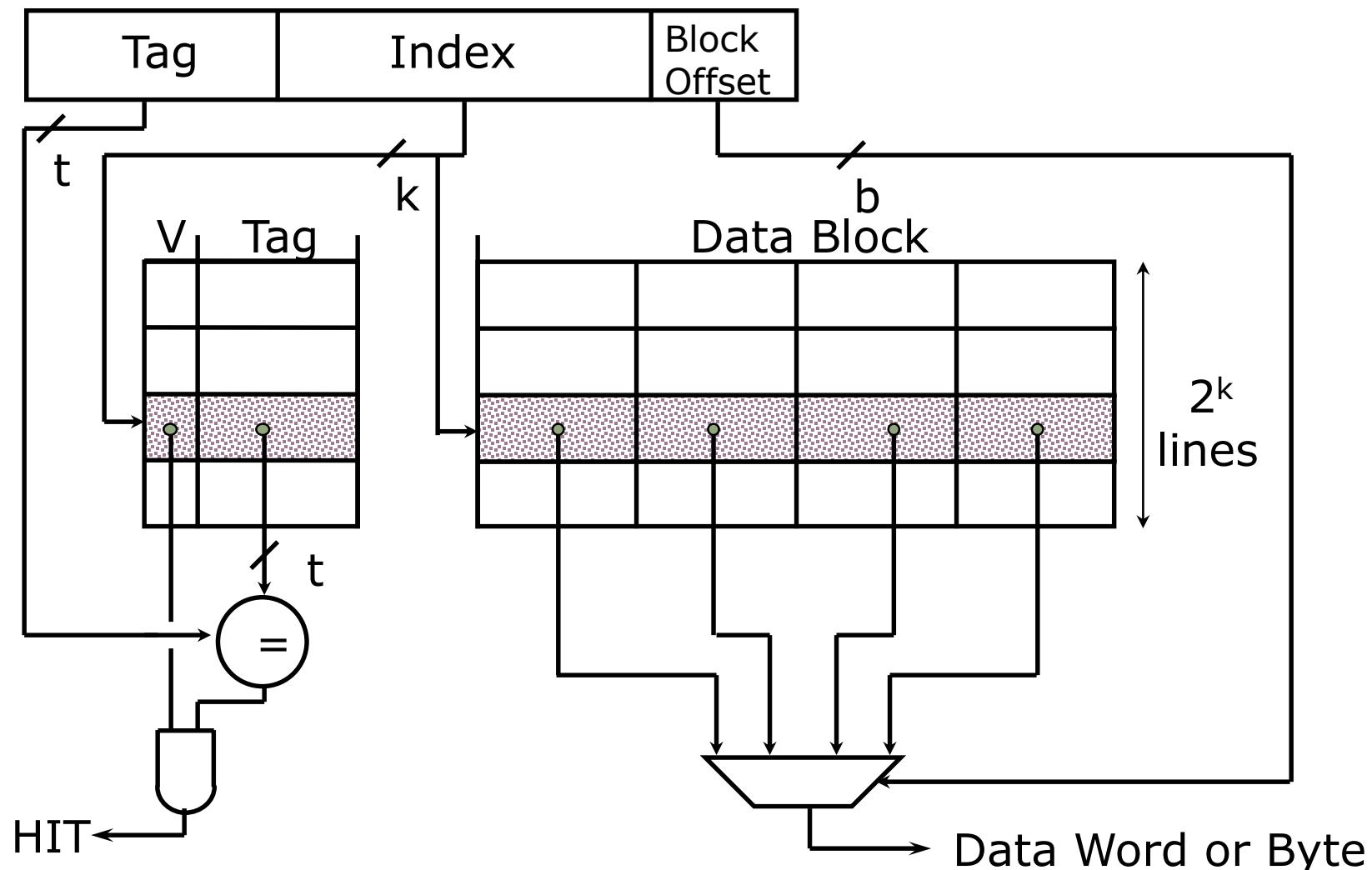
Thank you!

Advanced Memory Operations

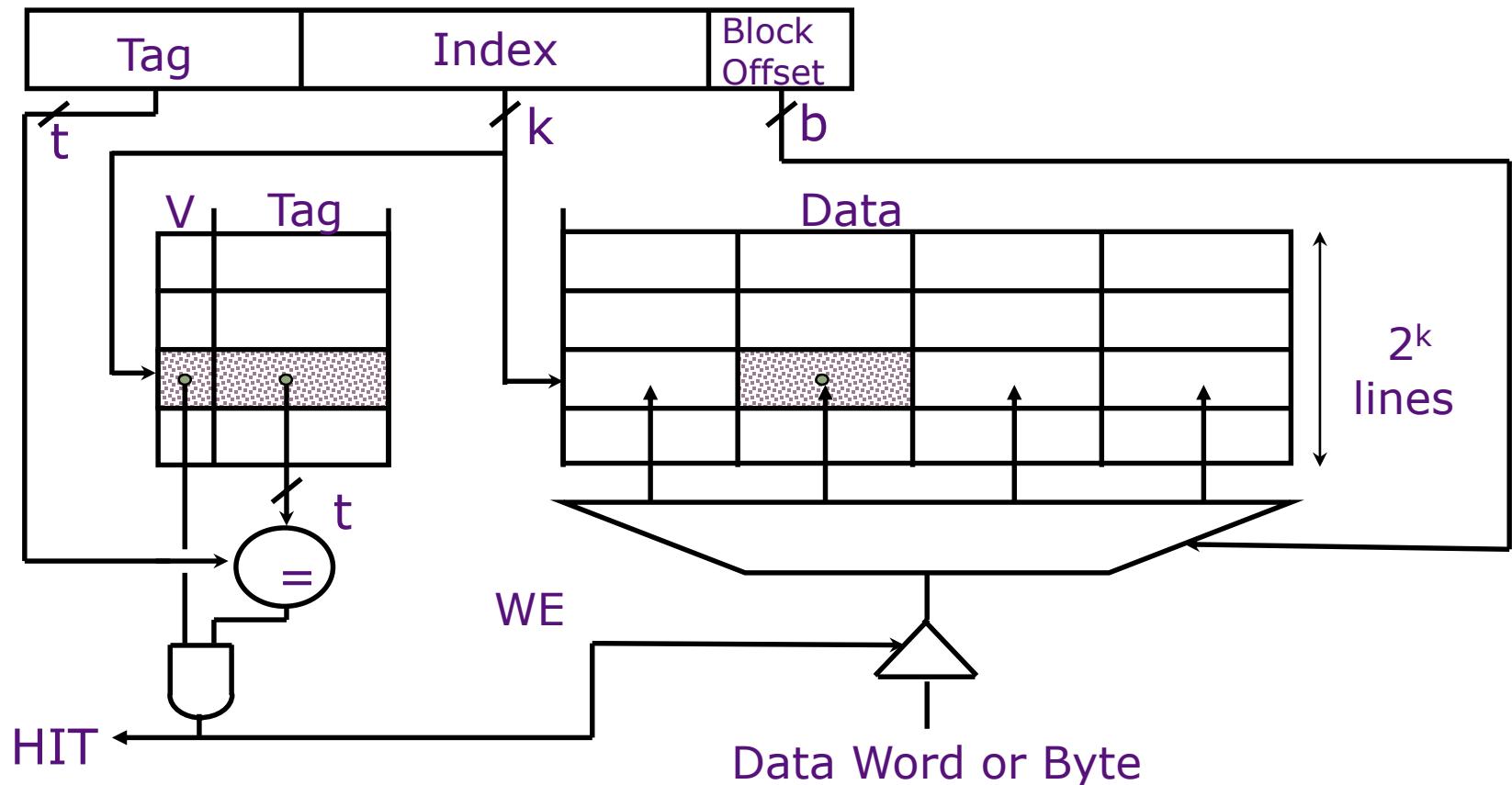
Daniel Sanchez

Computer Science and Artificial Intelligence Laboratory
M.I.T.

Reminder: Direct-Mapped Cache



Write Performance



How does write timing compare to read timing?

Reducing Write Hit Time

Problem: Writes take two cycles in memory stage, one cycle for tag check plus one cycle for data write if hit

View: Treat as data dependence on micro-architectural value 'hit/miss'

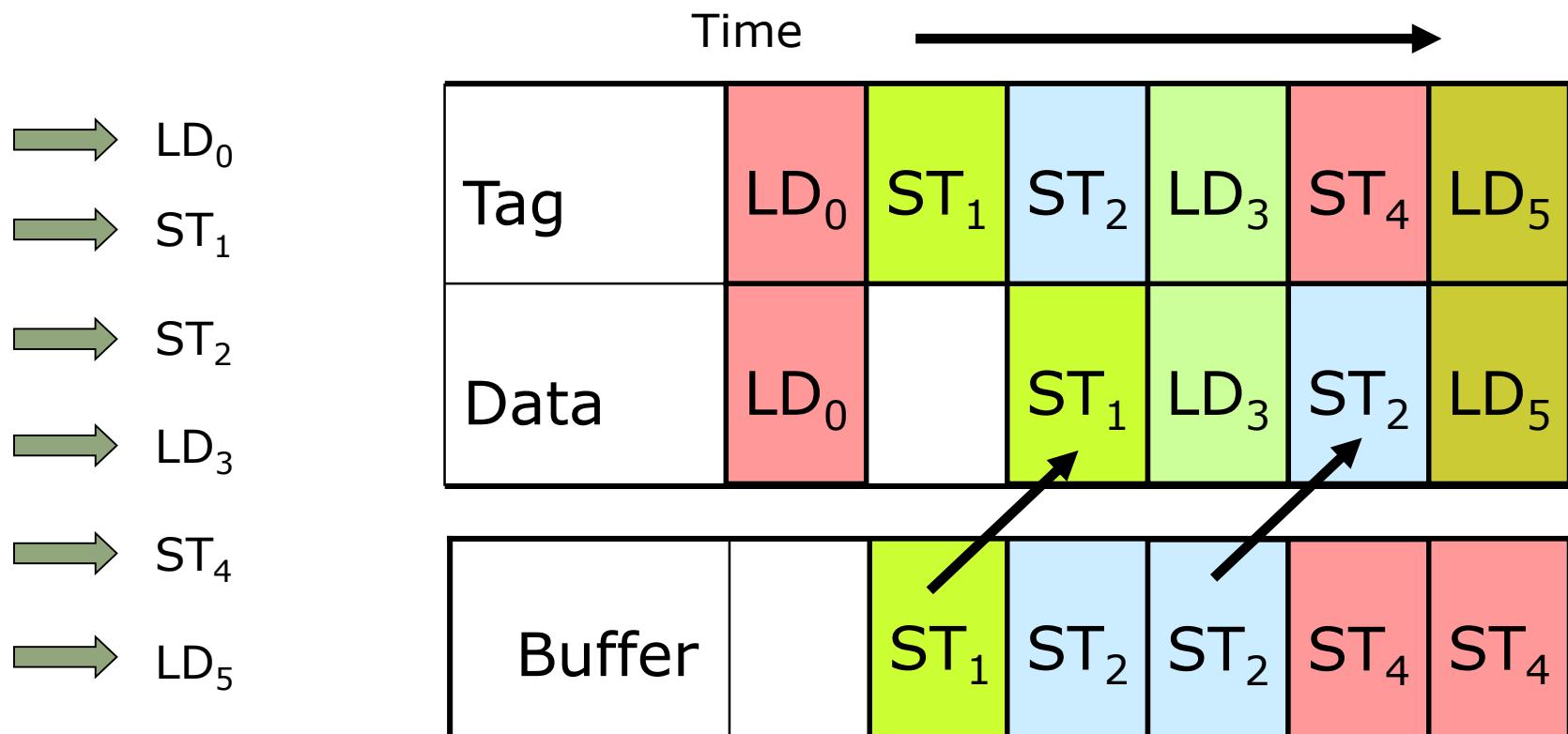
Solutions:

- Wait – delivering data as fast as possible:
 - Fully associative (CAM Tag) caches: Word line only enabled if hit
- Speculate predicting hit with greedy data update:
 - Design data RAM that can perform read and write in one cycle
 - Restore old value after tag miss (abort)
- Speculate predicting miss with lazy data update:
 - Hold write data for store in single buffer ahead of cache
 - Write cache data during next idle data access cycle (commit)

Pipelined/Delayed Write Timing

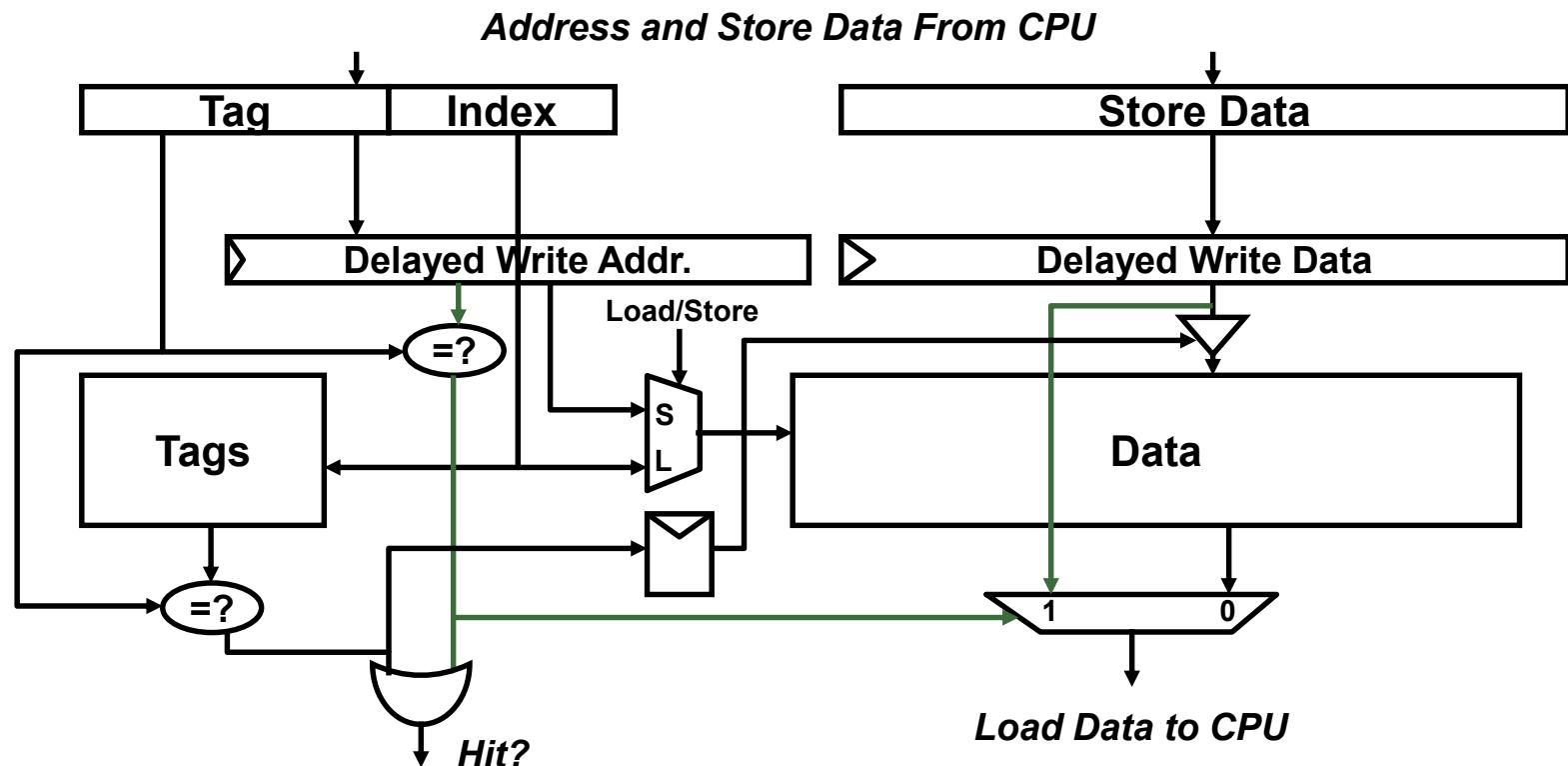
Problem: Need to commit lazily saved write data

Solution: Write data during idle data cycle of next store's tag check



Pipelining Cache Writes

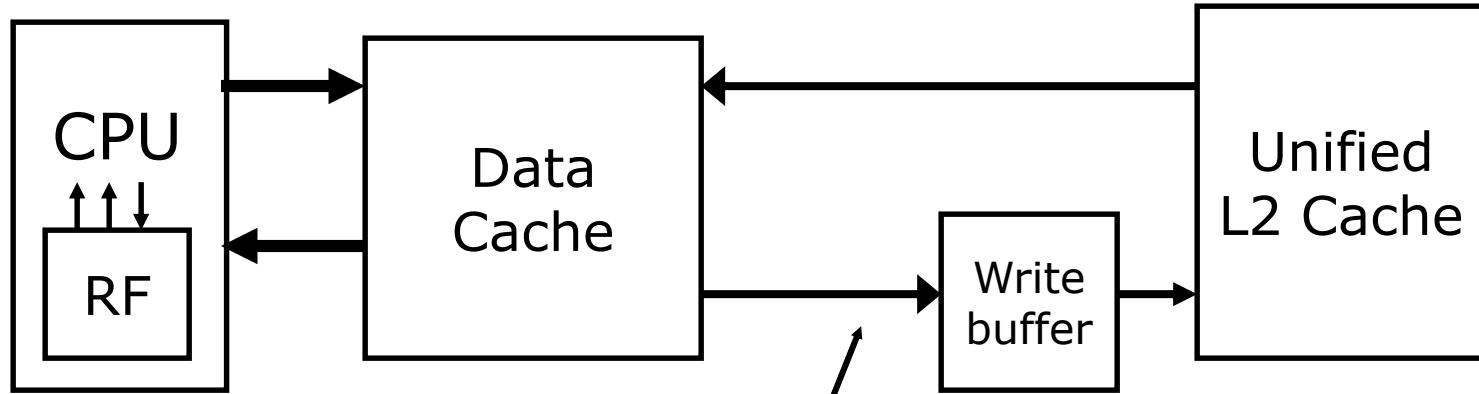
What if instruction needs data in delayed write buffer?



Write Policy Choices

- Cache hit:
 - **Write-through:** write both cache & memory
 - generally higher traffic but simplifies multi-processor design
 - **Write-back:** write cache only
(memory is written only when the entry is evicted)
 - a dirty bit per block can further reduce the traffic
- Cache miss:
 - **No-write-allocate:** only write to main memory
 - **Write-allocate** (*aka fetch on write*): fetch into cache
- Common combinations:
 - write-through and no-write-allocate
 - write-back with write-allocate

Reducing Read Miss Penalty



Evicted dirty lines for writeback cache
OR
All writes in writethrough cache

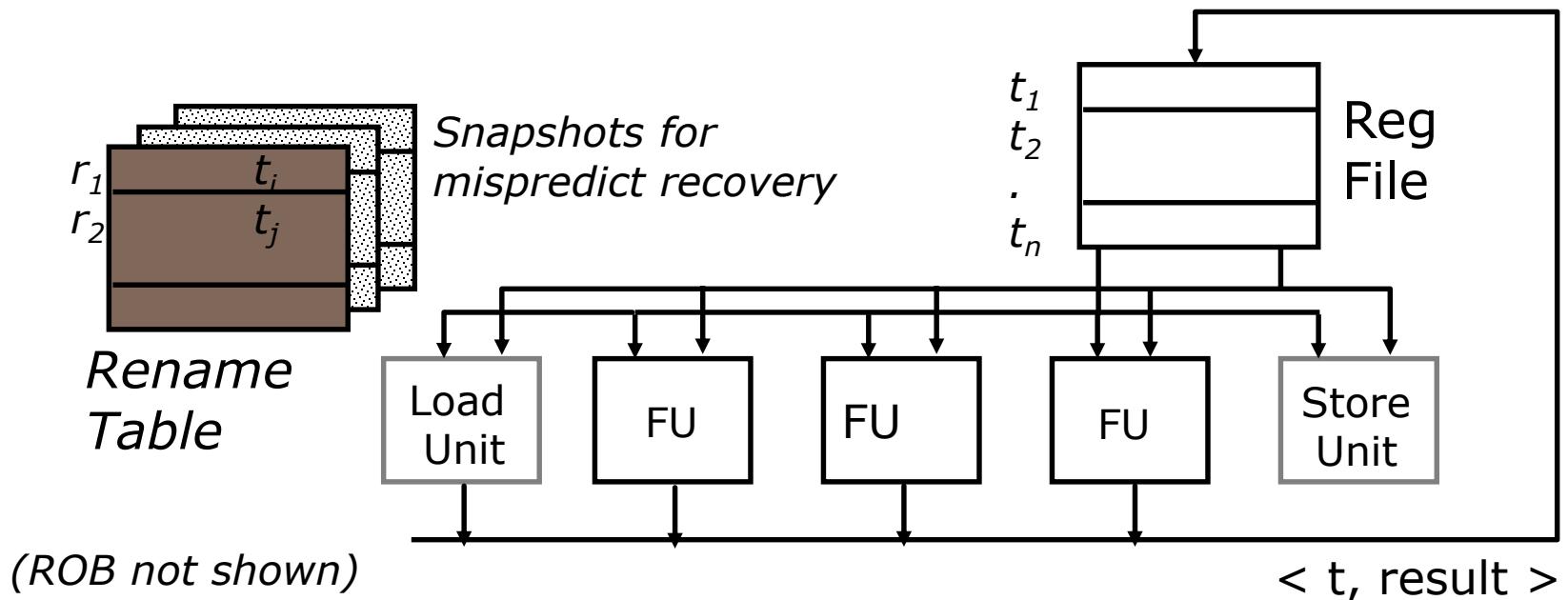
Problem: Write buffer may hold updated value of location needed by a read miss – RAW data hazard

Stall: On a read miss, wait for the write buffer to go empty

Bypass: Check write buffer addresses against read miss addresses, if no match, allow read miss to go ahead of writes, else, return value in write buffer

O-o-O With Physical Register File

(MIPS R10K, Alpha 21264, Pentium 4)



We've handled the register dependencies, but what about memory operations?

Speculative Loads / Stores

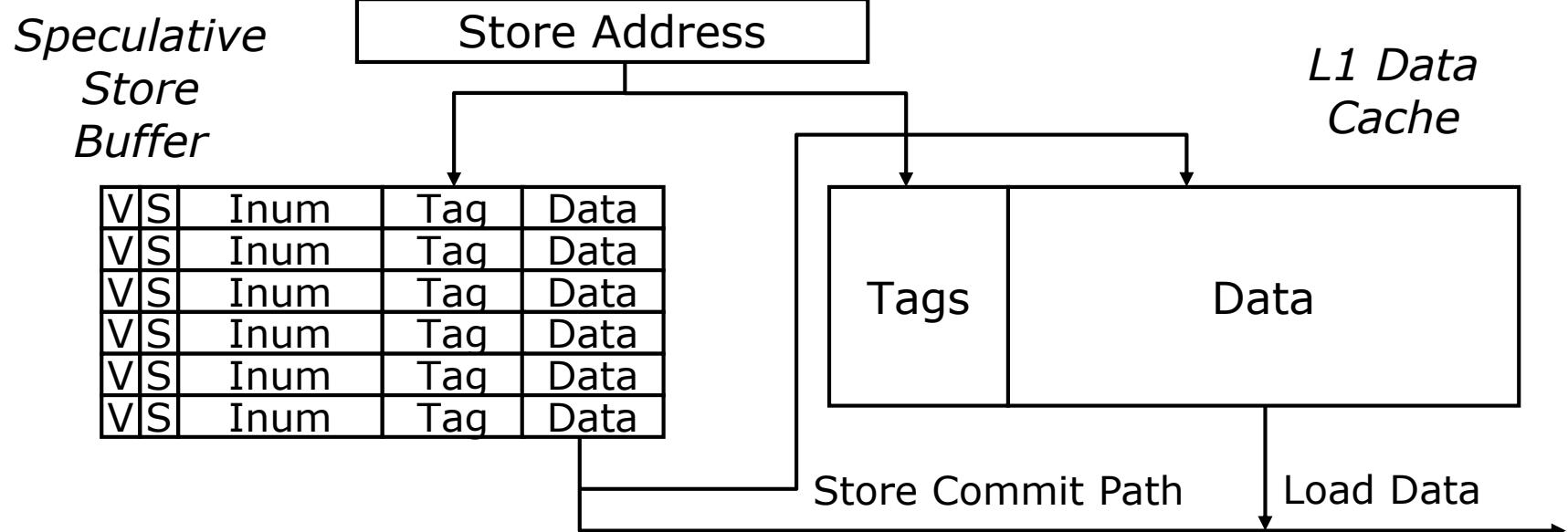
- Problem: Just like register updates, stores should not permanently change the architectural memory state until after the instruction is committed
- Choice: Data update policy: greedy or lazy?
Lazy: Add a speculative store buffer, a structure to lazily hold speculative store data.
- Choice: Handling of store-to-load data hazards:
stall, bypass, speculate...?
Bypass: ...

Store Buffer Responsibilities

- **Lazy store of data:** Buffer new data values for stores
- **Commit/abort:** The data from the oldest instructions must either be committed to memory or forgotten
- **Bypass:** Data from older instructions must be provided (or forwarded) to younger instructions before the older instruction is committed

Commits are generally done in order – why?

Store Buffer – Lazy data management



- On store execute:
 - mark valid and speculative; save tag, data, and instruction number
- On store commit:
 - clear speculative bit and eventually move data to cache
- On store abort:
 - clear valid bit

Store Buffer - Bypassing

What fields must be examined for bypassing?

Load Address

V	S	Inum	Tag	Data
V	S	Inum	Tag	Data
V	S	Inum	Tag	Data
V	S	Inum	Tag	Data
V	S	Inum	Tag	Data
V	S	Inum	Tag	Data
V	S	Inum	Tag	Data

- If data in both store buffer and cache, which should we use?
- If same address in store buffer twice, which should we use?
- Calculating entry needed in the store buffer can be considered a dependence on the index needed to access the store buffer. So store buffer bypassing can be managed speculatively by building a simple predictor that guesses that the specific entry in the store buffer the load needs. So what happens if we guessed the wrong entry?

Memory Dependencies

For registers, we used tags or physical register numbers to determine dependencies. What about memory operations?

st r1, (r2)
ld r3, (r4)

When is the load dependent on the store?

Does our ROB know this at issue time?

In-Order Memory Queue

st r1, (r2)
ld r3, (r4)

Stall naively:

- Execute all loads and stores in program order
=> Load and store cannot start execution until all previous loads and stores have completed execution
- Can still execute loads and stores speculatively, and out-of-order with respect to other instructions

Conservative O-o-O Load Execution

st r1, (r2)
ld r3, (r4)

Stall intelligently:

- Split execution of store instruction into two phases:
address calculation and data write
- Can execute load before store, if addresses known and $r4 \neq r2$
- Each load address compared with addresses of all previous
uncommitted stores (*can use partial conservative check,
e.g., bottom 12 bits of address*)
- Don't execute load if any previous store address not known

(MIPS R10K, 16 entry address queue)

Address Speculation

st r1, (r2)
ld r3, (r4)

1. Guess that $r4 \neq r2$, and execute load before store address known
2. If $r4 \neq r2$ commit...
3. But if $r4 == r2$, squash load and *all* following instructions
 - To support squash we need to hold all completed but uncommitted load/store addresses/data in program order

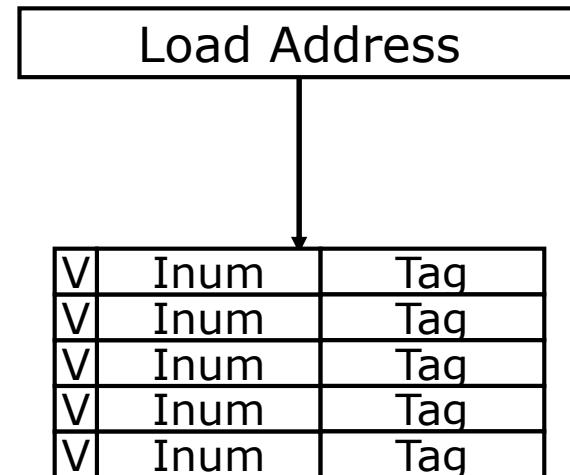
How do we resolve the speculation, i.e., detect when we need to squash?

Speculative Load Buffer

Speculation check:

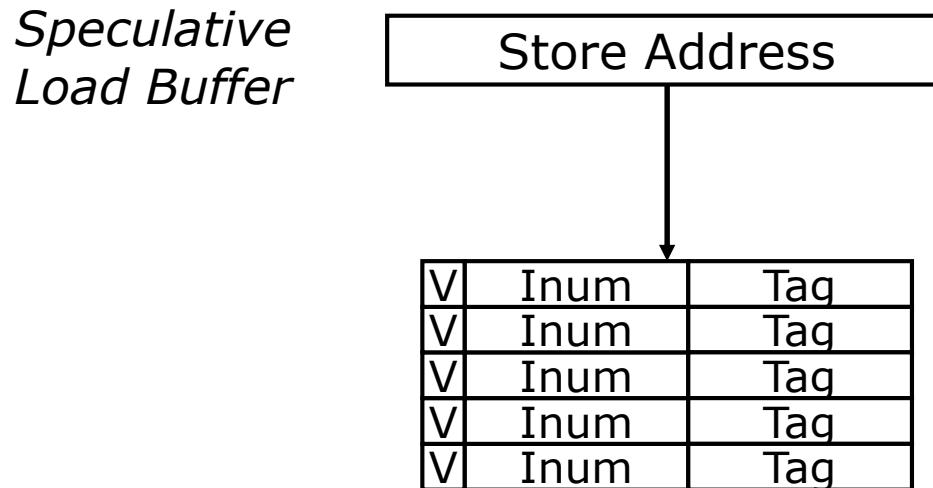
Detect if a load has executed before an earlier store to the same address – missed RAW hazard

Speculative Load Buffer



- On load execute:
 - mark entry valid, and instruction number and tag of data.
- On load commit:
 - clear valid bit
- On load abort:
 - clear valid bit

Speculative Load Buffer



- If data in load buffer with instruction younger than store:
 - Speculative violation – abort!=> Large penalty for inaccurate address speculation

Does tag match have to be perfect?

Memory Dependence Prediction

(Alpha 21264)

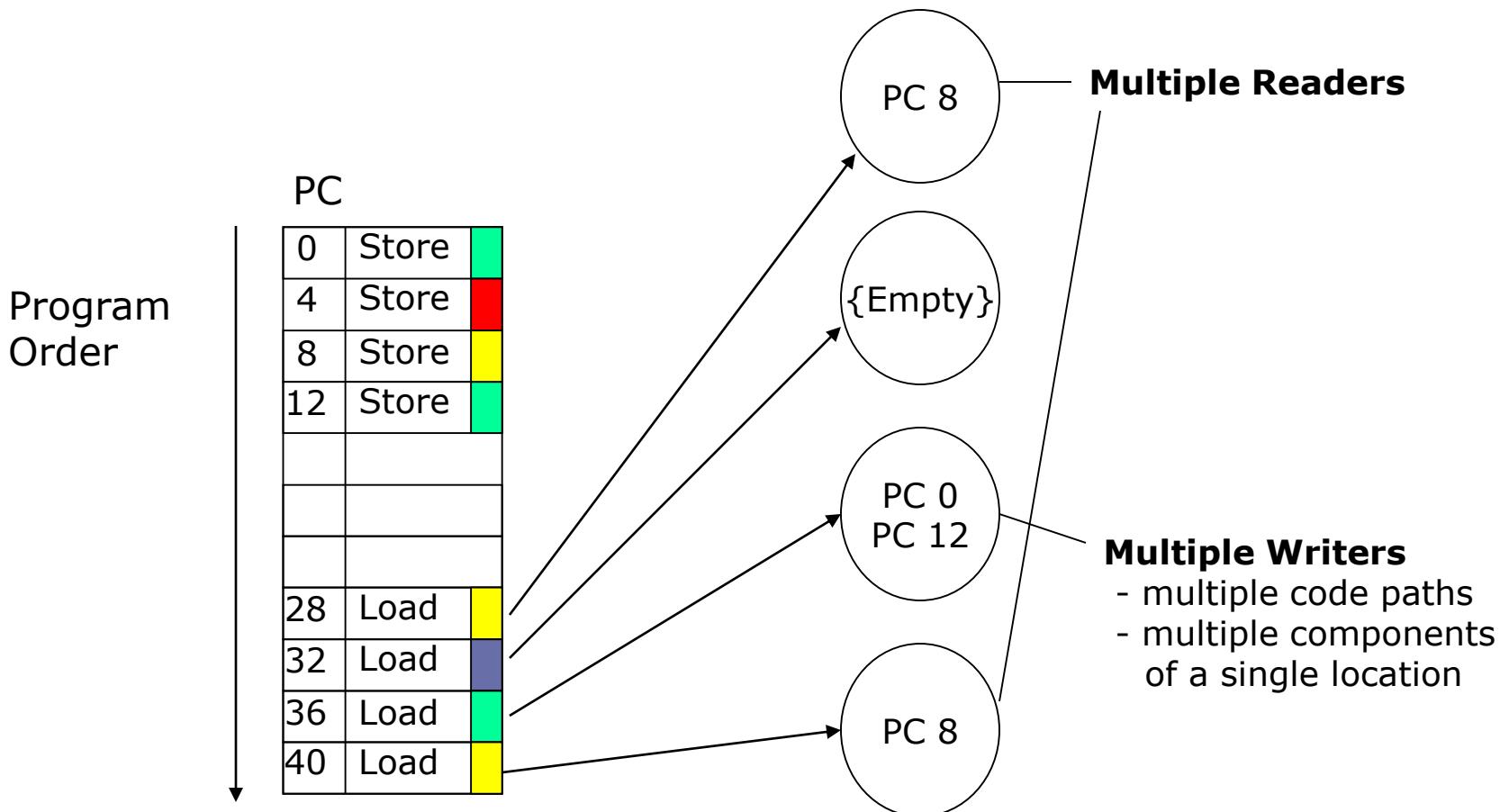
**st r1, (r2)
ld r3, (r4)**

1. Guess that $r4 \neq r2$ and execute load before store
2. If later find $r4 == r2$, squash load and all following instructions, but mark load instruction as *store-wait*
 - Subsequent executions of the same load instruction will wait for all previous stores to complete
 - Periodically clear *store-wait* bits

Notice the general problem of predictors that learn something but can't unlearn it

Store Sets

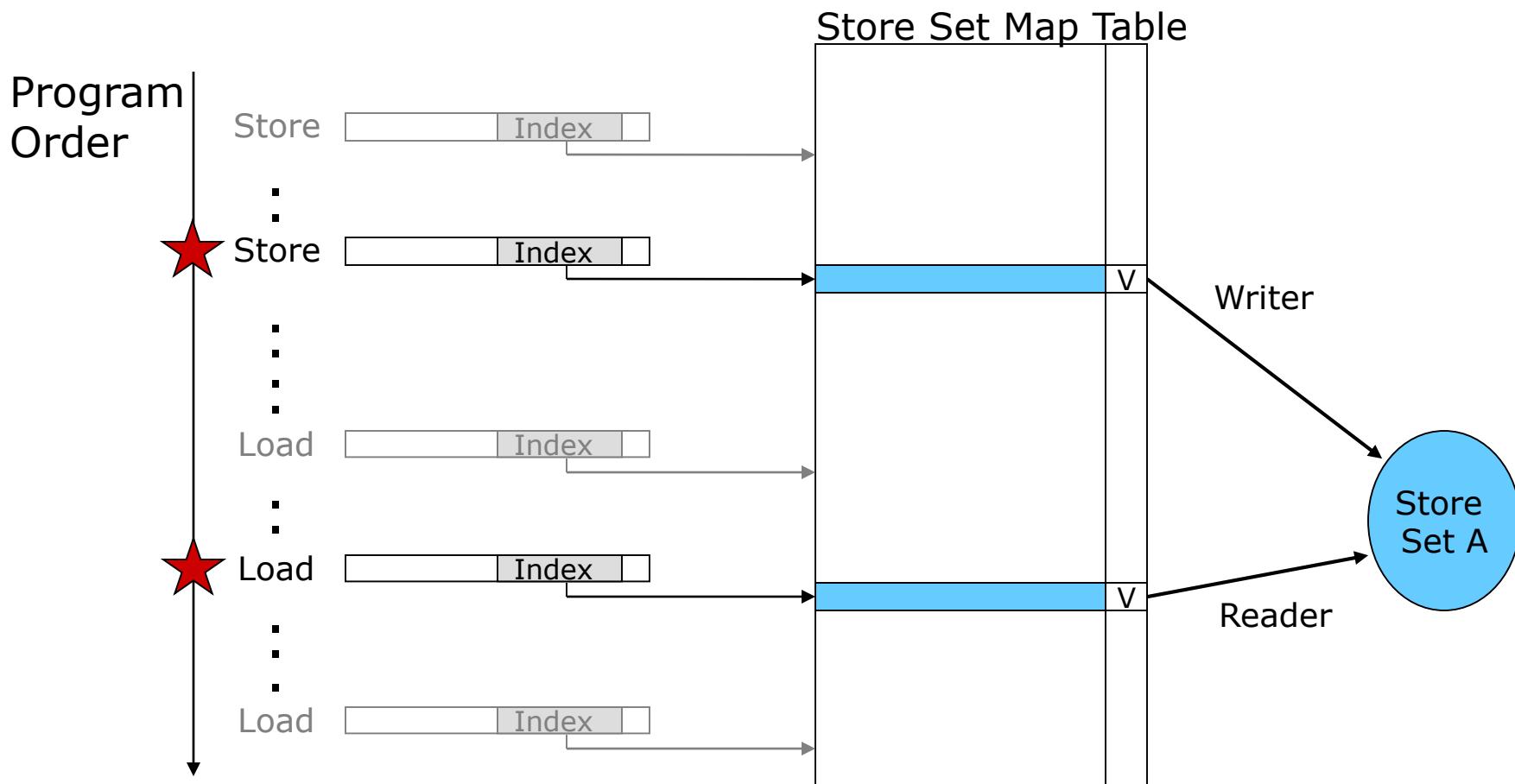
(Alpha 21464)



Memory Dependence Prediction using Store Sets

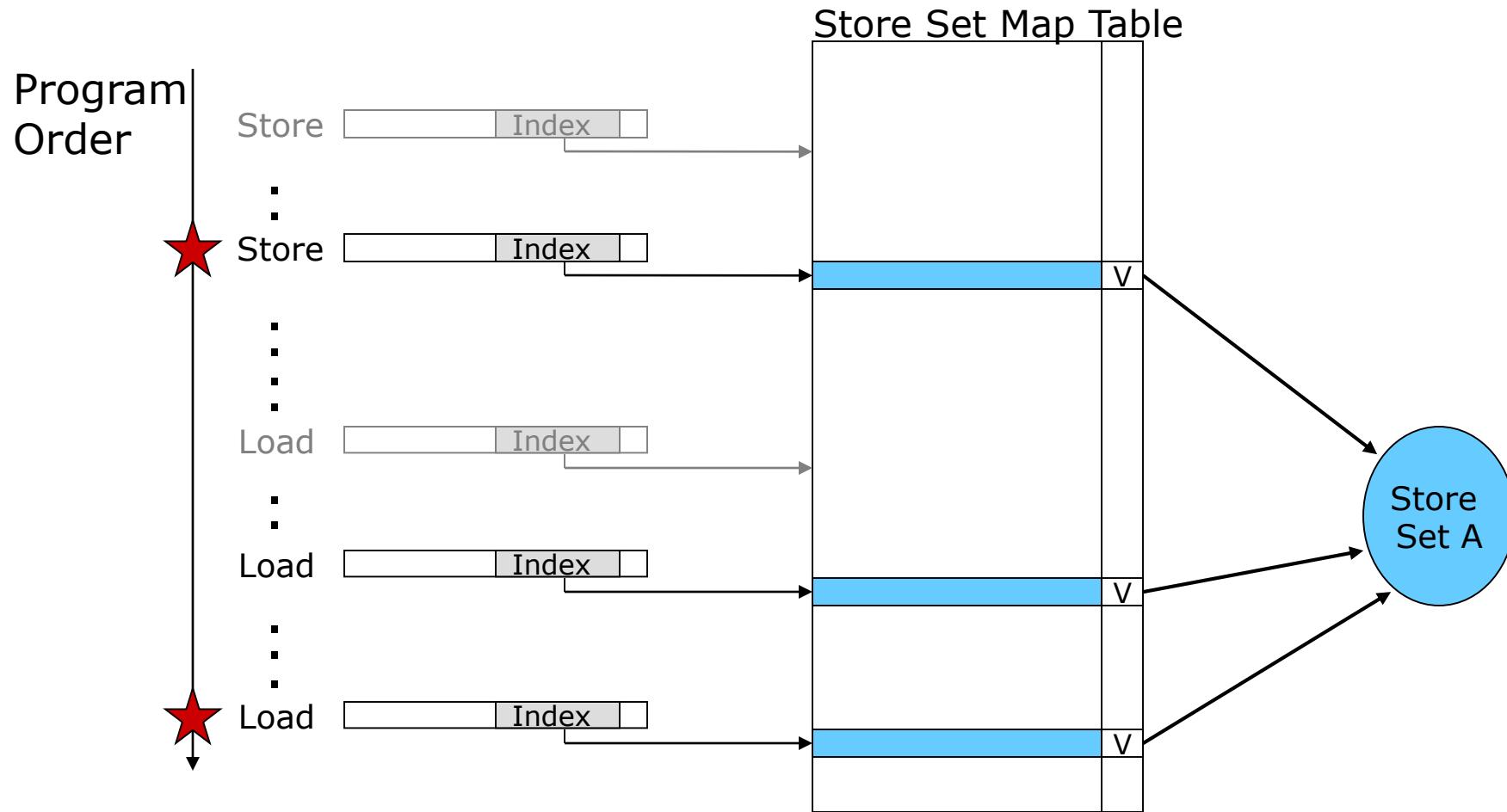
- A load must wait for any stores in its *store set* that have not yet executed
- The processor approximates each load's *store set* by initially allowing naïve speculation and recording memory-order violations

The Store Set Map Table



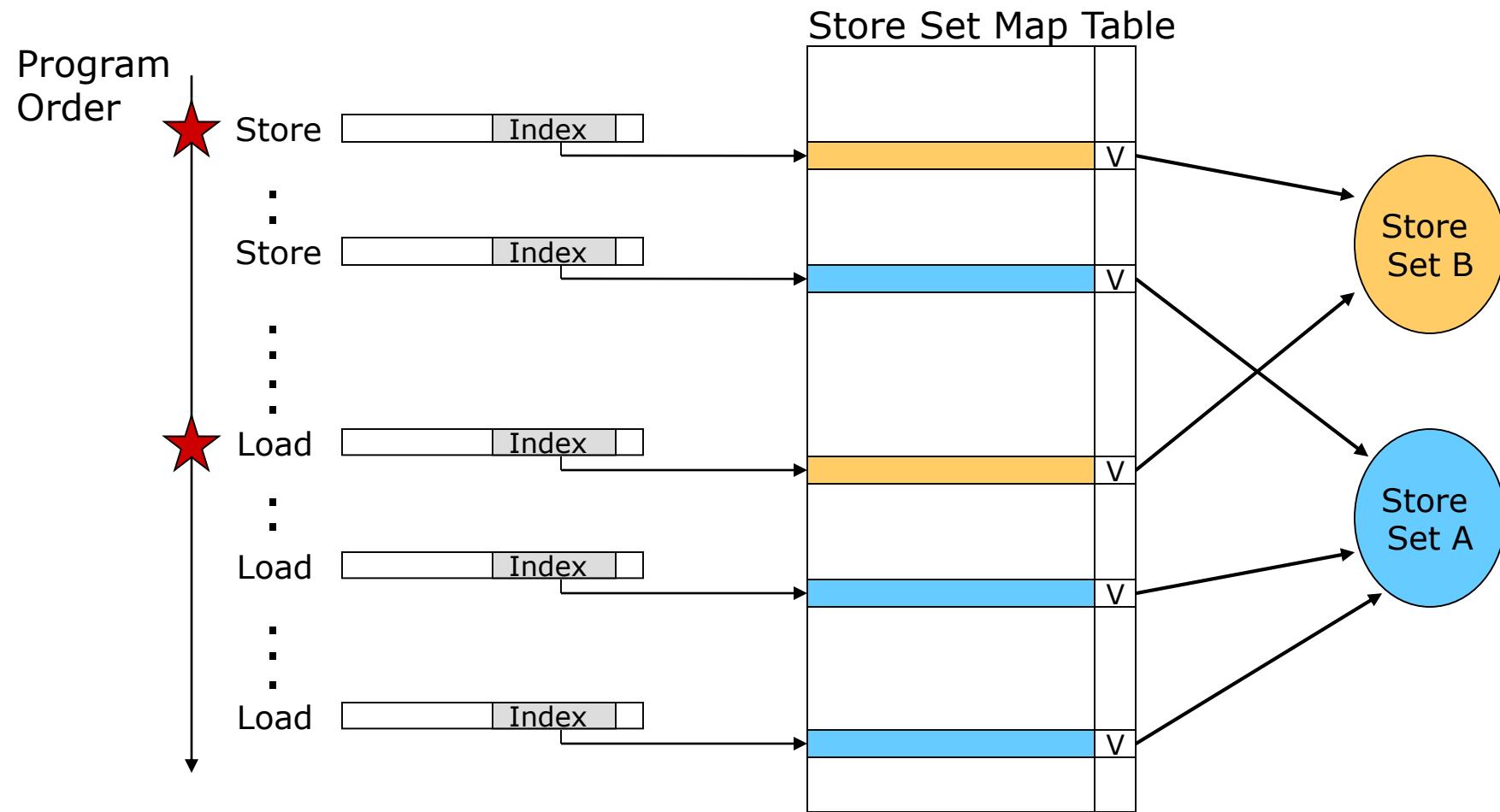
- Store/Load Pair causing Memory Order Violation

Store Set Sharing for Multiple Readers



- Store/Load Pair causing Memory Order Violation

Store Set Map Table, cont.



- Store/Load Pair causing Memory Order Violation

Prefetching

- Execution of a load 'depends' on the data it needs being in the cache...
- Speculate on future instruction and data accesses and fetch them into cache(s)
 - Instruction accesses easier to predict than data accesses
- Varieties of prefetching
 - Hardware prefetching
 - Software prefetching
 - Mixed schemes
- *How does prefetching affect cache misses?*

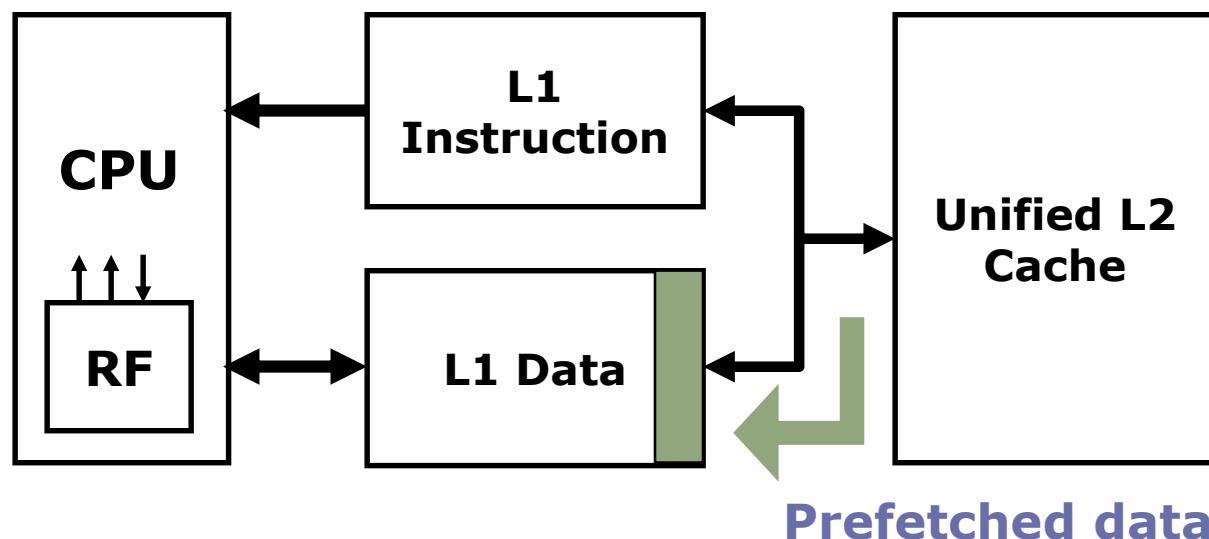
Compulsory

Conflict

Capacity

Issues in Prefetching

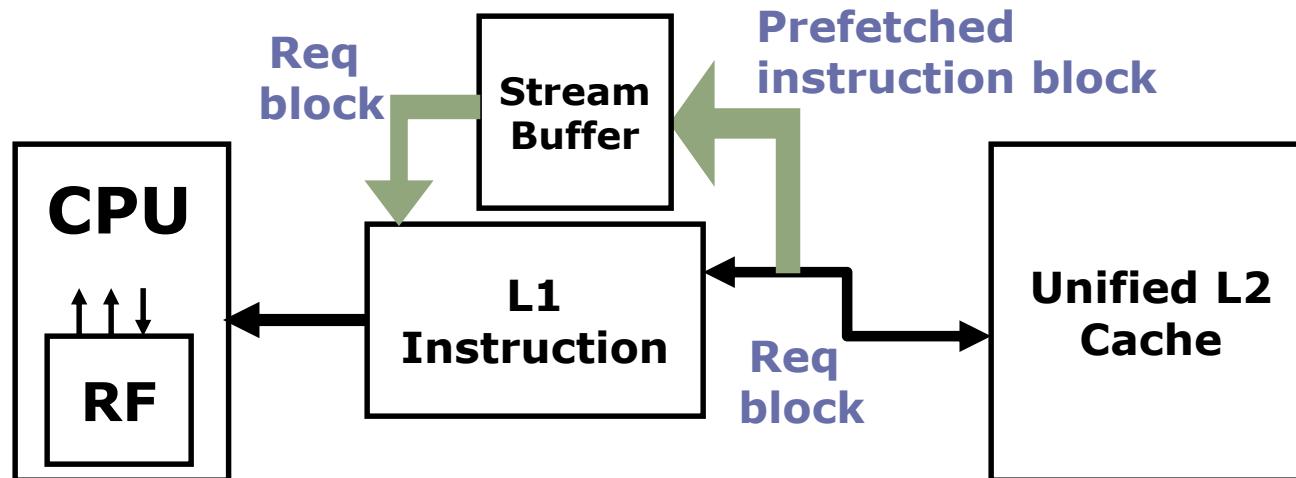
- Usefulness – should produce hits
- Timeliness – not late and not too early
- Cache and bandwidth pollution



Hardware Instruction Prefetching

Instruction prefetch in Alpha AXP 21064

- Fetch two blocks on a miss; the requested block (i) and the next consecutive block ($i+1$)
- Requested block placed in cache, and next block in instruction stream buffer
- If miss in cache but hit in stream buffer, move stream buffer block into cache and prefetch next block ($i+2$)



Hardware Data Prefetching

- Prefetch-on-miss:
 - Prefetch $b + 1$ upon miss on b
- One Block Lookahead (OBL) scheme
 - Initiate prefetch for block $b + 1$ when block b is accessed
 - Why is this different from doubling block size?*
 - Can extend to N-block lookahead (called *stream prefetching*)
- Strided prefetch
 - If observe sequence of accesses to block b , $b+N$, $b+2N$, then prefetch $b+3N$ etc.

Example: IBM Power 5 [2003] supports eight independent streams of strided prefetch per processor, prefetching 12 lines ahead of current access

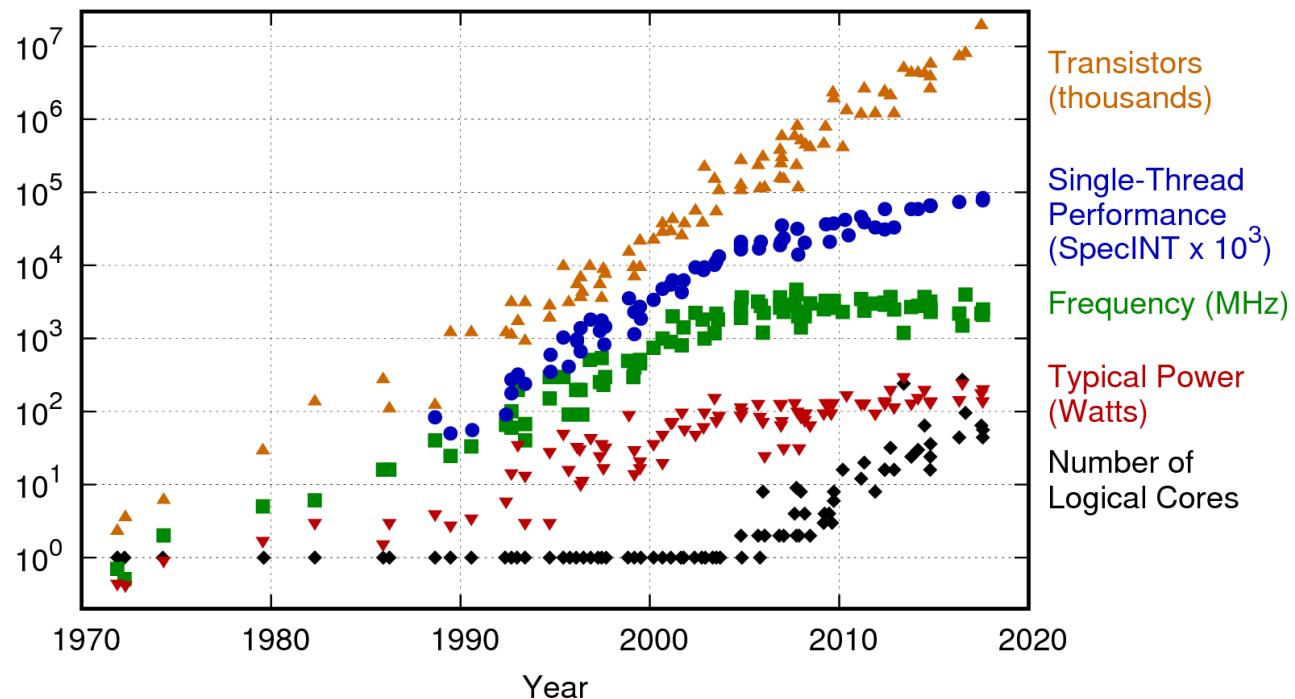
Thank you!

*Next lecture:
Cache Coherence*

Cache Coherence

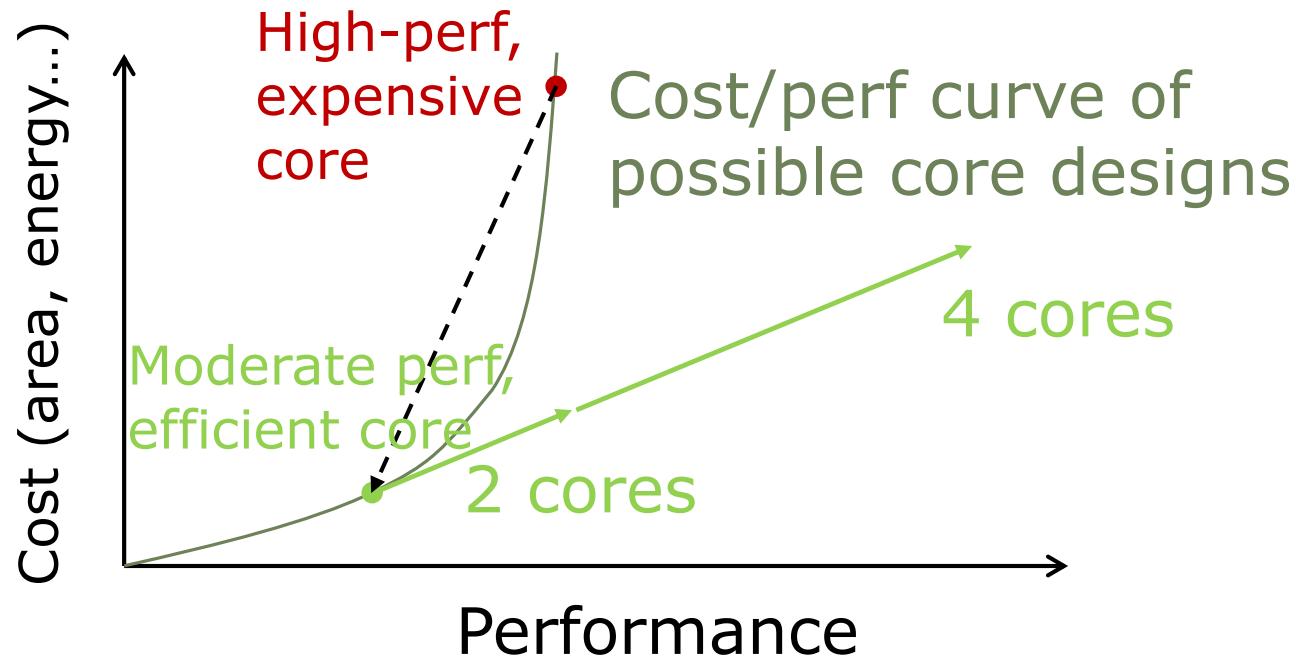
Daniel Sanchez
Computer Science & Artificial Intelligence Lab
M.I.T.

The Shift to Multicore



- Since 2005, improvements in system performance mainly due to increasing cores per chip
- Why?

Multicore Performance



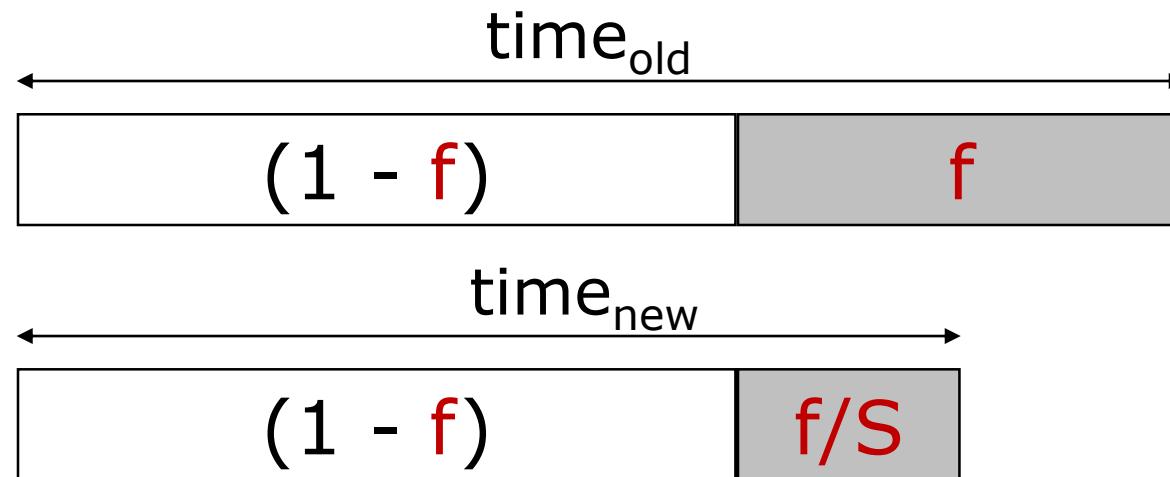
What factors may limit multicore performance?

Amdahl's Law

- Speedup = time_{without enhancement} / time_{with enhancement}
- Suppose an enhancement speeds up a fraction f of a task by a factor of S

$$\text{time}_{\text{new}} = \text{time}_{\text{old}} \cdot ((1-f) + f/S)$$

$$S_{\text{overall}} = 1 / ((1-f) + f/S)$$



Corollary: Make the common case fast

Amdahl's Law and Parallelism

- Say you write a program that can do 90% of the work in parallel, but the other 10% is sequential
- What is the maximum speedup you can get by running on a multicore machine?

$$S_{\text{overall}} = 1 / ((1-f) + f/S)$$

$$f = 0.9, S=\infty \rightarrow S_{\text{overall}} = 10$$

What f do you need to use a 1000-core machine well?

Communication Models

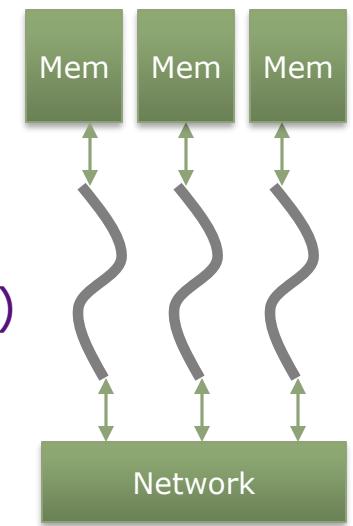
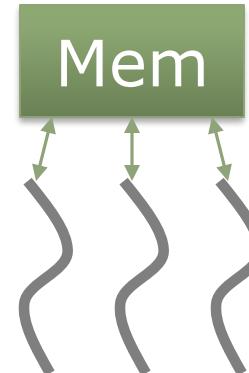
- Shared memory:

- Single address space
- Implicit communication by reading/writing memory
 - Data
 - Control (semaphores, locks, barriers, ...)
- Low-level programming model: threads

- Message passing:

- Separate address spaces
- Explicit communication by send/recv messages
 - Data
 - Control (blocking msgs, barriers, ...)
- Low-level programming model:
processes + inter-process communication (e.g., MPI)

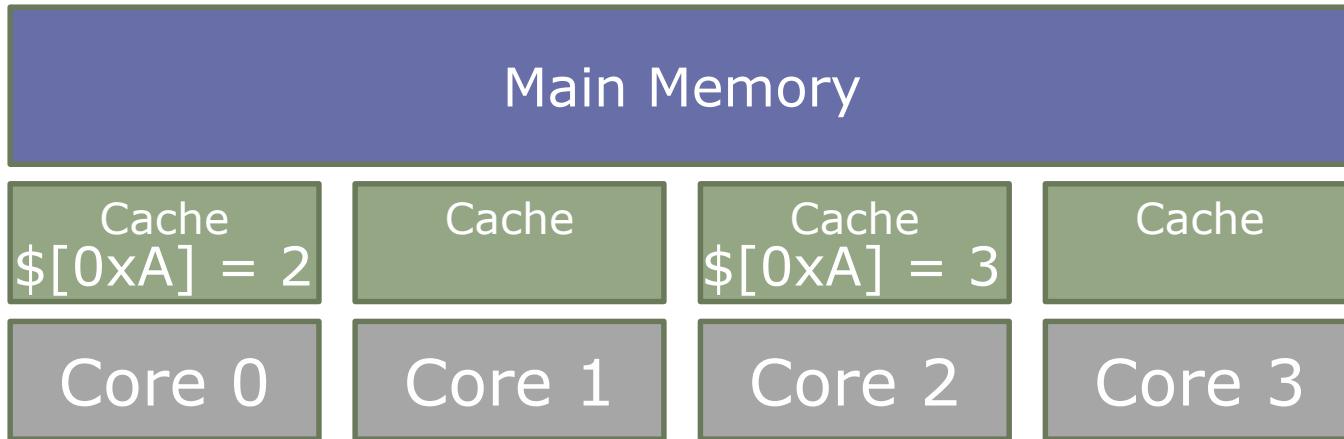
- Pros/cons of each model?



Coherence & Consistency

- Shared memory systems:
 - Have **multiple private caches** for performance reasons
 - Need to provide the illusion of a single shared memory
- Intuition: A read should return the most recently written value
 - What is “most recent”?
- Formally:
 - Coherence: What values can a read return?
 - Concerns reads/writes to a single memory location
 - Consistency: When do writes become visible to reads?
 - Concerns reads/writes to multiple memory locations

Cache Coherence Avoids Stale Data



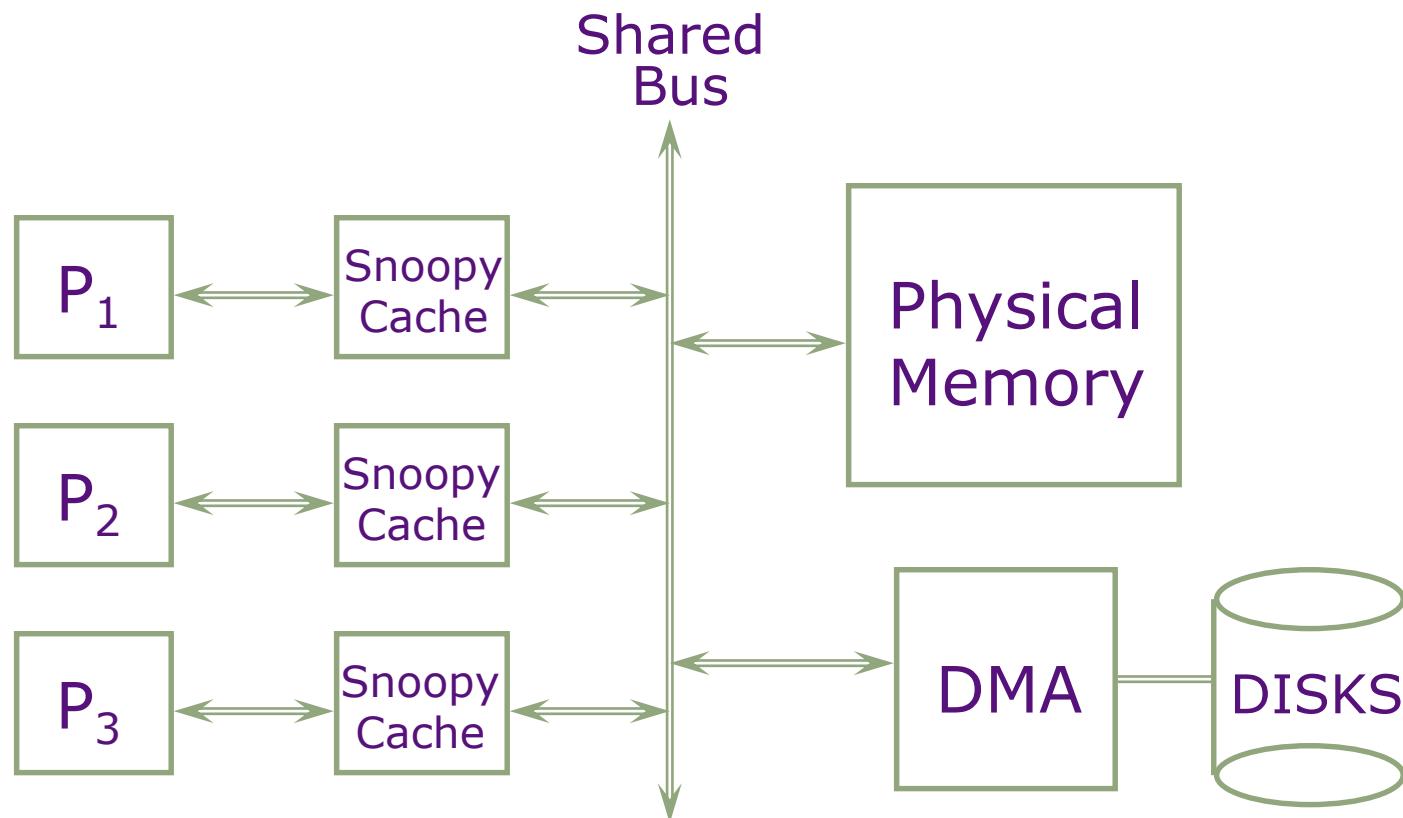
- A **cache coherence protocol** controls cache contents to avoid stale cache lines

Implementing Cache Coherence

- Coherence protocols must enforce two rules:
 - *Write propagation*: Writes eventually become visible to all processors
 - *Write serialization*: Writes to the same location are serialized (all processors see them in the same order)
- How to ensure write propagation?
 - *Write-invalidate protocols*: Invalidate all other cached copies before performing the write
 - *Write-update protocols*: Update all other cached copies after performing the write
- How to track sharing state of cached data and serialize requests to the same address?
 - *Snooping-based protocols*: All caches observe each other's actions through a shared bus (bus is the serialization point)
 - *Directory-based protocols*: A coherence directory tracks contents of private caches and serializes requests (directory is the serialization point)

Snooping-Based Coherence

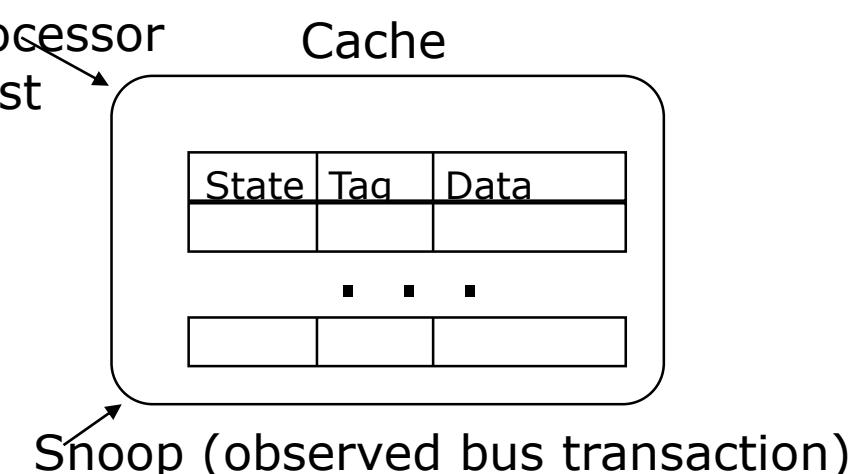
(Goodman, 1983)



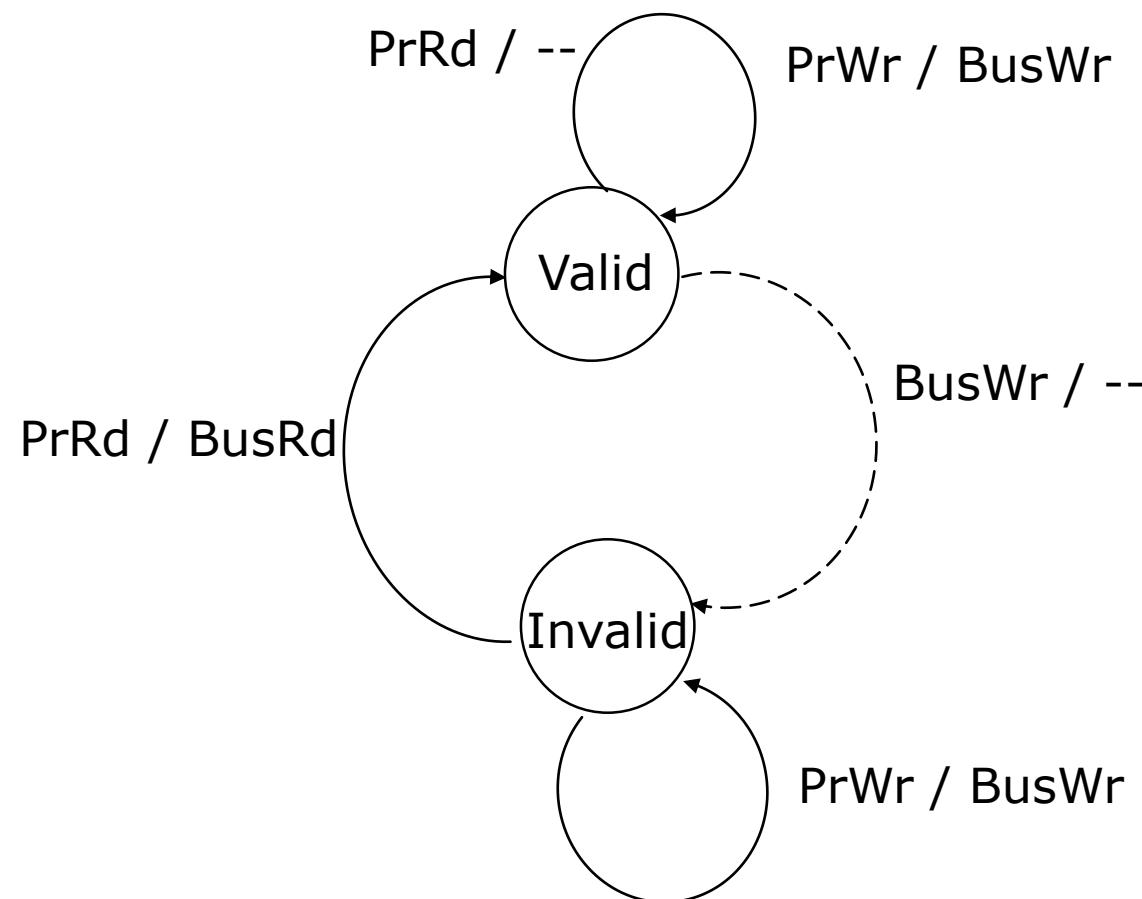
Caches watch (snoop on) bus to keep all processors' view of memory coherent

Snooping-Based Coherence

- Bus provides serialization point
 - Broadcast, totally ordered
- Controller
 - One cache controller for each core “snoops” all bus transactions
 - Controller
 - Responds to requests from core and the bus
 - changes state of the selected cache block
 - generates bus transactions to access data or invalidate
- Snoopy protocol (FSM)
 - State-transition diagram
 - Actions
- Handling writes:
 - Write-invalidate
 - Write-update



A Simple Protocol: Valid/Invalid (VI)

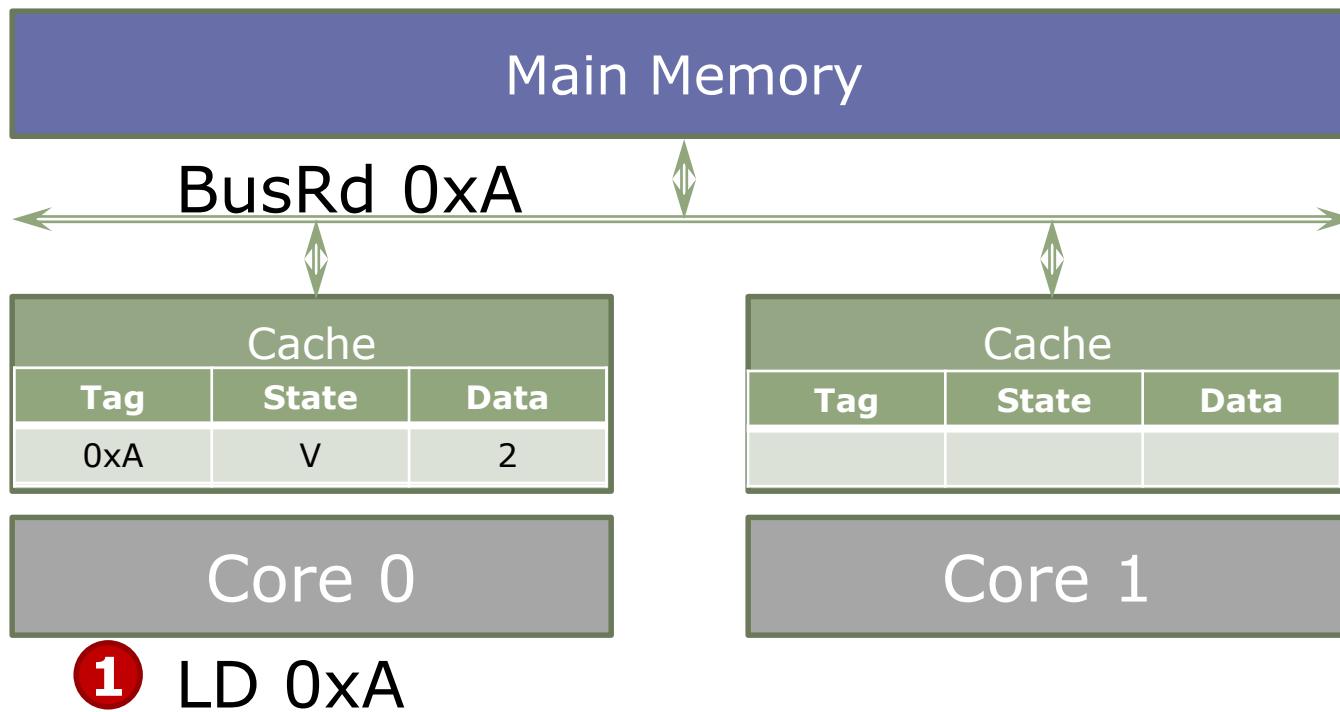


- Assume write-through caches
- Transition nomenclature:

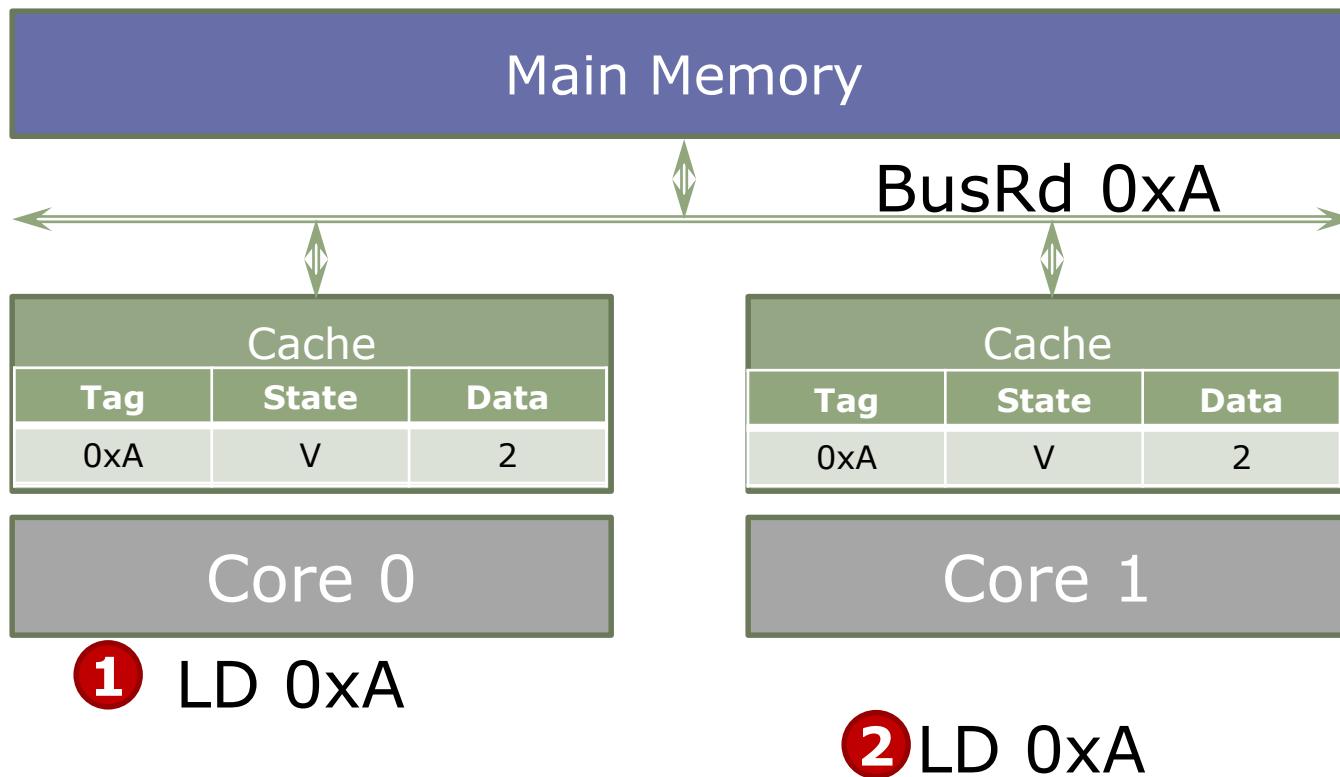
*triggering action /
taken action(s)*

Actions
Processor Read (PrRd)
Processor Write (PrWr)
Bus Read (BusRd)
Bus Write (BusWr)

Valid/Invalid Example

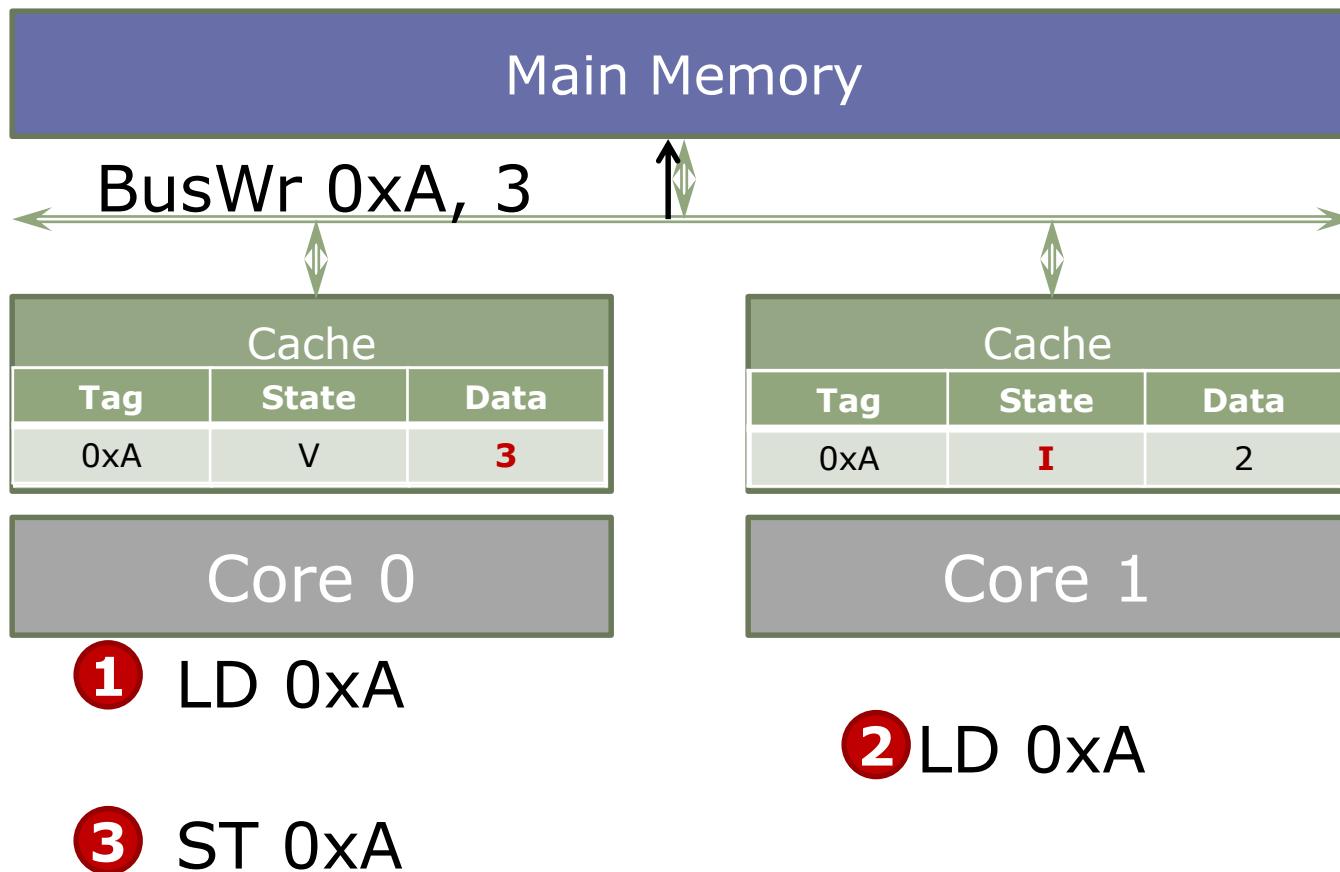


Valid/Invalid Example

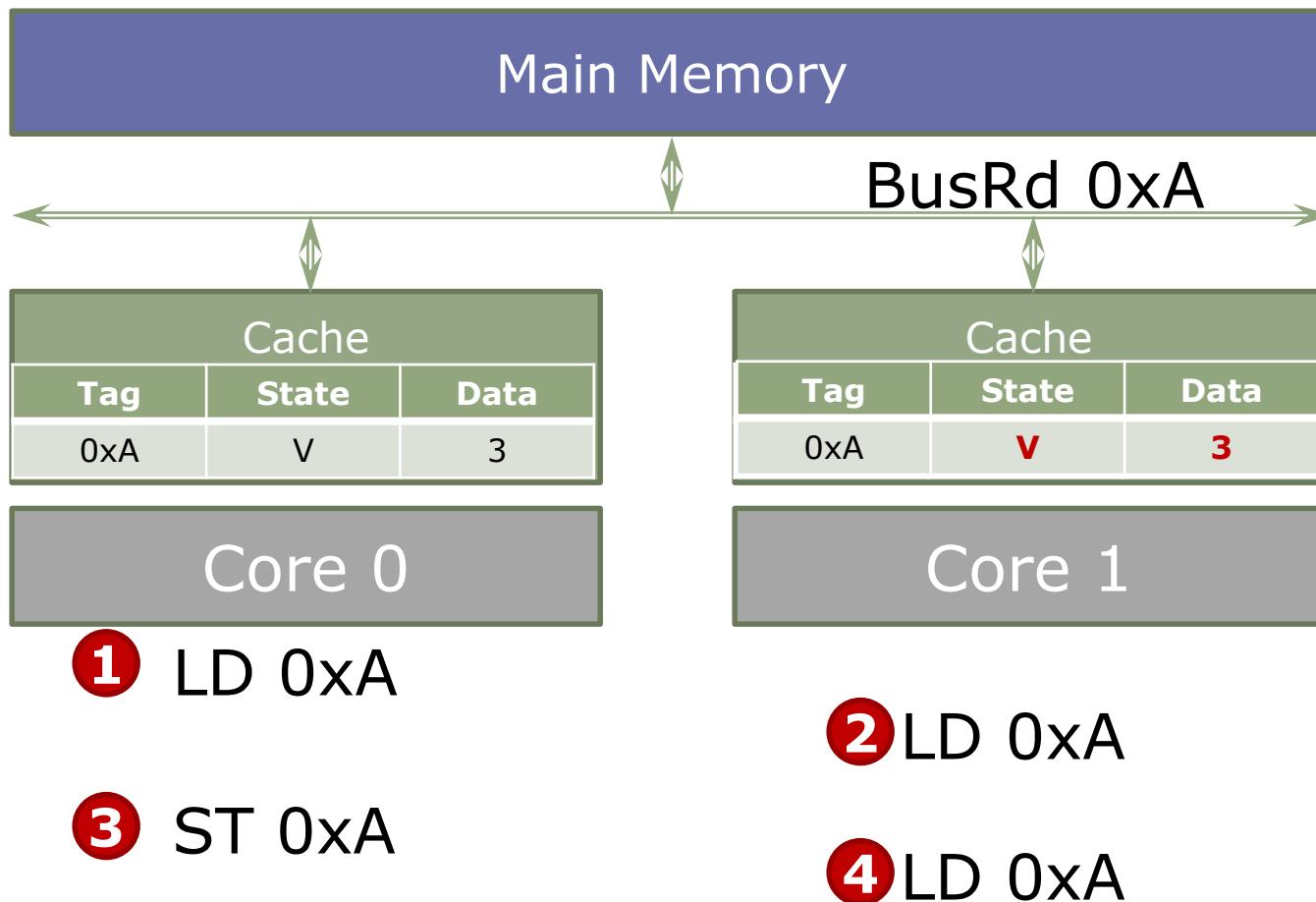


Additional loads satisfied locally, without BusRd

Valid/Invalid Example



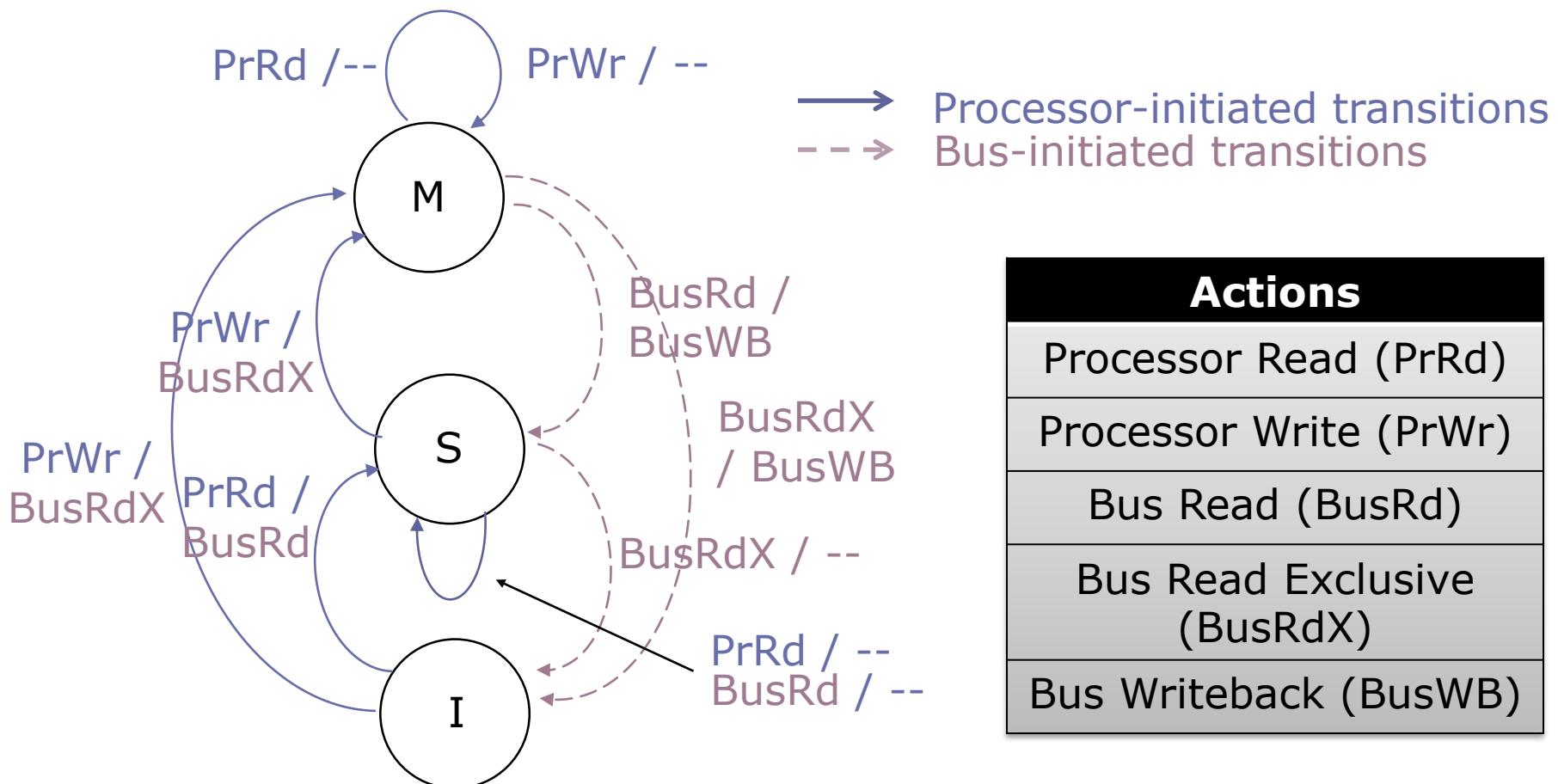
Valid/Invalid Example



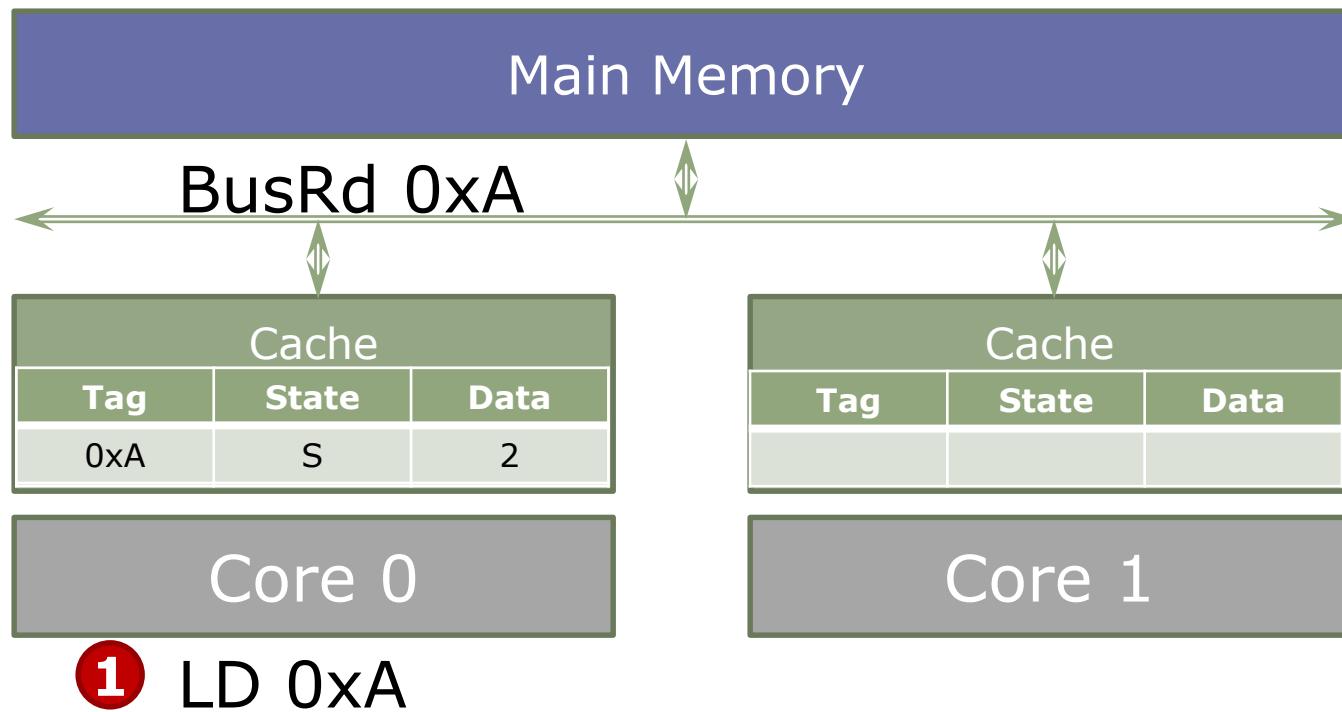
VI Problems?

Modified/Shared/Invalid (MSI) Protocol

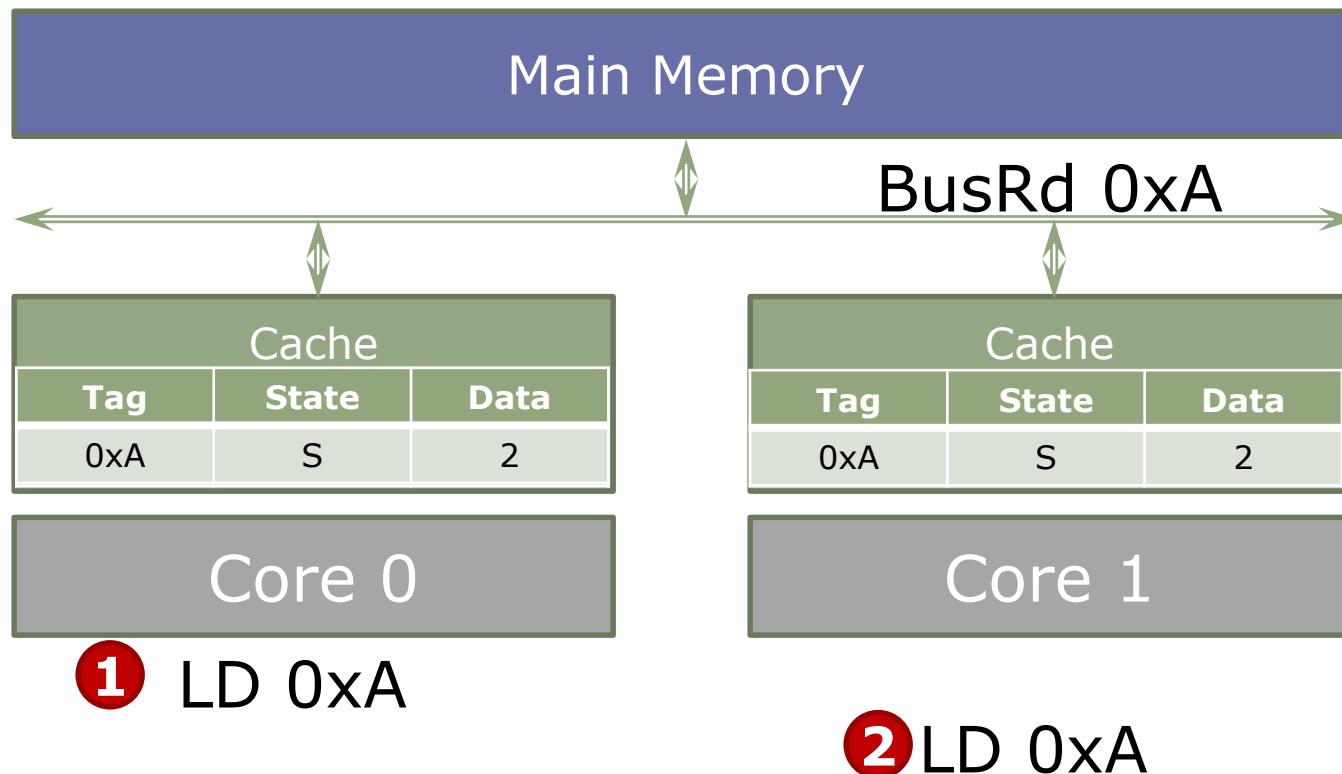
- Allows writeback caches + satisfying writes locally



MSI Example

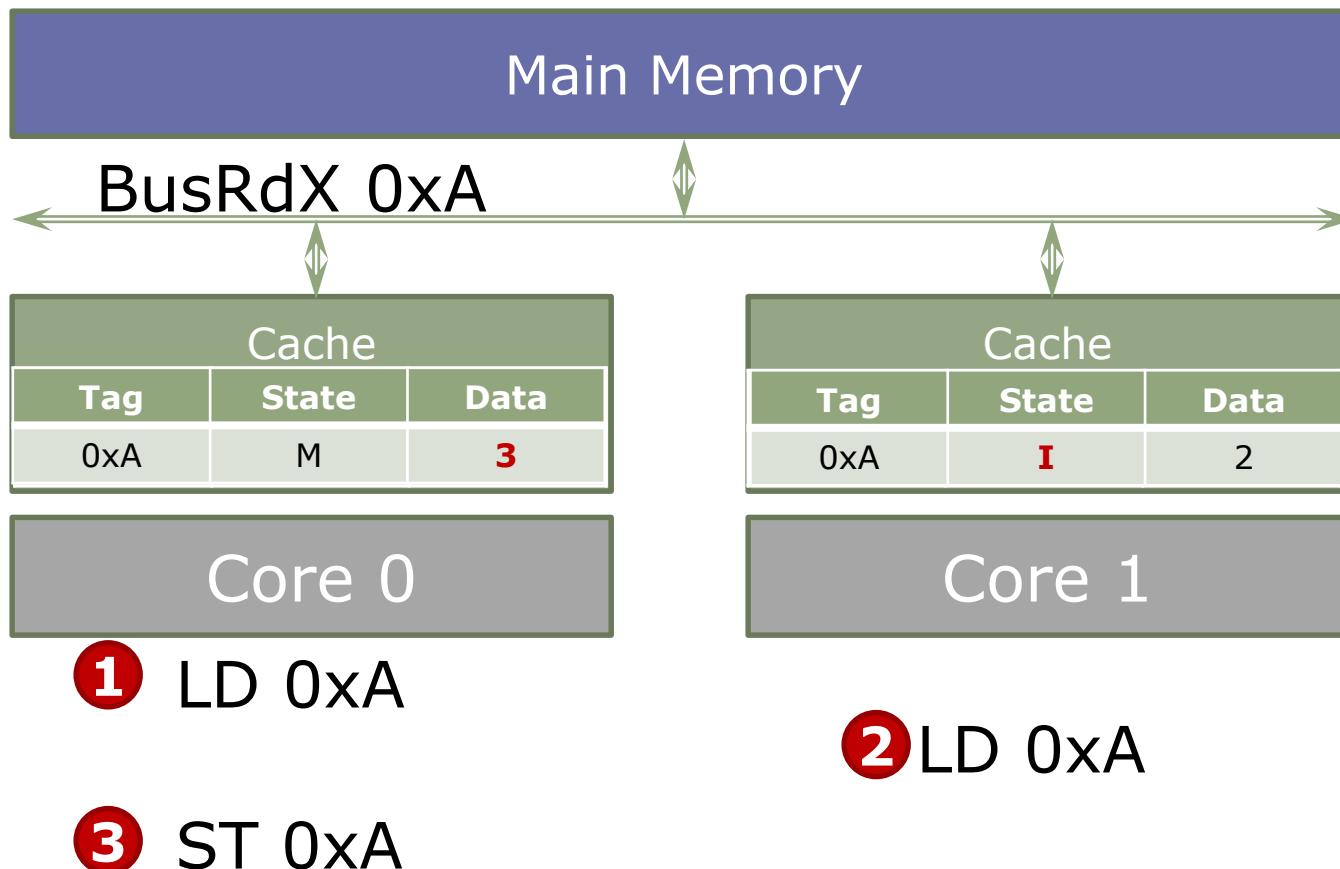


MSI Example



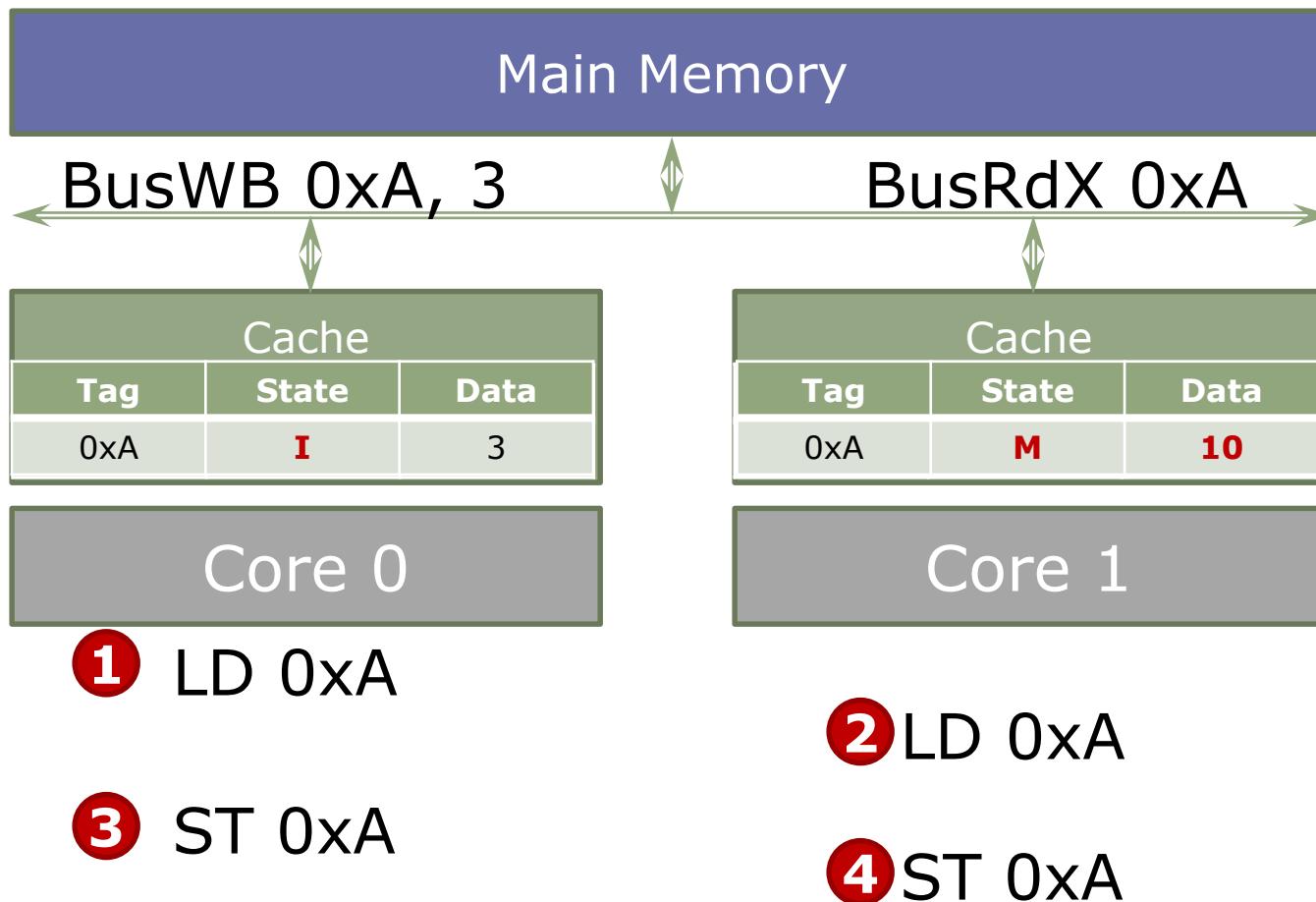
Additional loads satisfied locally, without BusRd
(like in VI)

MSI Example

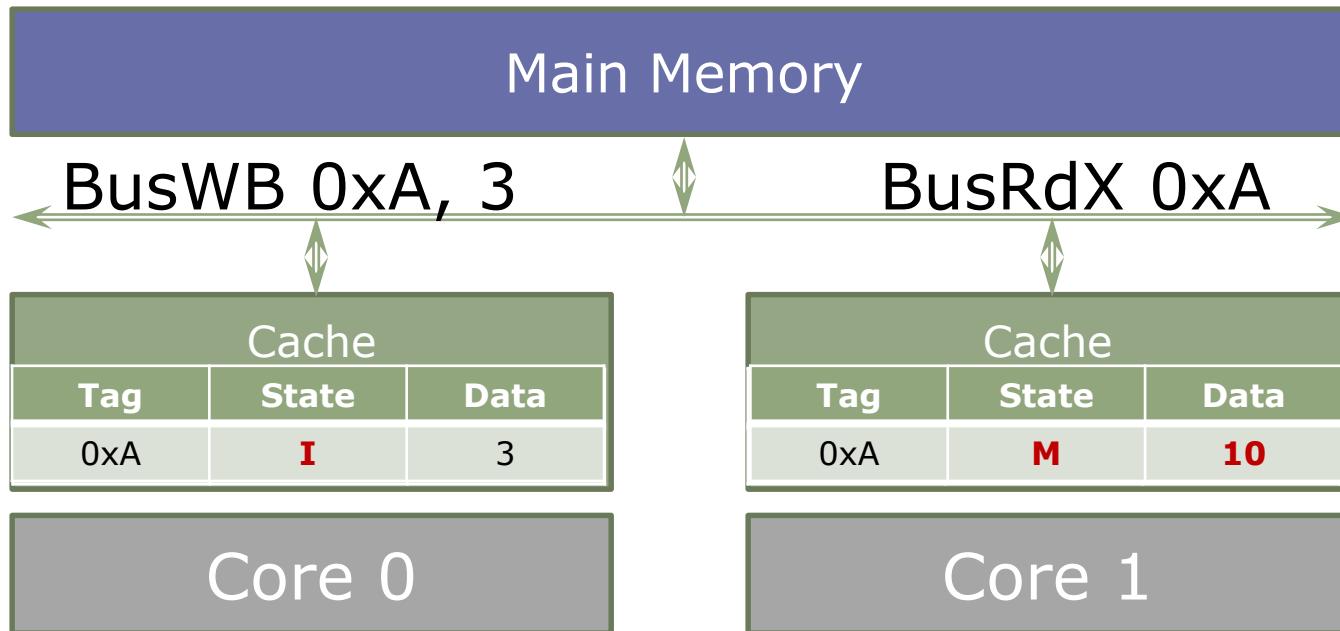


Additional loads *and stores* from core 0 satisfied locally,
without bus transactions (unlike in VI)

MSI Example

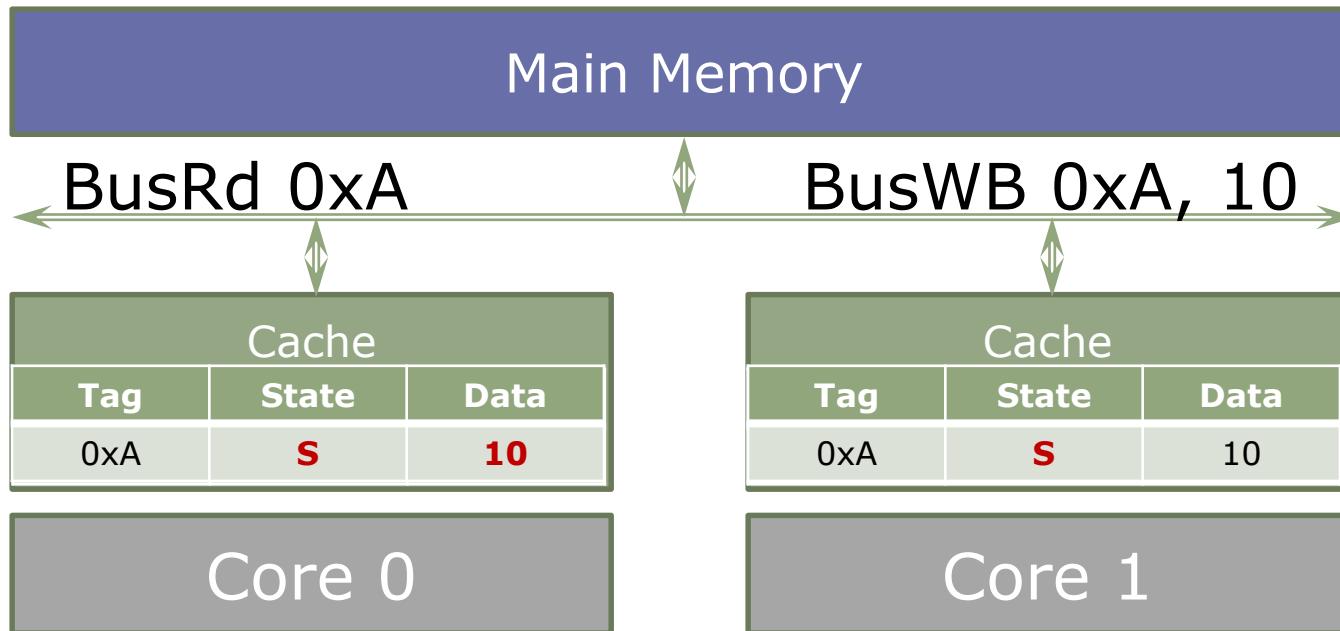


Cache interventions



- MSI allows caches to serve writes without updating memory, so main memory can have stale data
 - Core 0's cache needs to supply data
 - But main memory may also respond!
- Cache must override response from main memory

MSI Example



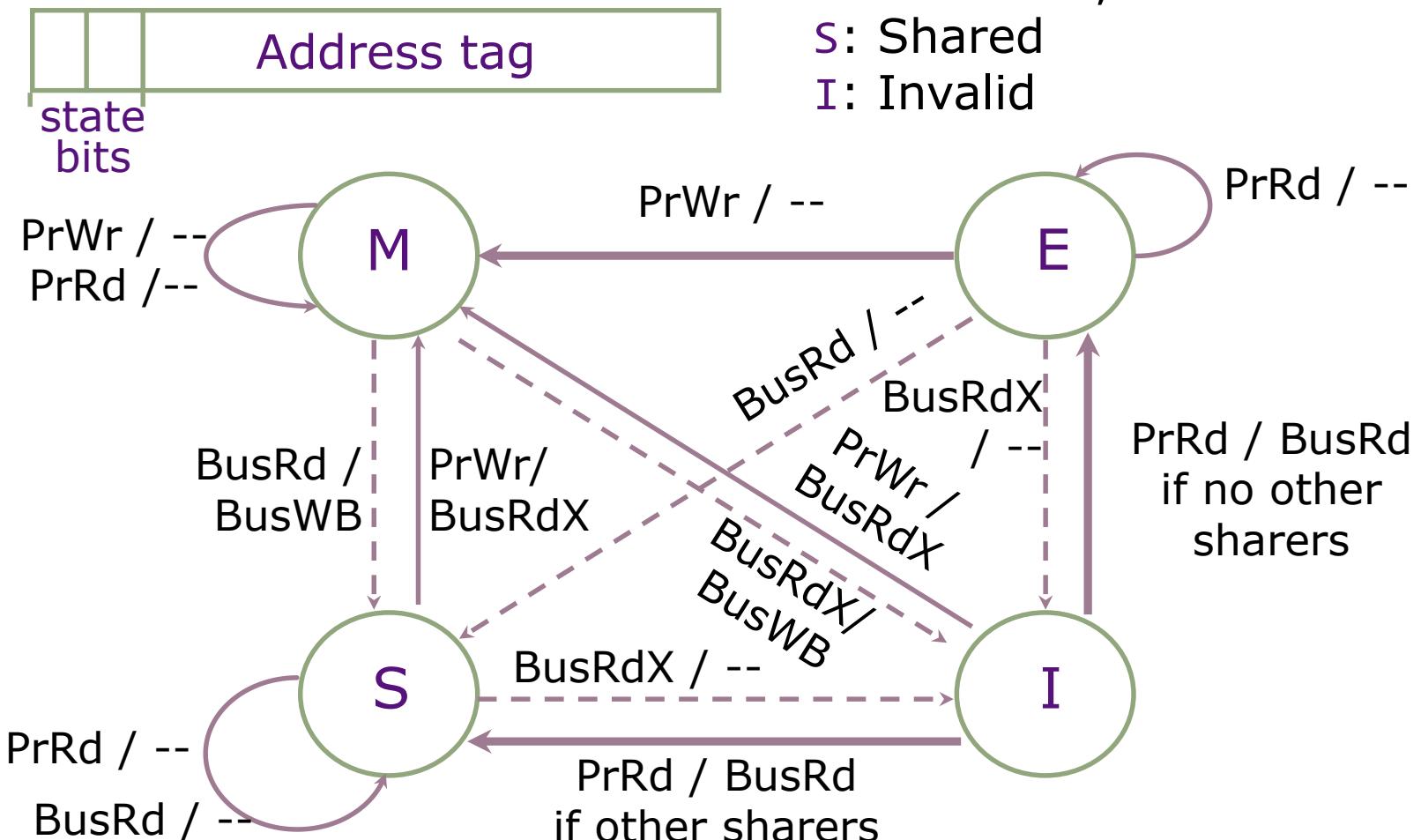
MSI Optimizations: Exclusive State

- Observation: Doing read-modify-write sequences on private data is common
 - What's the problem with MSI?
- Solution: E state (exclusive, clean)
 - If no other sharers, a read acquires line in E instead of S
 - Writes silently cause E→M (exclusive, dirty)

MESI: An Enhanced MSI protocol

increased performance for private read-write data

Each cache line has a tag

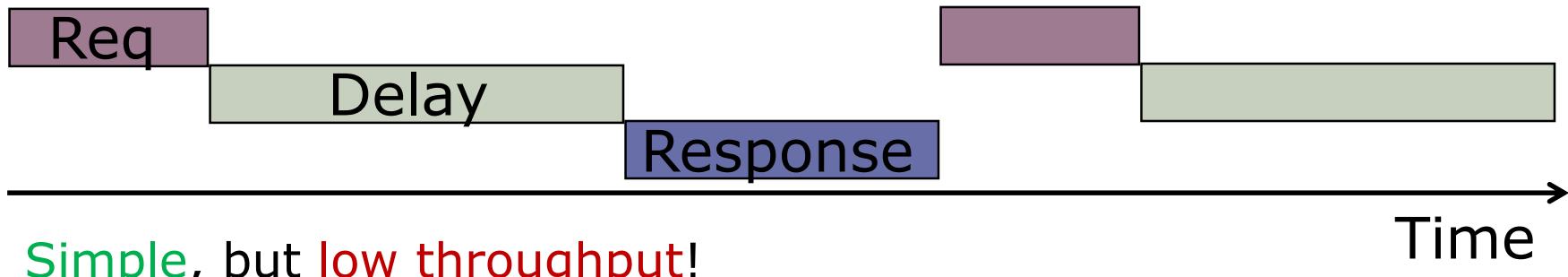


MSI Optimizations: Owner State

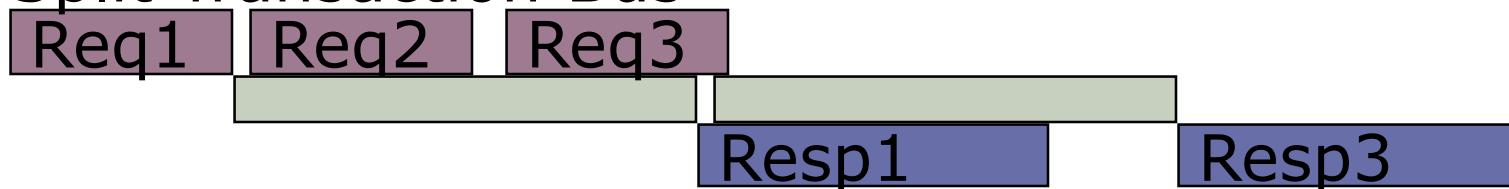
- Observation: On $M \rightarrow S$ transitions, must write back line!
 - What happens with frequent read-write sharing?
 - Can we defer the write after S ?
- Solution: O state (Owner)
 - $O = S + \text{responsibility to write back}$
 - On $M \rightarrow S$ transition, one sharer (typically the one who had the line in M) retains the line in O instead of S
 - On eviction, O writes back line (or another sharer does $S \rightarrow O$)
- MSI, MESI, MOSI, MOESI...
 - Typically E if private read-write $>>$ shared read-only (common)
 - Typically O only if writebacks are expensive (main mem vs L3)

Split-Transaction and Pipelined Buses

Atomic Transaction Bus



Split-Transaction Bus

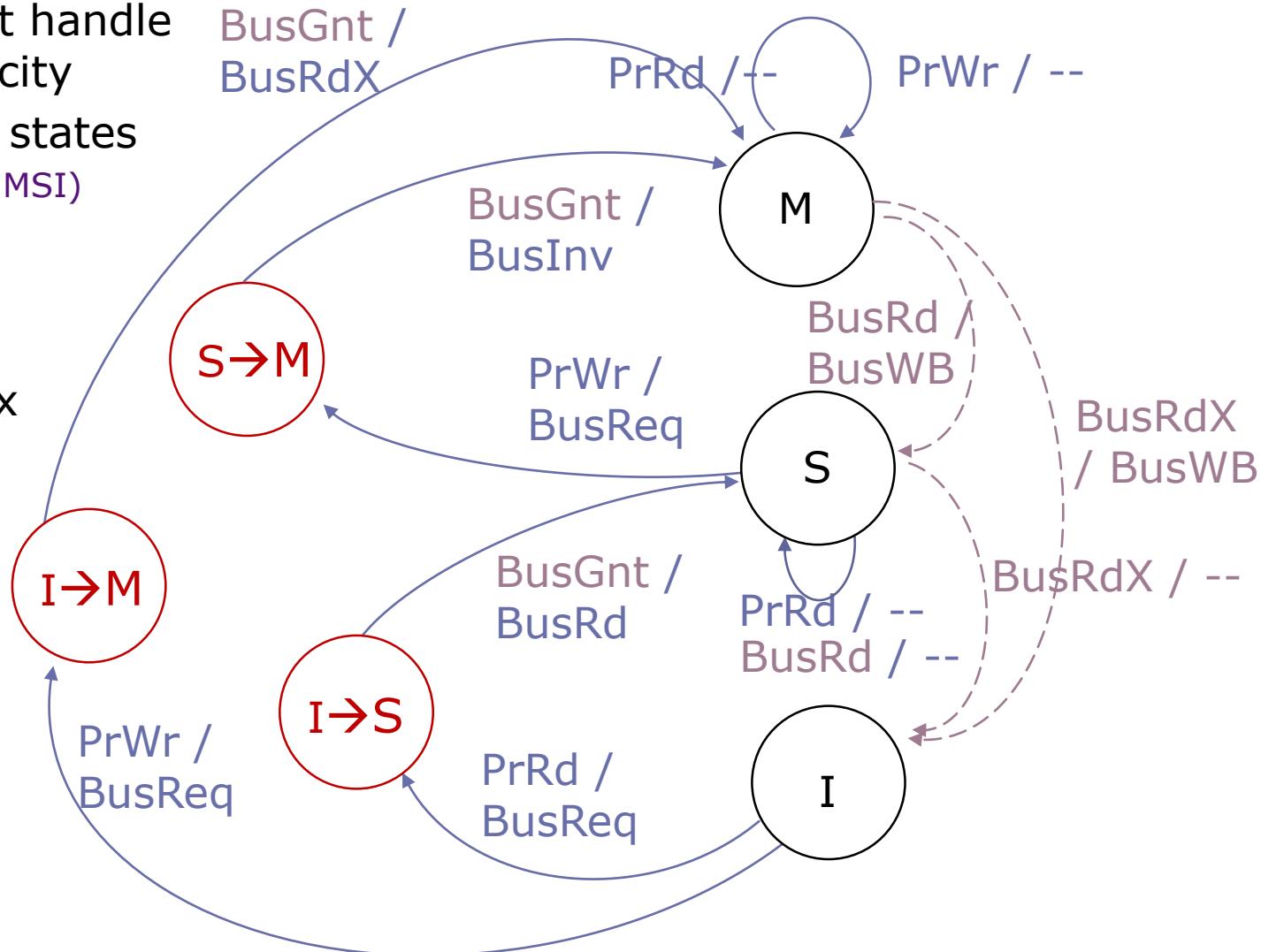


- Supports multiple simultaneous transactions
 - Higher throughput
 - Responses may arrive out of order
- Often implemented as multiple buses (req+resp)

Non-Atomicity → Transient States

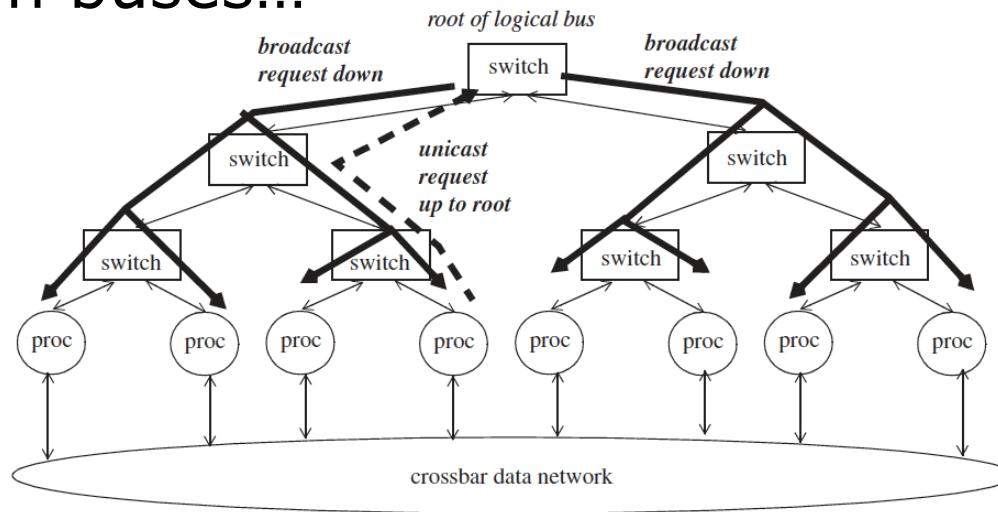
- Protocol must handle lack of atomicity
- Two types of states
 - Stable (e.g. MSI)
 - Transient
- Split + race transitions
- More complex

Actions	
Bus Request (BusReq)	PrWr / BusReq
Bus Grant (BusGnt)	PrRd / BusReq



Scaling Cache Coherence

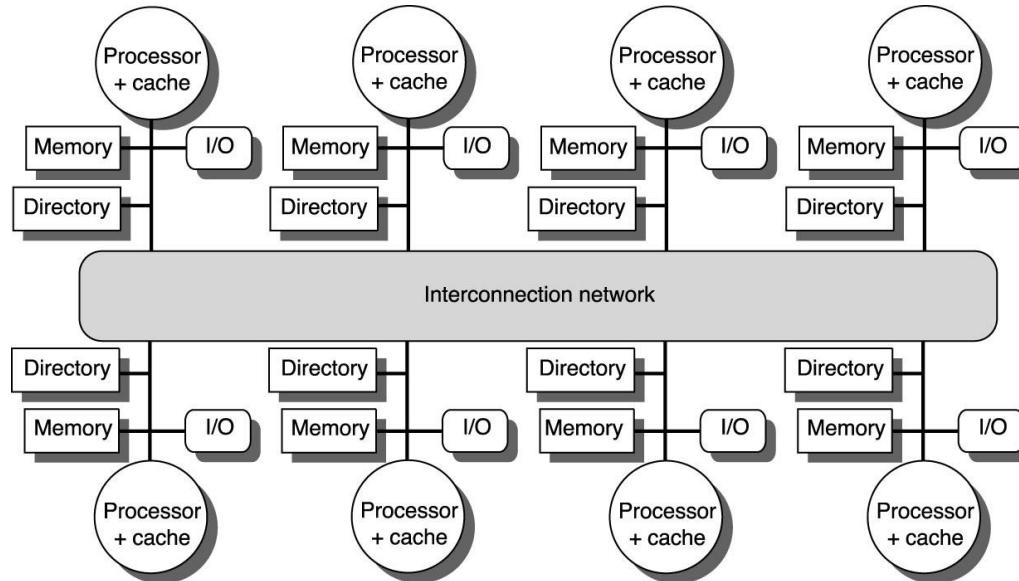
- Can implement ordered interconnects that scale better than buses...



Starfire E10000 (drawn with only eight processors for clarity). A coherence request is *unicast* up to the root, where it is serialized, before being *broadcast* down to all processors

- ... but broadcast is fundamentally unscalable
 - Bandwidth, energy of transactions with 100s of cache snoops?

Directory-Based Coherence



- Route all coherence transactions through a directory
 - Tracks contents of private caches → No broadcasts
 - Serves as ordering point for conflicting requests → Unordered networks

(more on next lecture)

Coherence and False Sharing

Performance Issue #1



A cache block contains more than one word and cache coherence is done at the block-level and not word-level

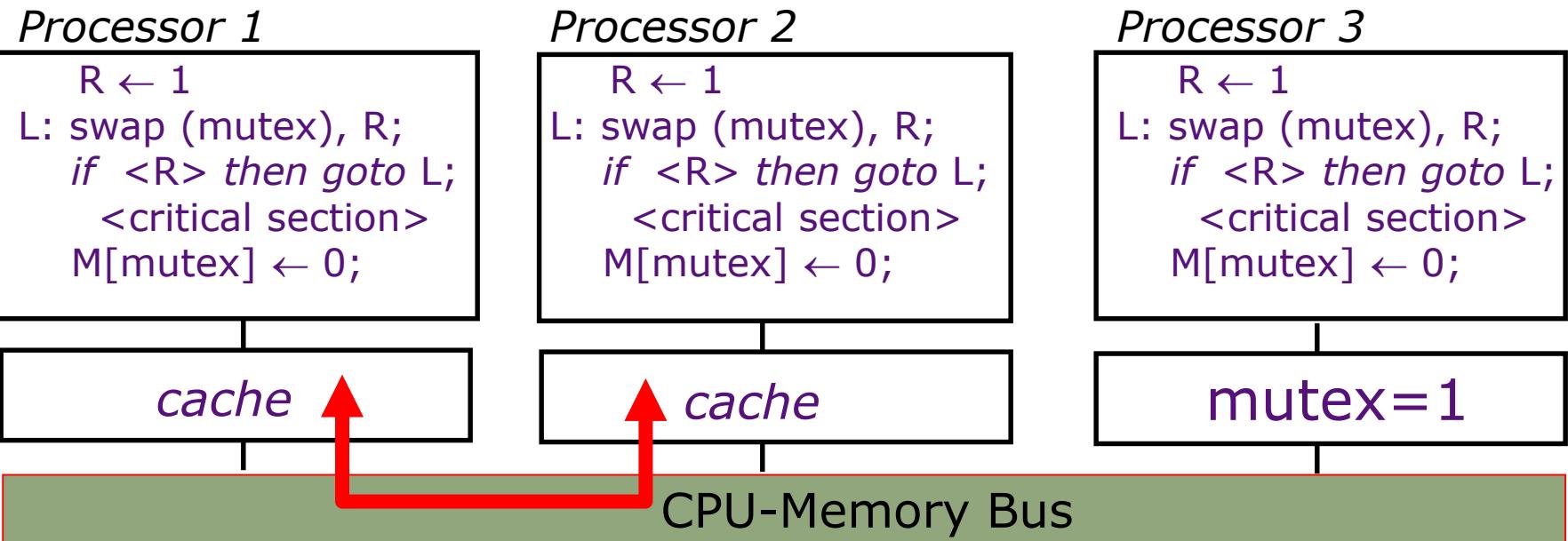
Suppose P_1 writes word_i and P_2 writes word_k and both words have the same block address.

What can happen?

How to address this problem?

Coherence and Synchronization

Performance Issue #2



Cache coherence protocols will cause **mutex** to *ping-pong* between P1's and P2's caches.

Ping-ponging can be reduced by first reading the **mutex** location (*non-atomically*) and executing a swap only if it is found to be zero (*test&test&set*).

Coherence and Bus Occupancy

Performance Issue #3

- In general, an *atomic read-modify-write* instruction requires two memory (bus) operations without intervening memory operations by other processors
- In a multiprocessor setting, bus needs to be locked for the entire duration of the atomic read and write operation
 - ⇒ expensive for simple buses
 - ⇒ *very expensive* for split-transaction buses
- modern processors use
 - load-reserve*
 - store-conditional*

Load-reserve & Store-conditional

Special register(s) to hold reservation flag and address, and the outcome of store-conditional

Load-reserve R, (a):

```
<flag, adr> ← <1, a>;  
R ← M[a];
```

Store-conditional (a), R:

```
if <flag, adr> == <1, a>  
then cancel other procs'  
reservation on a;  
M[a] ← <R>;  
status ← succeed;  
else status ← fail;
```

If the snooper sees a store transaction to the address in the reserve register, the reserve bit is set to **0**

- Several processors may reserve 'a' simultaneously
- These instructions are like ordinary loads and stores with respect to the bus traffic

Performance:

Load-reserve & Store-conditional

The total number of memory (bus) transactions is not necessarily reduced, but splitting an atomic instruction into load-reserve & store-conditional:

- *increases bus utilization* (and reduces processor stall time), especially in split-transaction buses
- *reduces cache ping-pong effect* because processors trying to acquire a mutex do not have to perform stores each time

Thank you!

*Next lecture: Directory-based
Cache Coherence*

Directory-Based Cache Coherence

Daniel Sanchez

Computer Science and Artificial Intelligence Lab
M.I.T.

Maintaining Cache Coherence

It is sufficient to have hardware such that

- only one processor at a time has write permission for a location
- no processor can load a stale copy of the location after a write

⇒ A correct approach could be:

write request:

The address is *invalidated* in all other caches *before* the write is performed

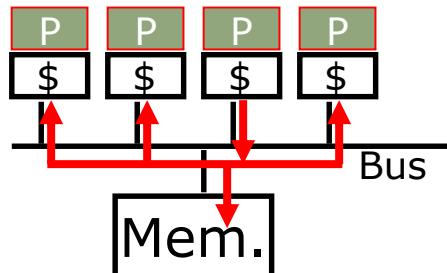
read request:

If a dirty copy is found in some cache, a write-back is performed before the memory is read

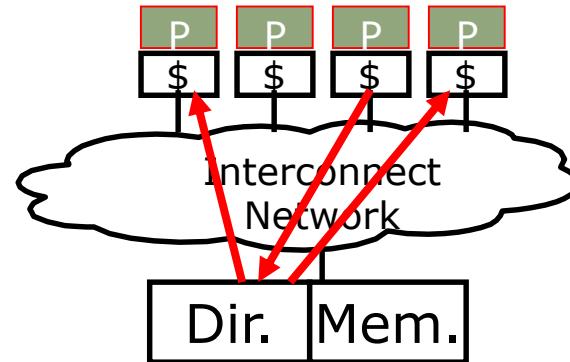
Directory-Based Coherence

(Censier and Feautrier, 1978)

Snoopy Protocols



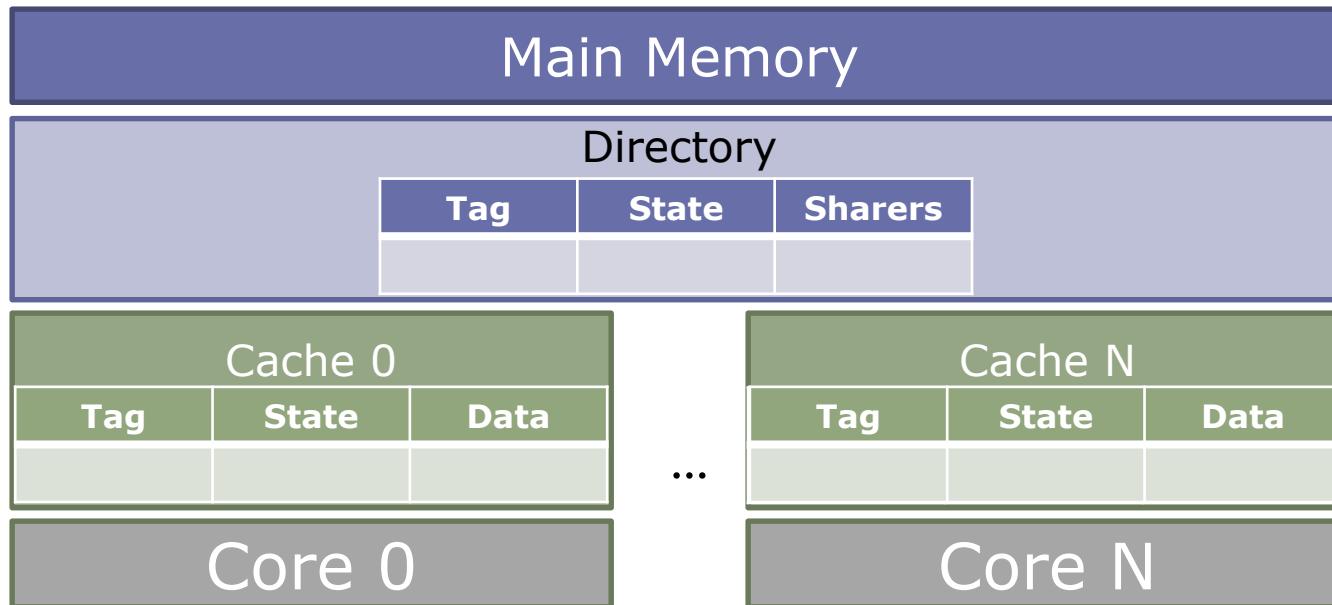
Directory Protocols



- Snoopy schemes broadcast requests over memory bus
- Difficult to scale to large numbers of processors
- Requires additional bandwidth to cache tags for snoop requests

- Directory schemes send messages to only those caches that might have the line
- Can scale to large numbers of processors
- Requires extra directory storage to track possible sharers

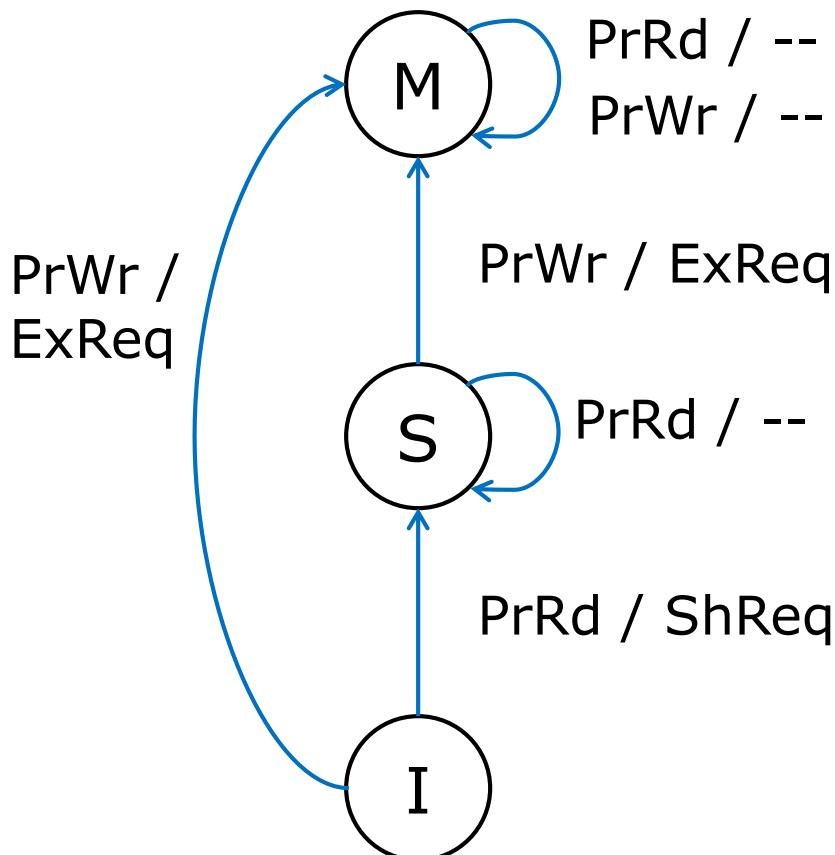
An MSI Directory Protocol



- Cache states: Modified (M) / Shared (S) / Invalid (I)
- Directory states:
 - Uncached (Un): No sharers
 - Shared (Sh): One or more sharers with read permission (S)
 - Exclusive (Ex): A single sharer with read & write permissions (M)
- Transient states not drawn for clarity; for now, assume no racing requests

MSI Protocol: Caches (1/3)

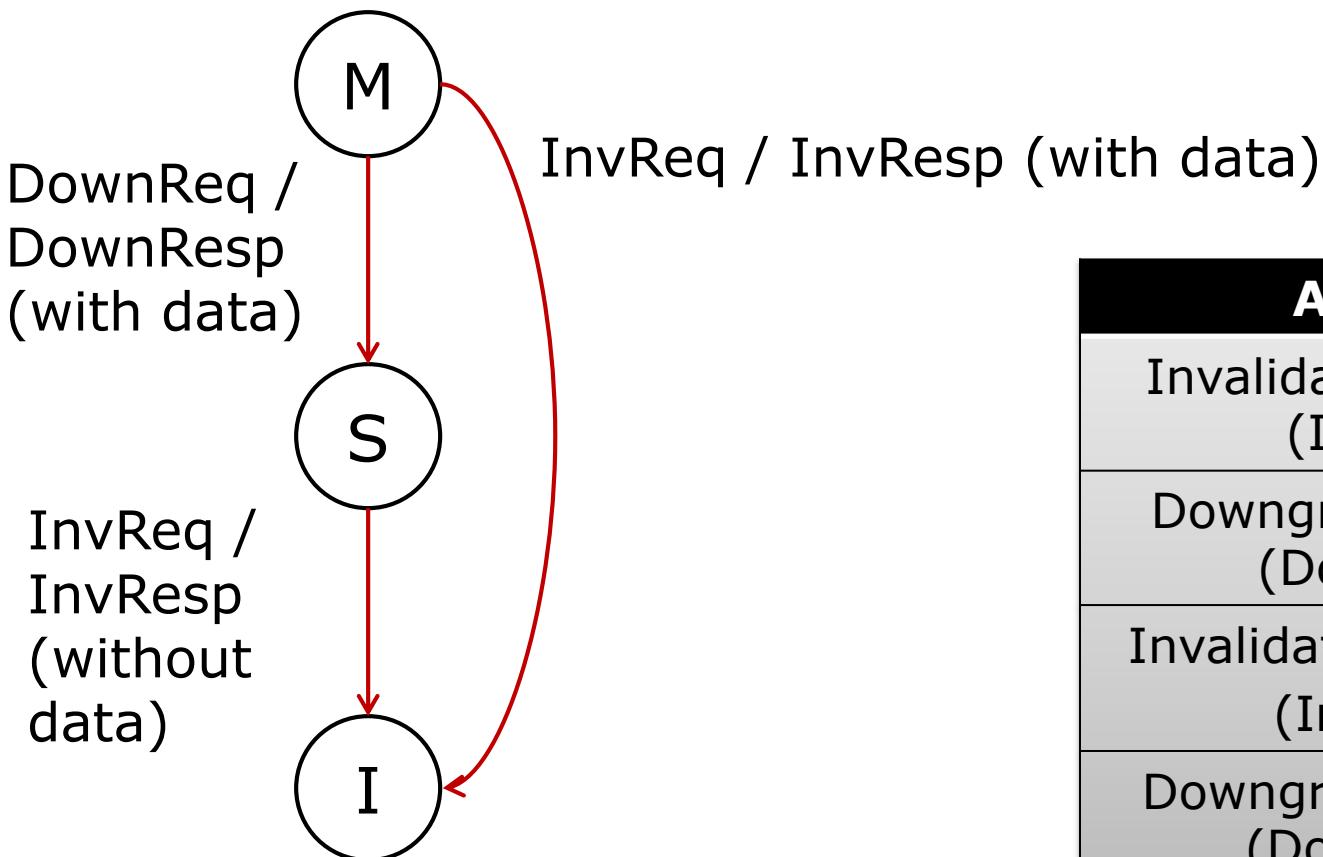
Transitions initiated by processor accesses:



Actions
Processor Read (PrRd)
Processor Write (PrWr)
Shared Request (ShReq)
Exclusive Request (ExReq)

MSI Protocol: Caches (2/3)

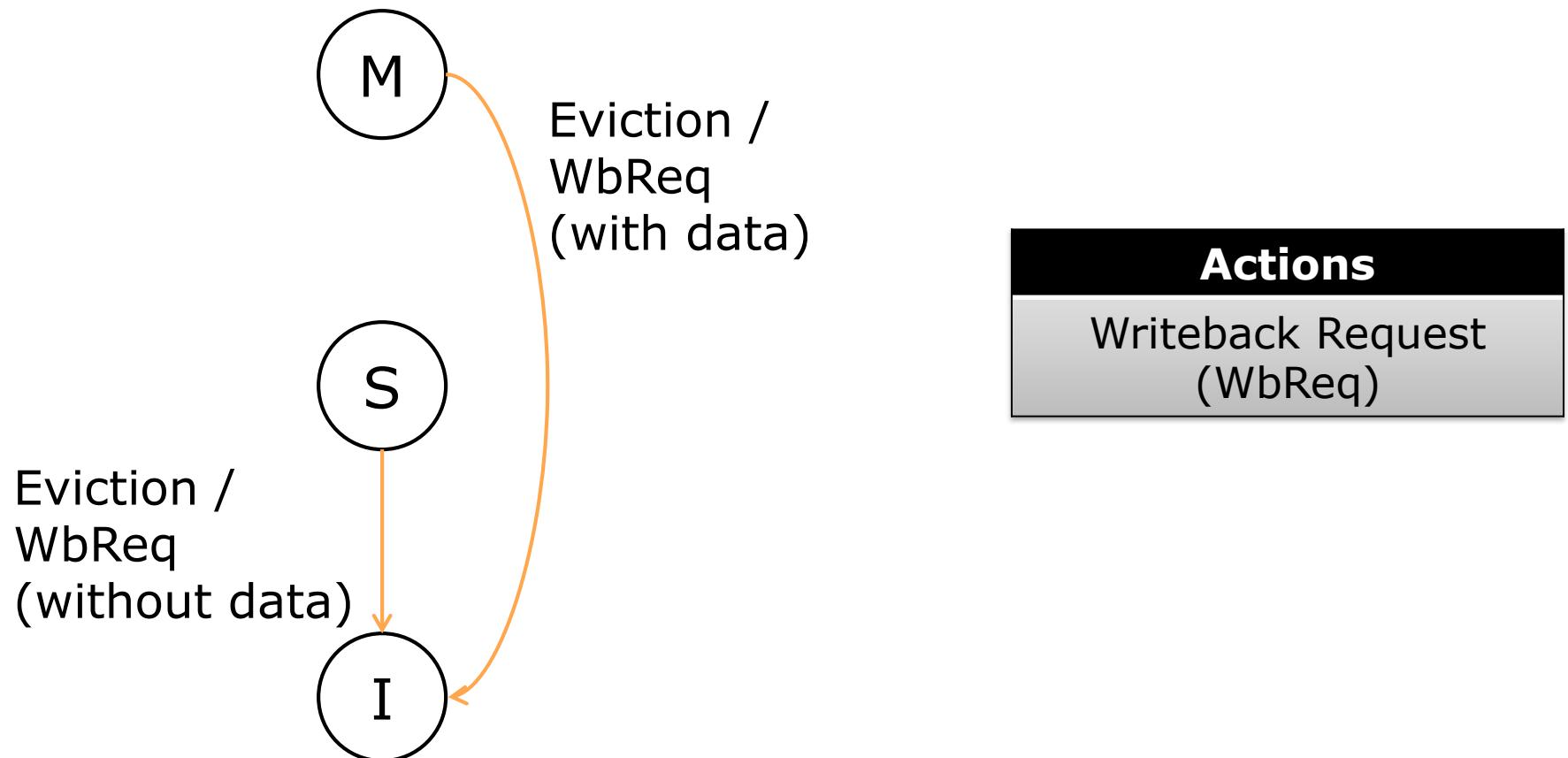
Transitions initiated by directory requests:



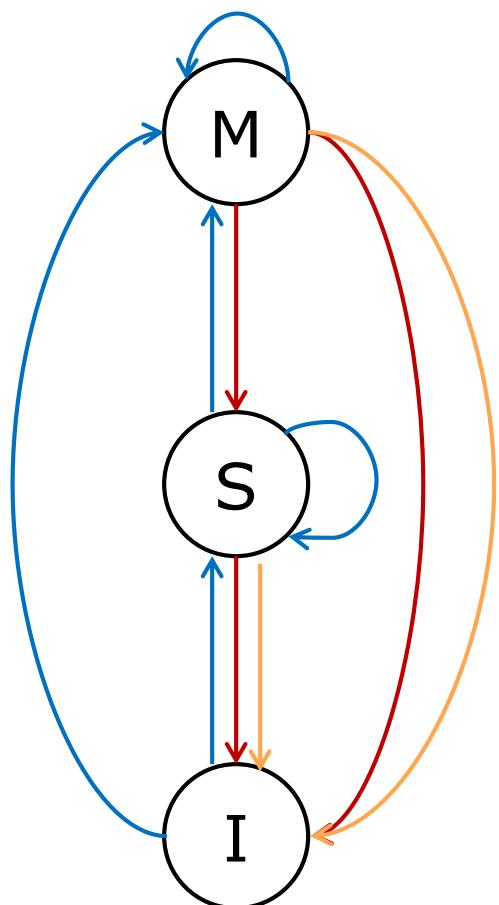
Actions
Invalidation Request (InvReq)
Downgrade Request (DownReq)
Invalidation Response (InvResp)
Downgrade Response (DownResp)

MSI Protocol: Caches (3/3)

Transitions initiated by evictions:



MSI Protocol: Caches

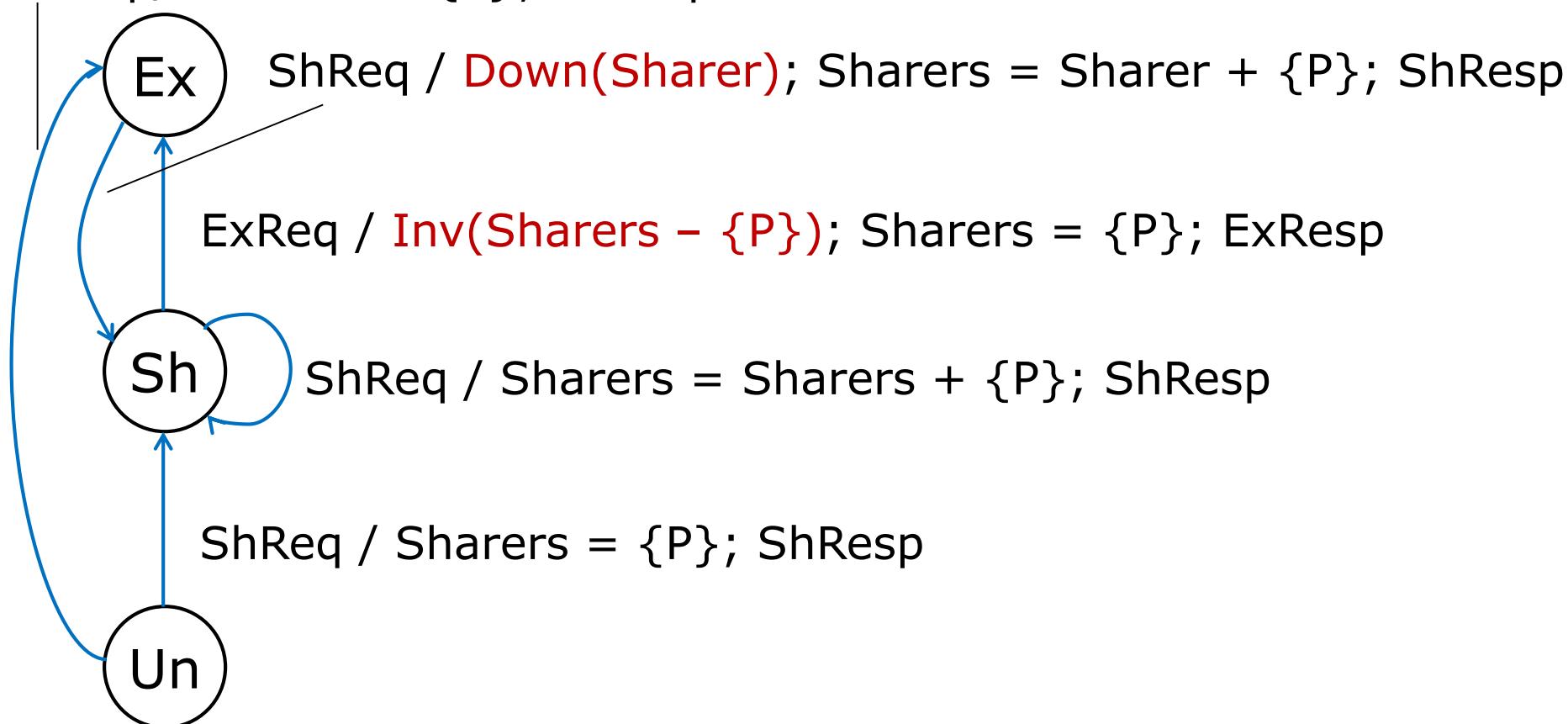


- Transitions initiated by processor accesses
- Transitions initiated by directory requests
- Transitions initiated by evictions

MSI Protocol: Directory (1/2)

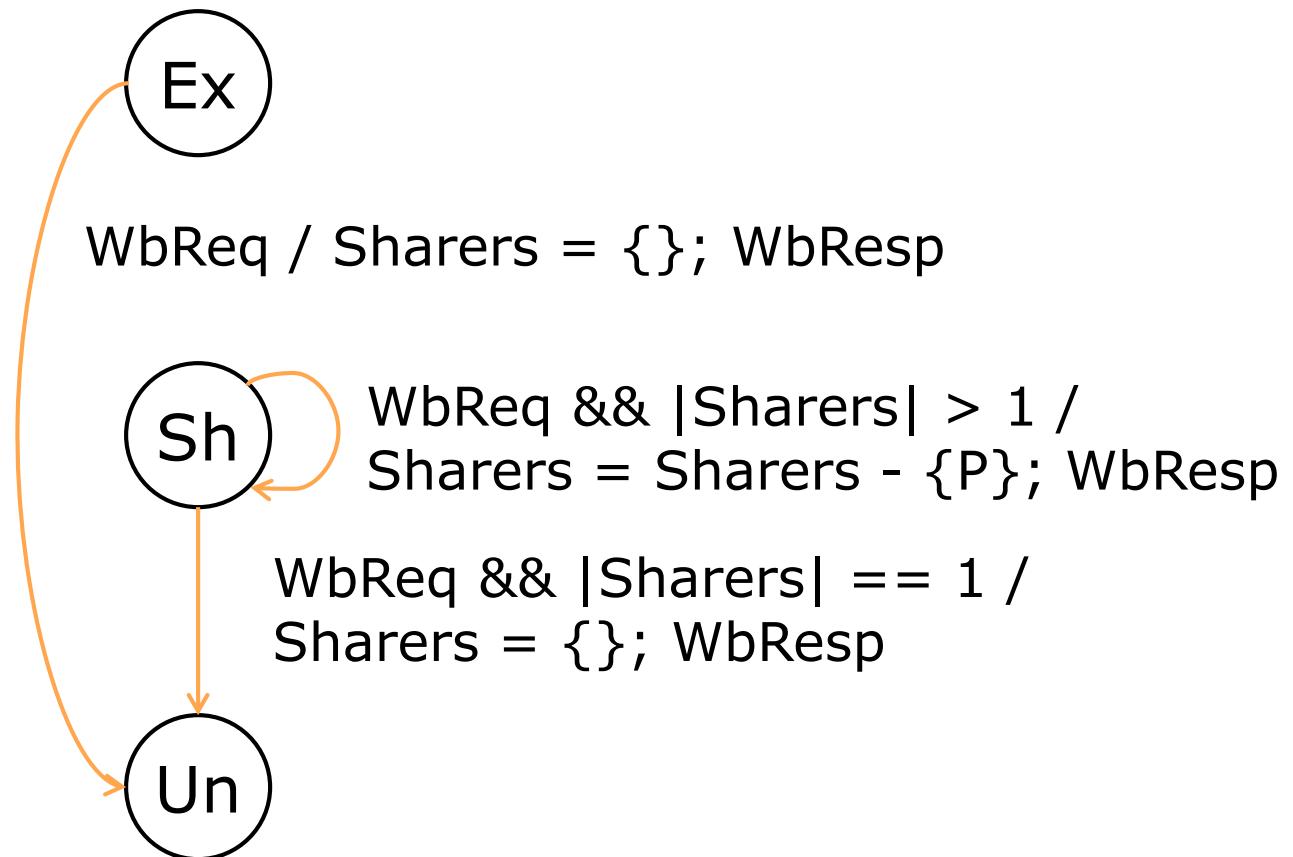
Transitions initiated by data requests:

ExReq / Sharers = {P}; ExResp

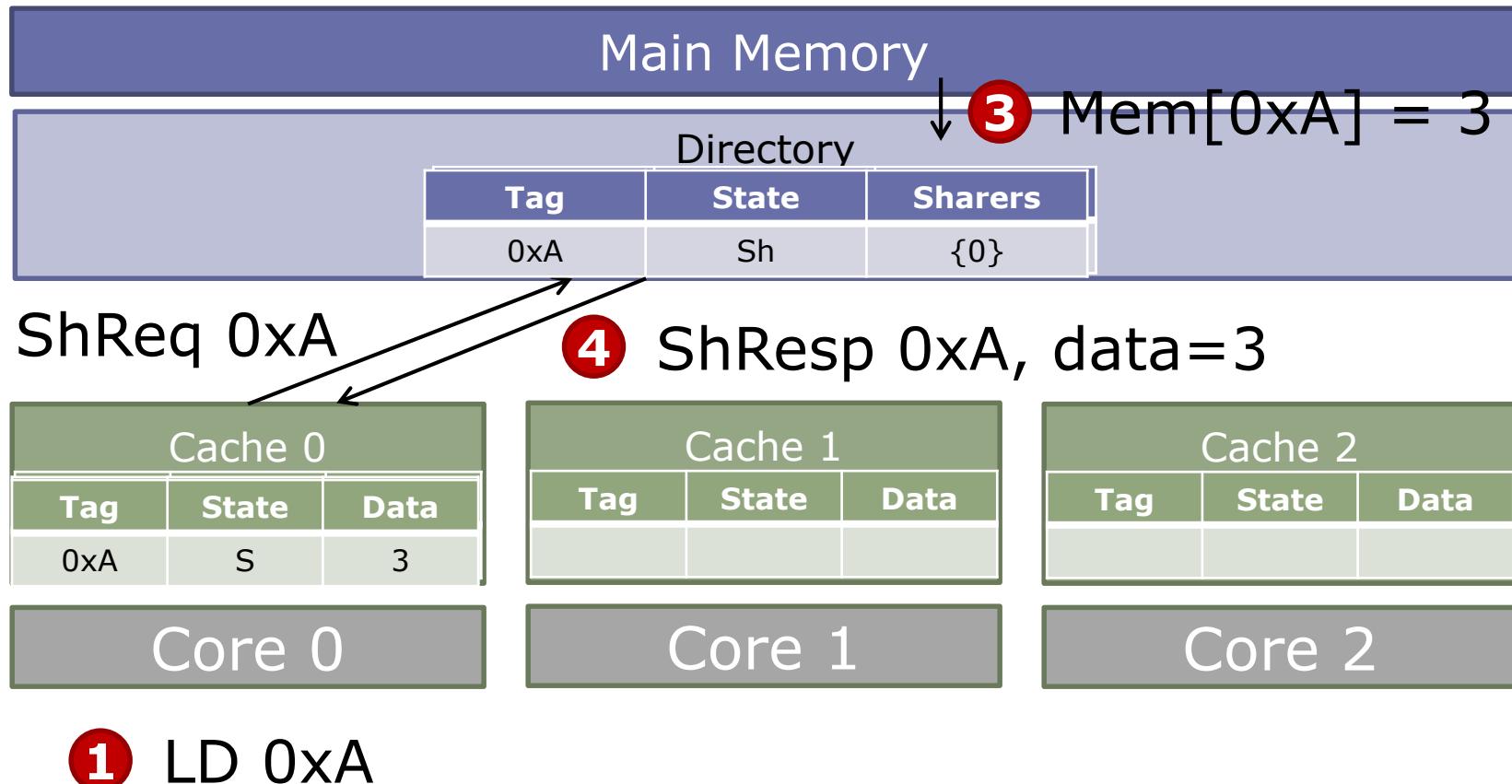


MSI Protocol: Directory (2/2)

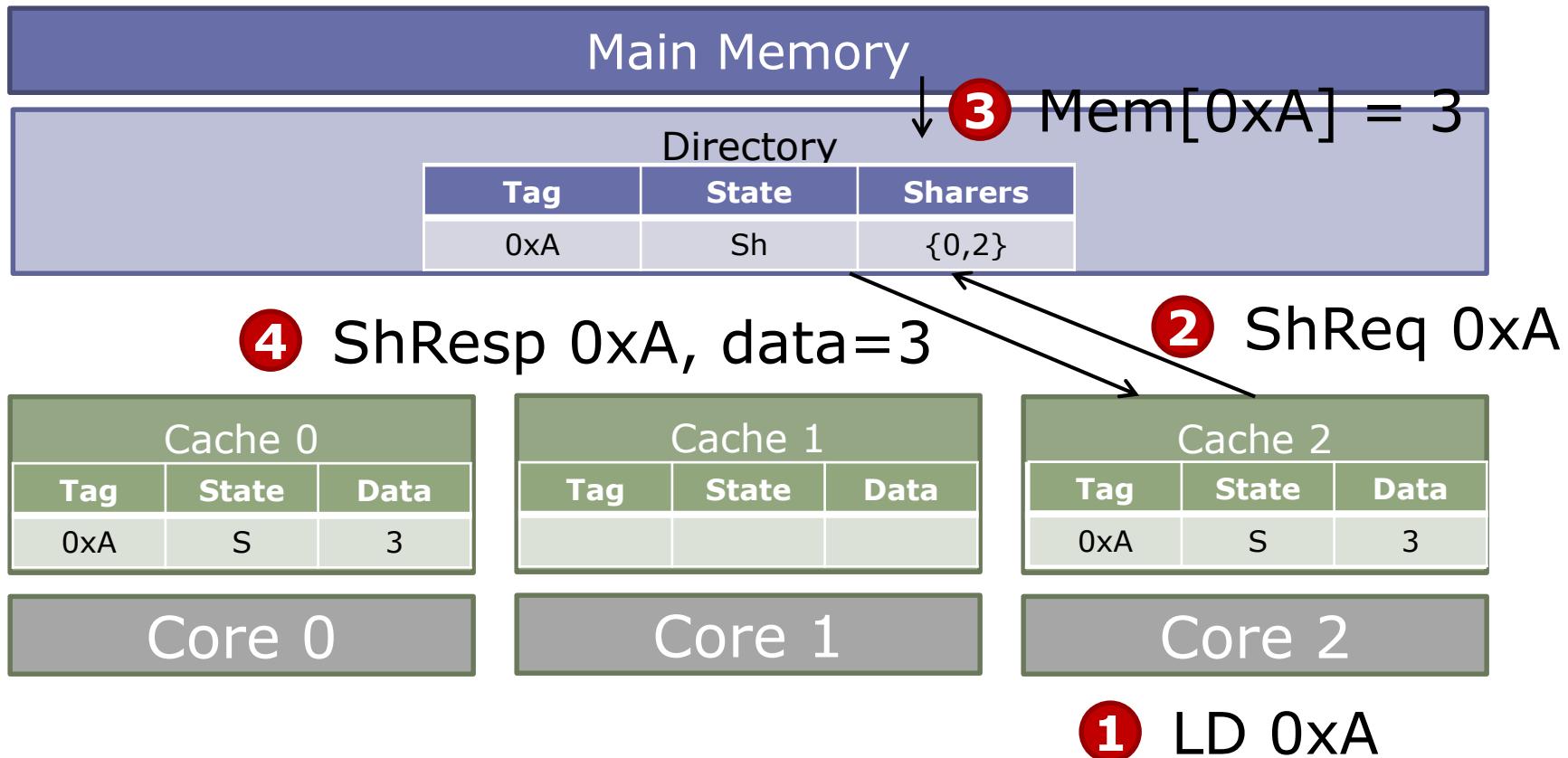
Transitions initiated by writeback requests:



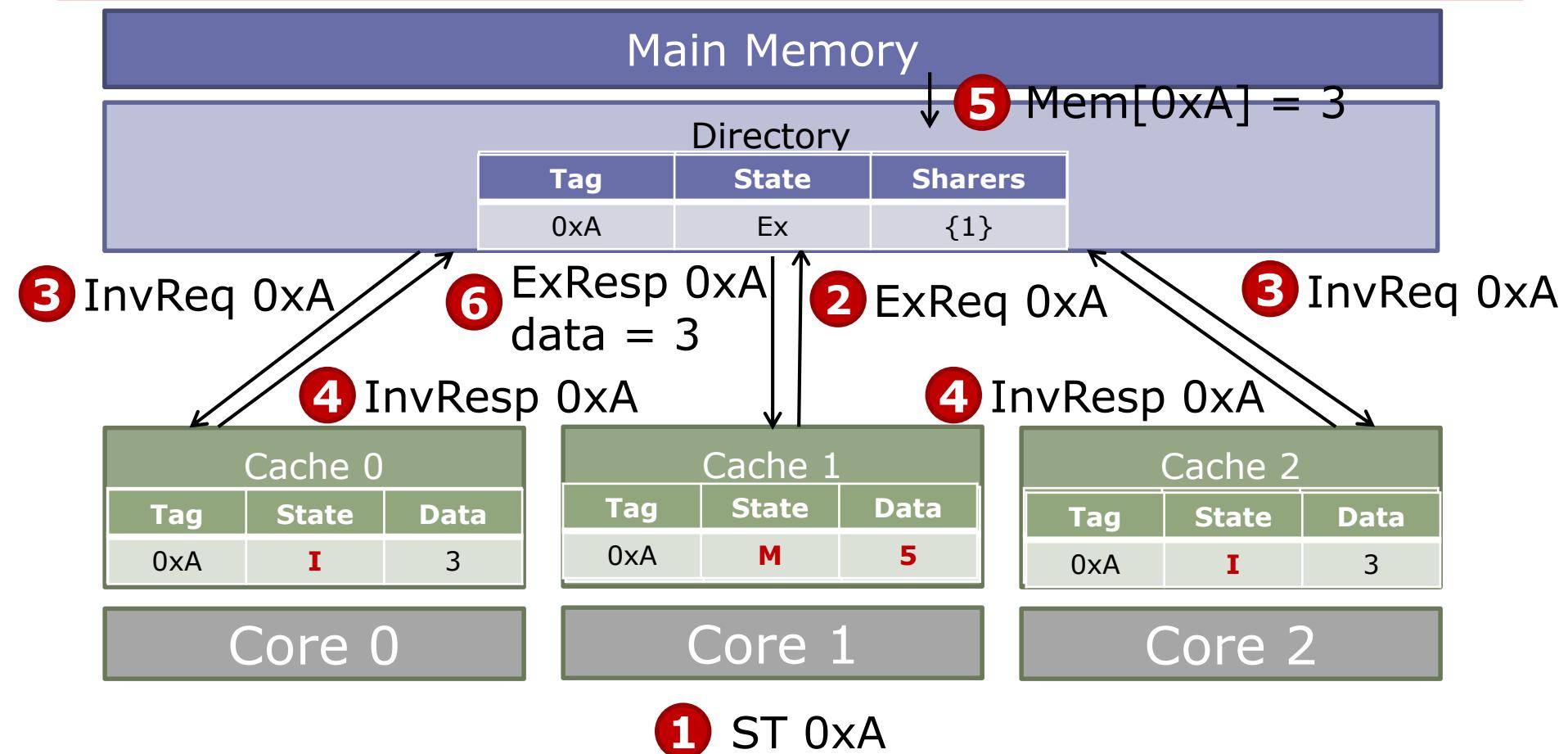
MSI Directory Protocol Example



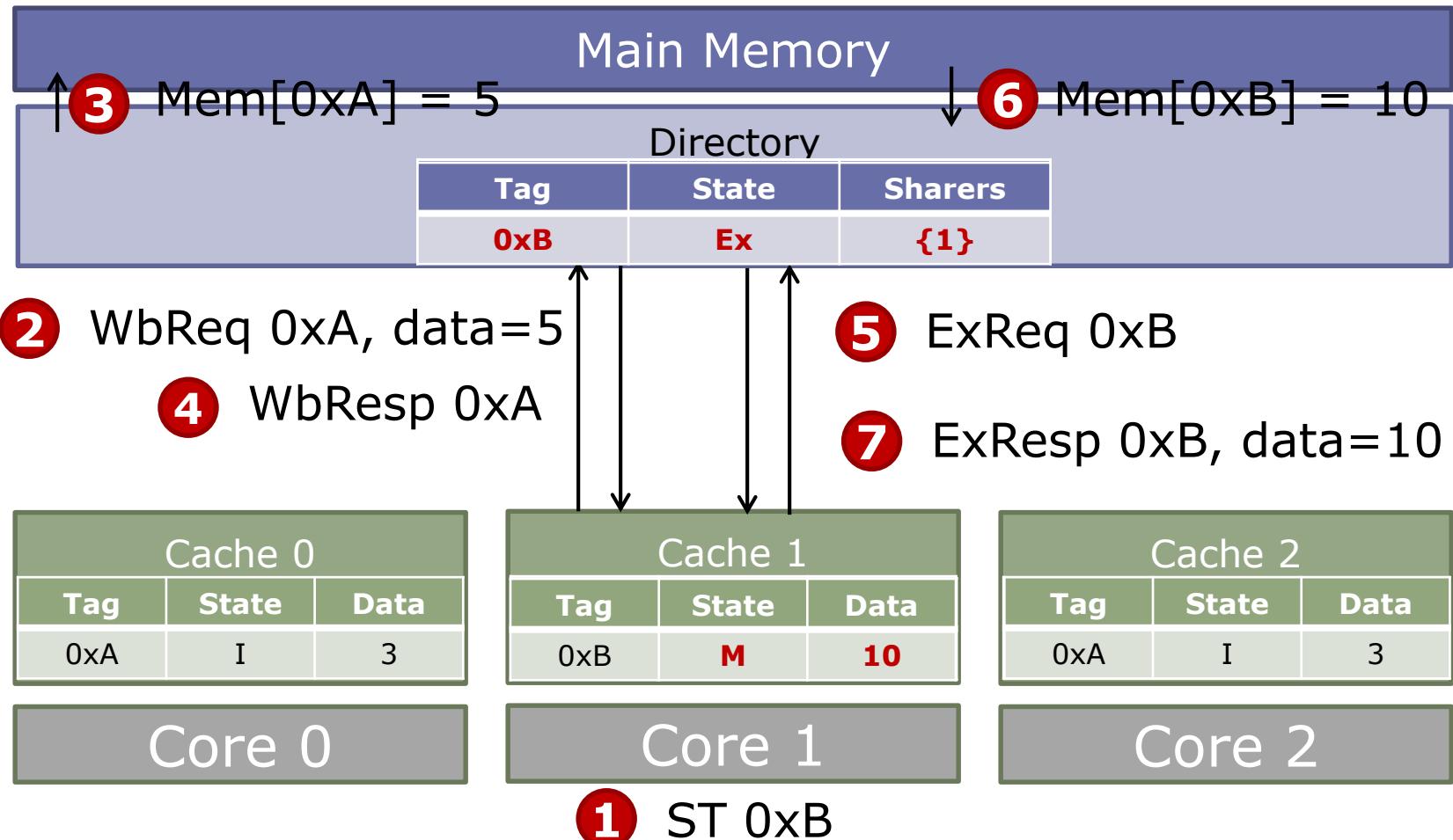
MSI Directory Protocol Example



MSI Directory Protocol Example



MSI Directory Protocol Example



Why are 0xA's wb and 0xB's req serialized?
Possible solutions?

Miss Status Handling Register

MSHR – Holds load misses and writes outside of cache

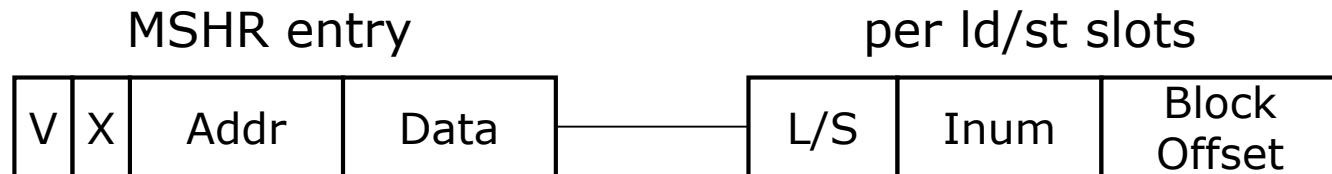
MSHR entry

V	X	Addr	Data
---	---	------	------

- On eviction/writeback
 - No free MSHR entry: stall
 - Allocate new MSHR entry
 - When channel available send WBReq and data
 - Deallocate entry on WBResp

Miss Status Handling Register

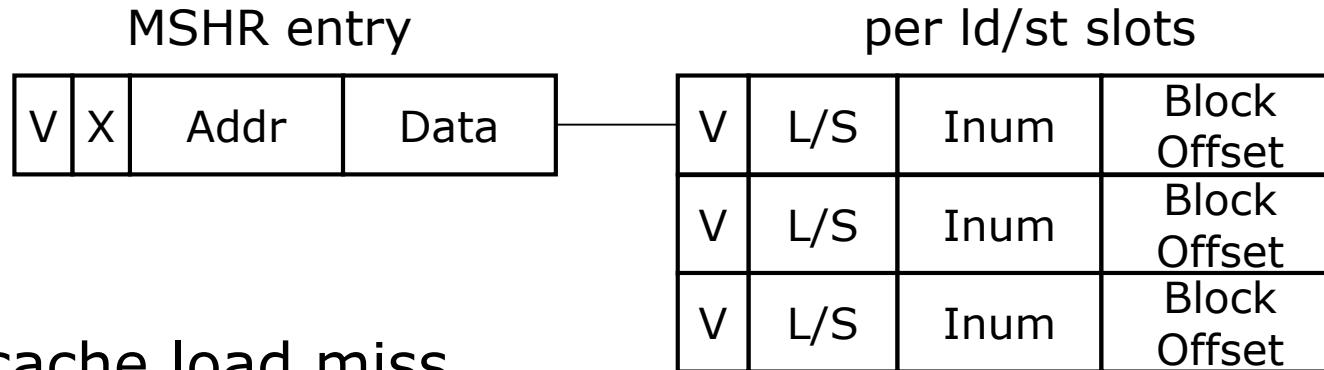
MSHR – Holds load misses and writes outside of cache



- On cache load miss
 - No free MSHR entry: stall
 - Allocate new MSHR entry
 - Send ShReq (or ExReq)
 - On *Resp forward data to CPU and cache
 - Deallocate MSHR

Miss Status Handling Register

MSHR – Holds load misses and writes outside of cache



- On cache load miss
 - Look for matching address in MSHR
 - If not found
 - If no free MSHR entry: stall
 - Allocate new MSHR entry and fill in
 - If found, just fill in per Id/st slot
 - Send ShReq (or ExReq)
 - On *Resp forward data to CPU and cache
 - Deallocate MSHR

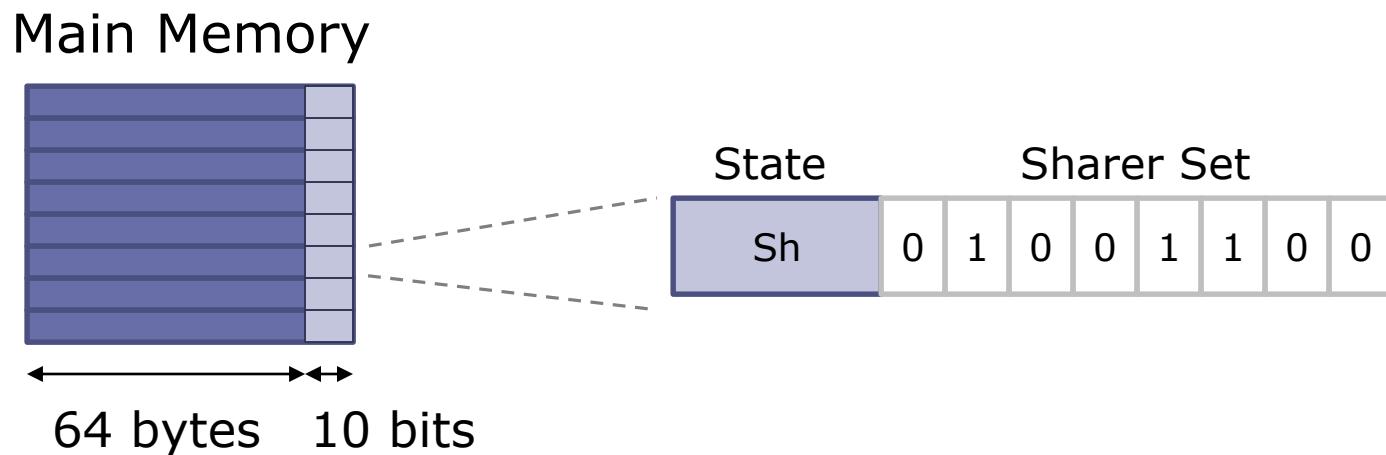
Per Id/st slots allow servicing multiple requests with one entry

Directory Organization

- Requirement: Directory needs to keep track of all the cores that are sharing a cache block
- Challenge: For each block, the space needed to hold the list of sharers grows with number of possible sharers...

Flat, Memory-based Directories

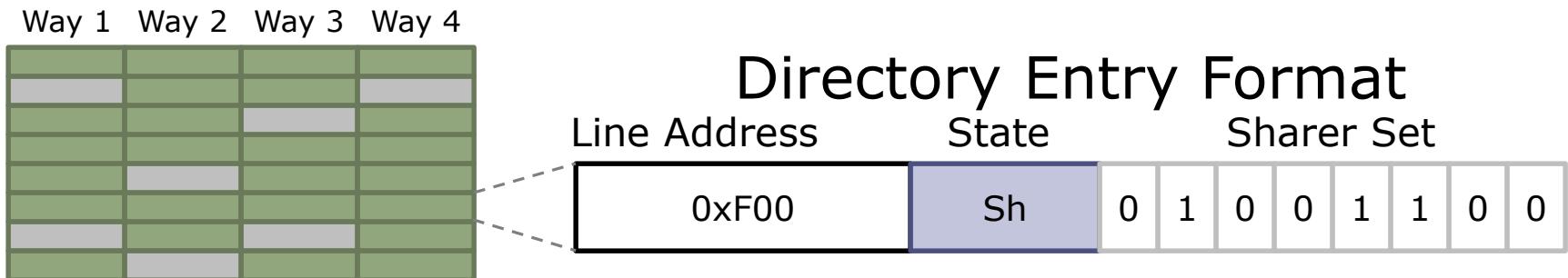
- Dedicate a few bits of main memory to store the state and sharers of every line
- Encode sharers using a bit-vector



- ✓ Simple
- ✗ Slow
- ✗ Very inefficient with many processors ($\sim P$ bits/line)

Sparse Full-Map Directories

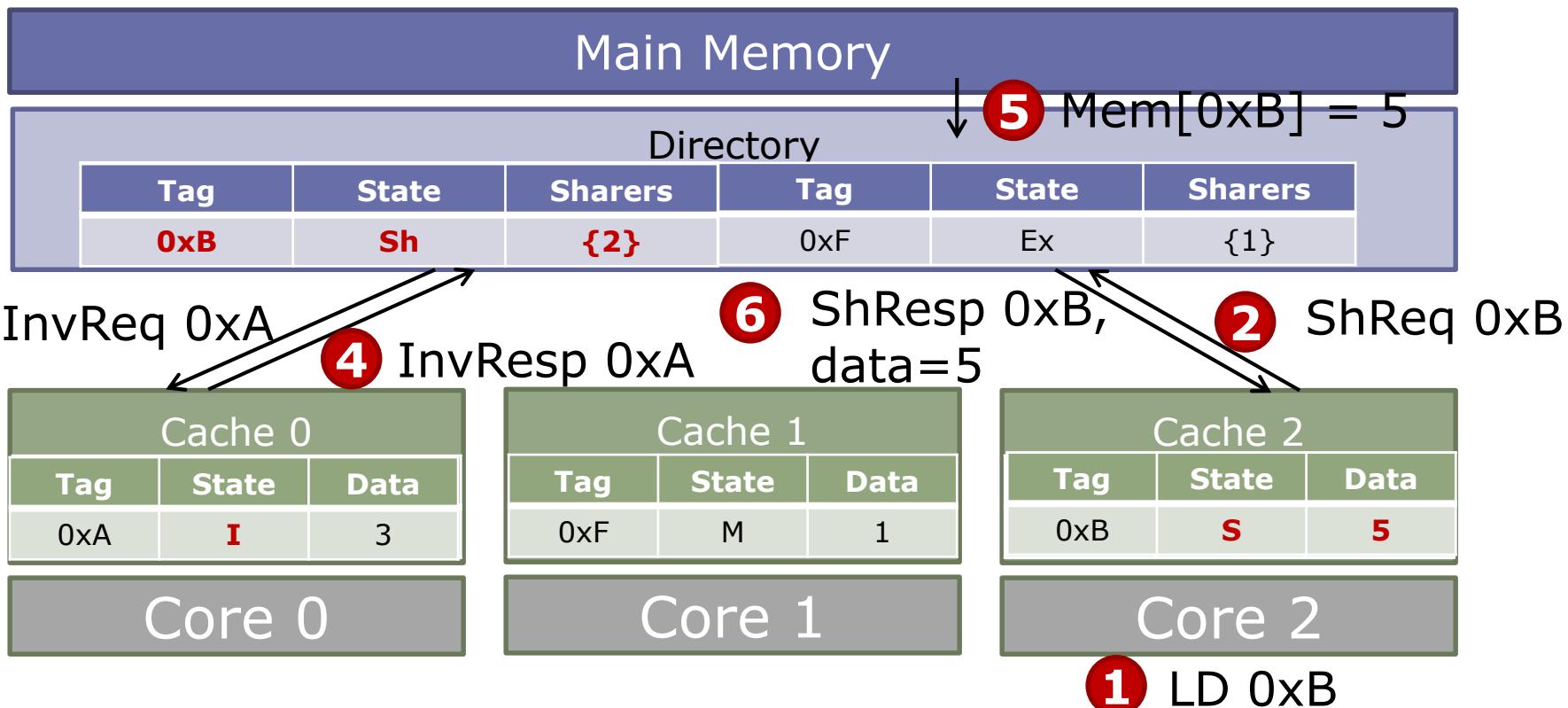
- Not every line in the system needs to be tracked – only those in private caches!
- Idea: Organize directory as a cache



- ✓ Low latency, energy-efficient
- ✗ Bit-vectors grow with # cores → Area scales poorly
- ✗ Limited associativity → Directory-induced invalidations

Directory-Induced Invalidations

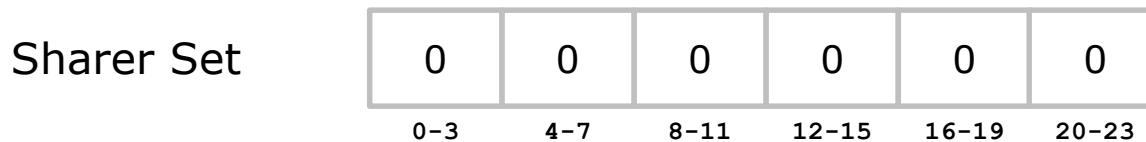
- To retain inclusion, must invalidate all sharers of an entry before reusing it for another address
- Example: 2-way set-associative sparse directory



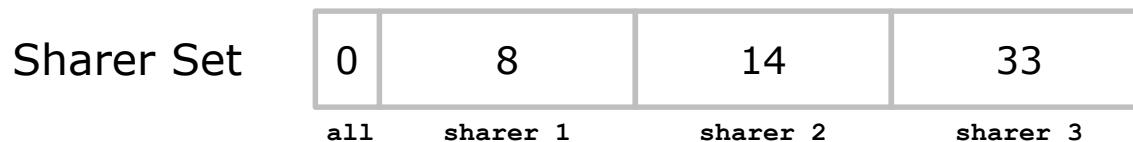
How many entries should the directory have?

Inexact Representations of Sharer Sets

- Coarse-grain bit-vectors (e.g., 1 bit per 4 cores)



- Limited pointers: Maintain a few sharer pointers, on overflow mark 'all' and broadcast (or invalidate another sharer)

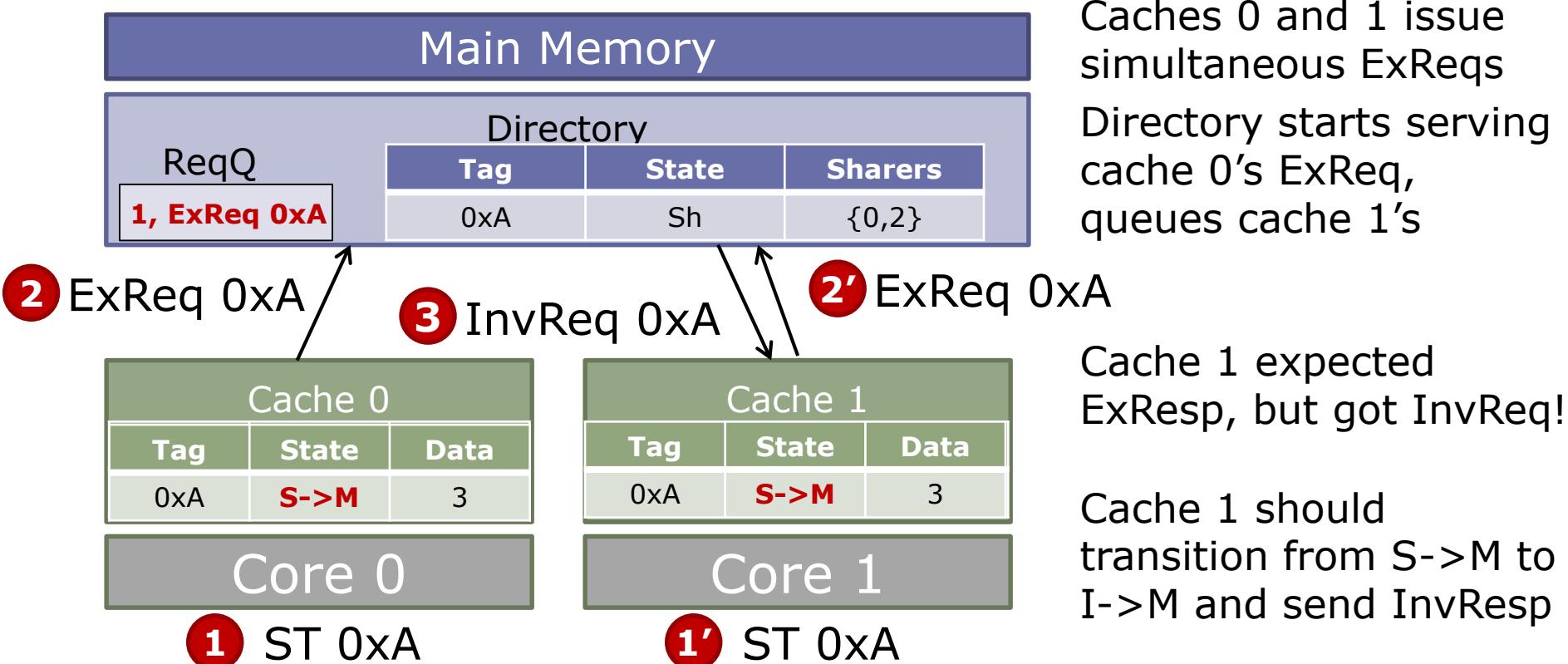


- Allow false positives (e.g., Bloom filters)

- ✓ Reduced area & energy
- ✗ Overheads still not scalable (these techniques simply play with constant factors)
- ✗ Inexact sharers → Broadcasts, invalidations or spurious invalidations and downgrades

Protocol Races

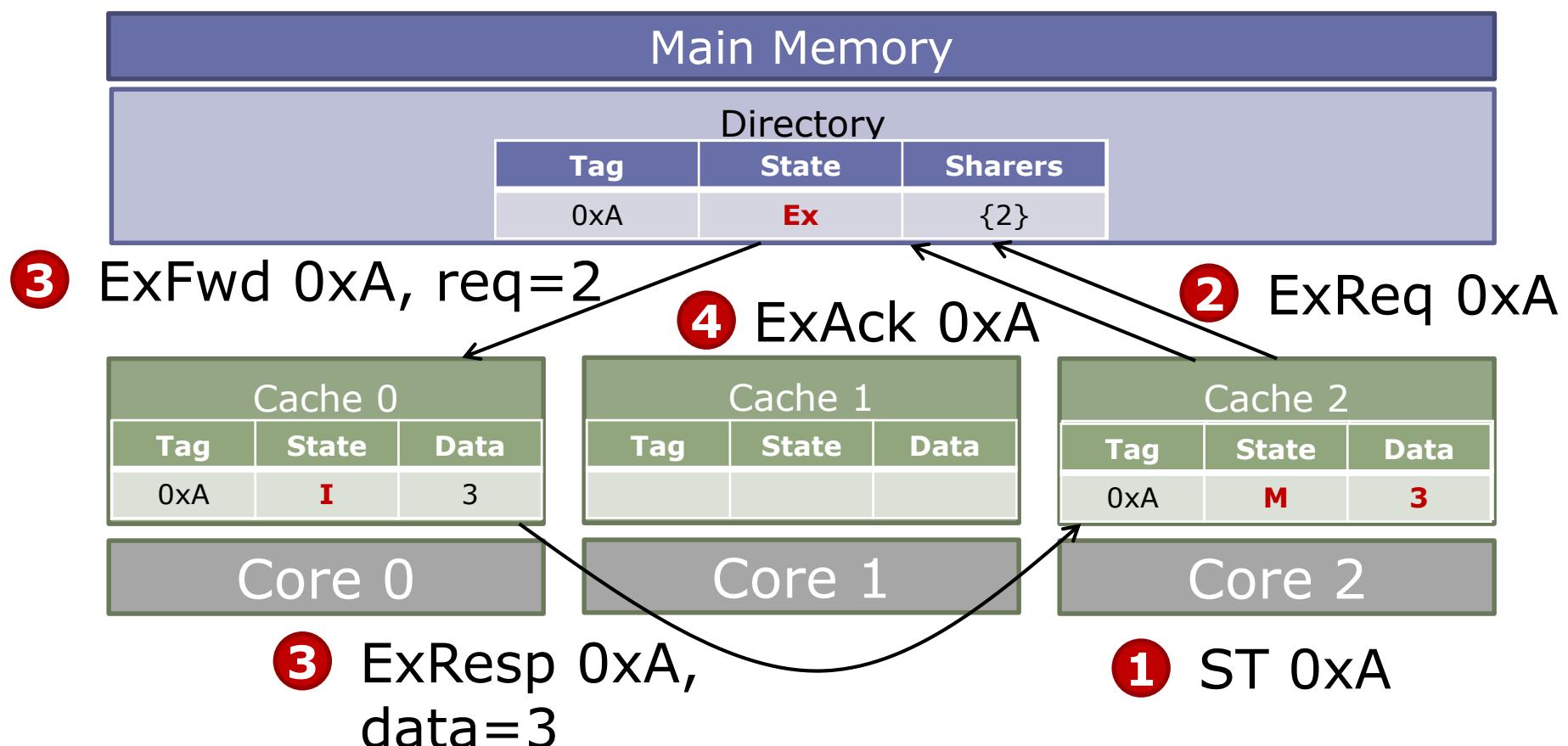
- Directory serializes multiple requests for the same address
 - Same-address requests are queued or NACKed and retried
- But races still exist due to conflicting requests
- Example: Upgrade race



Extra Hops and 3-Hop Protocols

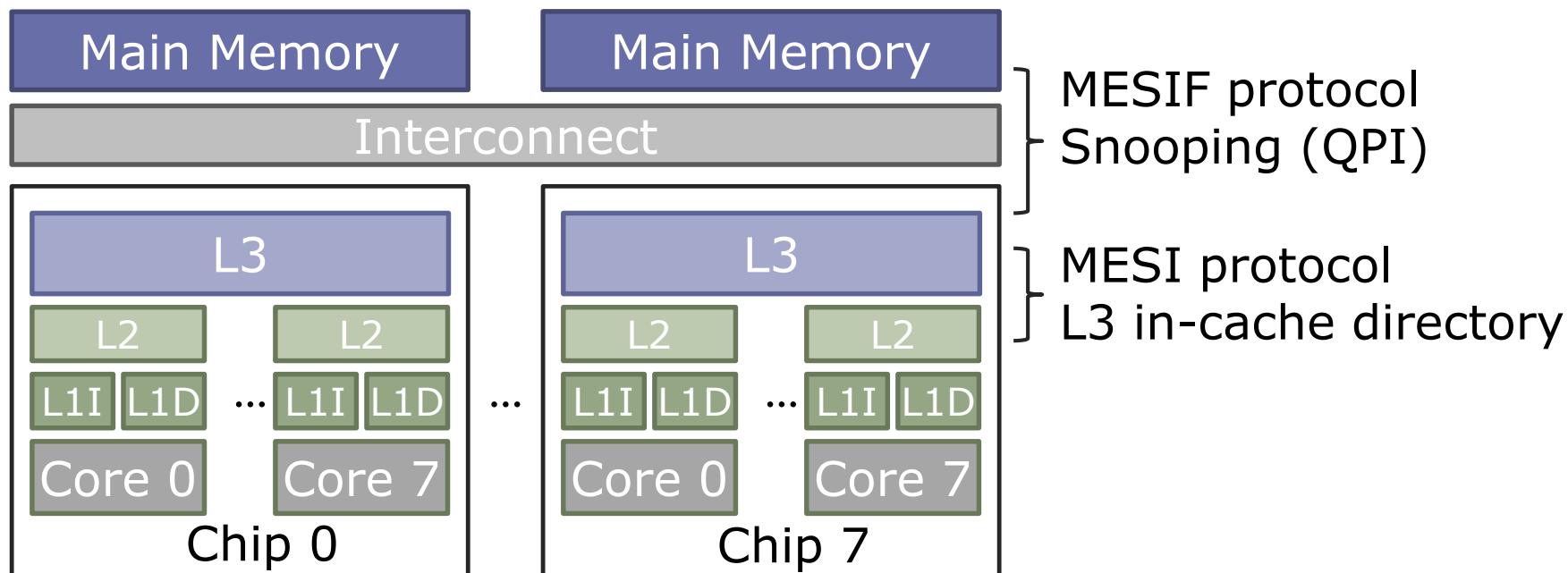
Reducing Protocol Latency

- Problem: Data in another cache needs to pass through the directory, adding latency
- Optimization: Forward data to requester directly



Coherence in Multi-Level Hierarchies

- Can use the same or different protocols to keep coherence across multiple levels
- Key invariant: Ensure sufficient permissions in all intermediate levels
- Example: 8-socket Xeon E7 (8 cores/socket)



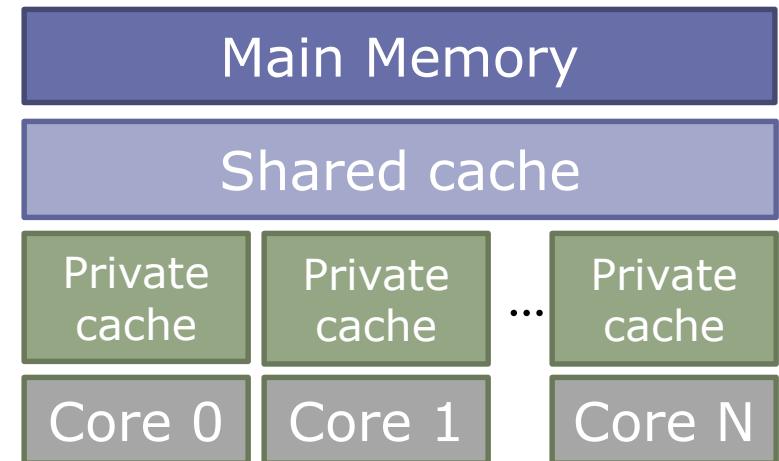
In-Cache Directories

- Common multicore memory hierarchy:

- 1+ levels of private caches
 - A shared last-level cache
 - Need to enforce coherence among private caches

- Idea: Embed the directory information in shared cache tags

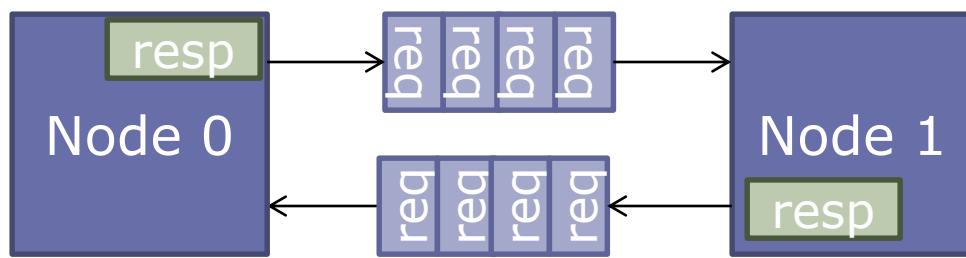
- Shared cache must be inclusive



- ✓ Avoids tag overheads & separate lookups
- ✗ Can be inefficient if shared cache size \gg sum(private cache sizes)

Avoiding Protocol Deadlock

- Protocols can cause deadlocks even if network is deadlock-free! (*more on this later*)



Example: Both nodes saturate all intermediate buffers with requests to each other, blocking responses from entering the network

- Solution: Separate *virtual networks*
 - Different sets of virtual channels and endpoint buffers
 - Same physical routers and links
- Most protocols require at least 2 virtual networks (for requests and replies), often >2 needed

Implementing Atomic Instructions

- In general, an *atomic read-modify-write* instruction requires two memory operations without intervening memory operations by other processors
- Implementation options:
 - *With snoopy coherence, lock the bus* -> expensive
 - *With directory-based coherence, lock the line in the cache (prevent invalidations or evictions until atomic op finishes)*
-> complex
- Modern processors often use
load-reserve
store-conditional

Load-reserve & Store-conditional

Special register(s) to hold reservation flag and address, and the outcome of store-conditional

Load-reserve R, (a):

```
<flag, adr> ← <1, a>;  
R ← M[a];
```

Store-conditional (a), R:

```
if <flag, adr> == <1, a>  
then cancel other procs'  
reservation on a;  
M[a] ← <R>;  
status ← succeed;  
else status ← fail;
```

If the cache receives an invalidation to the address in the reserve register, the reserve bit is set to **0**

- Several processors may reserve 'a' simultaneously
- These instructions are like ordinary loads and stores with respect to the bus traffic

Load-Reserve/Store-Conditional

Swap implemented with Ld-Reserve/St-Conditional

Swap(R1, mutex):

L: Ld-Reserve R2, (mutex)
 St-Conditional (mutex), R1
 if (status == fail) goto L
 R1 <- R2

Performance:

Load-reserve & Store-conditional

The total number of coherence transactions is not necessarily reduced, but splitting an atomic instruction into load-reserve & store-conditional:

- *increases utilization* (and reduces processor stall time), especially in split-transaction buses and directories
- *reduces cache ping-pong effect* because processors trying to acquire a semaphore do not have to perform stores each time

Thank you!

*Next Lecture:
Consistency and
Relaxed Memory Models*

Memory Consistency Models

Daniel Sanchez

Computer Science and Artificial Intelligence Lab
M.I.T.

Coherence vs Consistency

- Cache coherence makes private caches invisible to software
 - Concerns reads/writes to a single memory location
- Memory consistency models precisely specify how memory behaves with respect to read and write operations from multiple processors
 - Concerns reads/writes to multiple memory locations

Why Consistency Matters

Initial memory contents

a: 0

flag: 0

Processor 1

Store (a), 10;

Store (flag), 1;

Processor 2

L: Load r1, (flag);

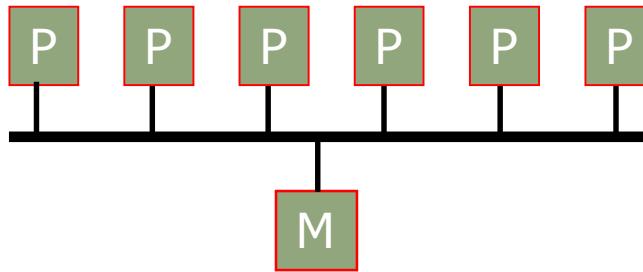
if $r_1 == 0$ goto L;

Load r2, (a);

- What value does r2 hold after both processors finish running this code?

Sequential Consistency

A Straightforward Memory Model

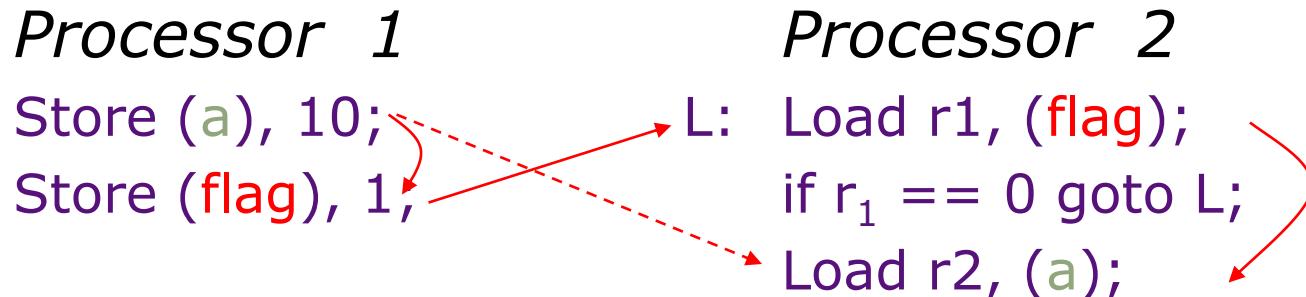


“A system is *sequentially consistent* if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in the order specified by the program”

Leslie Lamport

Sequential Consistency =
arbitrary *order-preserving interleaving*
of memory references of sequential programs

Sequential Consistency



- In-order instruction execution
- Atomic loads and stores

SC is easy to understand, but architects and compiler writers want to violate it for performance

Memory Model Issues

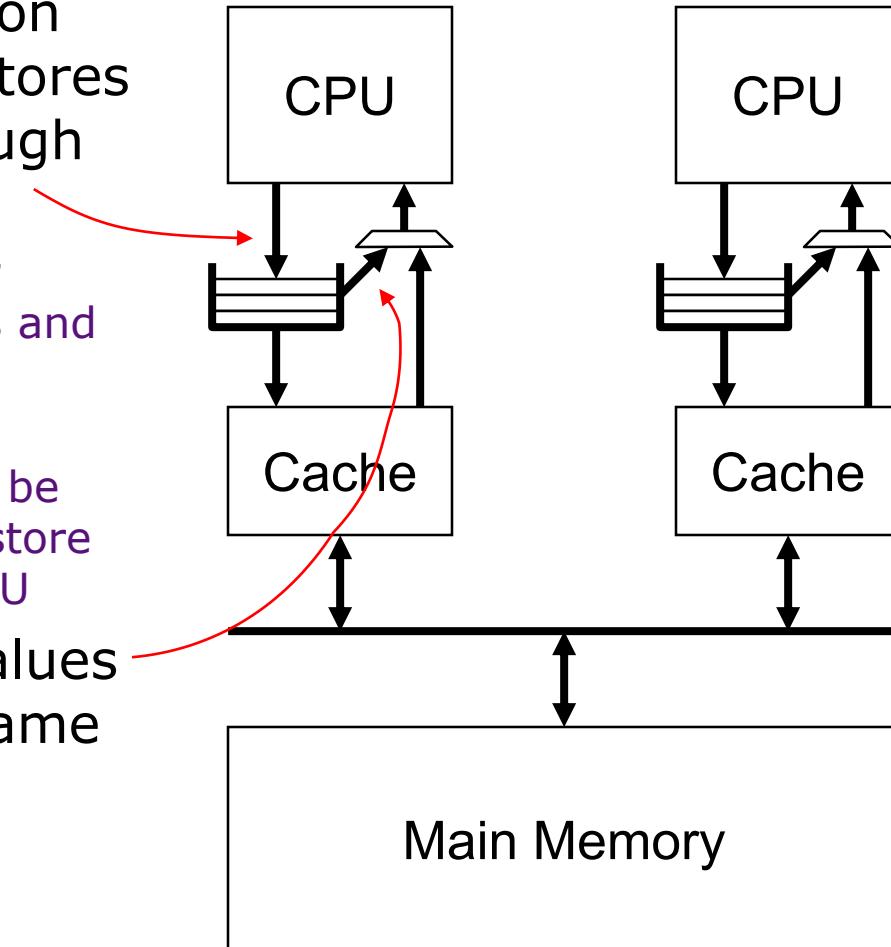
Architectural optimizations that are correct for uniprocessors often violate sequential consistency and result in a new memory model for multiprocessors

Consistency Models

- Sequential Consistency
 - All reads and writes in order
- Relaxed Consistency (one or more of the following)
 - Loads may be reordered after loads
 - e.g., PA-RISC, Power, Alpha
 - Loads may be reordered after stores
 - e.g., PA-RISC, Power, Alpha
 - Stores may be reordered after stores
 - e.g., PA-RISC, Power, Alpha, PSO
 - Stores may be reordered after loads
 - e.g., PA-RISC, Power, Alpha, PSO, TSO, x86
 - Other more esoteric characteristics
 - e.g., Alpha

Committed Store Buffers

- CPU can continue execution while earlier committed stores are still propagating through memory system
 - Processor can commit other instructions (including loads and stores) while first store is committing to memory
 - Committed store buffer can be combined with speculative store buffer in an out-of-order CPU
- Local loads can bypass values from buffered stores to same address



Example 1: Store Buffers

<i>Process 1</i>	<i>Process 2</i>
Store (flag_1),1;	Store (flag_2),1;
Load r_1 , (flag_2);	Load r_2 , (flag_1);

Question: Is it possible that $r_1=0$ and $r_2=0$?

- *Sequential consistency:*
- *Suppose Loads can go ahead of Stores waiting in the store buffer:*

Total Store Order (TSO):

Sun SPARC, IBM 370

Initially, all memory locations contain zeros

Example 2: Store-Load Bypassing

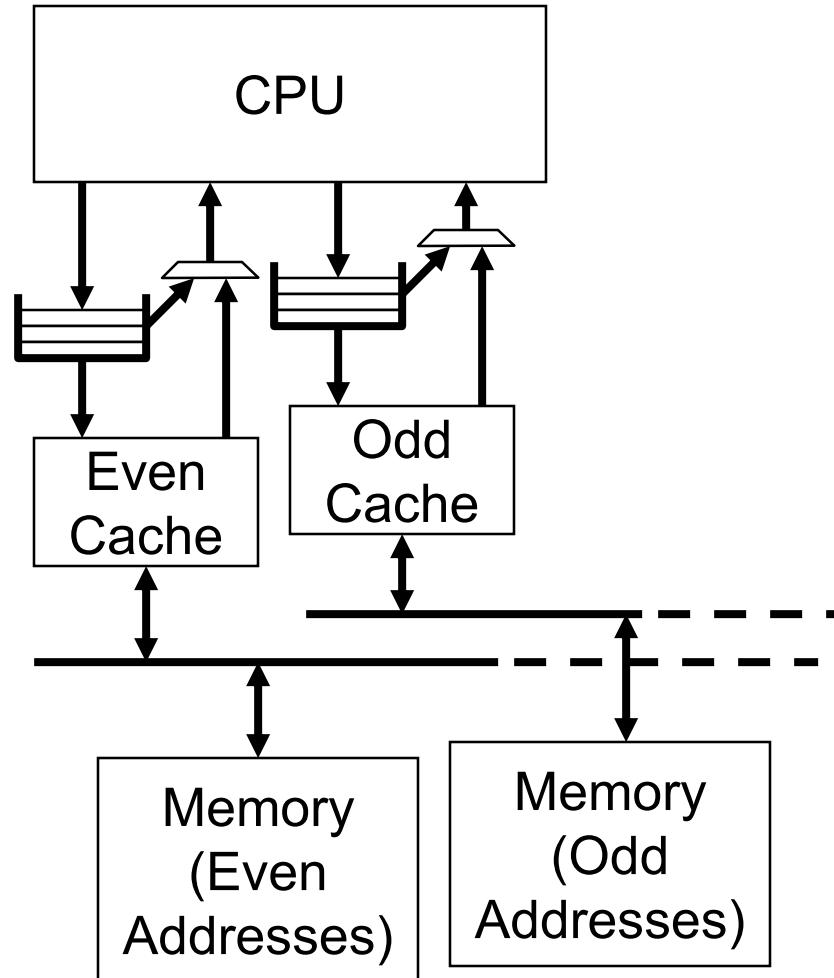
<i>Process 1</i>	<i>Process 2</i>
Store (flag ₁), 1;	Store (flag ₂), 1;
Load r ₃ , (flag ₁);	Load r ₄ , (flag ₂);
Load r ₁ , (flag ₂);	Load r ₂ , (flag ₁);

Question: Do extra Loads have any effect?

- *Sequential consistency:*
- *Suppose Store-Load bypassing is permitted in the store buffer*
 - No effect in Sparc's TSO model, still not SC
 - In IBM 370, a load cannot return a written value until it is visible to other processors => implicitly adds a memory fence, looks like SC

Interleaved Memory System

- Achieve greater throughput by spreading memory addresses across two or more parallel memory subsystems
 - In snooping system, can have two or more snoops in progress at same time (e.g., Sun UE10K system has four interleaved snooping busses)
 - Greater bandwidth from main memory system as two memory modules can be accessed in parallel



Example 3: Non-FIFO Store buffers

<i>Process 1</i>	<i>Process 2</i>
Store (a), 1;	Load r_1 , (flag);
Store (flag), 1;	Load r_2 , (a);

Question: Is it possible that $r_1=1$ but $r_2=0$?

- *Sequential consistency:*
- *With non-FIFO store buffers:*

Sparc's PSO memory model

Example 4: Non-Blocking Caches

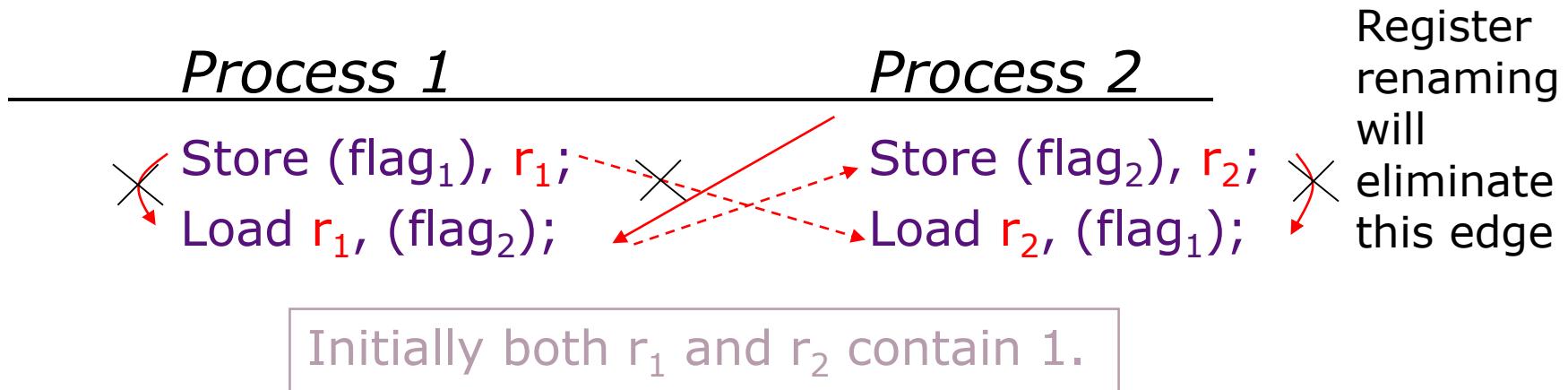
<i>Process 1</i>	<i>Process 2</i>
Store (a), 1;	Load r_1 , (flag);
Store (flag), 1;	Load r_2 , (a);

Question: Is it possible that $r_1=1$ but $r_2=0$?

- *Sequential consistency:*
- *Assuming stores are ordered:*

Alpha, Sparc's RMO, PowerPC's WO

Example 5: Register Renaming



Question: Is it possible that $r_1=0$ but $r_2=0$?

- *Sequential consistency: No*
- *Register renaming: Yes because it removes anti-dependencies*

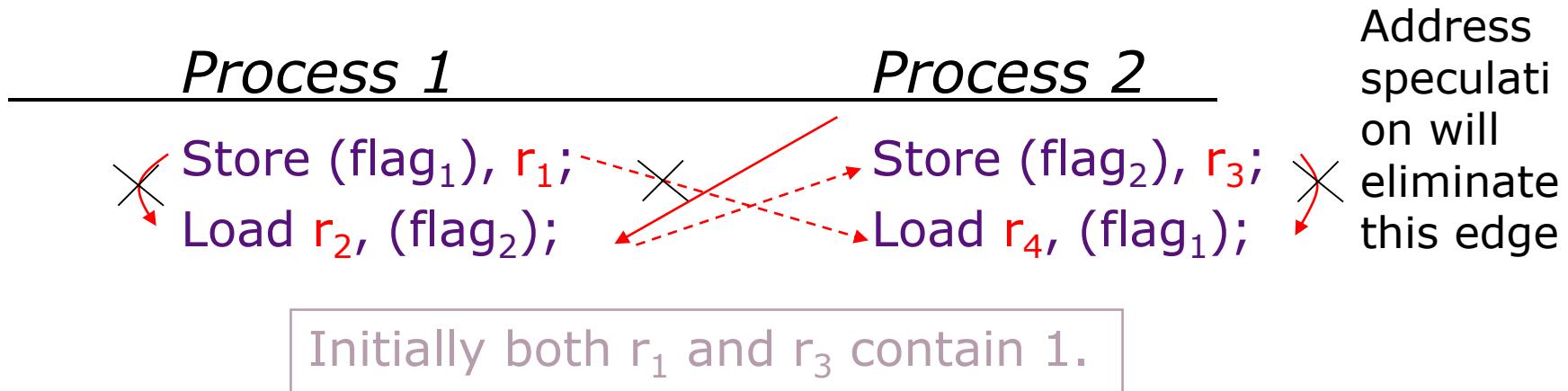
Example 6: Speculative Execution

<i>Process 1</i>	<i>Process 2</i>
Store (a), 1;	L: Load r_1 , (flag);
Store (flag), 1;	if $r_1 == 0$ goto L; Load r_2 , (a);

Question: Is it possible that $r_1=1$ but $r_2=0$?

- *Sequential consistency:*
- *With speculative loads:*

Example 7: Address Speculation



Question: Is it possible that r₂=0 but r₄=0?

- *Sequential consistency:*
- *Address speculation:*

Flag₁ and flag₂ are registers pointing at memory locations

Example 8: Store Atomicity

<u>Process 1</u>	<u>Process 2</u>	<u>Process 3</u>	<u>Process 4</u>
Store (a),1;	Store (a),2;	Load r ₁ , (a); Load r ₂ , (a);	Load r ₃ , (a); Load r ₄ , (a);

*Question: Is it possible that r₁=1 and r₂=2
but r₃=2 and r₄=1 ?*

- Sequential consistency:
- *Even if Loads on a processor are ordered,
the different ordering of stores can be
observed if the Store operation is not
atomic.*

Example 9: Causality

<i>Process 1</i>	<i>Process 2</i>	<i>Process 3</i>
Store (flag ₁),1;	Load r ₁ , (flag ₁); Store (flag ₂),1;	Load r ₂ , (flag ₂); Load r ₃ , (flag ₁);

Question: Is it possible that r₁=1 and r₂=1 but r₃=0 ?

- *Sequential consistency:*
- *With load/load reordering:*

Alpha

Weaker Memory Models & Memory Fence Instructions

- Architectures with weaker memory models provide memory fence instructions to prevent otherwise permitted reorderings of loads and stores

Store (a_1), r2;

Fence_{wr}

Load r1, (a_2);

The Load and Store can be reordered if $a_1 \neq a_2$.
Insertion of Fence_{wr} will disallow this reordering

Similarly: Fence_{rr}; Fence_{rw}; Fence_{ww};

SUN's Sparc: MEMBAR;

MEMBARRR; MEMBARRW; MEMBARWR; MEMBARWW

PowerPC: Sync; EIEIO

Enforcing Ordering using Fences

Processor 1

Store (a),10;
Store (flag),1;

Processor 2

L: Load r_1 , (flag);
if $r_1 == 0$ goto L;
Load r_2 , (a);

Processor 1

Store (a),10;
Fence_{ww};
Store (flag),1;

Processor 2

L: Load r_1 , (flag);
if $r_1 == 0$ goto L;
Fence_{rr};
Load r_2 , (a);

Weak ordering

Weaker (Relaxed) Memory Models



- Hard to understand and remember
- Unstable - *Modèle de l'année*
- Abandon weaker memory models in favor of implementing SC

Implementing SC

1. The memory operations of each individual processor appear to all processors in the order the requests are made to the memory
 - *Provided by cache coherence*, which ensures that all processors observe the same order of loads and stores to an address
2. Any execution is the same as if the operations of all the processors were executed in some sequential order
 - Provided by enforcing a dependence between each memory operation and the following one

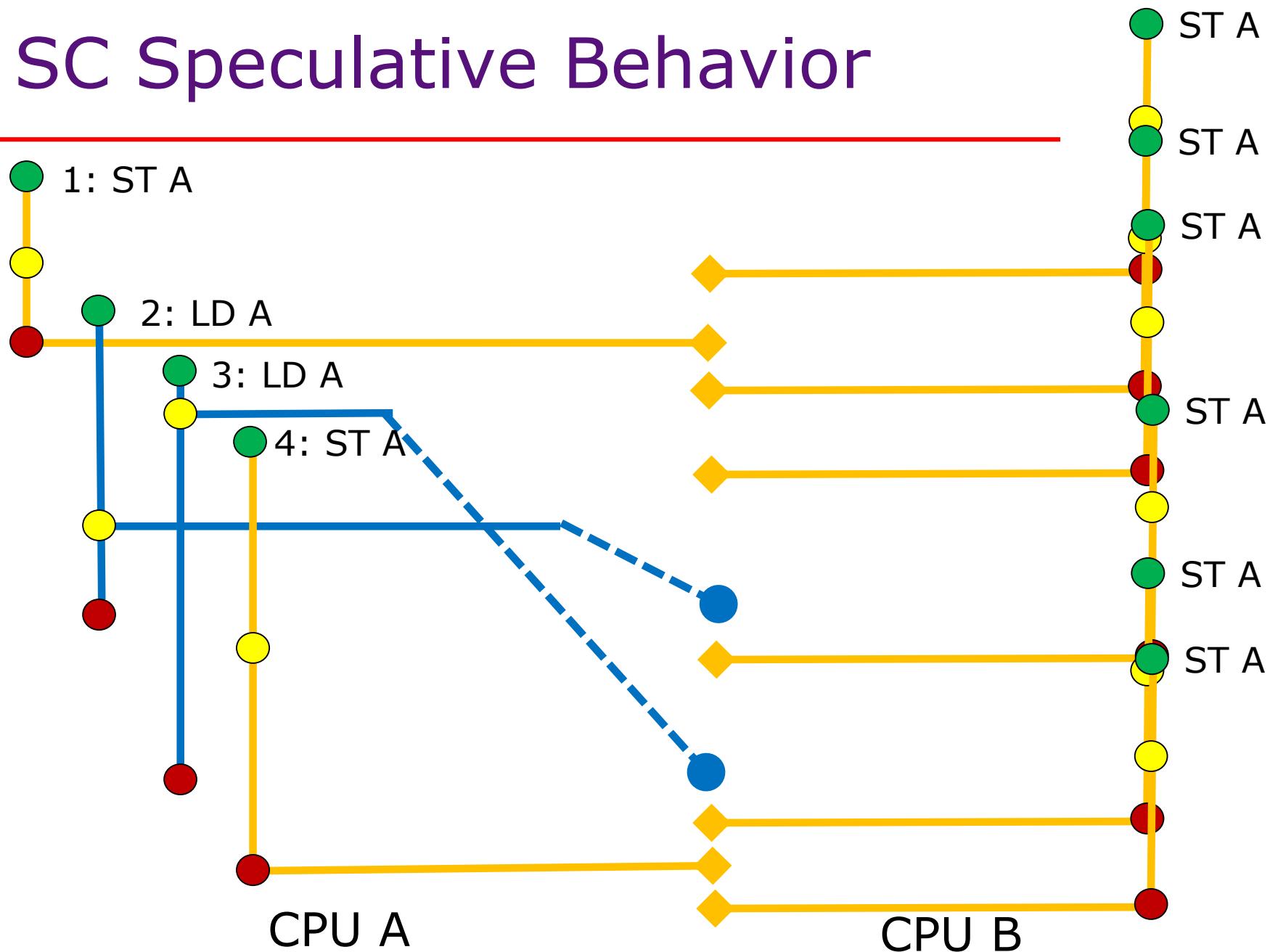
SC Data Dependence

- *Stall*
 - Use *in-order execution and blocking caches*
 - Cache coherence plus allowing a processor to have only one request in flight at a time will provide SC
- *Change architecture \Rightarrow Relaxed memory models*
 - Use *OOO and non-blocking caches*
 - Cache coherence and allowing multiple concurrent requests (to different addresses) gives high performance
 - Add fence operations to force ordering when needed
- *Speculate...*

Sequential Consistency Speculation

- Local load-store ordering uses standard OOO mechanism
- Globally non-speculative stores
 - Stores execute at commit -> stores are in-order!
- Globally speculative loads
 - **Guess** at issue that the memory location used by a load will not change between issue and commit of the instruction
 - this is equivalent to loads happening in-order at commit
 - **Check** at commit by remembering all loads addresses starting at issue and watching for writes to that location.
 - **Data Management** for rollback relies on the basic out-of-order speculative data management used for uni-processor rollback and instruction re-execution.

SC Speculative Behavior



Properly Synchronized Programs

- Very few programmers do programming that relies on SC; instead, they use higher-level synchronization primitives
 - locks, semaphores, monitors, atomic transactions
- A “properly synchronized program” is one where each shared writable variable is protected (say, by a lock) so that there is no race in updating the variable
 - There is still race to get the lock
 - There is no way to check if a program is properly synchronized
- For properly synchronized programs, instruction reordering does not matter as long as updated values are committed before leaving a locked region

Release Consistency

[Garachorloo 1990]

- Only care about inter-processor memory ordering at thread synchronization points, not in between
- Can treat all synchronization instructions as the only ordering points

...

Acquire(lock) // All following loads get most recent written values

... Read and write shared data ..

Release(lock) // All preceding writes are globally visible before
// lock is freed.

...

Takeaways

- SC is too low level a programming model. High-level programming should be based on critical sections & locks, atomic transactions, monitors, ...
- High-level parallel programming should be oblivious of memory model issues
 - Programmer should not be affected by changes in the memory model
- ISA definition for Load, Store, Memory Fence, synchronization instructions should
 - Be precise
 - Permit maximum flexibility in hardware implementation
 - Permit efficient implementation of high-level parallel constructs

Thank you!

*Next Lecture:
On-Chip Networks*

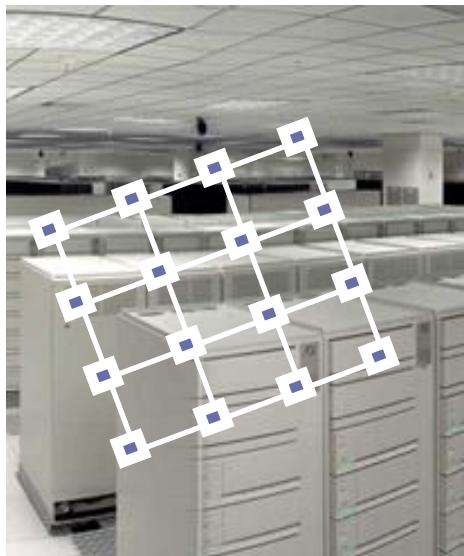
On-Chip Networks I: Topology/Flow Control

Daniel Sanchez

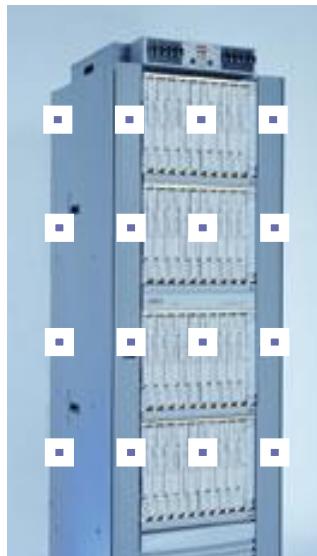
Computer Science & Artificial Intelligence Lab
M.I.T.

History: From interconnection networks to on-chip networks

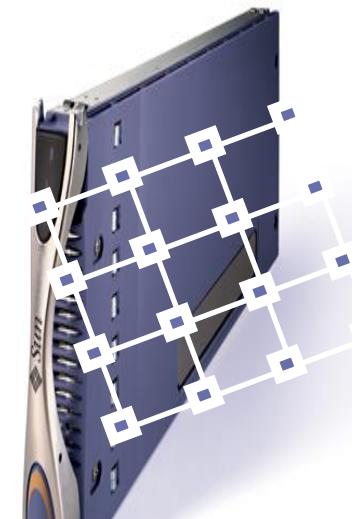
Box-to-box
networks



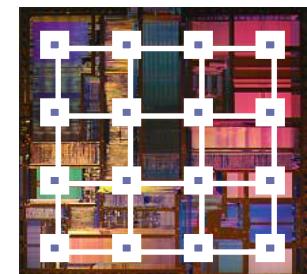
Board-to-board
networks



Chip-to-chip
networks



On-chip
networks

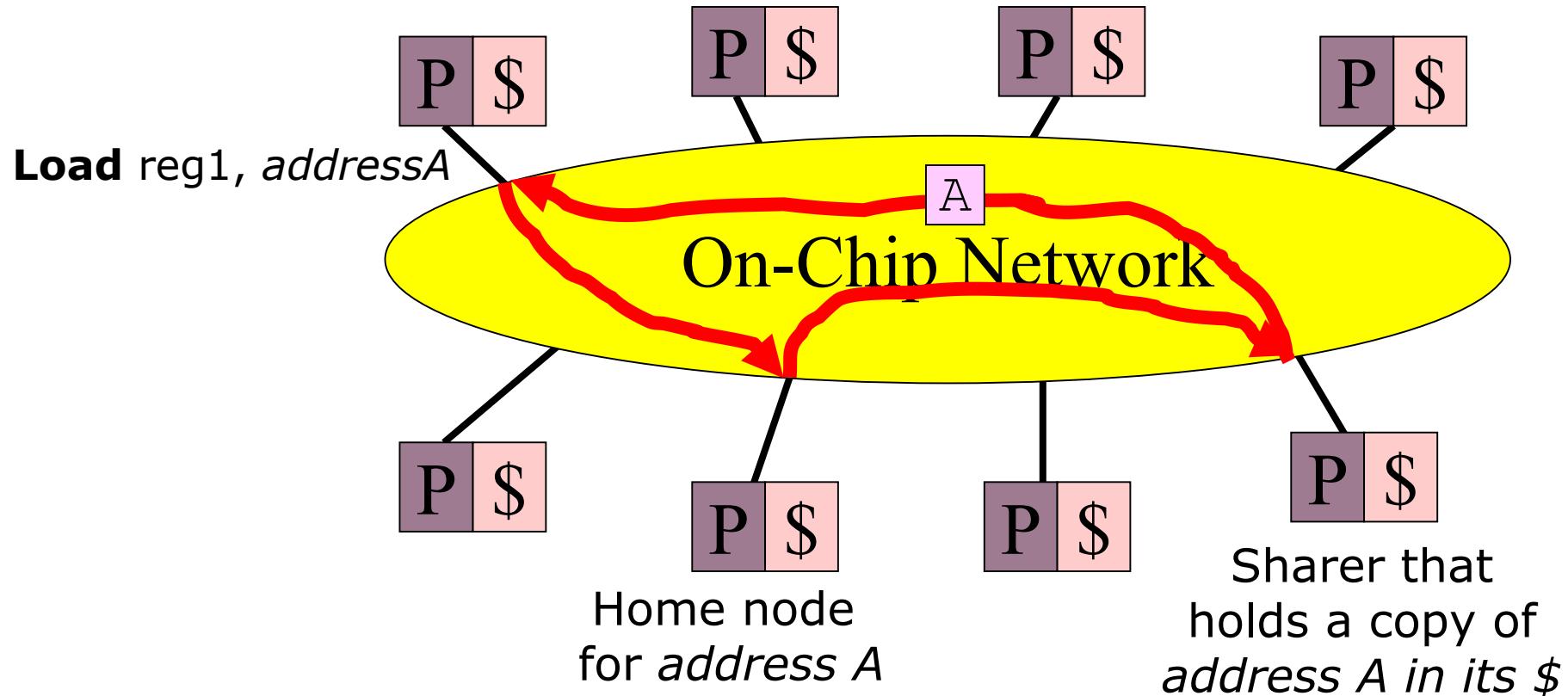


Focus on on-chip networks connecting caches in shared-memory processors

Multi-Chip: Supercomputers, Data Centers, Internet Routers, Servers
On-Chip: Servers, Laptops, Phones, HDTVs, Access routers

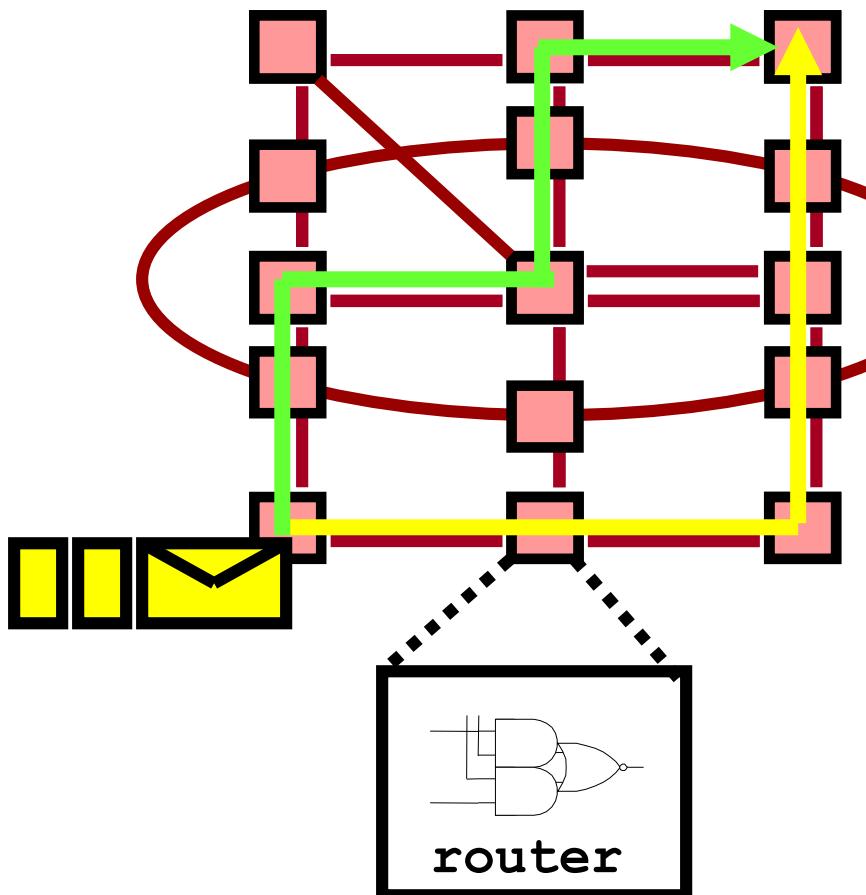
What's an on-chip network?

E.g. Cache-coherent chip multiprocessor



**Network transports cache coherence messages
and cache lines between processor cores**

Designing an on-chip network



- Topology
- Flow control
- Router microarchitecture
- Routing

Interconnection Network Architecture

- *Topology*: How to connect the nodes up?
(processors, memories, router line cards, ...)
- *Routing*: Which path should a message take?
- *Flow control*: How is the message actually forwarded from source to destination?
- *Router microarchitecture*: How to build the routers?
- *Link microarchitecture*: How to build the links?

Topology

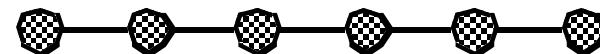
Topological Properties

- *Diameter*
- *Average Distance*
- *Bisection Bandwidth*

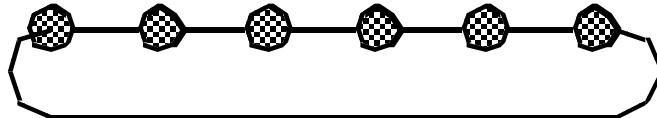
Topological Properties

- *Routing Distance* - number of links on route
- *Diameter* - maximum routing distance
- *Average Distance*
- A network is *partitioned* by a set of links if their removal disconnects the graph
- *Bisection Bandwidth* is the bandwidth crossing a minimal cut that divides the network in half

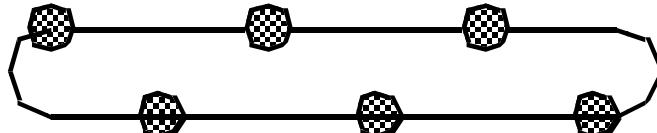
Linear Arrays and Rings



Linear Array



Torus



Torus arranged to use short wires

Route A -> B given by relative address $R = B-A$

Linear Array Ring (1-D Torus)

Diameter?

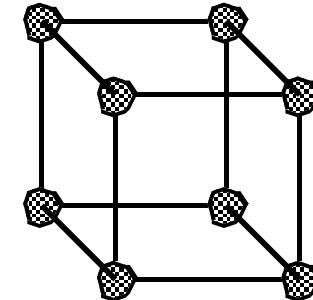
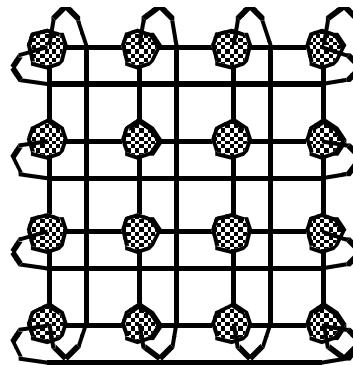
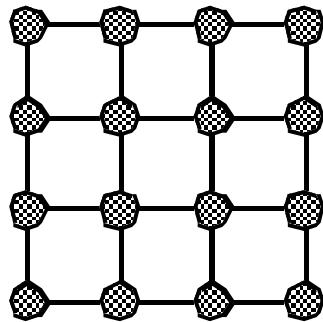
Average distance?

Bisection bandwidth?

- **Torus Examples:**

- FDDI, SCI, FiberChannel Arbitrated Loop, Intel Xeon

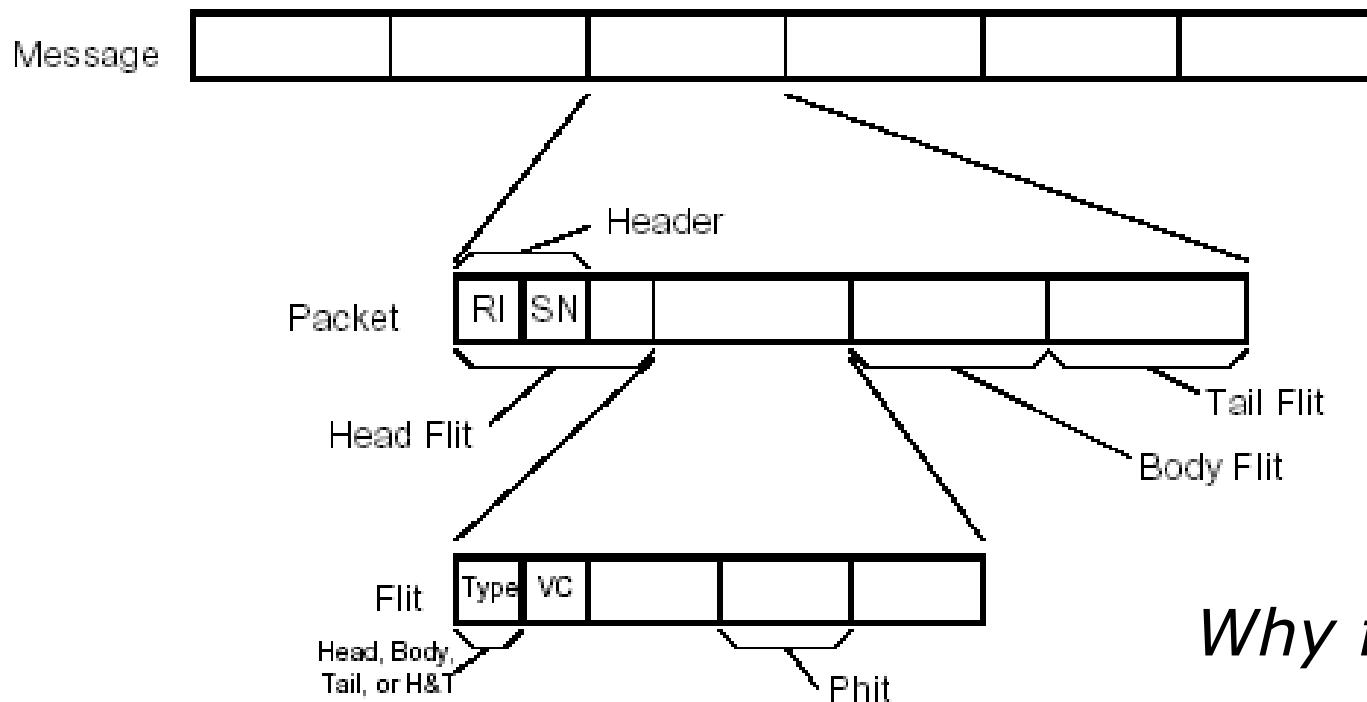
Multidimensional Meshes and Tori



- d -dimensional array
 - $n = k_{d-1} \times \dots \times k_0$ nodes
 - described by d -vector of coordinates (i_{d-1}, \dots, i_0)
- d -dimensional k -ary mesh: $N = k^d$
 - $k = \sqrt[d]{N}$
 - described by d -vector of radix k coordinate
- d -dimensional k -ary torus (or k -ary d -cube)

Routing & Flow Control Overview

Messages, Packets, Flits, Phits



Packet: Basic unit of routing and sequencing

- Limited size (e.g. 64 bits – 64 KB)

Flit (flow control digit): Basic unit of bandwidth/storage allocation

- All flits in packet follow the same path

Phit (physical transfer digit): data transferred in single clock

Routing vs Flow Control

- Routing algorithm chooses path that packets should follow to get from source to destination
- Flow control schemes allocate resources (buffers, links, control state) to packets traversing the network
- Our approach: Bottom-up
 - Today: Flow control, assuming routes are set
 - Next lecture: Routing algorithms

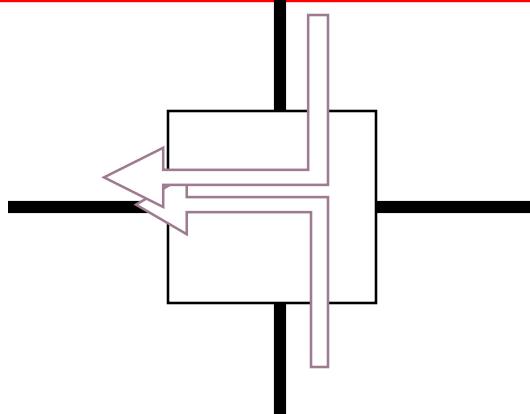
Properties of Routing Algorithms

- Deterministic/Oblivious
 - Route determined by (source, dest), not intermediate state (i.e. traffic)
- Adaptive
 - Route influenced by traffic along the way
- Minimal
 - Only selects shortest paths
- Deadlock-free
 - No traffic pattern can lead to a situation where no packets move forward

(more in next lecture)

Flow Control

Contention



- Two packets trying to use the same link at the same time
 - Limited or no buffering
- Problem arises because we are sharing resources
 - Sharing bandwidth and buffers

Flow Control Protocols

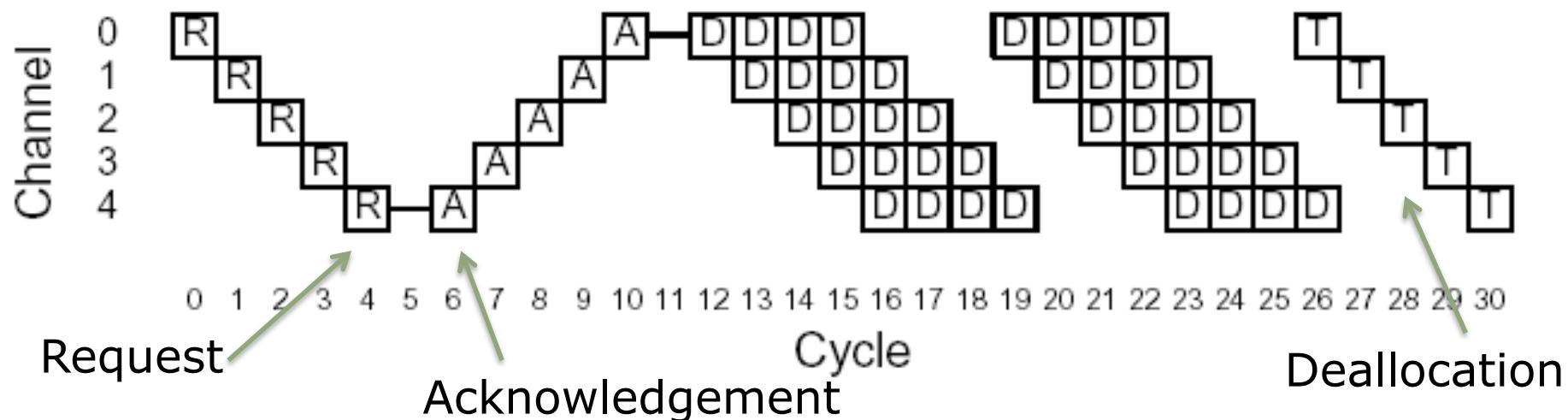
- Bufferless
 - Circuit switching
 - Dropping
 - Misrouting
- Buffered
 - Store-and-forward
 - Virtual cut-through
 - Wormhole
 - Virtual-channel



Circuit Switching

- Form a circuit from source to dest
- Probe to set up path through network
- Reserve all links
- Data sent through links
- Bufferless

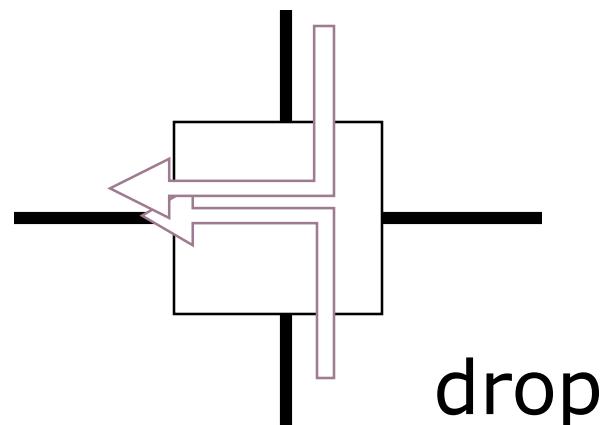
Time-space View: Circuit Switching



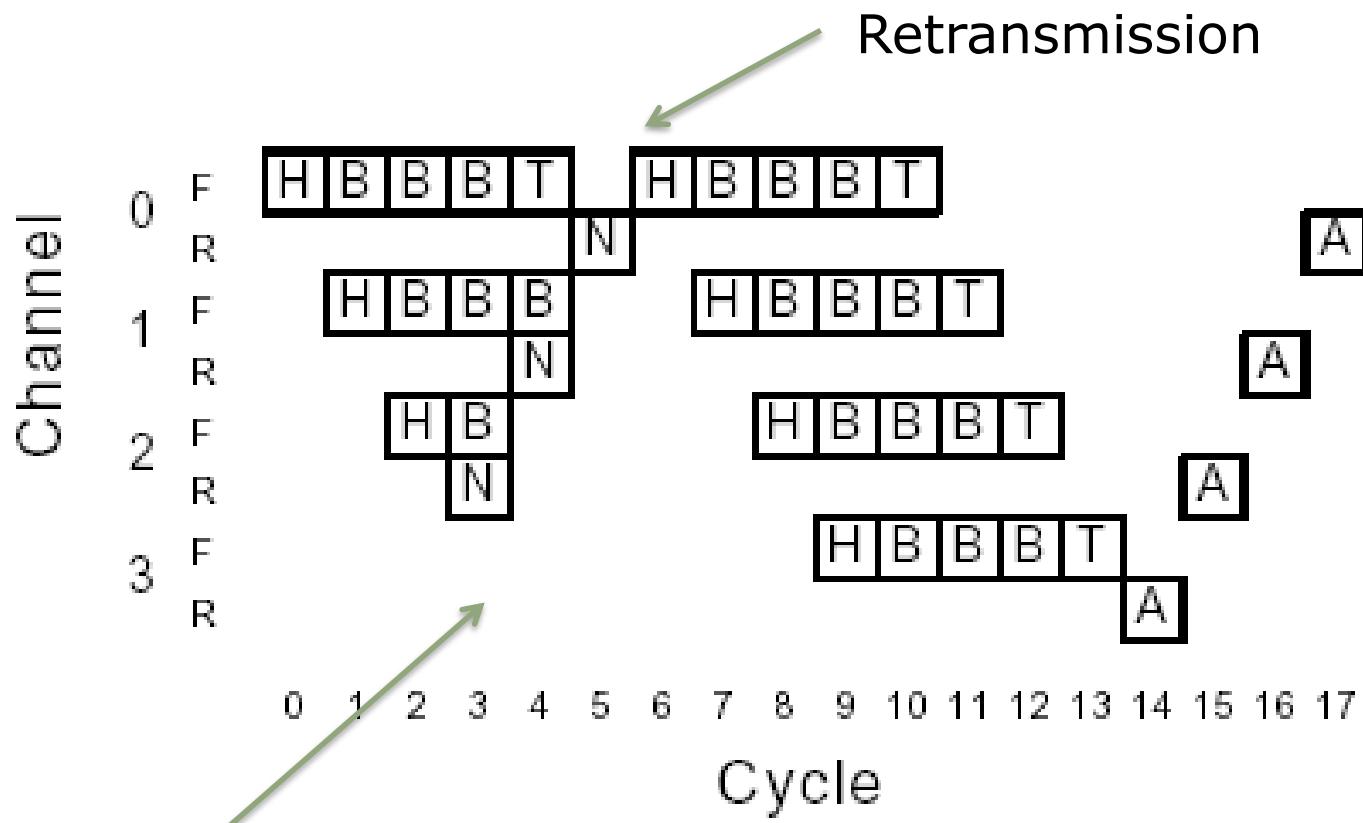
- *Why is this good?*
- *Why is it not?*

Speculative Flow Control: Dropping

- If two things arrive and I don't have resources, drop one of them
- Flow control protocol on the Internet



Time-space Diagram: Dropping

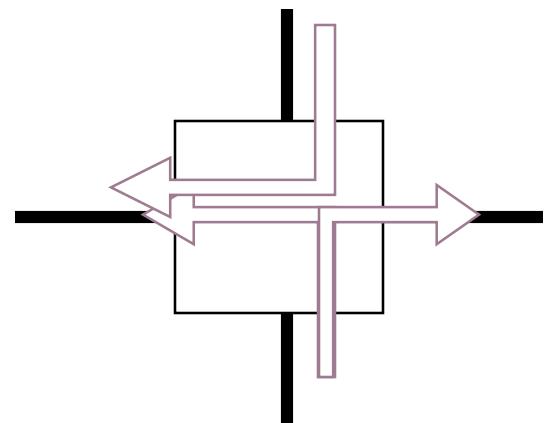


Unable to allocate channel 3

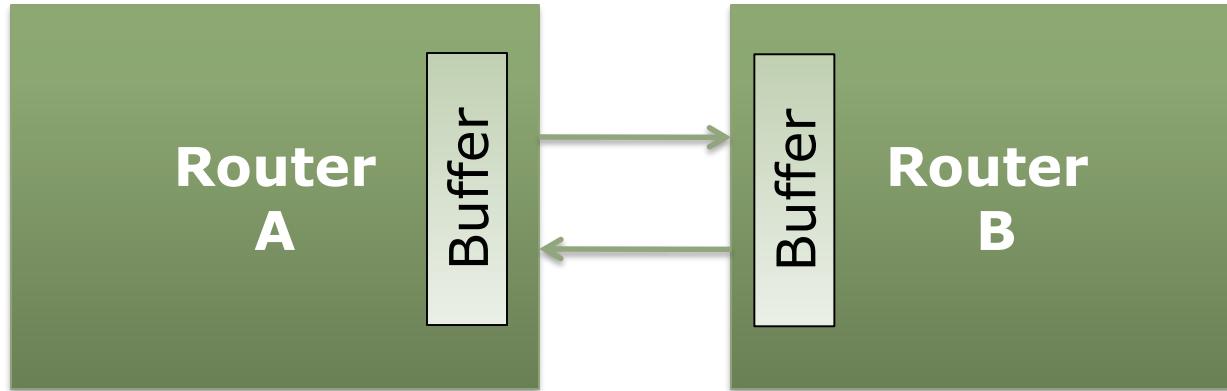
Disadvantages?

Less Simple Flow Control: Misrouting

- If only one message can enter the network at each node, and one message can exit the network at each node, the network can never be congested. Right?
- Philosophy behind misrouting: intentionally route away from congestion
- No need for buffering
- Problems?



Buffered Routing



- Link-level flow control:
 - Given that you can't drop packets, how to manage the buffers?
When can you send stuff forward, when not?
- Metrics of interest:
 - Throughput/Latency
 - Buffer utilization (turnaround time)

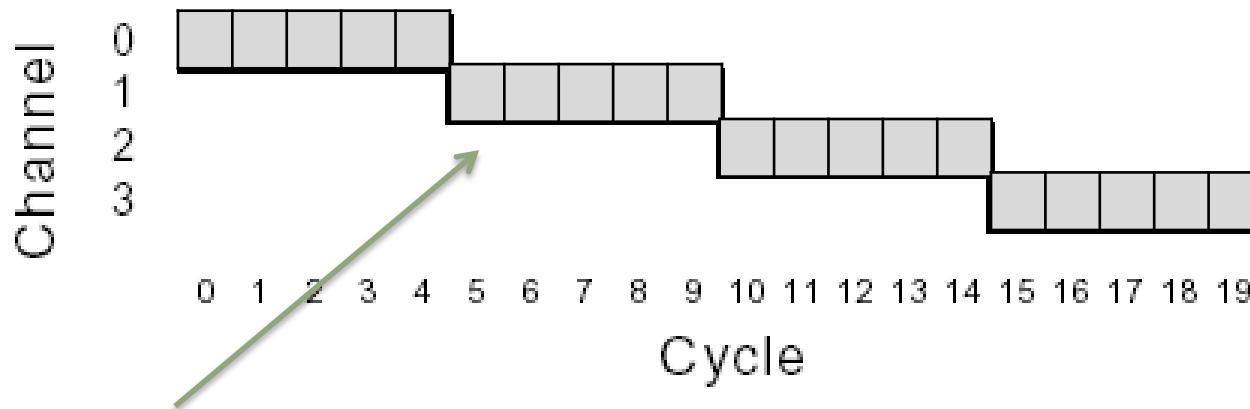
Techniques for link backpressure

- Naïve stall-based (on/off):
 - Can source send or not?
- Sophisticated stall-based (credit-based):
 - How many flits can be sent to the next node?
- Speculative (ack/nack):
 - Guess can always send, but keep copy
 - Resolve if send was successful (ack/nack)
 - On ack – drop copy
 - On nack - resend

Store-and-Forward (packet-based, no flits)

- Strategy:
 - Make intermediate stops and wait until the entire packet has arrived before you move on
- Advantage:
 - Other packets can use intermediate links

Time-space View: Store-and-Forward



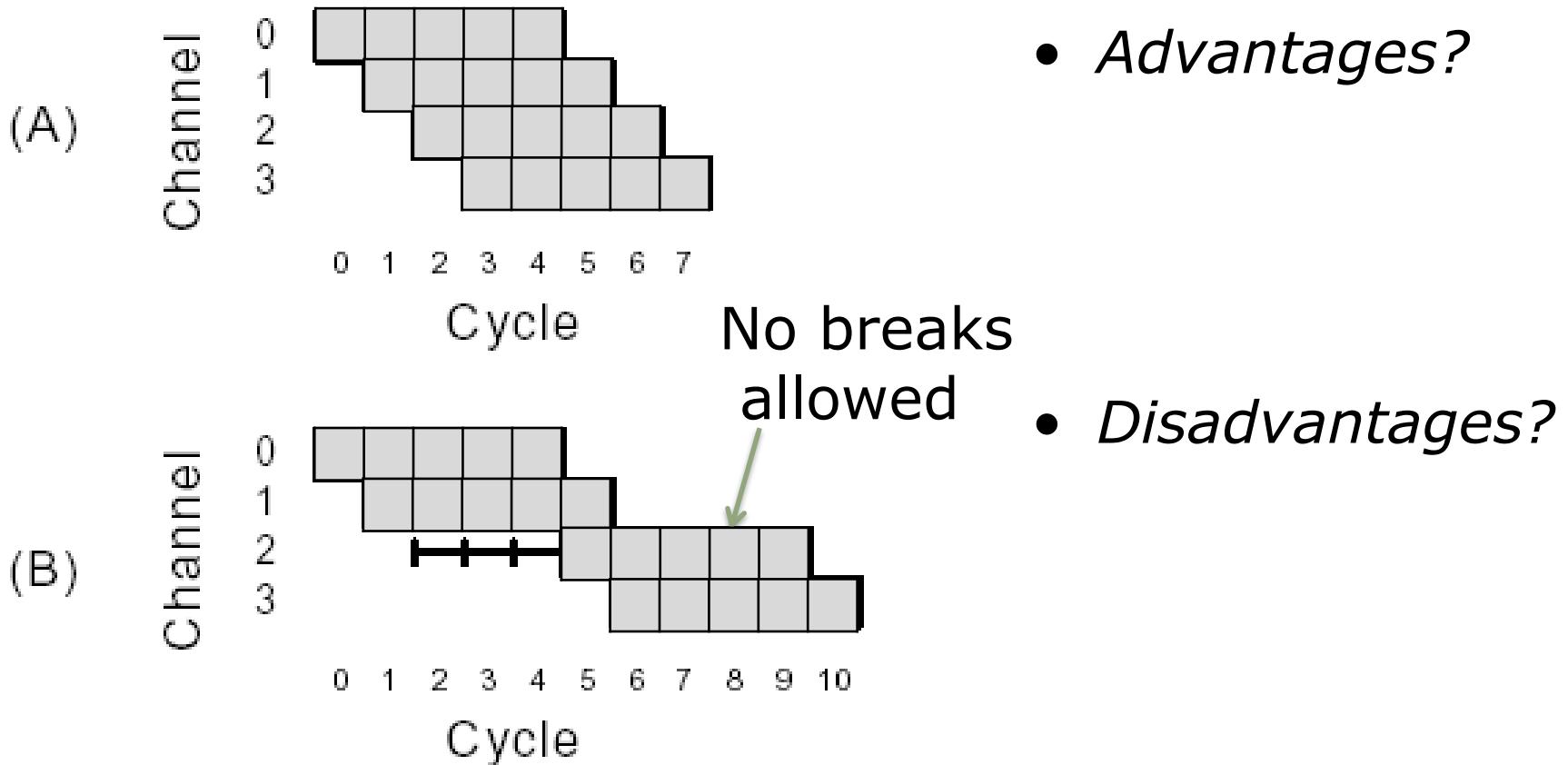
Could be allocated at a much later time without packet dropping

- Buffering allows packet to wait for channel
- *Drawback?*

Virtual Cut-through (packet-based)

- Why wait till entire message has arrived at each intermediate stop?
- The head flit of the packet can dash off first
- When the head gets blocked, whole packet gets blocked at one intermediate node
- Used in Alpha 21364

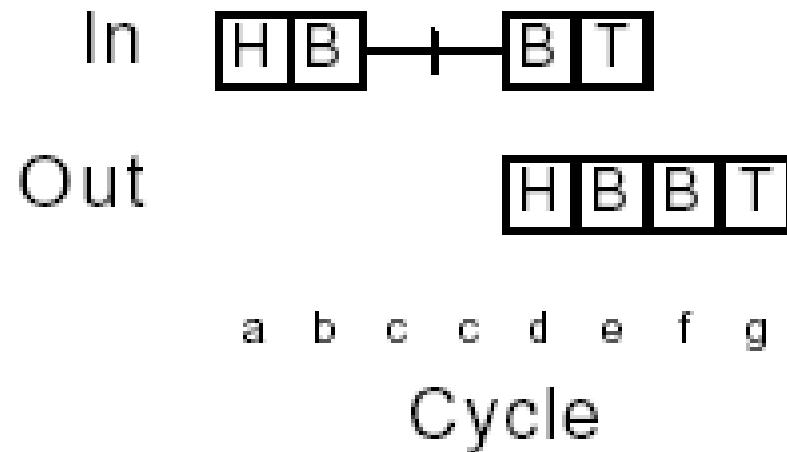
Time-space View: Virtual Cut-through



Flit-Buffer Flow Control: Wormhole

- When a packet blocks, just block wherever the pieces (flits) of the message are at that time.
- Operates like cut-through but with channel and buffers allocated to flits rather than packets
 - Channel state (virtual channel) allocated to **packet** so body flits can follow head flit

Time-space View: Wormhole

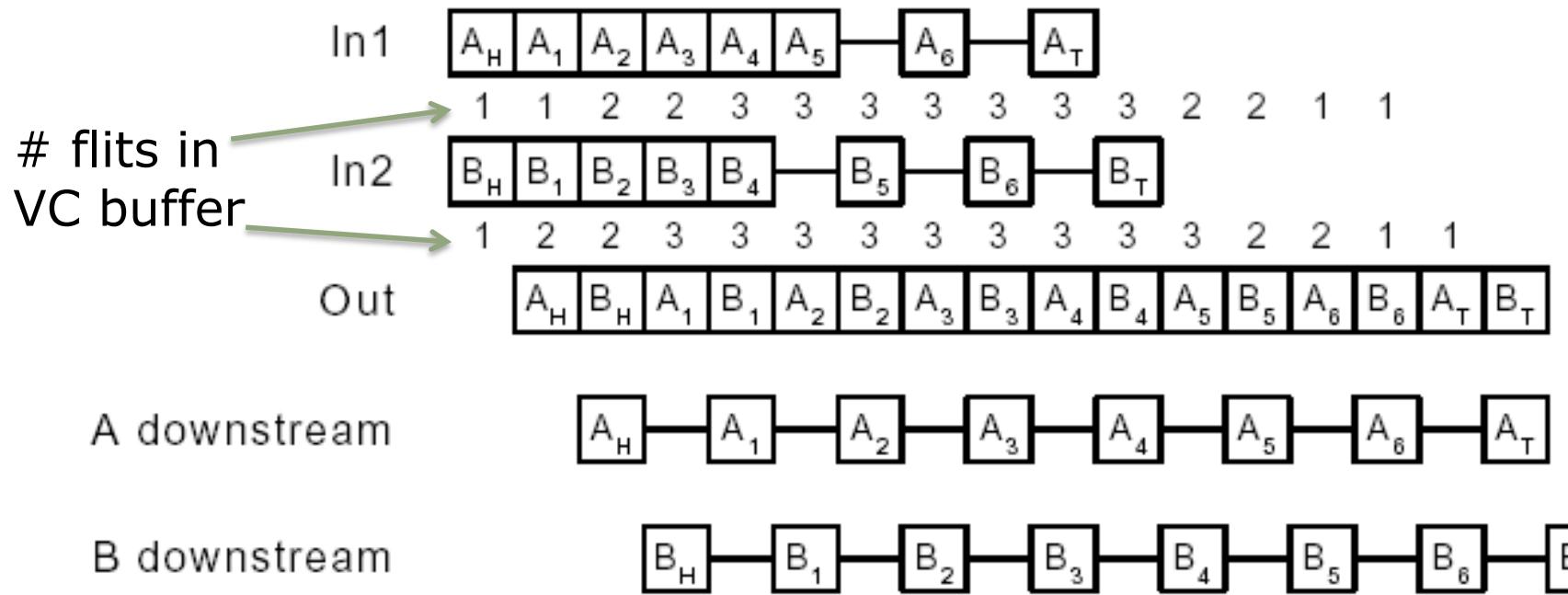


- *Advantages?*
- *Disadvantages?*

Virtual-Channel (VC) Flow Control

- When a message blocks, instead of holding on to links so others can't use them, hold on to **virtual** links
- Multiple queues in buffer storage
 - Like lanes on the highway
- Virtual channel can be thought of as channel state and flit buffers

Time-space View: Virtual-Channel



- *Advantages?*
- *Disadvantages?*

Thank you!

Next Lecture:
Router (Switch) Microarchitecture
Routing Algorithms

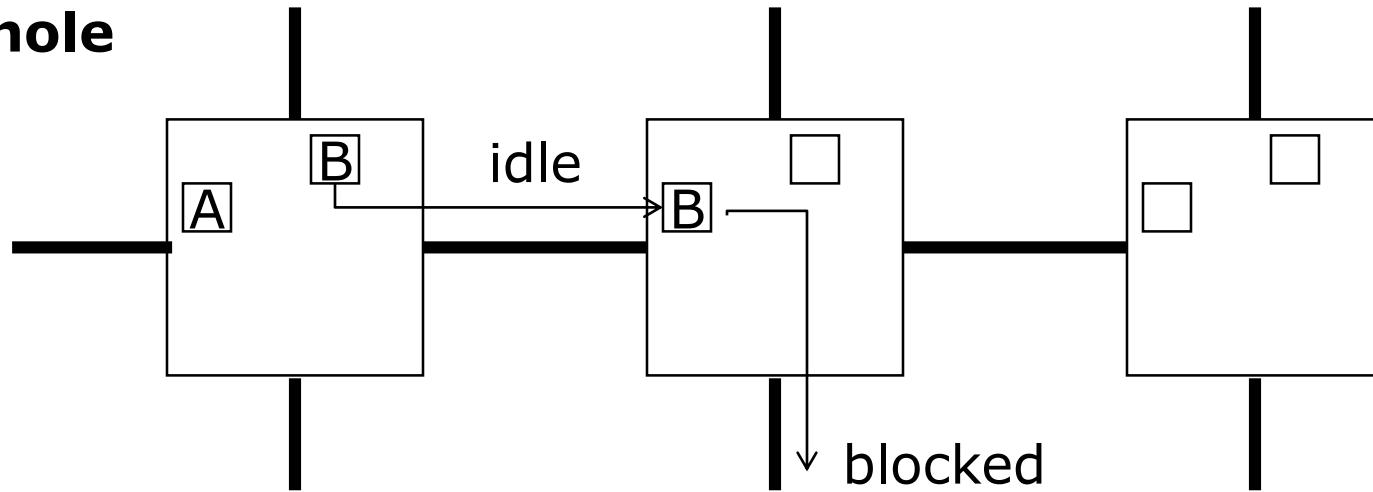
On-Chip Networks II: Router Microarchitecture & Routing

Daniel Sanchez
Computer Science & Artificial Intelligence Lab
M.I.T.

Recap: Wormhole Flow Control

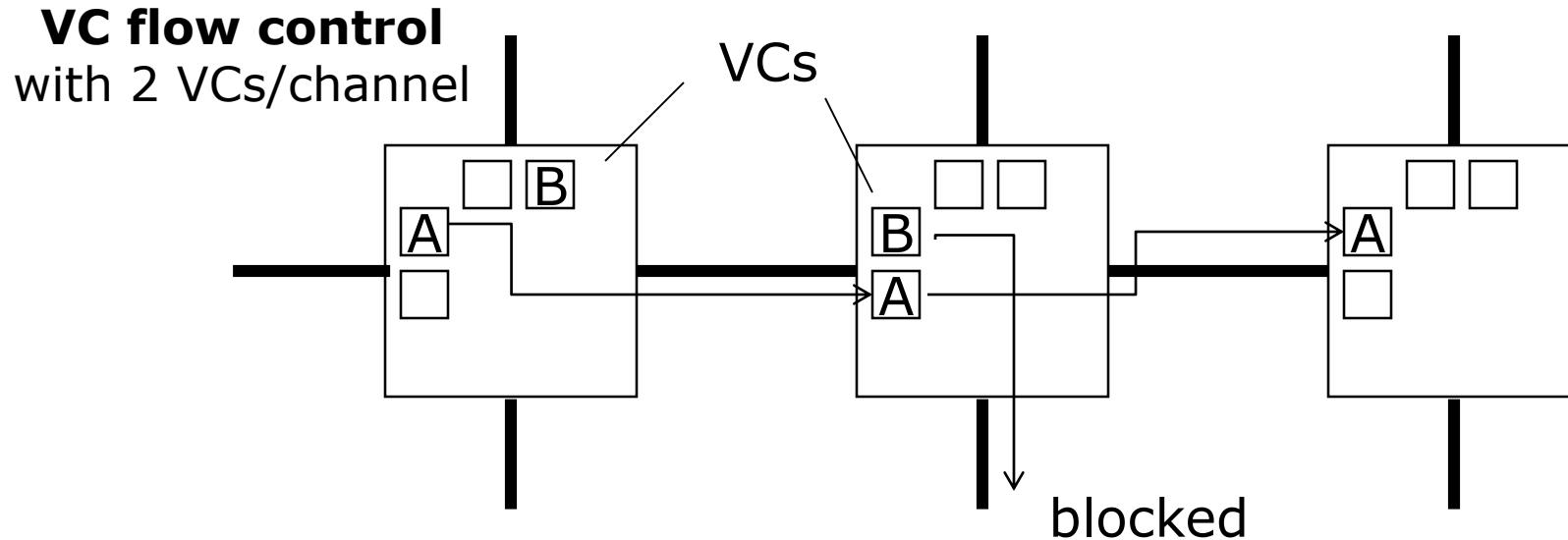
- Each router manages buffers in flits
- Each packet is sent through output link as soon as possible (without waiting for all its flits to arrive)
- Router buffers are not large enough to hold full packet → on congestion, packet's flits often buffered across routers
- Problem: On congestion, links assigned to a blocked packet cannot be used by other packets

Wormhole

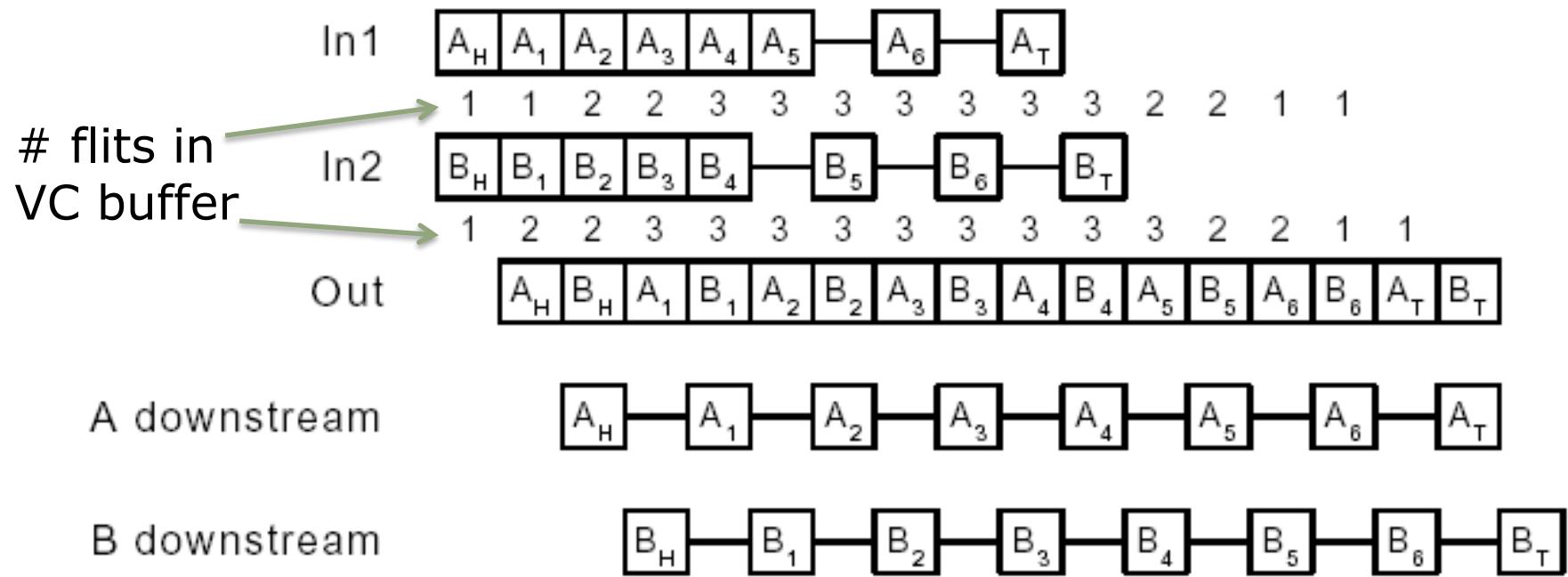


Recap: Virtual-Channel Flow Control

- When a packet blocks, instead of holding on to channel, hold on to **virtual channel**
- Virtual channel (VC) = channel state + flit buffers
- Multiple virtual channels reduce blocking
- Ex: Wormhole (=1 VC/channel) vs 2 VCs/channel



Time-Space View: Virtual-Channel



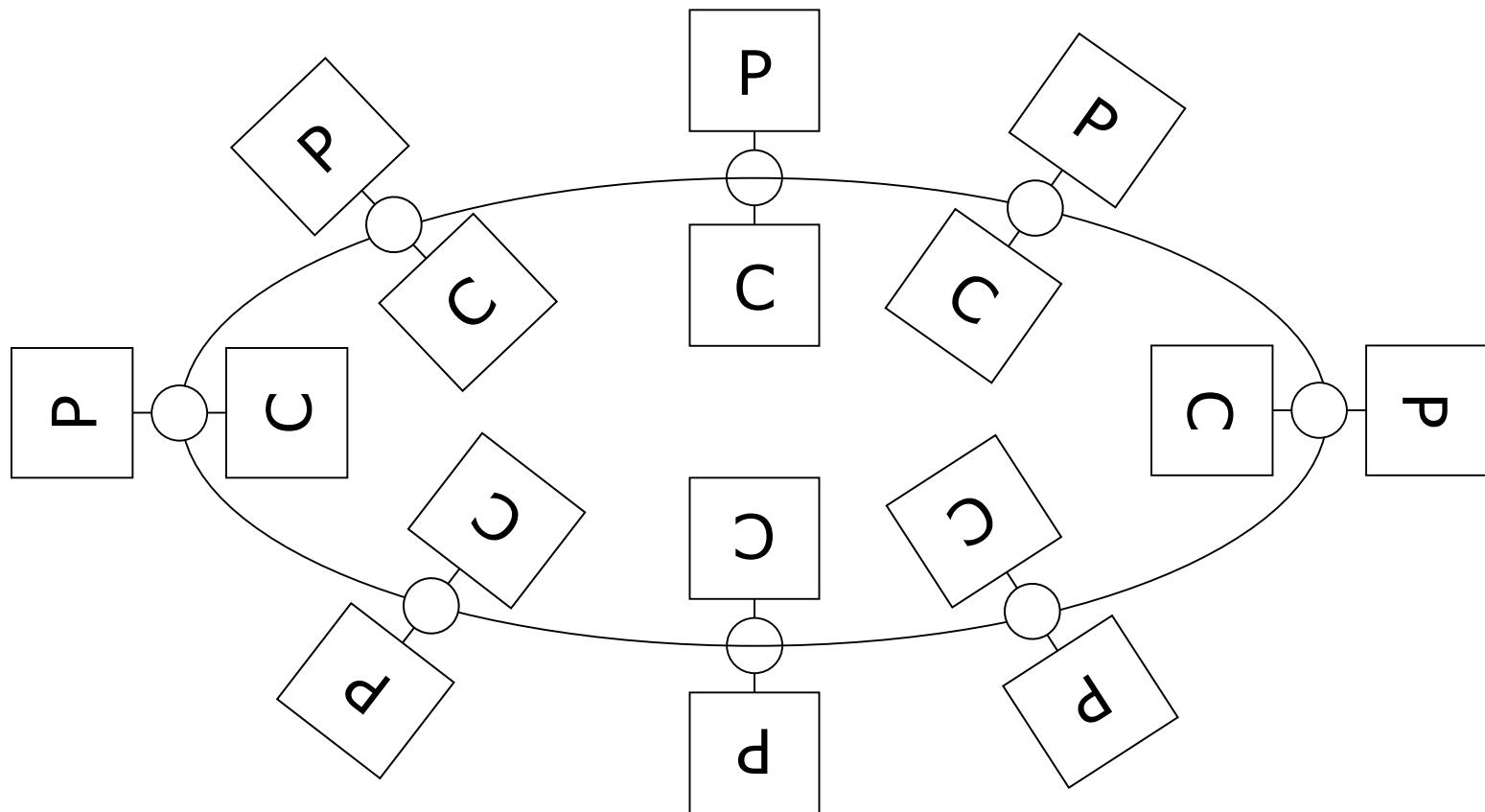
- Advantages?
- Disadvantages?

Interconnection Network Architecture

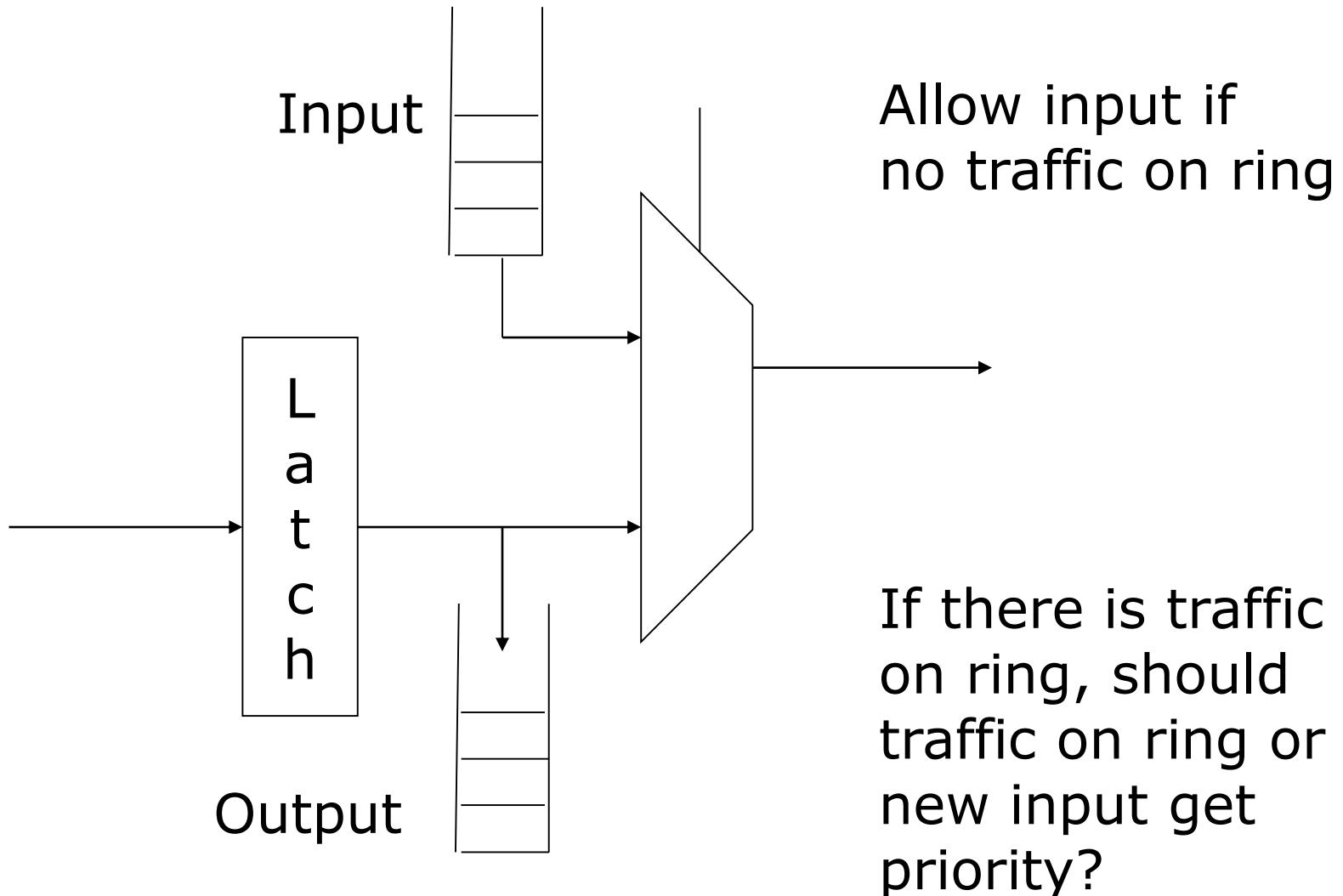
- *Topology*: How to connect the nodes up?
(processors, memories, router line cards, ...)
- *Routing*: Which path should a message take?
- *Flow control*: How is the message actually forwarded from source to destination?
- *Router microarchitecture*: How to build the routers?
- *Link microarchitecture*: How to build the links?

Router Microarchitecture

Ring-based Interconnect



Ring Stop



Ring Flow Control: Priorities



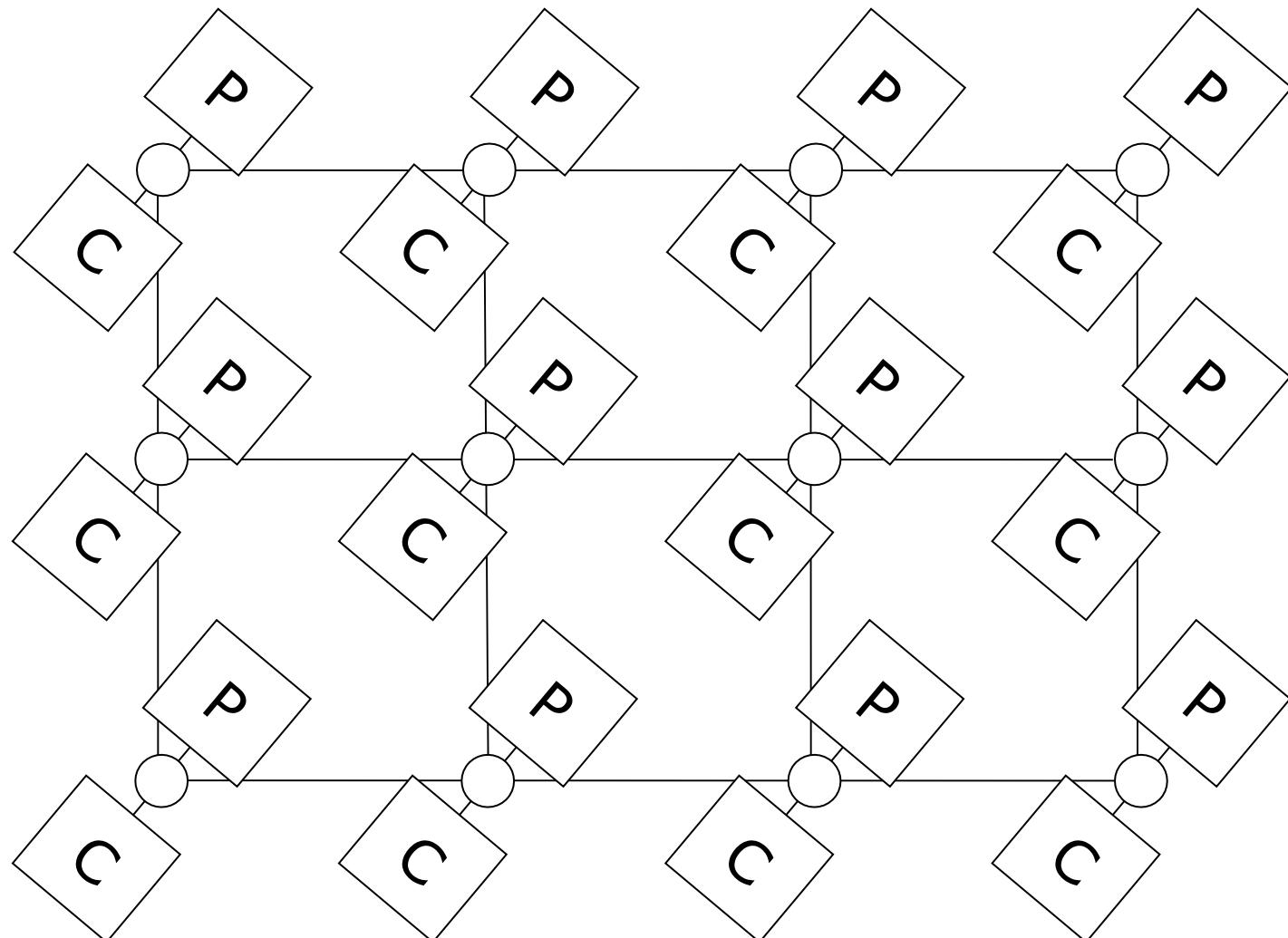
Rotary Rule – traffic in ring has priority

Ring Flow Control: Bounces

- What if traffic on the ring cannot get delivered, e.g., if output FIFO is full?
- One alternative: Continue on ring (bounce)
- *What are the consequences of such bounces?*

General Interconnect

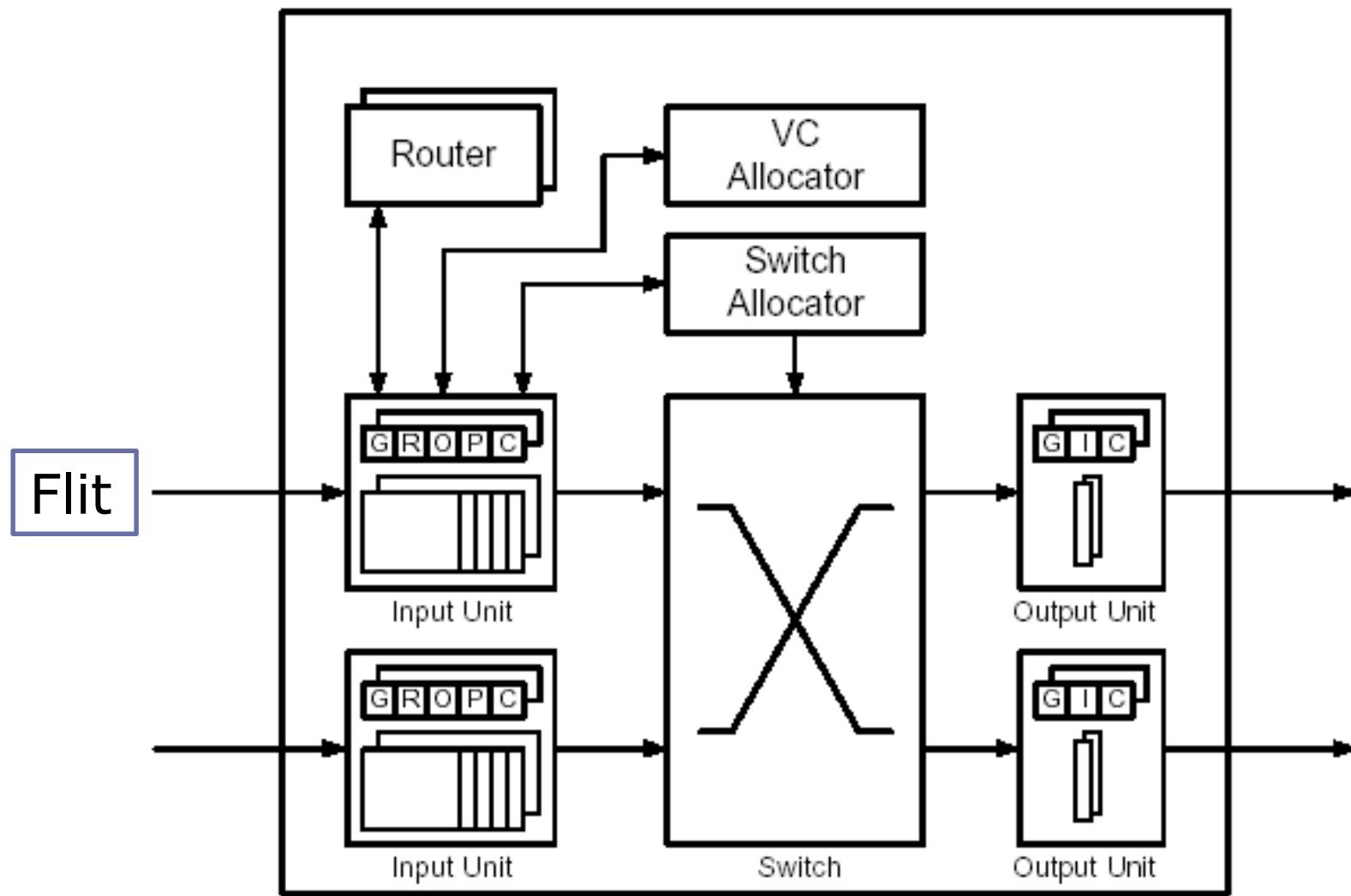
Tilera, Knights Landing...



What's In A Router?

- It's a system as well
 - Logic – State machines, Arbiters, Allocators
 - Control data movement through router
 - Idle, Routing, Waiting for resources, Active
 - Memory – Buffers
 - Store flits before forwarding them
 - SRAMs, registers, processor memory
 - Communication – Switches
 - Transfer flits from input to output ports
 - Crossbars, multiple crossbars, fully-connected, bus

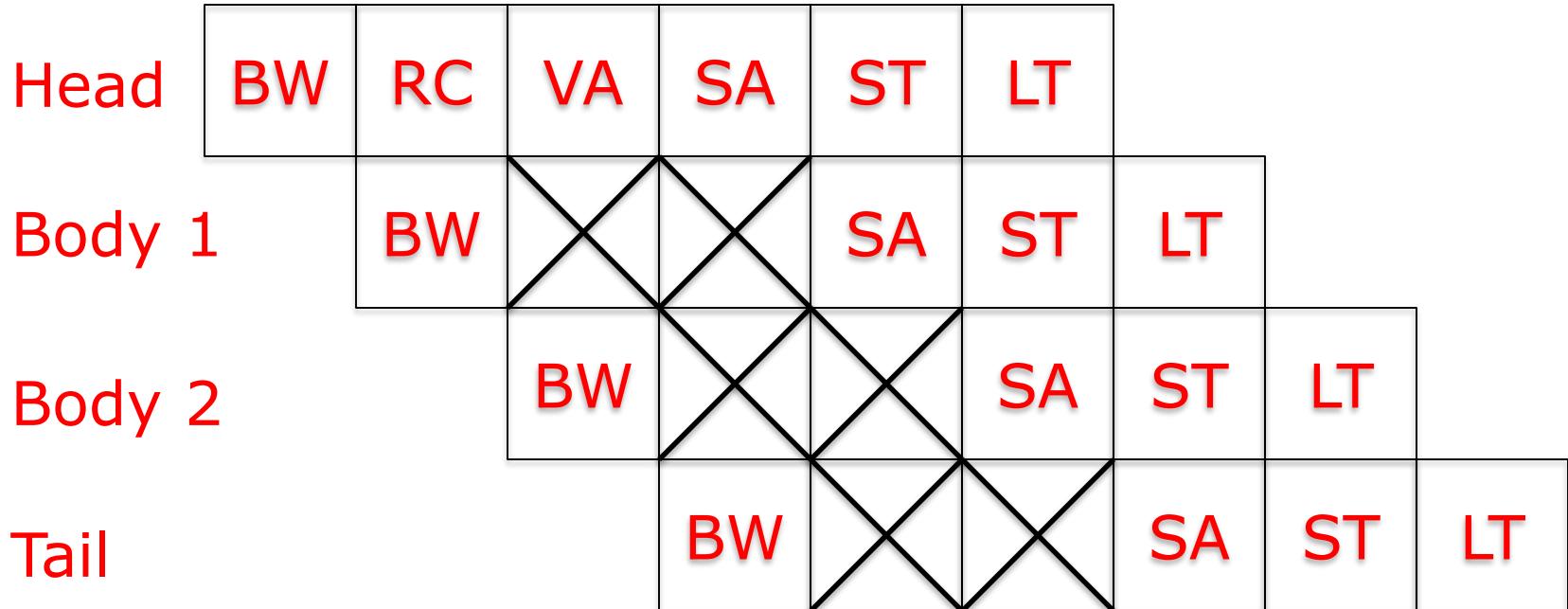
Virtual-channel Router



Router Pipeline vs. Processor Pipeline

- Logical stages:
 - BW
 - RC
 - VA
 - SA
 - BR
 - ST
 - LT
 - Different flits go through different stages
 - Different routers have different variants
 - E.g. speculation, lookaheads, bypassing
 - Different implementations of each pipeline stage
-
- Logical stages:
 - IF
 - ID
 - EX
 - MEM
 - WB
 - Different instructions go through different stages
 - Different processors have different variants
 - E.g. speculation, ISA
 - Different implementations of each pipeline stage

Baseline Router Pipeline

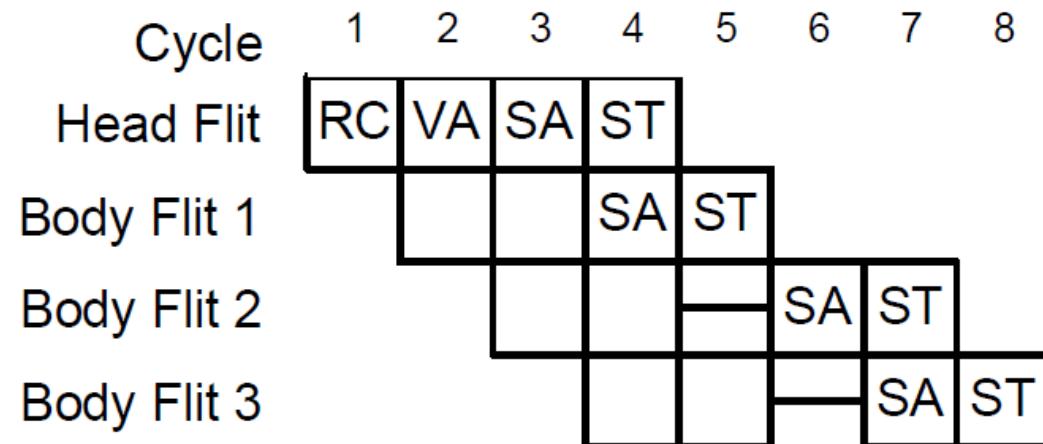
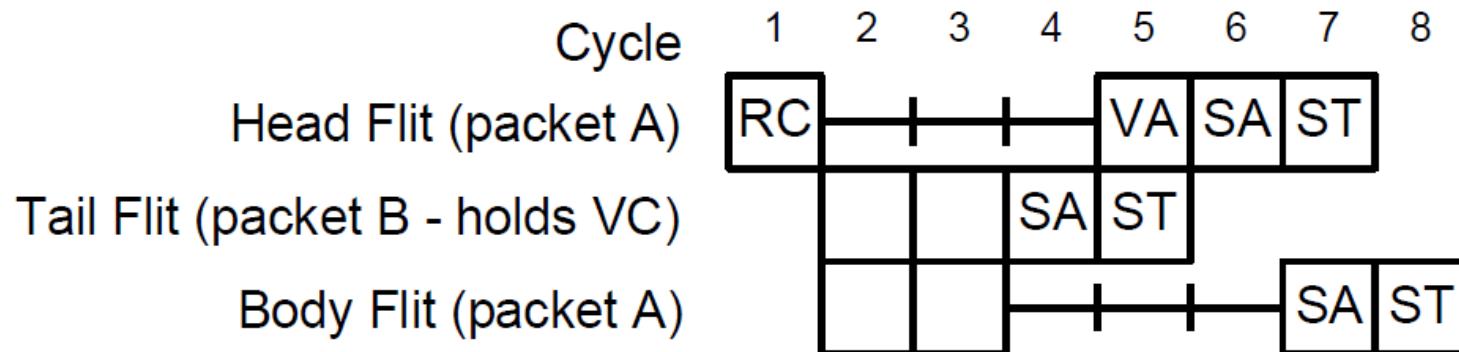


- Route computation performed once per packet
- Virtual channel allocated once per packet
- Body and tail flits inherit this info from head flit

Allocators In Routers

- VC Allocator
 - Input VCs requesting for a range of output VCs
 - Example: A packet of VC0 arrives at East input port. It's destined for west output port, and would like to get any of the VCs of that output port.
- Switch Allocator
 - Input VCs of an input port request for different output ports (e.g., One's going North, another's going West)
- “Greedy” algorithms used for efficiency
- What happens if allocation fails on a given cycle?

VC & Switch Allocation Stalls



Pipeline Optimizations: Lookahead Routing [Galles, SGI Spider Chip]

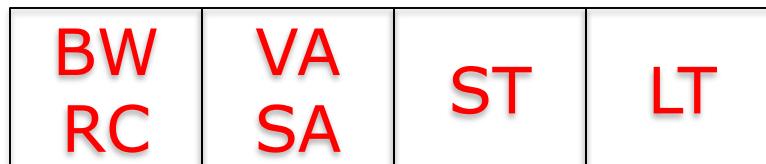
- At current router, perform route computation for next router



- Head flit already carries output port for next router
 - RC just has to read output → fast, can be overlapped with BW
 - Precomputing route allows flits to compete for VCs immediately after BW
 - Routing computation for the next hop (NRC) can be computed in parallel with VA
-
- Or simplify RC (e.g., X-Y routing is very fast)

Pipeline Optimizations: Speculative Switch Allocation [Peh&Dally, 2001]

- Assume that Virtual Channel Allocation stage will be successful
 - Valid under low to moderate loads
- If both successful, VA and SA are done in parallel



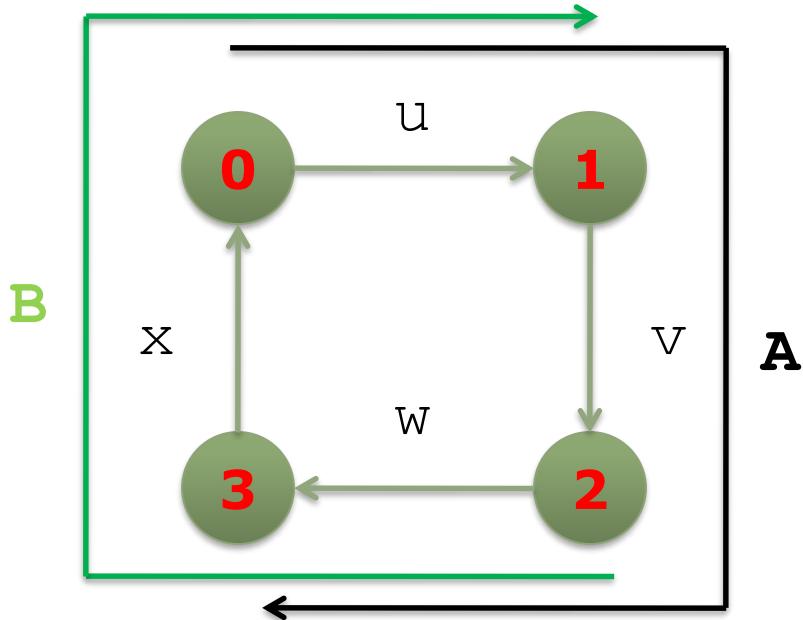
- If VA unsuccessful (no virtual channel returned)
 - Must repeat VA/SA in next cycle
- Prioritize non-speculative requests

Routing

Properties of Routing Algorithms

- Deterministic/Oblivious
 - route determined by (source, dest),
 - not intermediate state (i.e. traffic)
- Adaptive
 - route influenced by traffic along the way
- Minimal
 - only selects shortest paths
- Deadlock-free
 - no traffic pattern can lead to a situation where no packets move forward

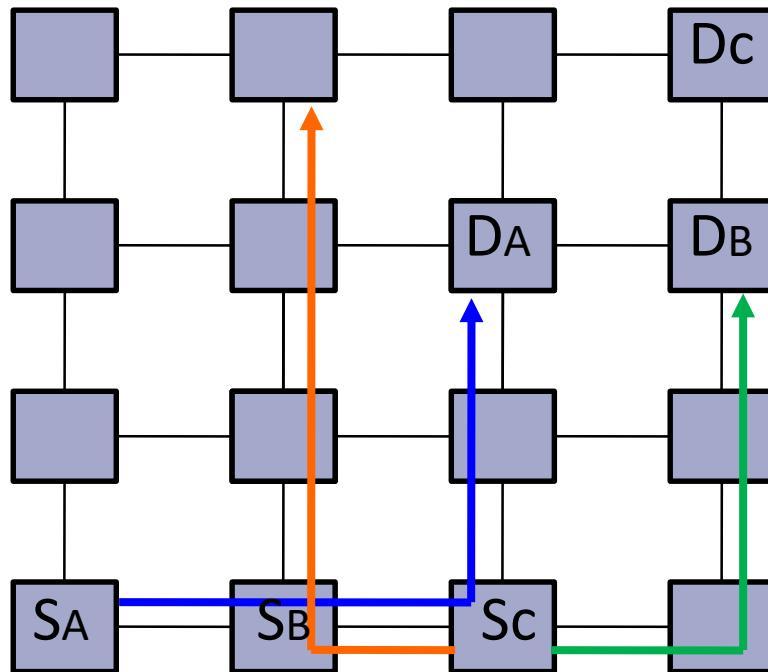
Network Deadlock



- Flow A holds u and v but cannot make progress until it acquires channel w
- Flow B holds channels w and x but cannot make progress until it acquires channel u

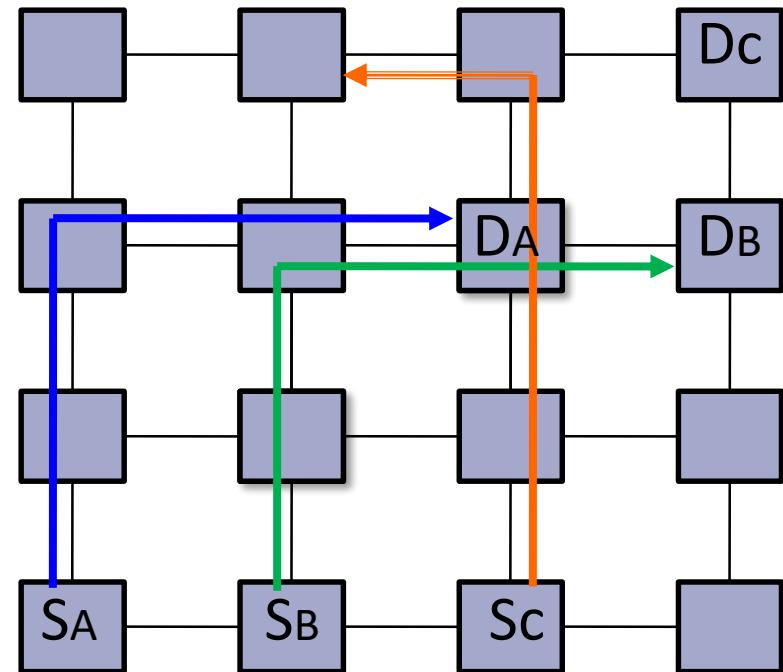
Dimension-Order Routing

XY-order



Uses 2 out of 4 turns

YX-order

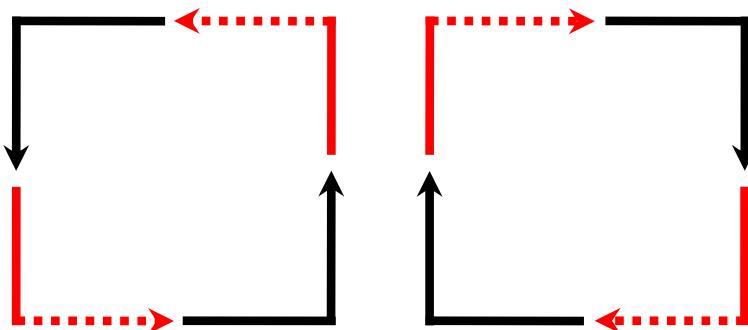


Uses 2 out of 4 turns

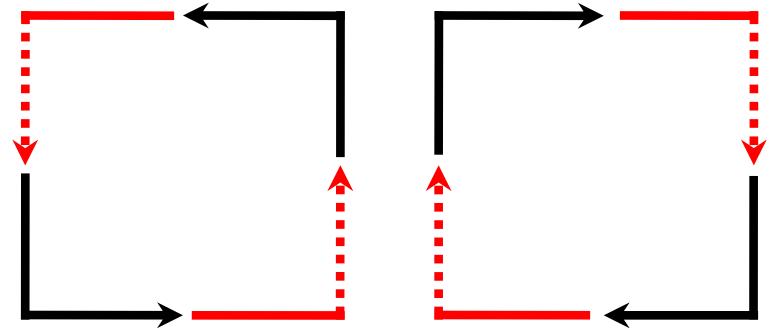
XY is deadlock free, YX is deadlock free, what about XY+YX?

DOF – Turns allowed

- One way of looking at whether a routing algorithm is deadlock free is to look at the turns allowed.
- Deadlocks may occur if turns can form a cycle



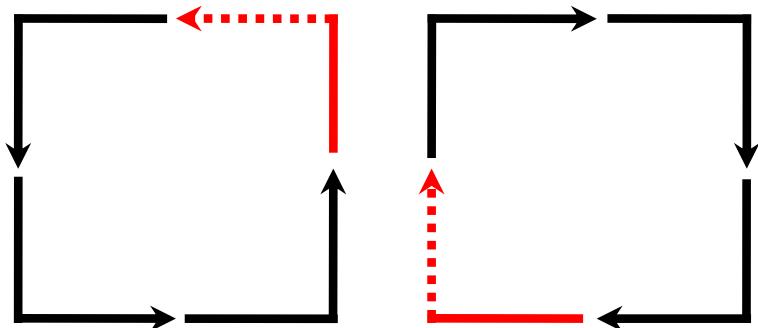
XY Model



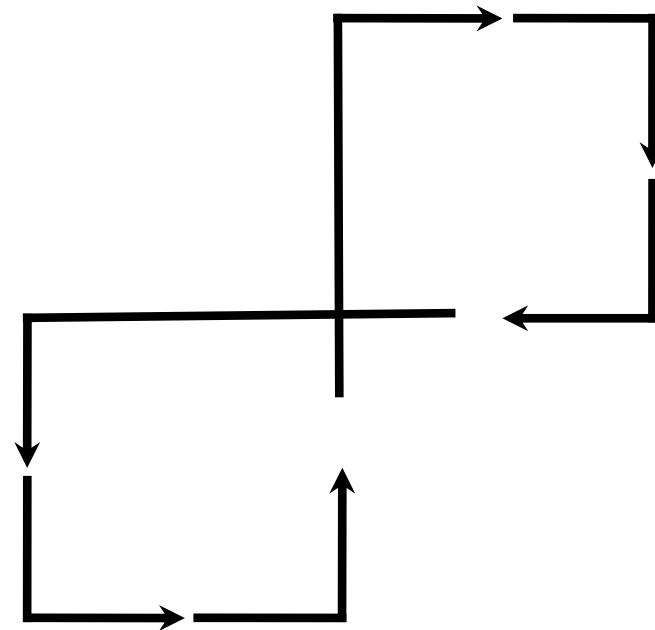
YX Model

Allowing more turns

- Allowing more turns may allow adaptive routing, but also **deadlock**

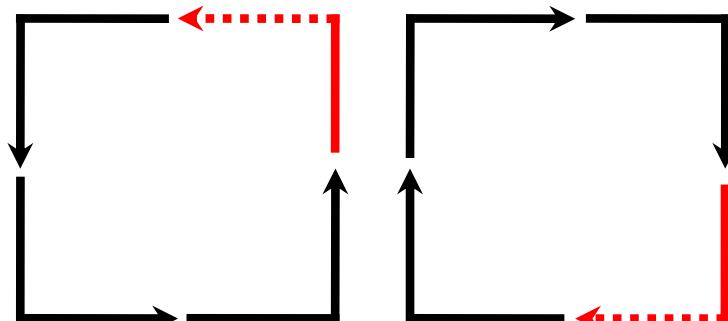


Six turn model

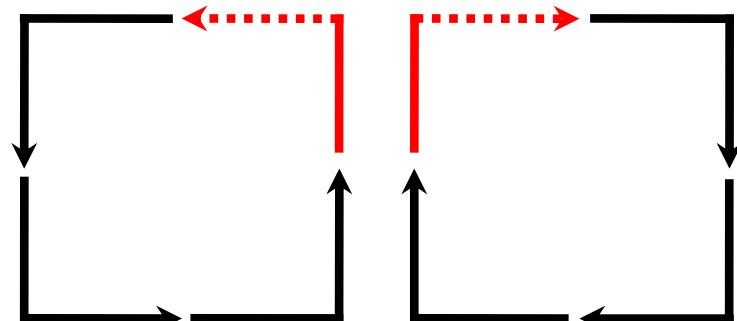


Turn Model [Glass and Ni, 1994]

- A systematic way of generating deadlock-free routes with small number of prohibited turns
- Deadlock-free if routes conform to at least ONE of the turn models (acyclic channel dependence graph)



West-First Turn Model



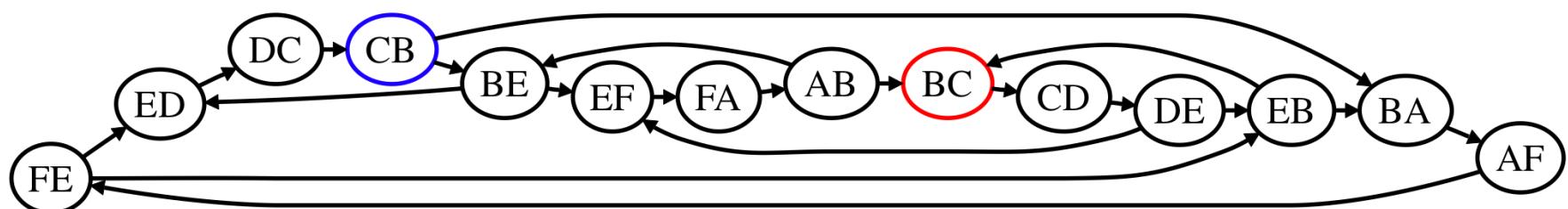
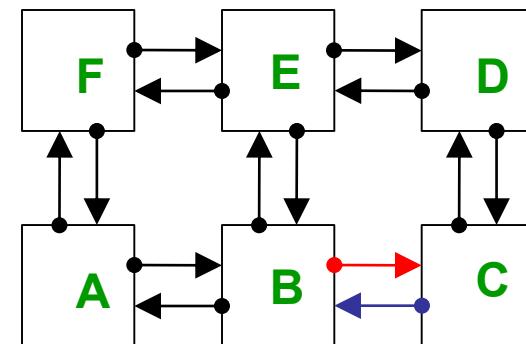
North-Last Turn Model

2-D Mesh and CDG

Can create a channel dependency graph (CDG) of the network.

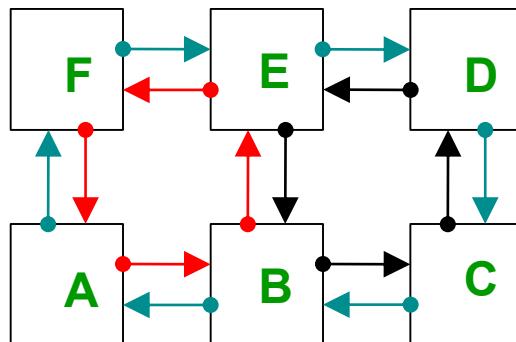
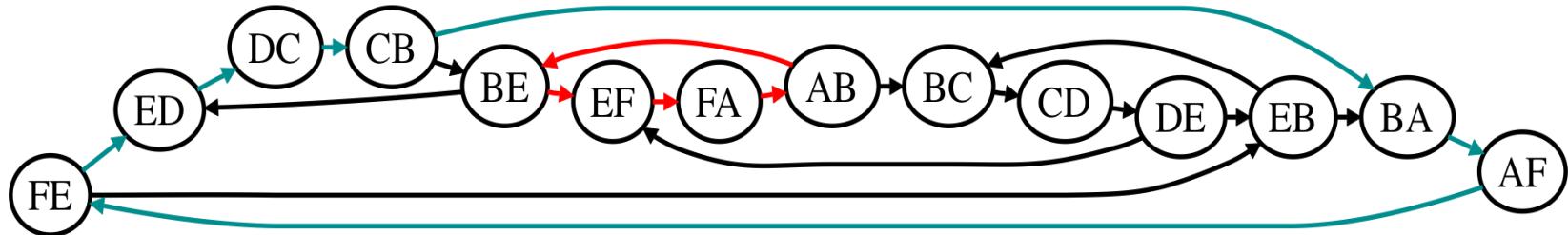
Vertices in the
CDG represent
network *links*

Disallowing
 180° turns, e.g.,
 $AB \rightarrow BA$



Cycles in CDG

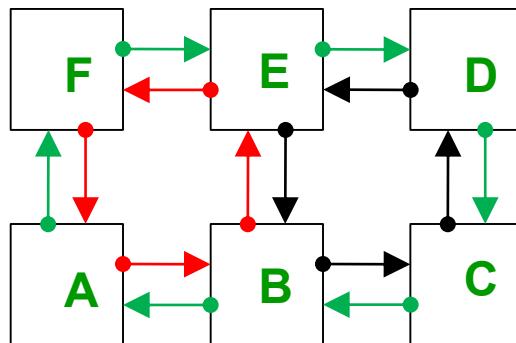
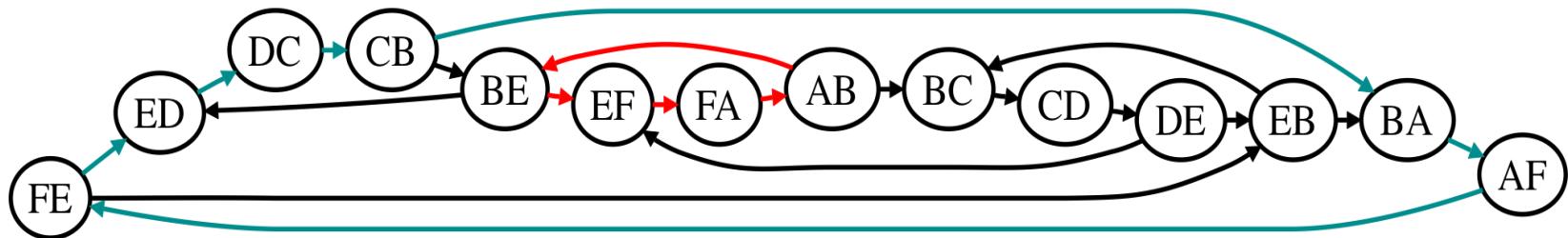
The channel dependency graph D derived from the network topology may contain many cycles



Flow routed through links AB, BE, EF
Flow routed through links EF, FA, AB
Deadlock!

Key Insight

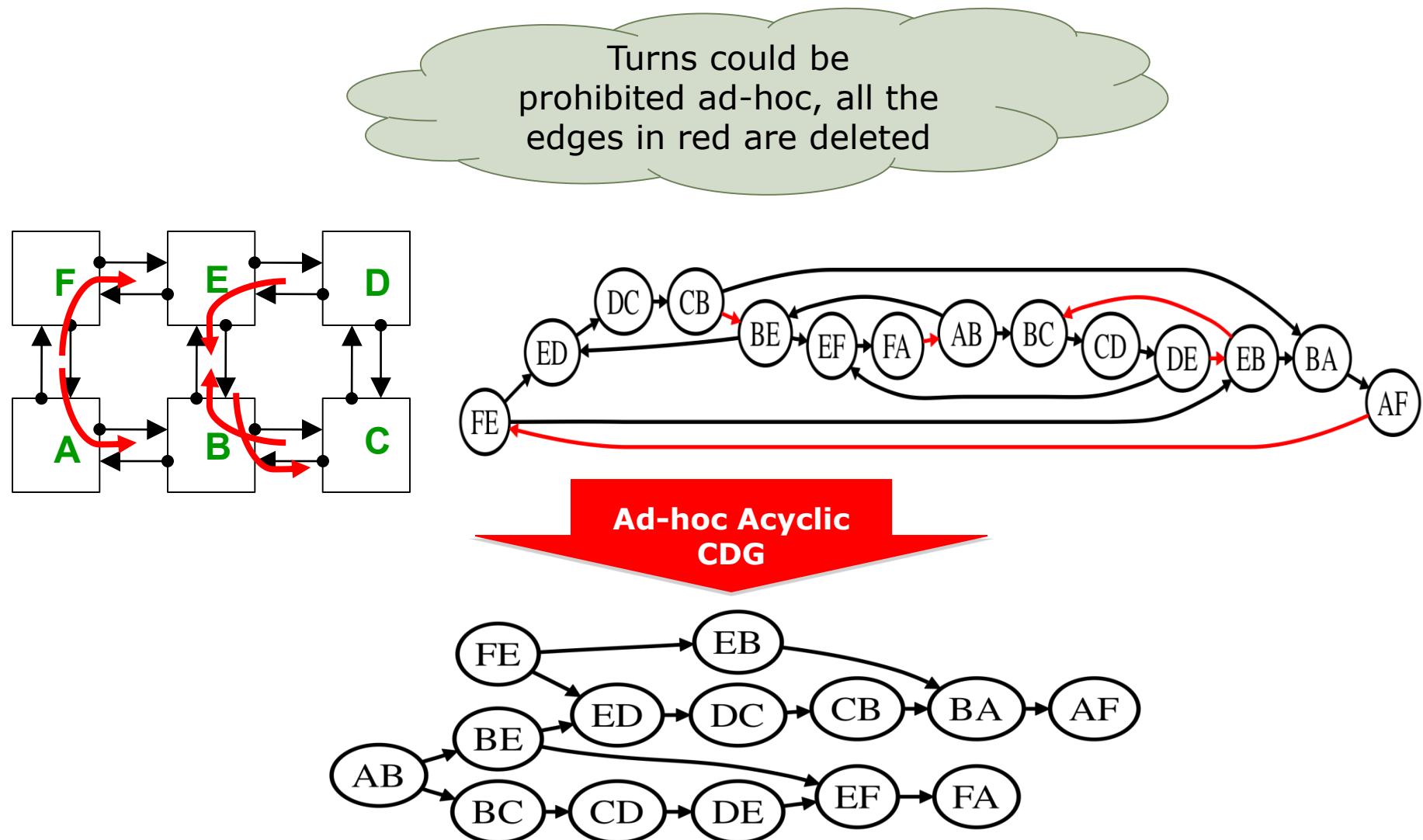
If routes of flows conform to acyclic CDG, then there will be no possibility of deadlock!



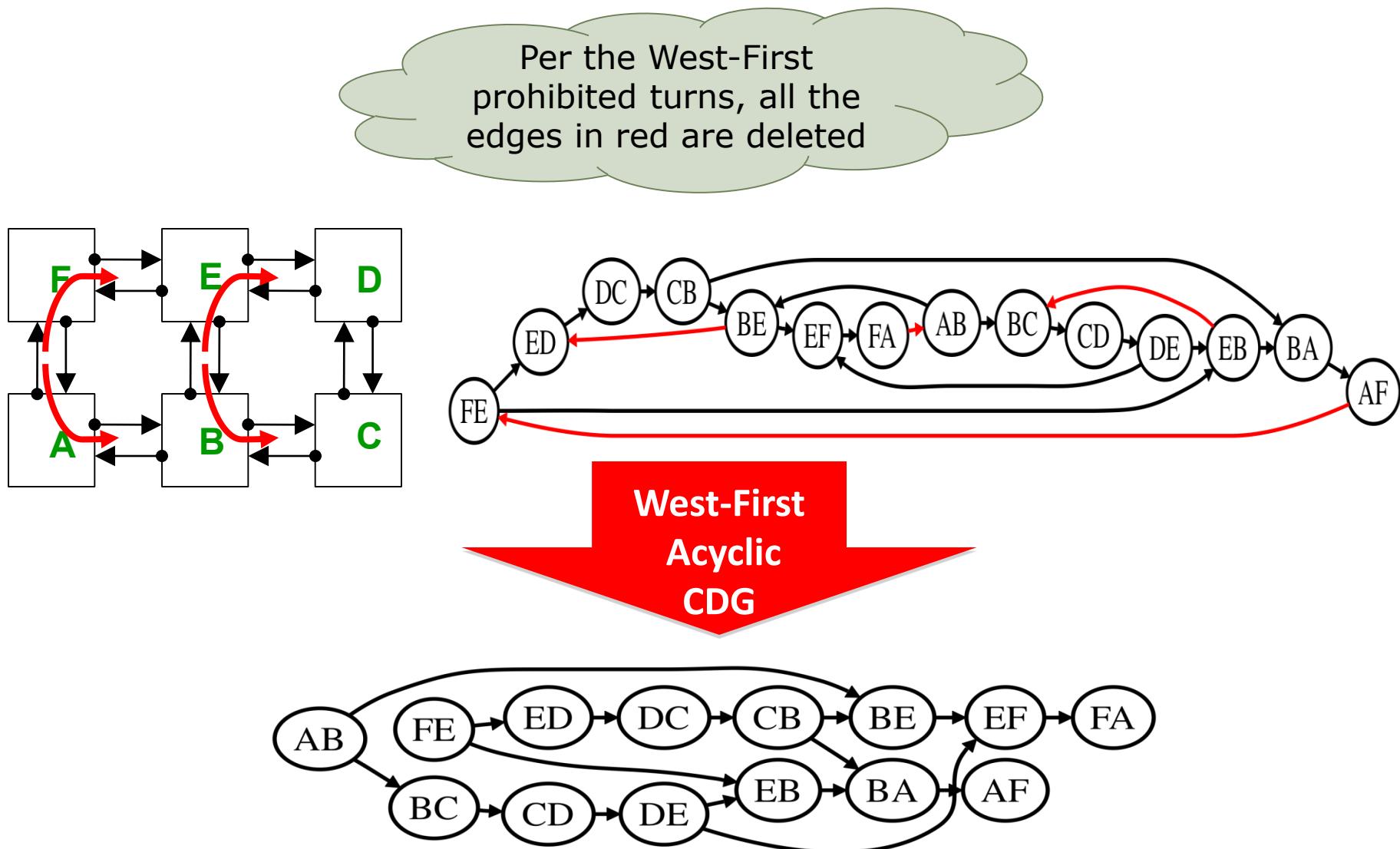
Disallow/Delete certain edges in CDG

Edges in CDG correspond to turns in network!

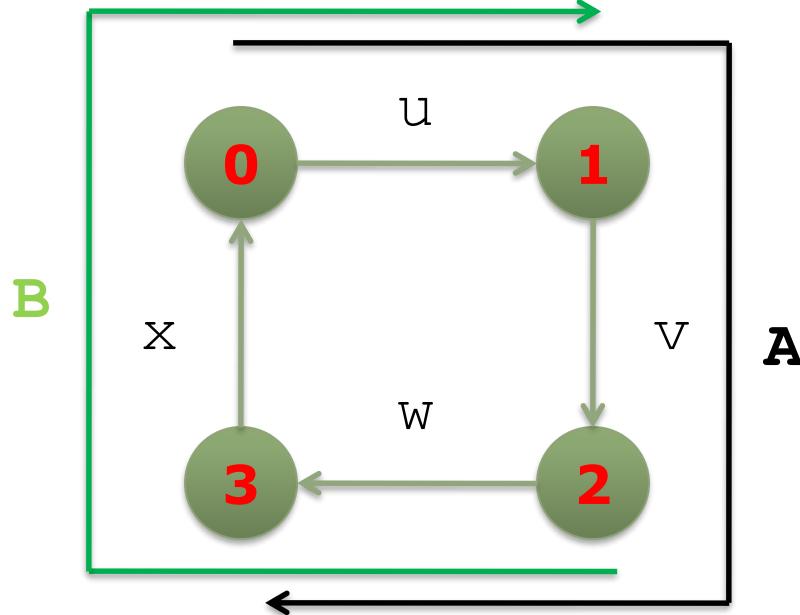
Acyclic CDG → Deadlock-free routes



West-first → Deadlock-free routes



Resource Conflicts → Deadlock

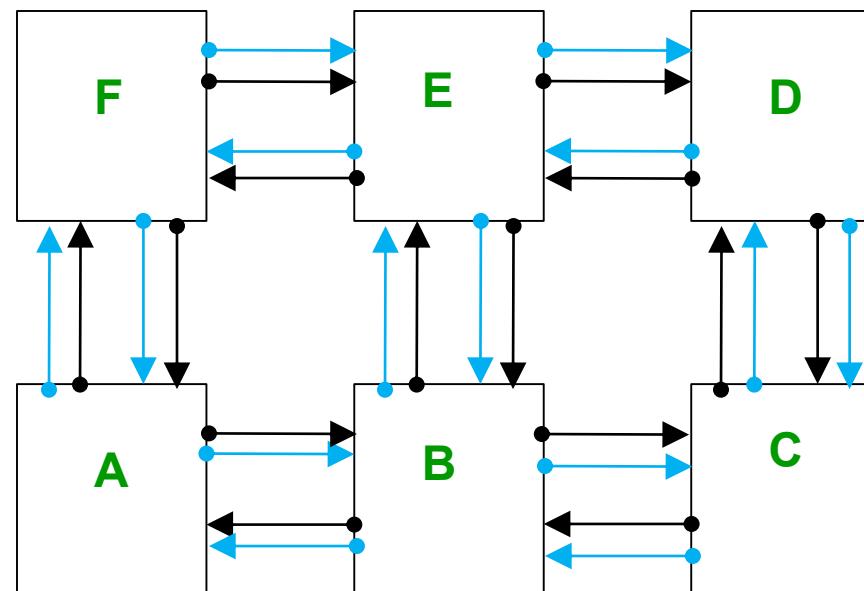


Routing deadlocks in wormhole routing result from Structural hazard at router resources, e.g., buffers.

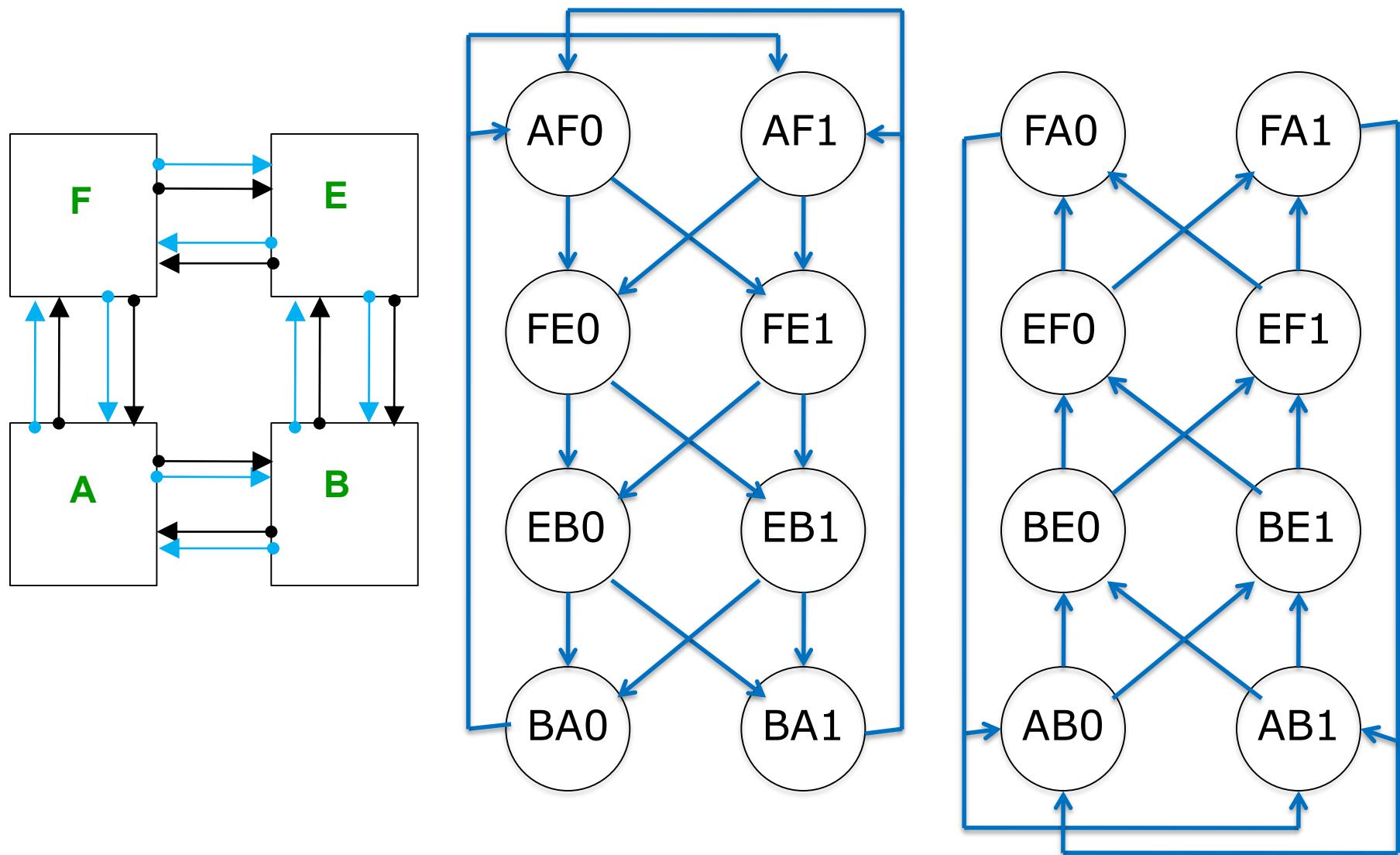
How can structural hazards be avoided?

Virtual Channels

- Virtual channels can be used to avoid deadlock by restricting VC allocation

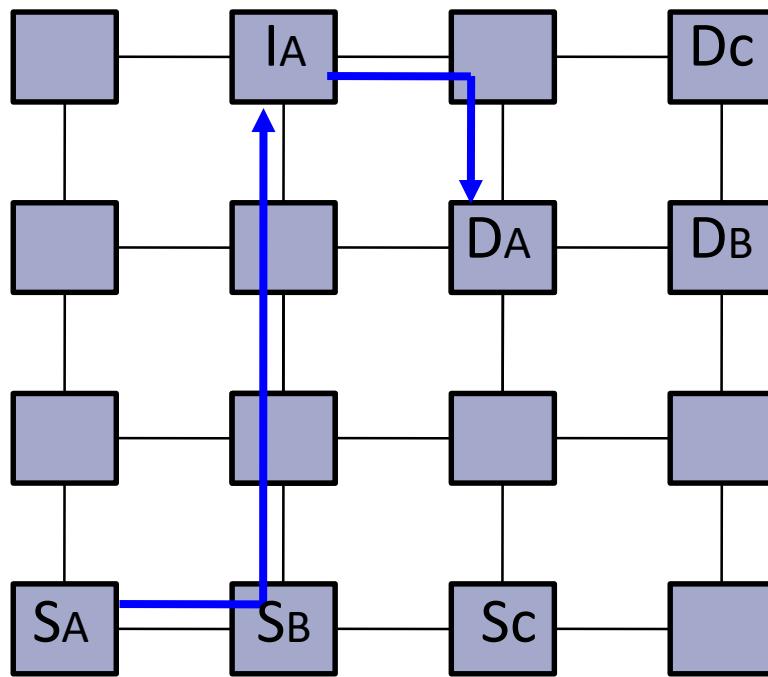


CDG and Virtual Channels



Randomized Routing: Valiant

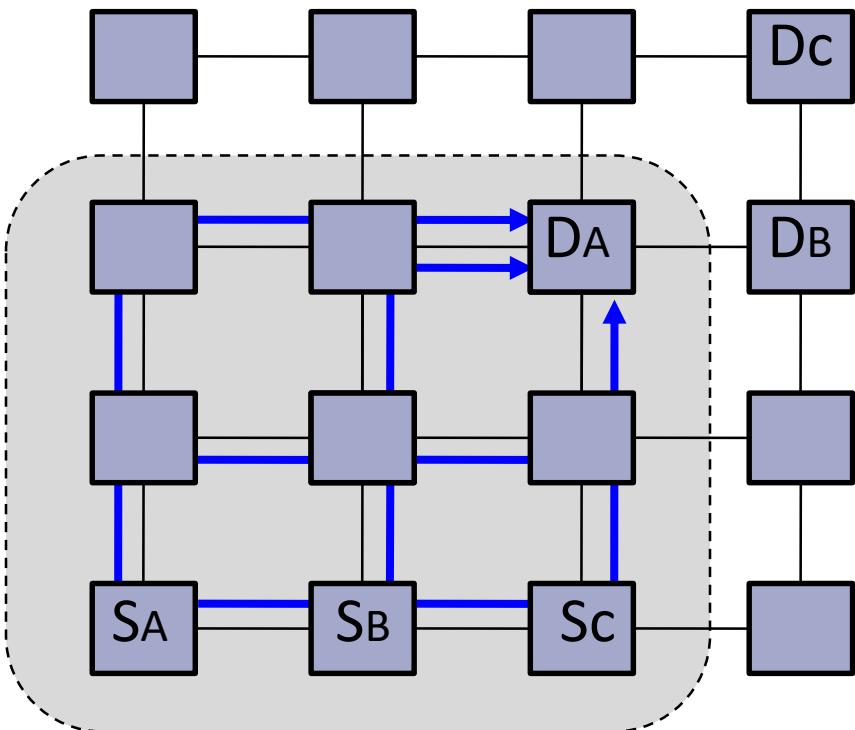
- Route each packet through a randomly chosen intermediate node



A packet, going from node SA to node DA, is first routed from SA to a randomly chosen intermediate node IA, before going from IA to final destination DA.

It helps load-balance the network and has a good worst-case performance at the expense of locality.

ROMM: Randomized, Oblivious Multi-phase Minimal Routing



To retain locality, choose intermediate node in the minimal quadrant

Equivalent to randomly selecting among the various minimal paths from source to destination

Thank you!

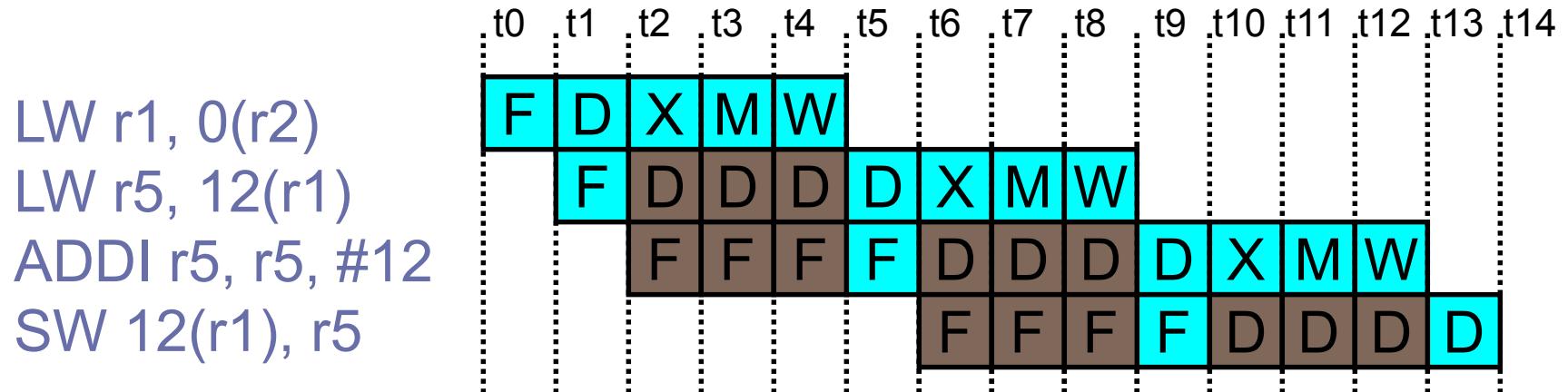
Next Lecture: Multithreading

Multithreading Architectures

Daniel Sanchez

Computer Science & Artificial Intelligence Lab
M.I.T.

Pipeline Hazards



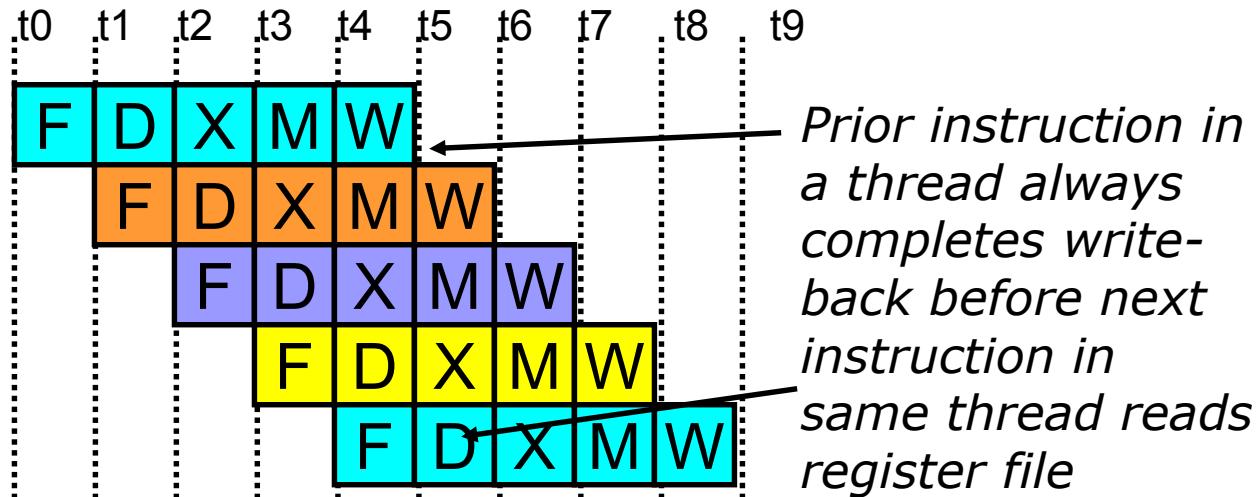
- Each instruction may depend on the previous one
 - What can be done to cope with this?
- Even bypassing, speculation and finding something else to do (via O-O-O) does not eliminate all delays

Multithreading

How can we guarantee no dependencies between instructions in a pipeline?

Interleave 4 threads, T1-T4, on non-bypassed 5-stage pipe

- T1: LW r1, 0(r2)
- T2: ADD r7, r1, r4
- T3: XORI r5, r4, #12
- T4: SW 0(r7), r5
- T1: LW r5, 12(r1)

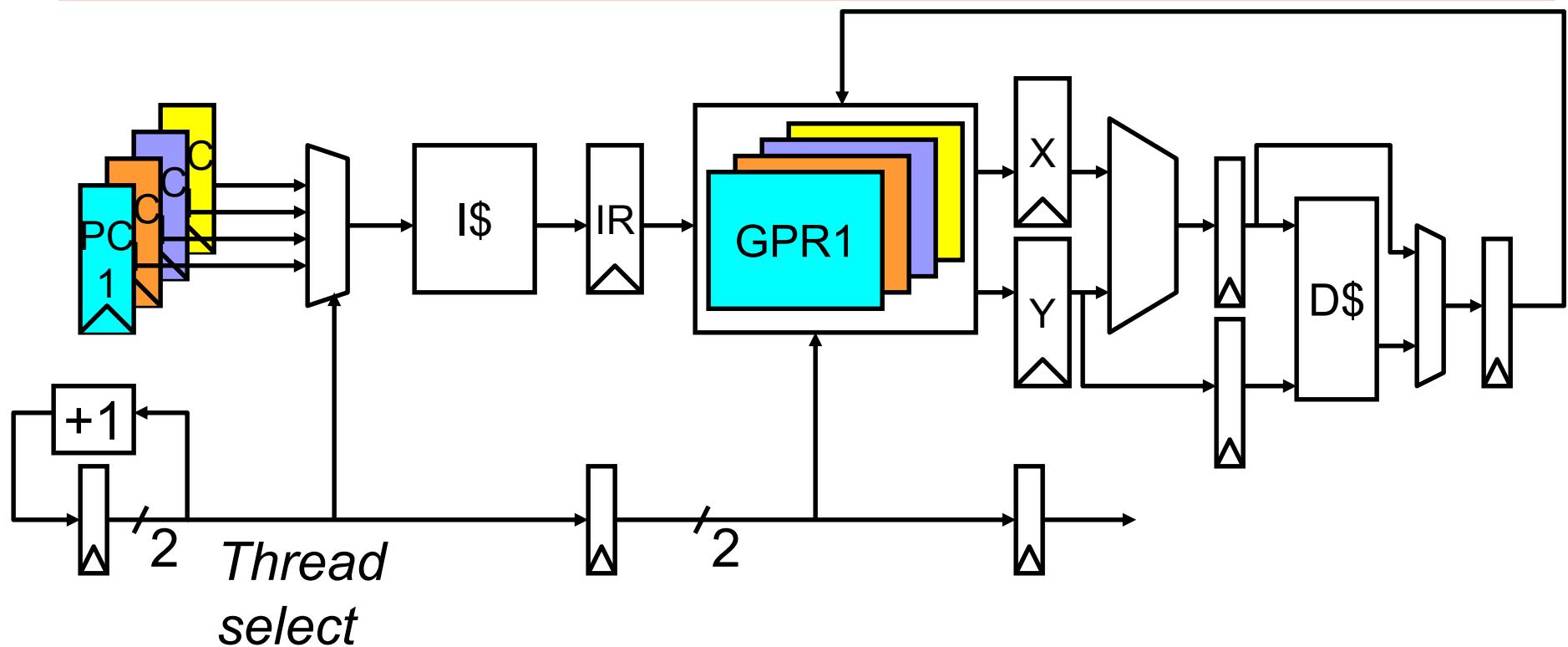


CDC 6600 Peripheral Processors (Cray, 1964)



- First commercial multithreaded hardware
- 10 “virtual” I/O processors
- Fixed interleave on simple pipeline
- Pipeline has 100ns cycle time
- Each virtual processor executes one instruction every 1000ns

Simple Multithreaded Pipeline



Have to carry thread select down pipeline to ensure correct state bits read/written at each pipe stage

Multithreading Costs

- Each thread needs its own user architectural state
 - PC
 - GPRs (CDC6600 PPUs – accumulator-based architecture)
- Also, needs its own system architectural state
 - Virtual memory page table base register
 - Exception handling registers
- *Other costs?*
- Appears to software (including OS) as multiple, albeit slower, CPUs

Thread Scheduling Policies

- Fixed interleave (*CDC 6600 PPUs, 1965*)
 - Each of N threads executes one instruction every N cycles
 - If thread not ready to go in its slot, insert pipeline bubble
- Software-controlled interleave (*TI ASC PPUs, 1971*)
 - OS allocates S pipeline slots among N threads
 - Hardware performs fixed interleave over S slots, executing whichever thread is in that slot
- Hardware-controlled thread scheduling (*HEP, 1982*)
 - Hardware keeps track of which threads are ready to go
 - Picks next thread to execute based on hardware priority scheme

Denelcor HEP

(Burton Smith, 1982)



First commercial machine to use hardware threading in main CPU

- 120 threads per processor
- 10 MHz clock rate
- Up to 8 processors
- Precursor to Tera MTA (Multithreaded Architecture)

Tera MTA (1990-97)

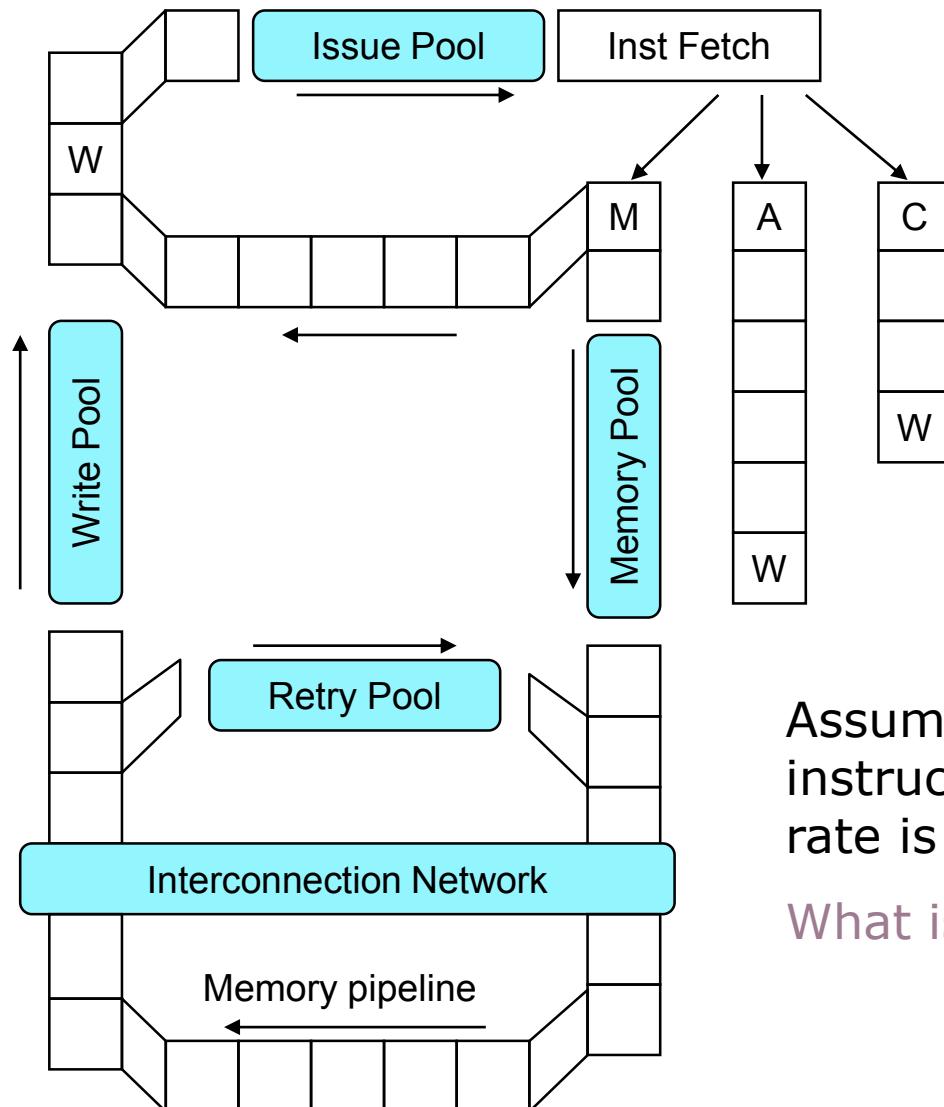


- Up to 256 processors
- Up to 128 active threads per processor
- Processors and memory modules populate a sparse 3D torus interconnection fabric
- Flat, shared main memory
 - No data cache
 - Sustains one main memory access per cycle per processor
- GaAs logic in prototype, 1KW/processor @ 260MHz
 - CMOS version, MTA-2, 50W/processor

MTA Architecture

- Each processor supports 128 active hardware threads
 - $1 \times 128 = 128$ stream status word (SSW) registers,
 - $8 \times 128 = 1024$ branch-target registers,
 - $32 \times 128 = 4096$ general-purpose registers
- Three operations packed into 64-bit instruction (short VLIW)
 - One memory operation,
 - One arithmetic operation, plus
 - One arithmetic or branch operation
- Thread creation and termination instructions
- Explicit 3-bit “lookahead” field in instruction gives number of subsequent instructions (0-7) that are independent of this one
 - Allows fewer threads to fill machine pipeline
 - Used for variable-sized branch delay slots

MTA Pipeline



- Every cycle, one instruction from one active thread is launched into pipeline
- Instruction pipeline is 21 cycles long
- Memory operations incur ~ 150 cycles of latency

Assuming a single thread issues one instruction every 21 cycles, and clock rate is 260 MHz...

What is single thread performance?

Coarse-Grain Multithreading

Tera MTA designed for supercomputing applications with large data sets and low locality

- No data cache
- Many parallel threads needed to hide large memory latency

Other applications are more cache friendly

- Few pipeline bubbles when cache getting hits
- Just add a few threads to hide occasional cache miss latencies
- Swap threads on cache misses

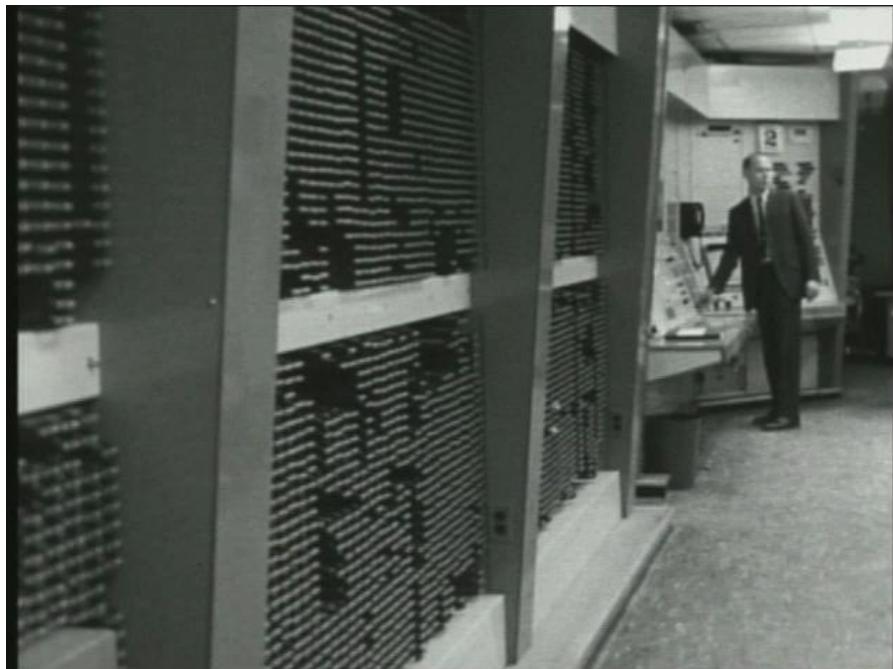
Multithreading Design Choices

- Fine-grained multithreading
 - Context switch among threads every cycle
- Coarse-grained multithreading
 - Context switch among threads every few cycles, e.g., on:
 - Function unit data hazard,
 - L1 miss,
 - L2 miss...
- Why choose one style over another?
- Choice depends on
 - Context-switch overhead
 - Number of threads supported (due to per-thread state)
 - Expected application-level parallelism...

TX-2: Multi-sequence computer (Wes Clark, Lincoln Labs, 1956)

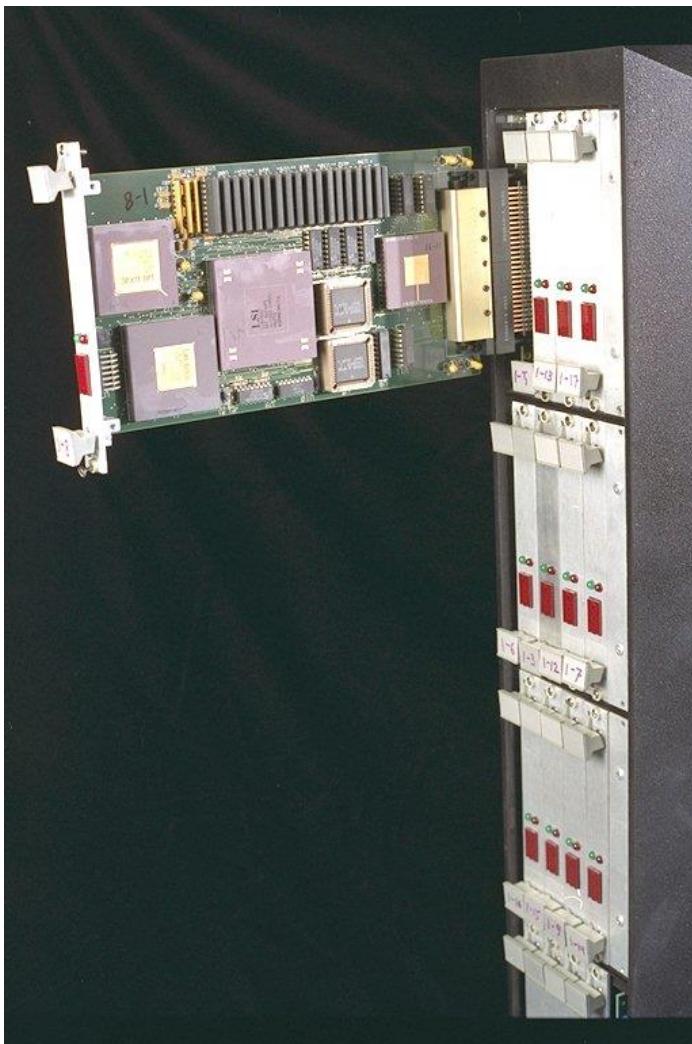
32 Instruction sequences (threads) with

- a fixed priority order among the threads, and
- executes many instructions in a thread - switches mediated by:
 - Instruction “break”/“dismiss” bits
 - Attention request from I/O



- Start-Over
- In-out alarms
- Arithmetic alarms (overflows, etc.)
- Magnetic tape units (multiple)
- High-speed printer
- Analog-to-digital converter
- Paper tape readers (multiple)
- Light pen
- Display (multiple)
- Memory Test Computer
- TX-O
- Digital-to-analog converter
- Paper tape punch
- Flexowriters (multiple)
- *Main sequences (three)

MIT Alewife (1990)

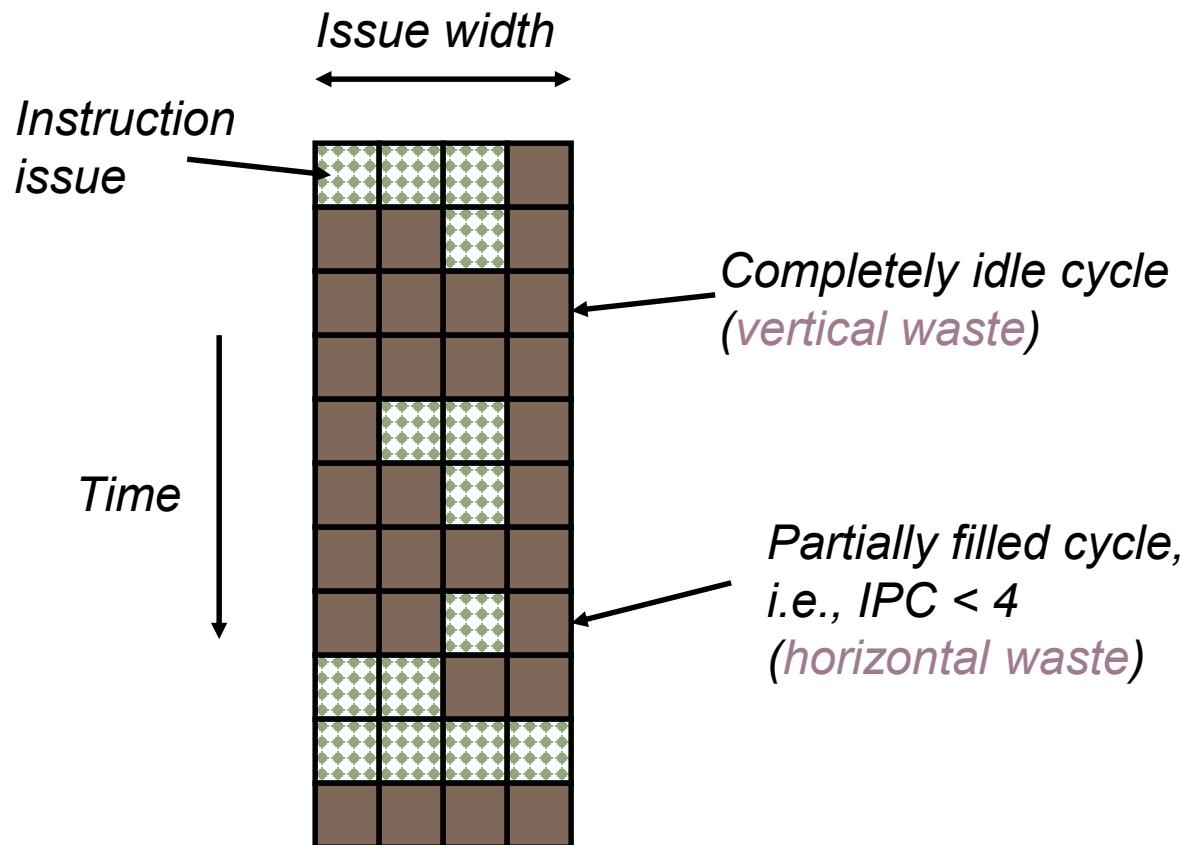


- Modified SPARC chips
 - Register windows hold different thread contexts
- Up to four threads per node
- Thread switch on local cache miss

IBM PowerPC RS64-IV (2000)

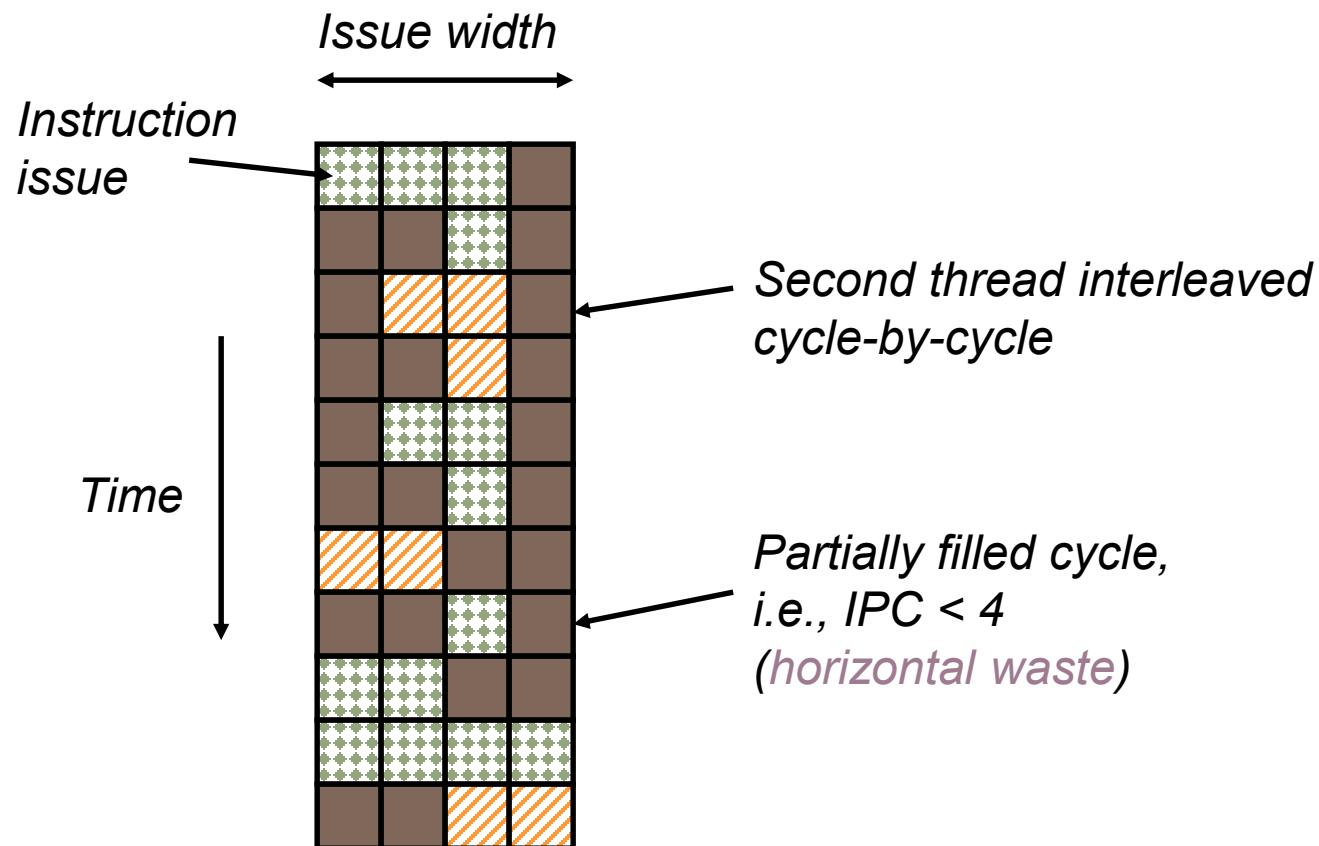
- Commercial coarse-grain multithreading CPU
- Based on PowerPC with quad-issue in-order five-stage pipeline
- Each physical CPU supports two virtual CPUs
- On L2 cache miss, pipeline is flushed and execution switches to second thread
 - Short pipeline minimizes flush penalty (4 cycles), small compared to memory access latency
 - Flush pipeline to simplify exception handling

Superscalar Machine Efficiency



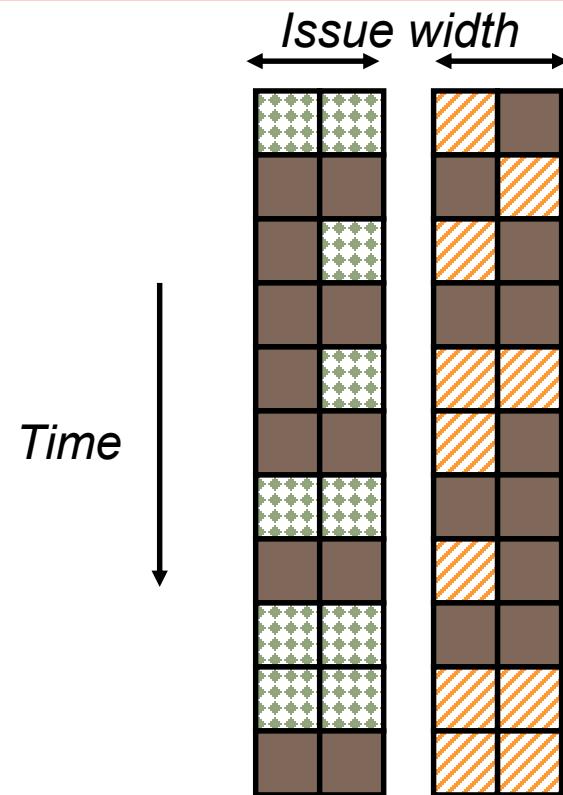
- *Why horizontal waste?*
- *Why vertical waste?*

Vertical Multithreading



- What is the effect of cycle-by-cycle interleaving?

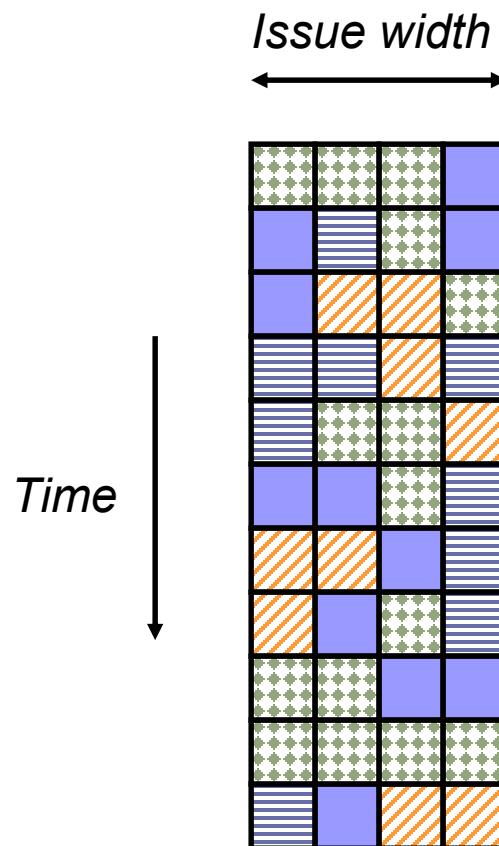
Chip Multiprocessing



- What is the effect of splitting into multiple processors?

Ideal Superscalar Multithreading

[Tullsen, Eggers, Levy, UW, 1995]



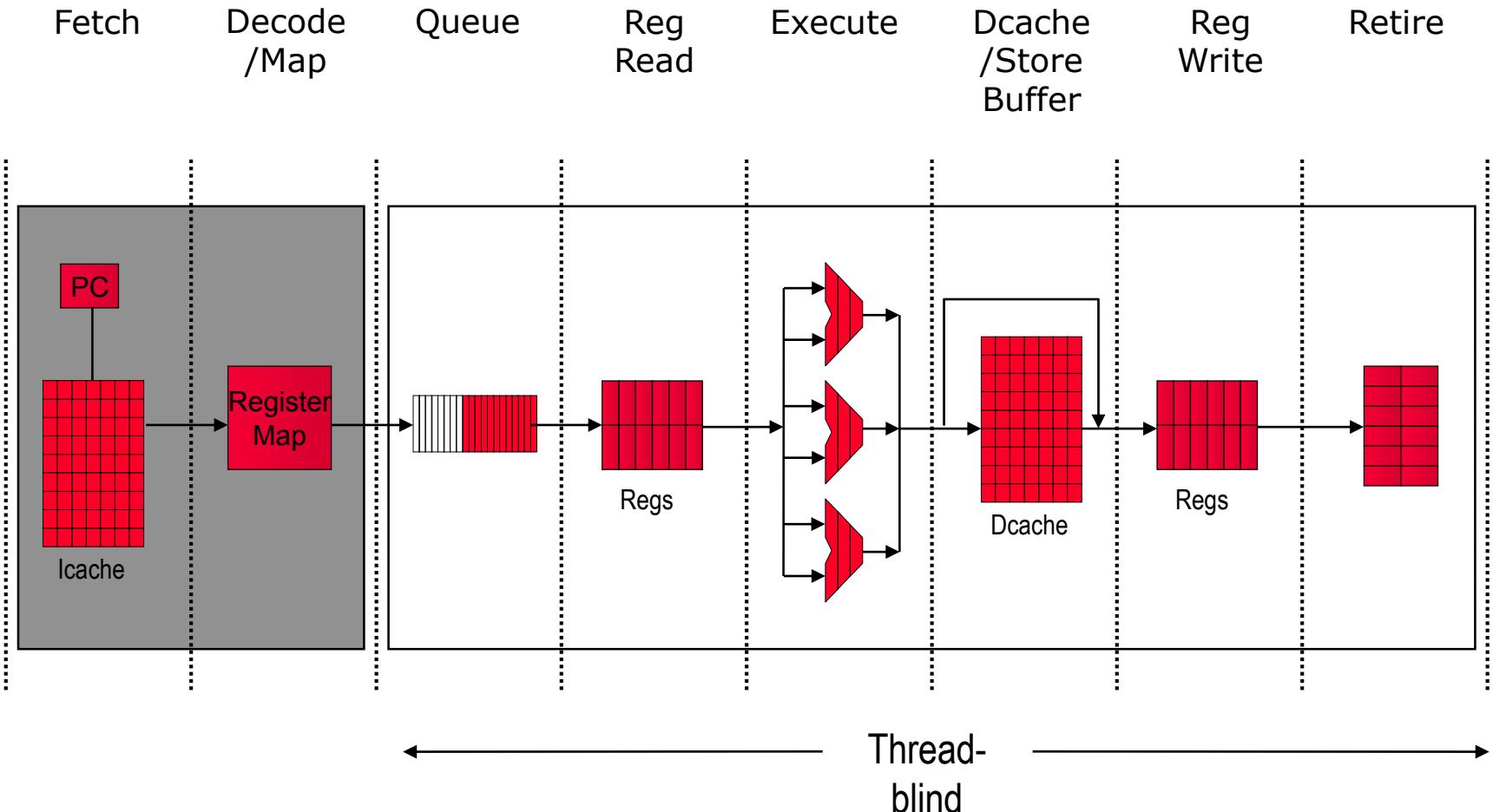
- Interleave multiple threads to multiple issue slots with no restrictions

O-o-O Simultaneous Multithreading

[Tullsen, Eggers, Emer, Levy, Stamm, Lo, DEC/UW, 1996]

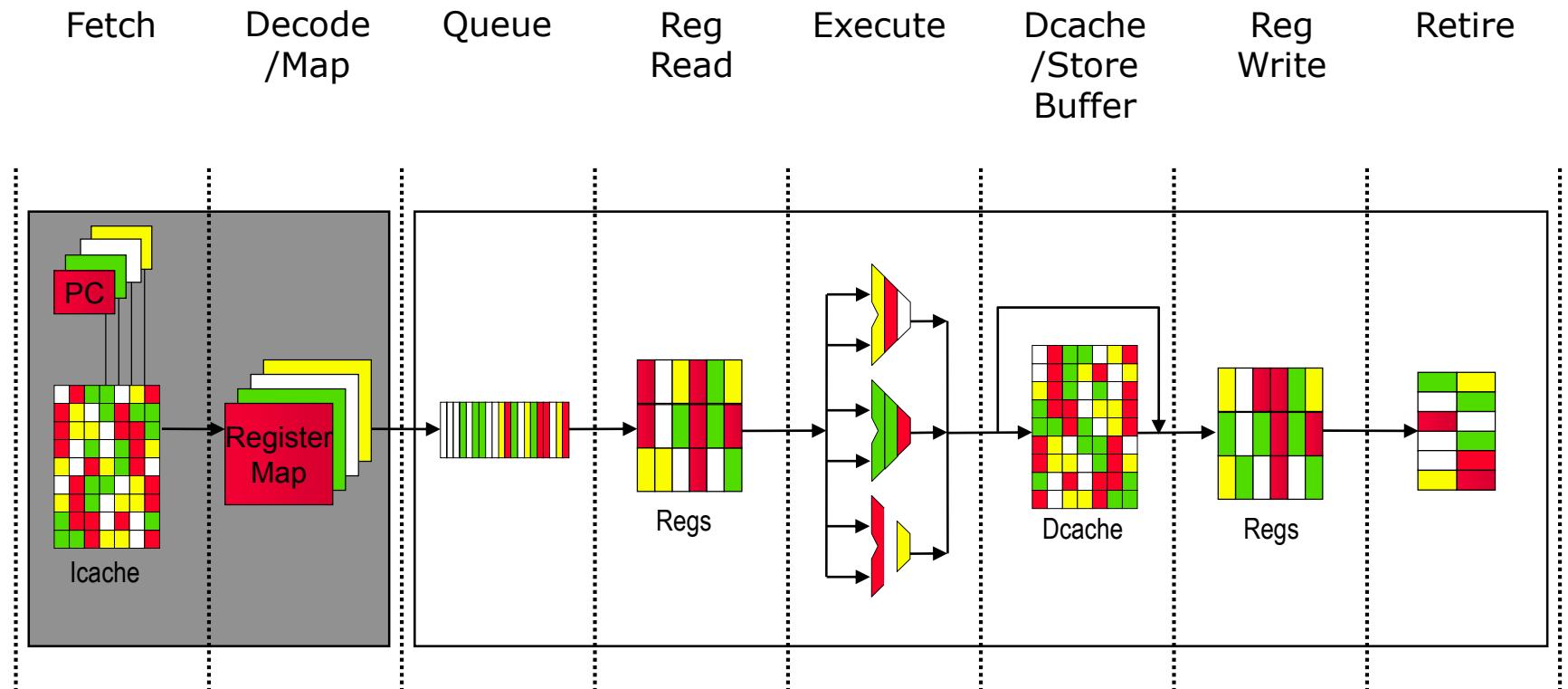
- Add multiple contexts and fetch engines and allow instructions fetched from different threads to issue simultaneously
- Utilize wide out-of-order superscalar processor issue queue to find instructions to issue from multiple threads
- OOO instruction window already has most of the circuitry required to schedule from multiple threads
- Any single thread can utilize whole machine

Basic Out-of-order Pipeline



[EV8 – Microprocessor Forum, Oct 1999]

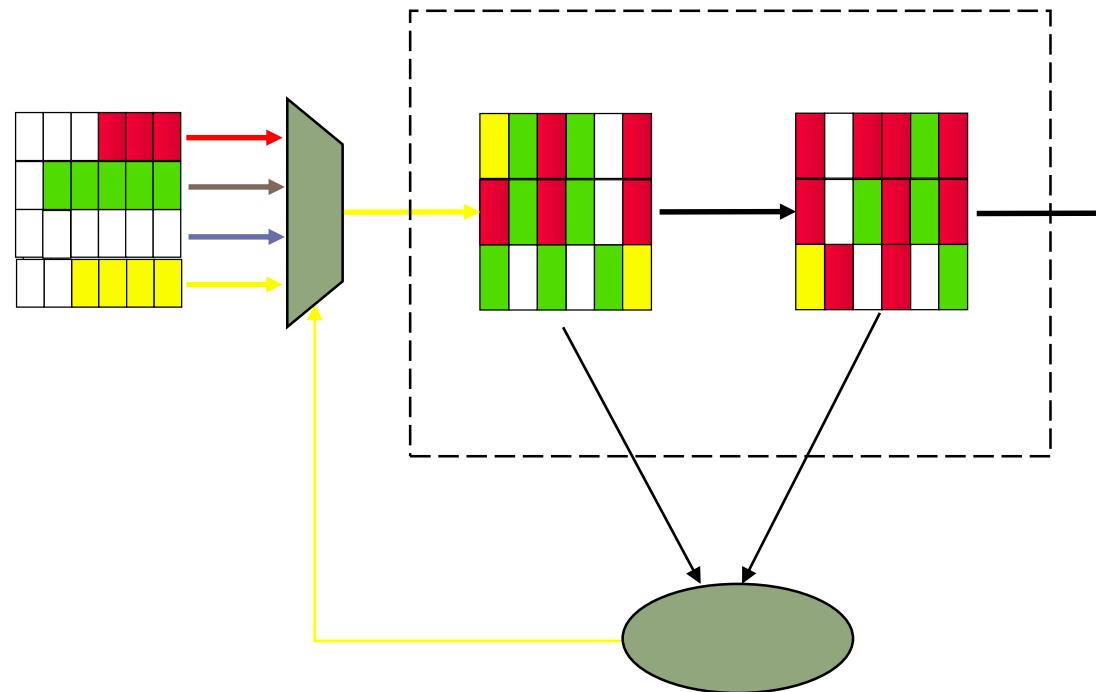
SMT Pipeline



[EV8 – Microprocessor Forum, Oct 1999]

Icount Choosing Policy

Fetch from thread with the least instructions in flight.



Why does this enhance throughput?

Why Does Icount Make Sense?

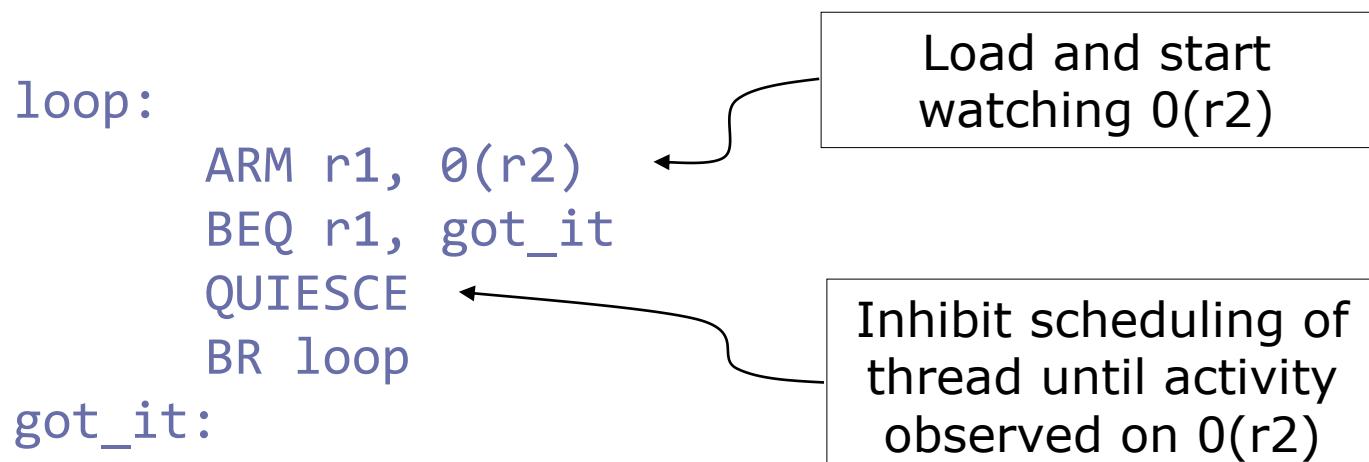
$$T = \frac{N}{L}$$

Assuming latency (L) is unchanged with the addition of threading.
For each thread i with original throughput T_i (and 4 threads):

$$T_i/4 = \frac{N/4}{L}$$

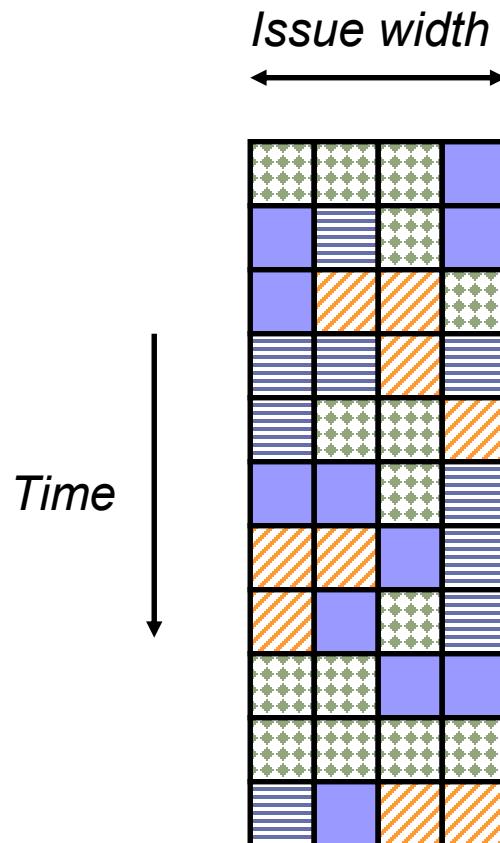
SMT Fetch Policies (Locks)

- Problem:
Spin looping thread consumes resources
- Solution:
Provide quiescing operation that allows a thread to sleep until a memory location changes

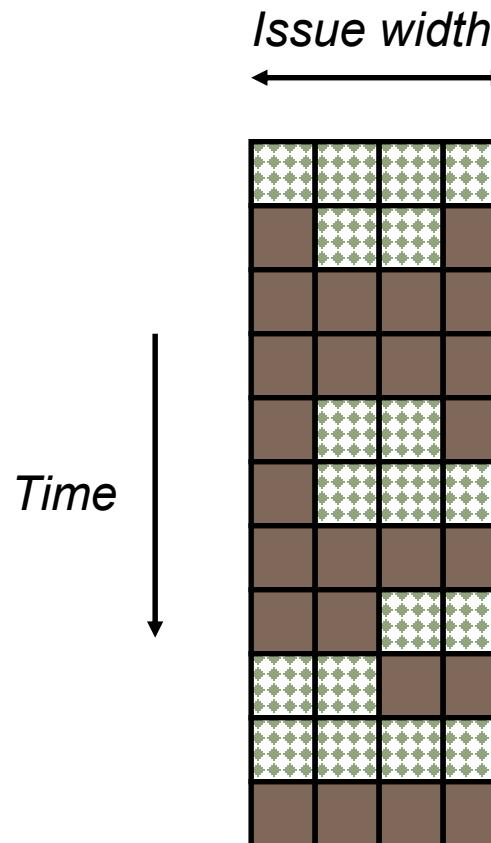


Adaptation to parallelism type

For regions with high thread level parallelism (TLP) entire machine width is shared by all threads



For regions with low thread level parallelism (TLP) entire machine width is available for instruction level parallelism (ILP)

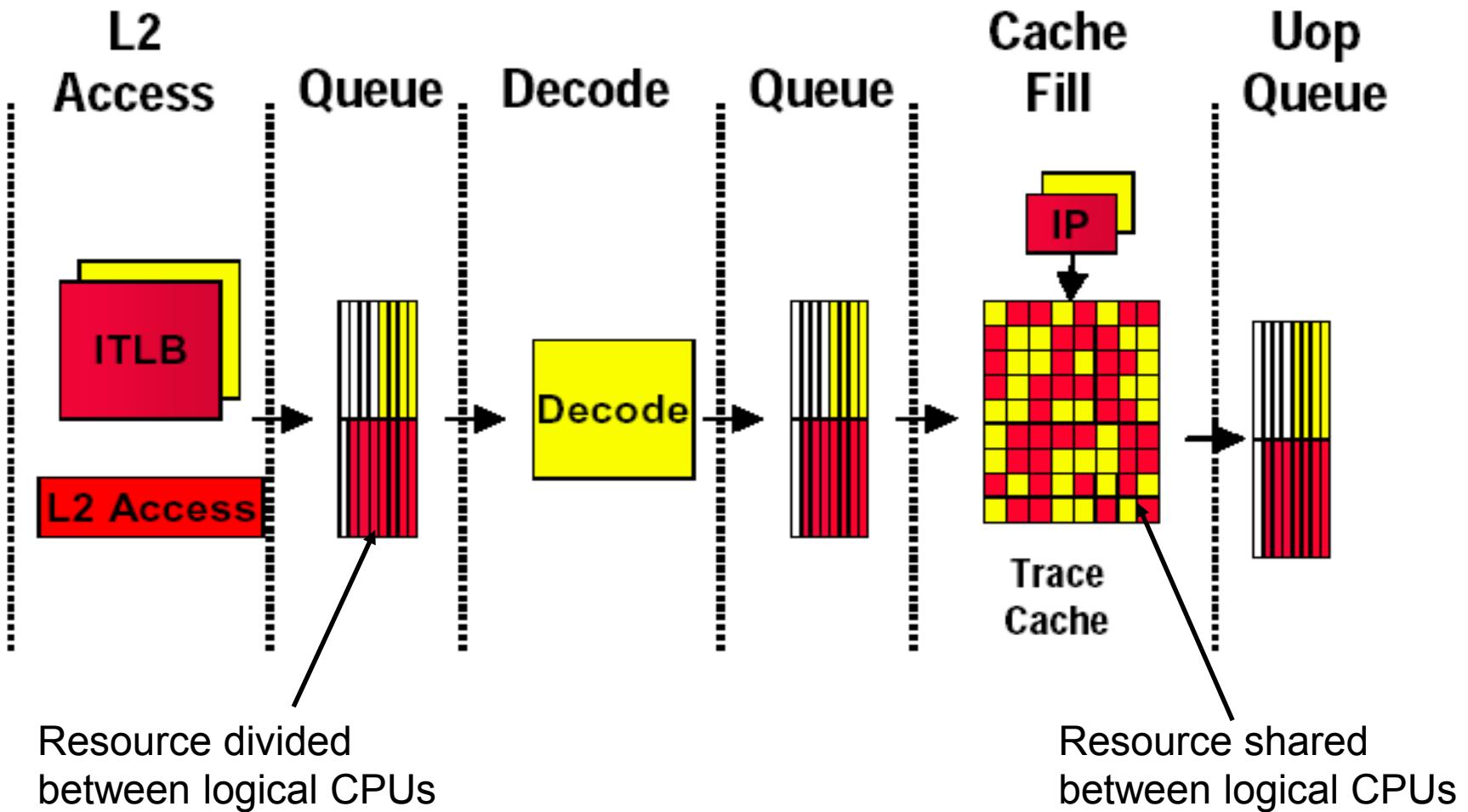


Pentium-4 Hyperthreading (2002)

- First commercial SMT design (2-way SMT)
 - Hyperthreading == SMT
- Logical processors share nearly all resources of the physical processor
 - Caches, execution units, branch predictors
- Die area overhead of hyperthreading ~ 5%
- When one logical processor is stalled, the other can make progress
 - No logical processor can use all entries in queues when two threads are active
- Processor running only one active software thread runs at approximately same speed with or without hyperthreading

Pentium-4 Hyperthreading

Front End

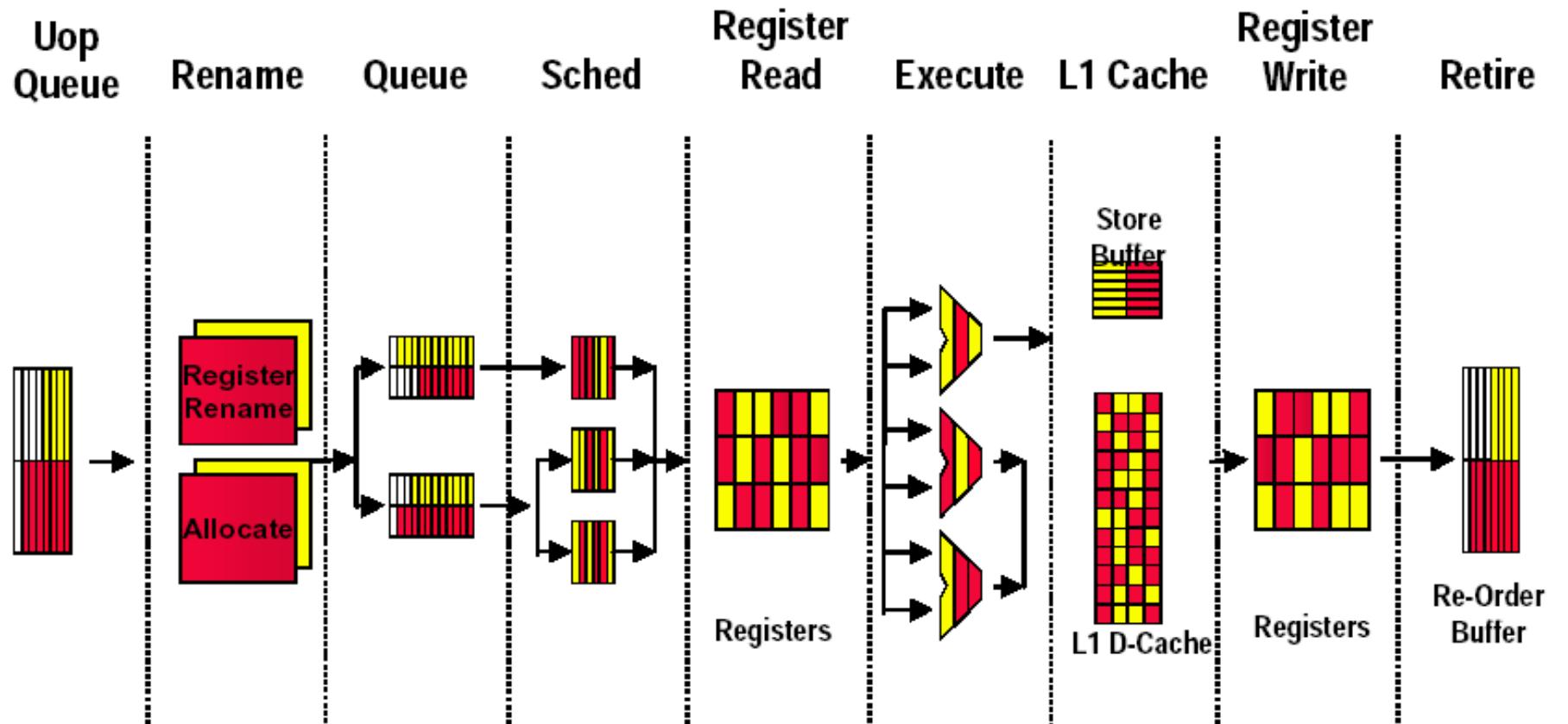


[Intel Technology Journal, Q1 2002]

Pentium-4 Branch Predictor

- Separate return address stacks per thread
Why?
- Separate first-level global branch history table
Why?
- Shared second-level branch history table, tagged with logical processor IDs

Pentium-4 Hyperthreading *Execution Pipeline*



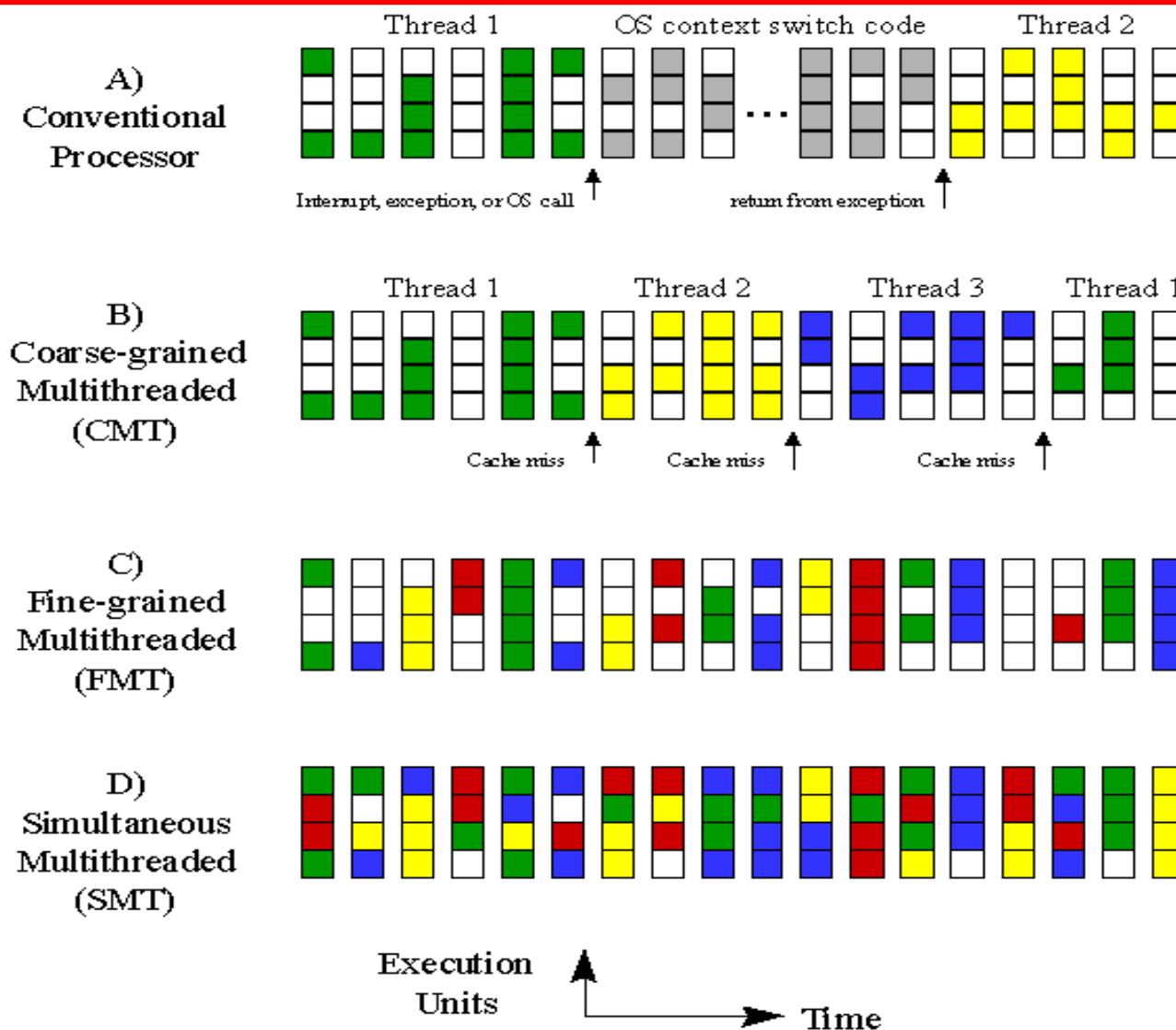
[Intel Technology Journal, Q1 2002]

April 27, 2021

MIT 6.823 Spring 2021

L18-31

Summary: Multithreading Styles



Thank you!

Microcoded and VLIW Processors

Daniel Sanchez
Computer Science & Artificial Intelligence Lab
M.I.T.

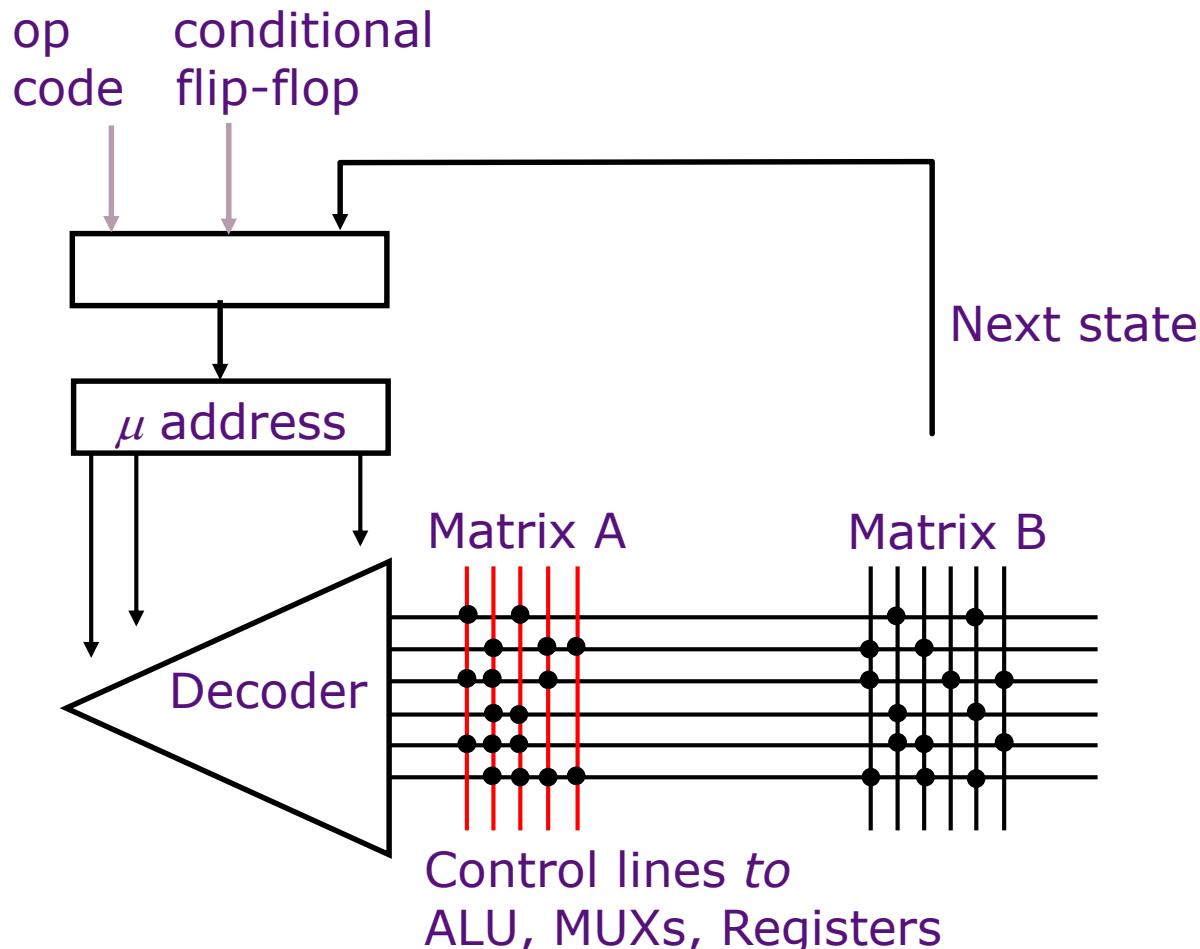
Hardwired vs Microcoded Processors

- All processors we have seen so far are hardwired:
The microarchitecture directly implements all the instructions in the ISA
- Microcoded processors add a layer of interpretation:
Each ISA instruction is executed as a sequence of simpler *microinstructions*
 - *Simpler implementation*
 - *Lower performance than hardwired ($CPI > 1$)*
- Microcoding common until the 80s, still in use today
(e.g., complex x86 instructions are decoded into multiple “micro-ops”)

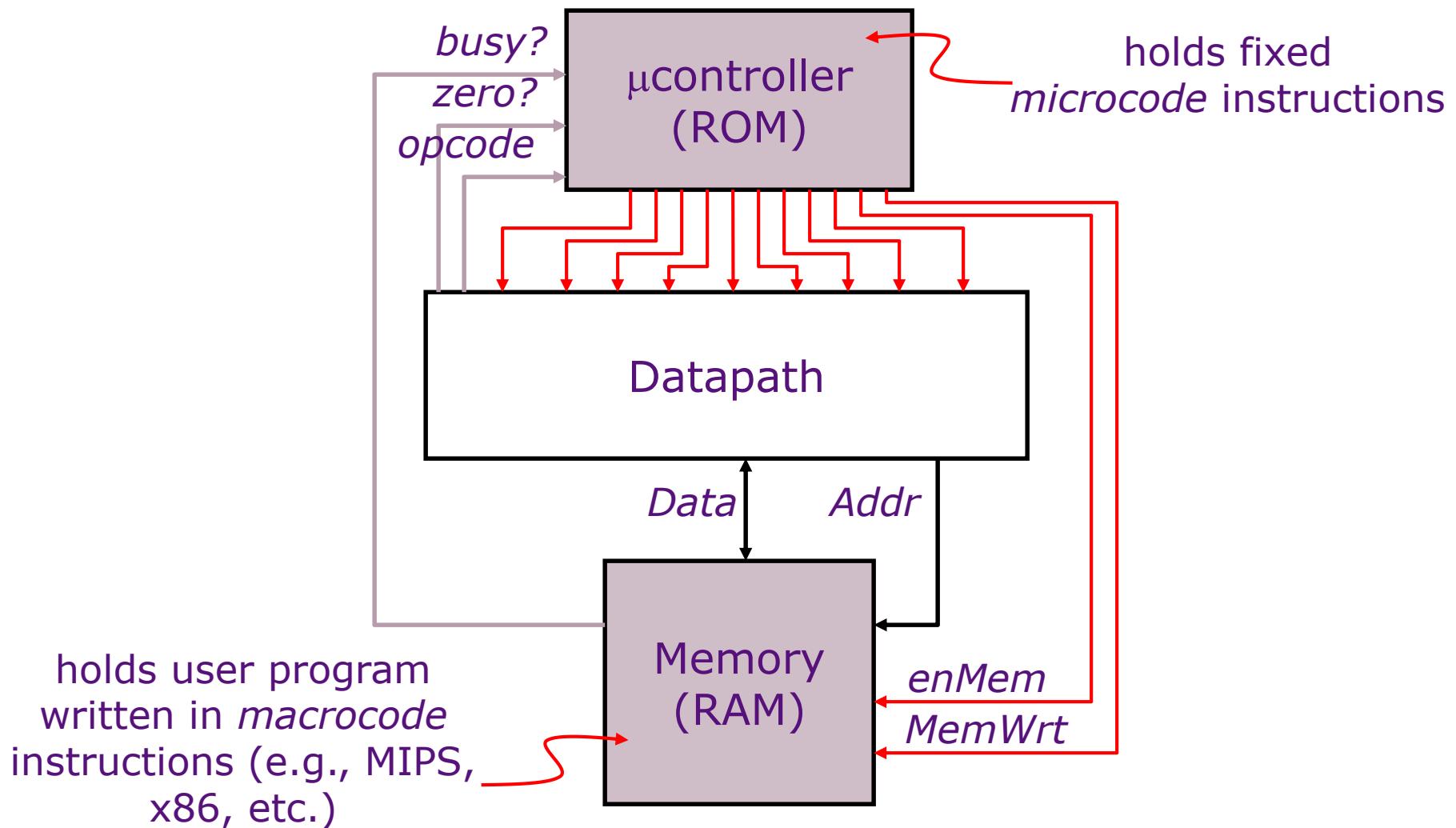
Microcontrol Unit

[Maurice Wilkes, 1954]

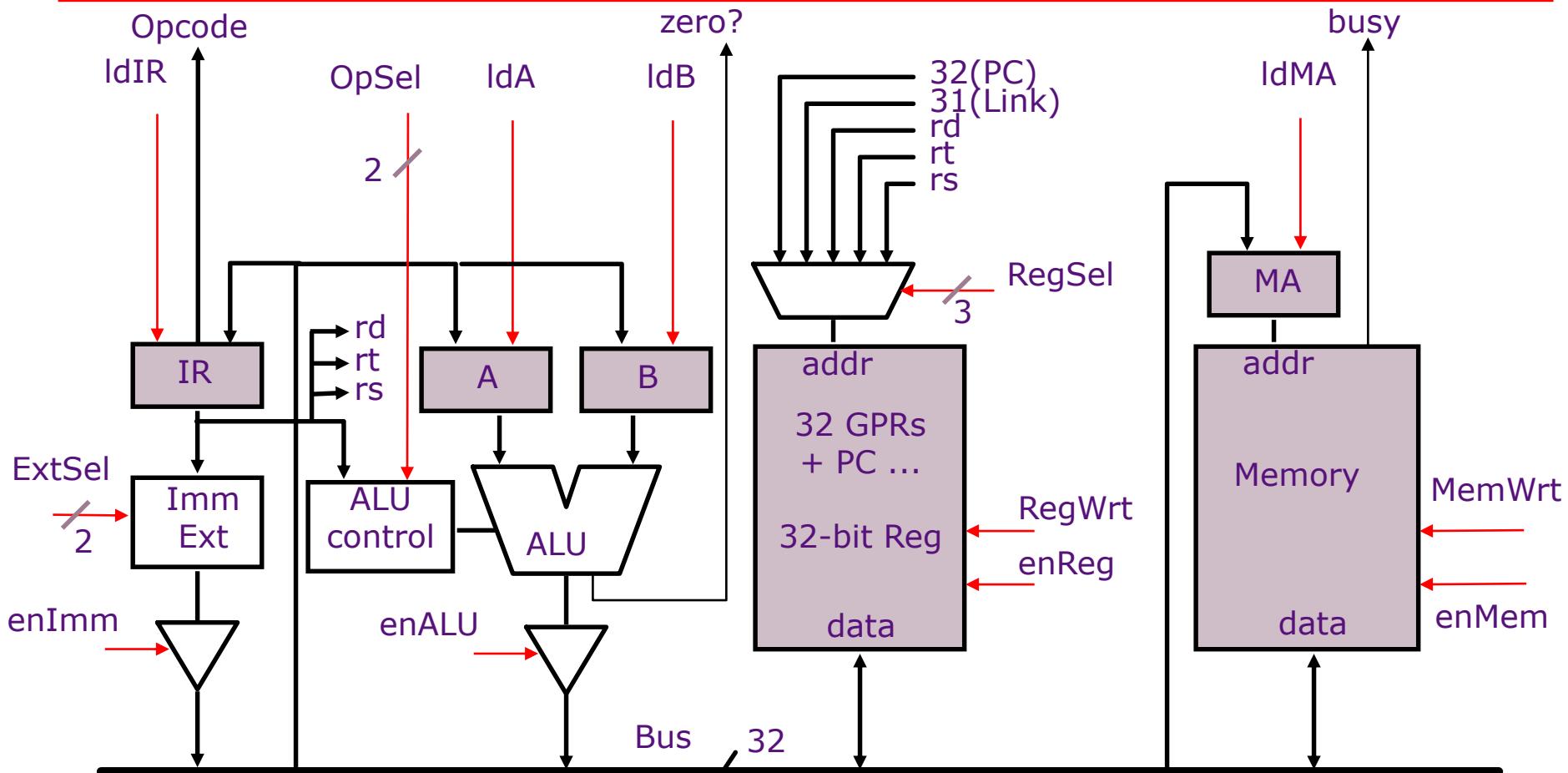
Embed the control logic state table in a read-only memory array



Microcoded Microarchitecture



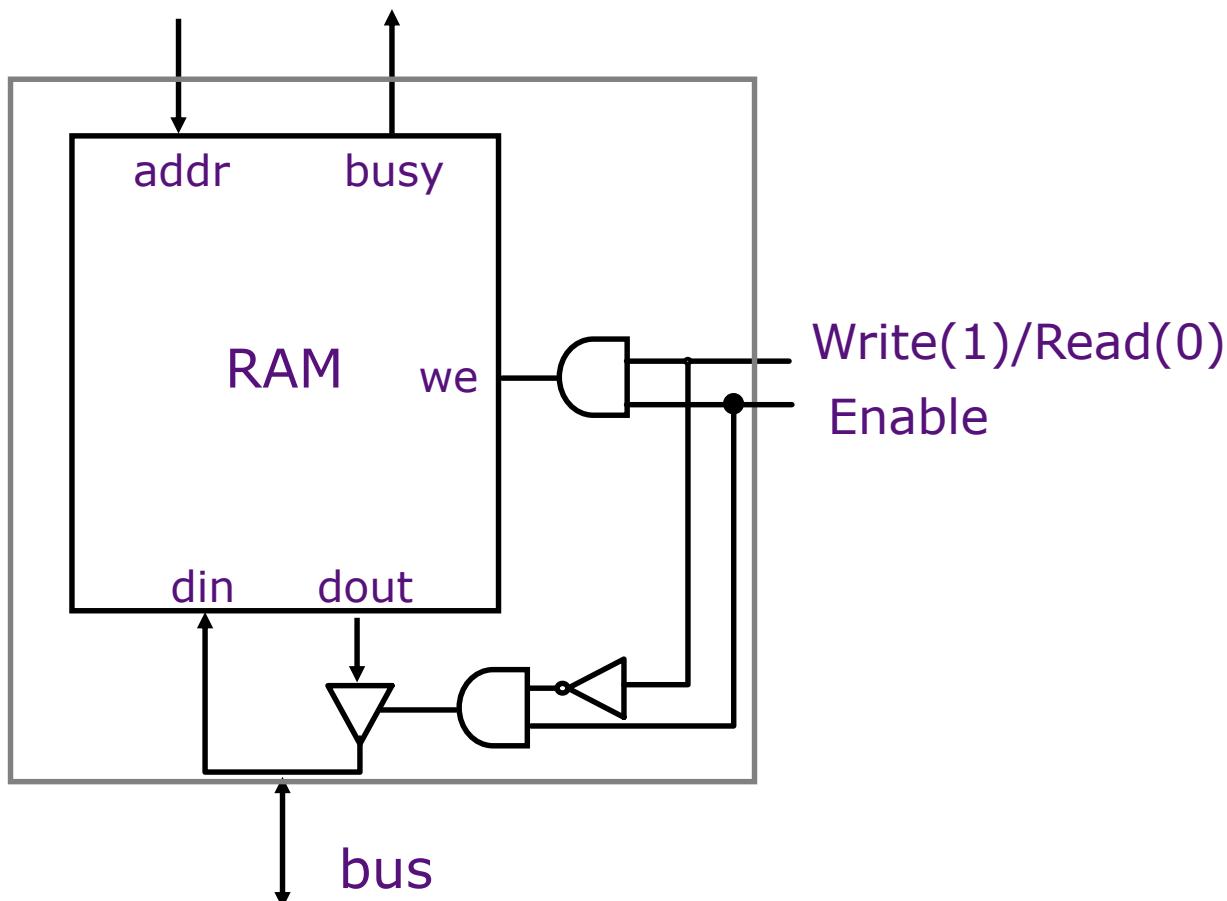
A Bus-based Datapath for MIPS



Microinstruction: register to register transfer (17 control signals)

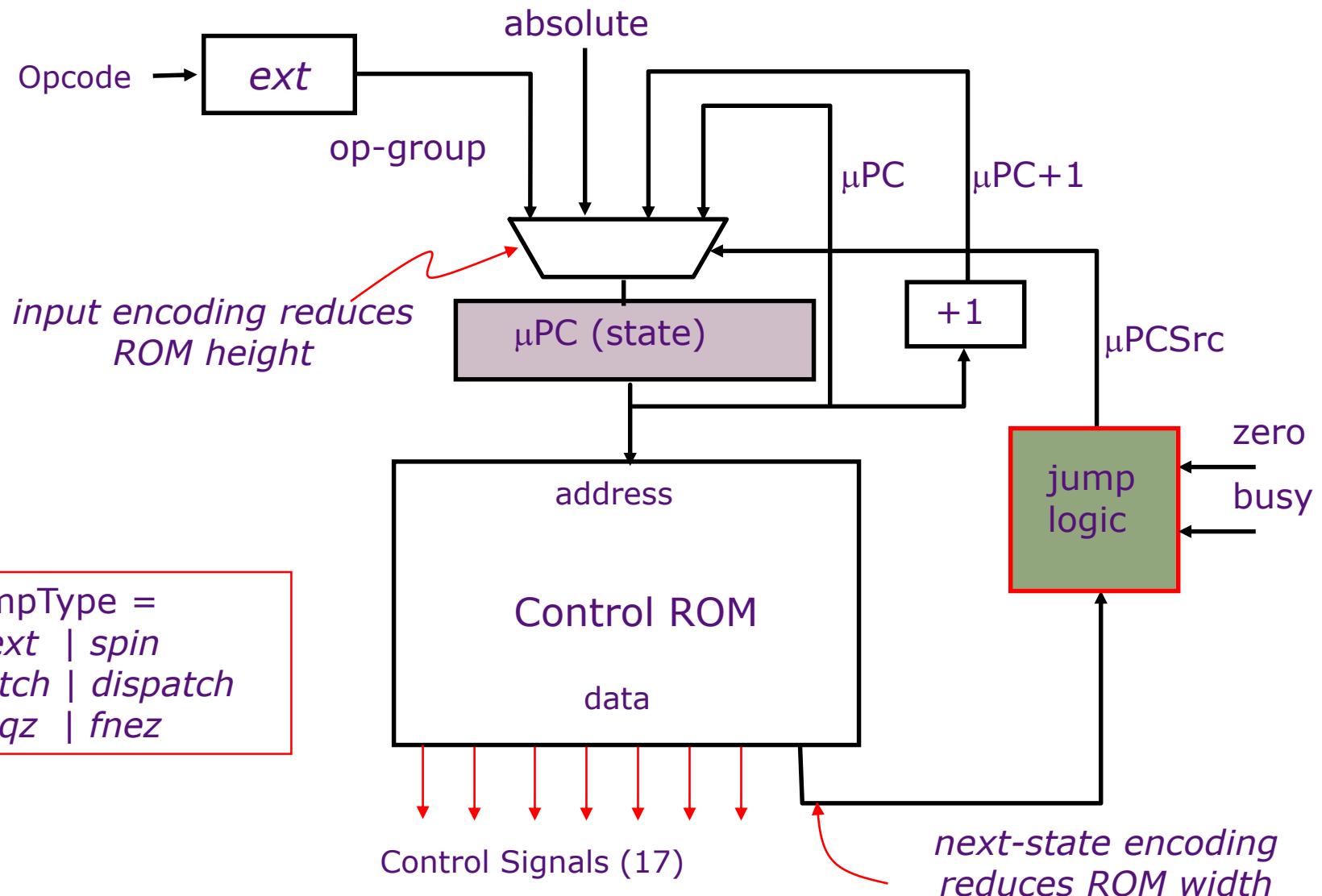
$\begin{array}{ll} \text{MA} & \leftarrow \text{PC} \\ \text{B} & \leftarrow \text{Reg}[\text{rt}] \end{array}$ means $\text{RegSel} = \text{PC}; \text{enReg} = \text{yes}; \text{IdMA} = \text{yes}$
 $\text{B} \leftarrow \text{Reg}[\text{rt}]$ means

Memory Module



- Assumption: Memory operates asynchronously and is slow compared to Reg-to-Reg transfers

Microcode Controller



Jump Logic

$\mu\text{PCSrc} = \text{Case } \mu\text{JumpTypes}$

next	\Rightarrow	$\mu\text{PC} + 1$
spin	\Rightarrow	if (busy) then μPC else $\mu\text{PC} + 1$
fetch	\Rightarrow	absolute
dispatch	\Rightarrow	op-group
feqz	\Rightarrow	if (zero) then absolute else $\mu\text{PC} + 1$
fnez	\Rightarrow	if (zero) then $\mu\text{PC} + 1$ else absolute

Instruction Execution

Execution of a MIPS instruction involves

1. instruction fetch
2. decode and register fetch
3. ALU operation
4. memory operation (optional)
5. write back to register file (optional)
+ the computation of the
next instruction address

Instruction Fetch

State	Control points	next-state
-------	----------------	------------

fetch₀ MA \leftarrow PC

fetch₁ IR \leftarrow Memory

fetch₂ A \leftarrow PC

fetch₃ PC \leftarrow A + 4

...

ALU₀ A \leftarrow Reg[rs]

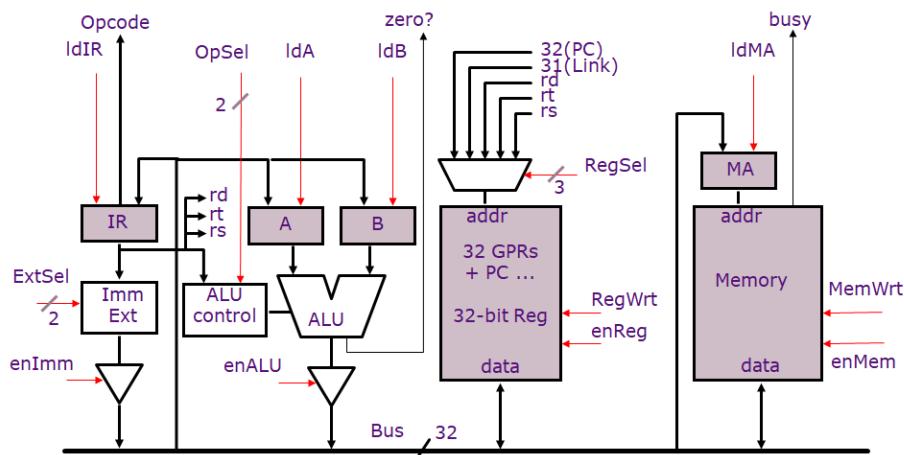
ALU₁ B \leftarrow Reg[rt]

ALU₂ Reg[rd] \leftarrow func(A,B)

ALUi₀ A \leftarrow Reg[rs]

ALUi₁ B \leftarrow sExt₁₆(Imm)

ALUi₂ Reg[rd] \leftarrow Op(A,B)



Load & Store

State	Control points	next-state
LW_0	$A \leftarrow \text{Reg}[rs]$	next
LW_1	$B \leftarrow \text{sExt}_{16}(\text{Imm})$	next
LW_2	$MA \leftarrow A+B$	next
LW_3	$\text{Reg}[rt] \leftarrow \text{Memory}$	spin
LW_4		fetch
SW_0	$A \leftarrow \text{Reg}[rs]$	next
SW_1	$B \leftarrow \text{sExt}_{16}(\text{Imm})$	next
SW_2	$MA \leftarrow A+B$	next
SW_3	$\text{Memory} \leftarrow \text{Reg}[rt]$	spin
SW_4		fetch

Branches

State	Control points	next-state
BEQZ_0	$A \leftarrow \text{Reg}[rs]$	next
BEQZ_1		fnez
BEQZ_2	$A \leftarrow \text{PC}$	next
BEQZ_3	$B \leftarrow \text{sExt}_{16}(\text{Imm} \ll 2)$	next
BEQZ_4	$\text{PC} \leftarrow A + B$	fetch
BNEZ_0	$A \leftarrow \text{Reg}[rs]$	next
BNEZ_1		feqz
BNEZ_2	$A \leftarrow \text{PC}$	next
BNEZ_3	$B \leftarrow \text{sExt}_{16}(\text{Imm} \ll 2)$	next
BNEZ_4	$\text{PC} \leftarrow A + B$	fetch

Jumps

State	Control points	next-state
J_0	$A \leftarrow PC$	next
J_1	$B \leftarrow IR$	next
J_2	$PC \leftarrow \text{JumpTarg}(A, B)$	fetch
JR_0	$A \leftarrow \text{Reg}[rs]$	next
JR_1	$PC \leftarrow A$	fetch
JAL_0	$A \leftarrow PC$	next
JAL_1	$\text{Reg}[31] \leftarrow A$	next
JAL_2	$B \leftarrow IR$	next
JAL_3	$PC \leftarrow \text{JumpTarg}(A, B)$	fetch
$JALR_0$	$A \leftarrow PC$	next
$JALR_1$	$B \leftarrow \text{Reg}[rs]$	next
$JALR_2$	$\text{Reg}[31] \leftarrow A$	next
$JALR_3$	$PC \leftarrow B$	fetch

VAX 11-780 Microcode (1978)

```

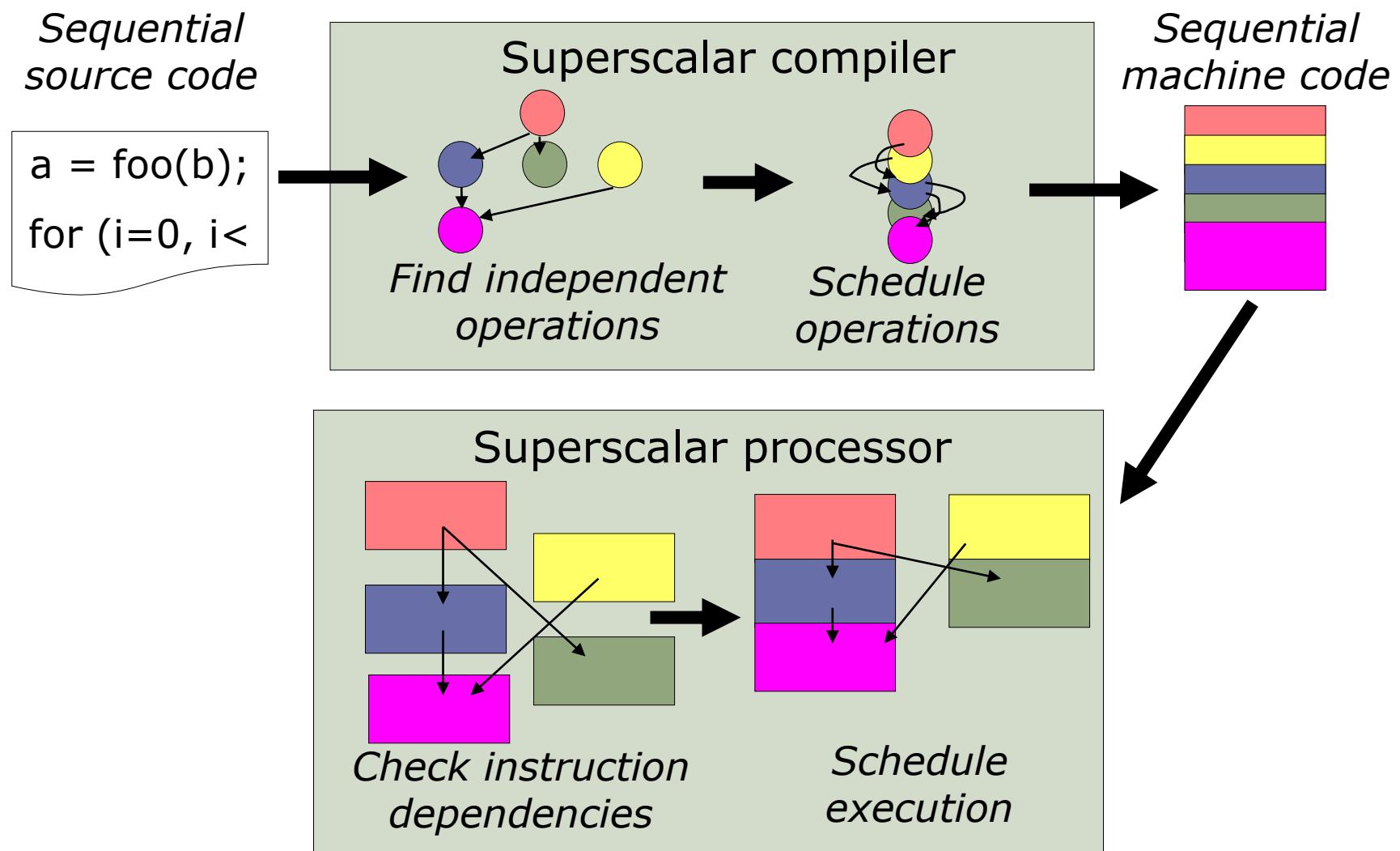
; P1WFUD,1 [600,1205]      MICRO2 1F(12)    26-May-81 14:58:1      VAX11/780 Microcode : PCS 01, FPLA 0D, WCS122
; CALL2 ,Mic [600,1205]      procedure cell

;29744 ;HERE FOR CALLG OR CALLS, AFTER PROBING THE EXTENT OF THE STACK
;29745
;29746 =0 ;-----;CALL SITE FOR MPUSH
;29747 CALL,7: D_0,AND,RC[T2], ;STRIP MASK TO BITS 11-0
;29748     CALL,J/MPUSH ;PUSH REGISTERS
;29749
;29750 ;-----;RETURN FROM MPUSH
;29751 CACHE_D[LONG], ;PUSH PC
;29752 LAB_R[SP] ; BY SP
;29753
;29754 ;-----;
;29755 CALL,8: R[SP]&VA_LA-K[,8] ;UPDATE SP FOR PUSH OF PC &
;29756
;29757 ;-----;
;29758 D_R[FPP] ;READY TO PUSH FRAME POINTER
;29759
;29760 =0 ;-----;CALL SITE FOR PSHSP
;29761 CACHE_D[LONG], ;STORE FP,
;29762 LAB_R[SP], ; GET SP AGAIN
;29763 SC_K[.FFF0], ;-16 TO SC
;29764     CALL,J/PSHSP
;29765
;29766 ;-----;
;29767 D_R[TAP], ;READY TO PUSH AP
;29768 Q_ID[PSL] ; AND GET PSW FOR COMBINATIO
;29769
;29770 ;-----;
;29771 CACHE_D[LONG], ;STORE OLD AP
;29772 Q_0,ANDNOT,K1.F1, ;CLEAR PSW<T,N,Z,V,C>
;29773 LAB_R[SP] ;GET SP INTO LATCHES AGAIN
;29774
;29775 ;-----;
;29776 PC&VA_RC[T1], FLUSH,IB ; LOAD NEW PC AND CLEAR OUT
;29777
;29778 ;-----;
;29779 D_DAL,SC, ;PSW TO D<31:16>
;29780 O_RC[T2], ;RECOVER MASK
;29781 SC-SC+K[.3], ;PUT -13 IN SC
;29782     LOAD,IB, PC-PC+1 ;START FETCHING SUBROUTINE I
;29783
;29784 ;-----;
;29785 D_DAL,SC, ;MASK AND PSW IN D<31:03>
;29786 O_RC[T4], ;GET LOW BITS OF OLD SP TO Q<1:0>
;29787 SC-SC+K[,A] ;PUT -3 IN SC
;29788

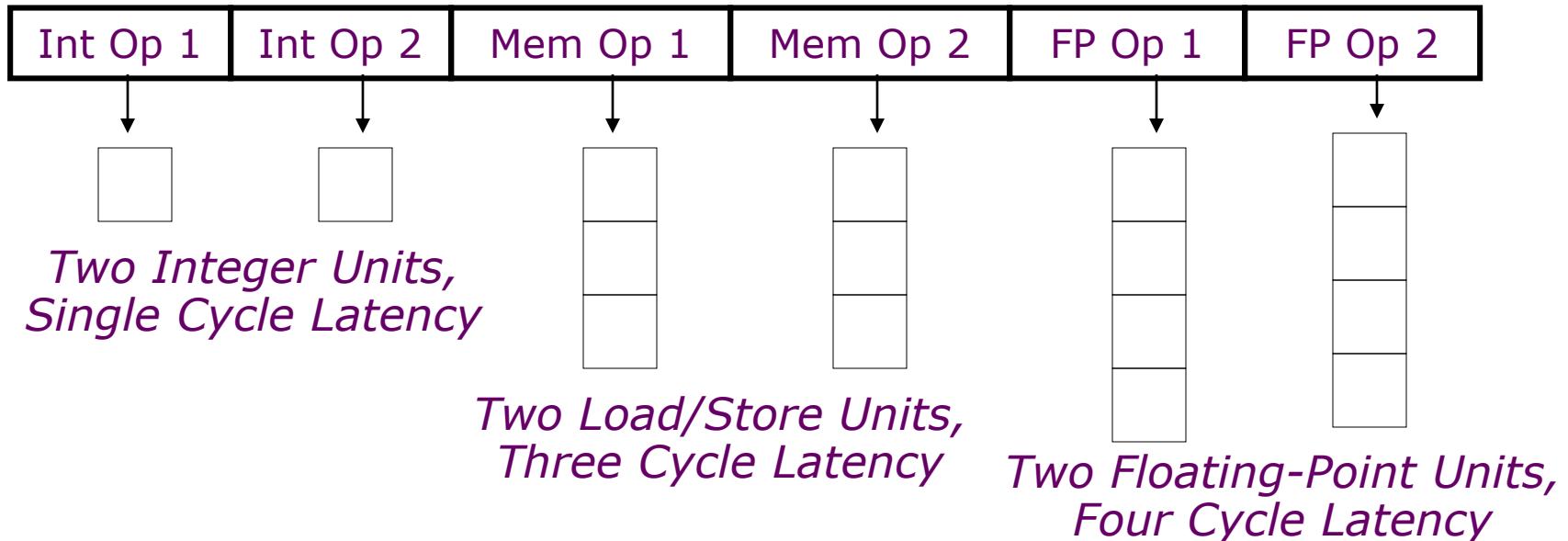
```

Very Long Instruction Word (VLIW) Processors

Sequential ISA Bottleneck



VLIW: Very Long Instruction Word



- Multiple operations packed into one instruction
- Each operation slot is for a fixed function
- Constant operation latencies are specified

VLIW Design Principles

The architecture:

- Allows operation parallelism within an instruction
 - No cross-operation RAW check
- Provides deterministic latency for all operations
 - Latency measured in ‘instructions’
 - No data use allowed before specified latency with no data interlocks

The compiler:

- Schedules (reorders) to maximize parallel execution
- Guarantees intra-instruction parallelism
- Schedules to avoid data hazards (no interlocks)
 - Typically separates operations with explicit NOPs

Early VLIW Machines

- FPS AP120B (1976)
 - scientific attached array processor
 - first commercial wide instruction machine
 - hand-coded vector math libraries using software pipelining and loop unrolling
- Multiflow Trace (1987)
 - commercialization of ideas from Fisher's Yale group including "trace scheduling"
 - available in configurations with 7, 14, or 28 operations/instruction
 - 28 operations packed into a 1024-bit instruction word
- Cydrome Cydra-5 (1987)
 - 7 operations encoded in 256-bit instruction word
 - rotating register file

Loop Execution

```
for (i=0; i<N; i++)  
    B[i] = A[i] + C;
```

Compile

```
loop: ld f1, 0(r1)  
      add r1, 8  
      fadd f2, f0, f1  
      sd f2, 0(r2)  
      add r2, 8  
      bne r1, r3, loop
```

loop:

Schedule

Int1	Int 2	M1	M2	FP+	FPx
add r1		ld			
				fadd	
add r2	bne	sd			

How many FP ops/cycle?

Loop Unrolling

```
for (i=0; i<N; i++)  
    B[i] = A[i] + C;
```

Unroll inner loop to
perform 4 iterations
at once

```
for (i=0; i<N; i+=4)  
{  
    B[i]      = A[i] + C;  
    B[i+1] = A[i+1] + C;  
    B[i+2] = A[i+2] + C;  
    B[i+3] = A[i+3] + C;  
}
```

Is this code correct?

Scheduling Loop Unrolled Code

Unroll 4 ways

```
loop: ld f1, 0(r1)
      ld f2, 8(r1)
      ld f3, 16(r1)
      ld f4, 24(r1)
      add r1, 32
      fadd f5, f0, f1
      fadd f6, f0, f2
      fadd f7, f0, f3
      fadd f8, f0, f4
      sd f5, 0(r2)
      sd f6, 8(r2)
      sd f7, 16(r2)
      sd f8, 24(r2)
      add r2, 32
      bne r1, r3, loop
```

loop:

Schedule

	Int1	Int 2	M1	M2	FP+	FPx
			ld f1			
			ld f2			
			ld f3			
	add r1		ld f4		fadd f5	
					fadd f6	
					fadd f7	
					fadd f8	
			sd f5			
			sd f6			
			sd f7			
	add r2	bne	sd f8			

How many FLOPS/cycle?

Software Pipelining

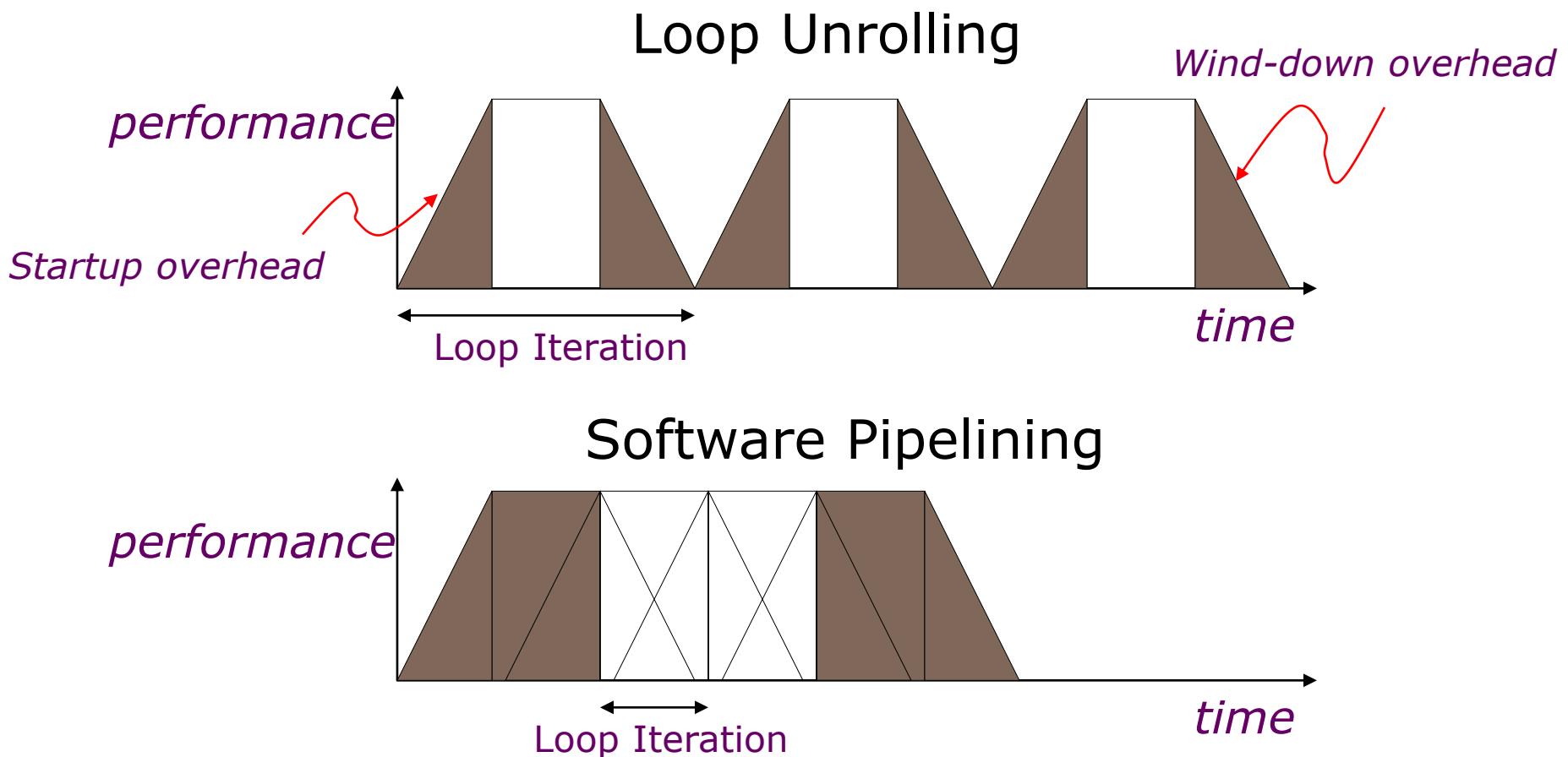
Unroll 4 ways first

```
loop: ld f1, 0(r1)
      ld f2, 8(r1)
      ld f3, 16(r1)
      ld f4, 24(r1)
      add r1, 32
      fadd f5, f0, f1
      fadd f6, f0, f2
      fadd f7, f0, f3
      fadd f8, f0, f4
      sd f5, 0(r2)
      sd f6, 8(r2)
      sd f7, 16(r2)
      add r2, 32
      sd f8, -8(r2)
      bne r1, r3, loop
```

	Int1	Int 2	M1	M2	FP+	FPx
prolog			ld f1			
			ld f2			
			ld f3			
	add r1		ld f4			
iterate			ld f1	fadd f5		
			ld f2	fadd f6		
			ld f3	fadd f7		
	add r1		ld f4	fadd f8		
loop:			ld f1	sd f5	fadd f5	
			ld f2	sd f6	fadd f6	
		add r2	ld f3	sd f7	fadd f7	
	add r1	bne	ld f4	sd f8	fadd f8	
epilog				sd f5	fadd f5	
				sd f6	fadd f6	
		add r2		sd f7	fadd f7	
			bne	sd f8	fadd f8	
				sd f5		

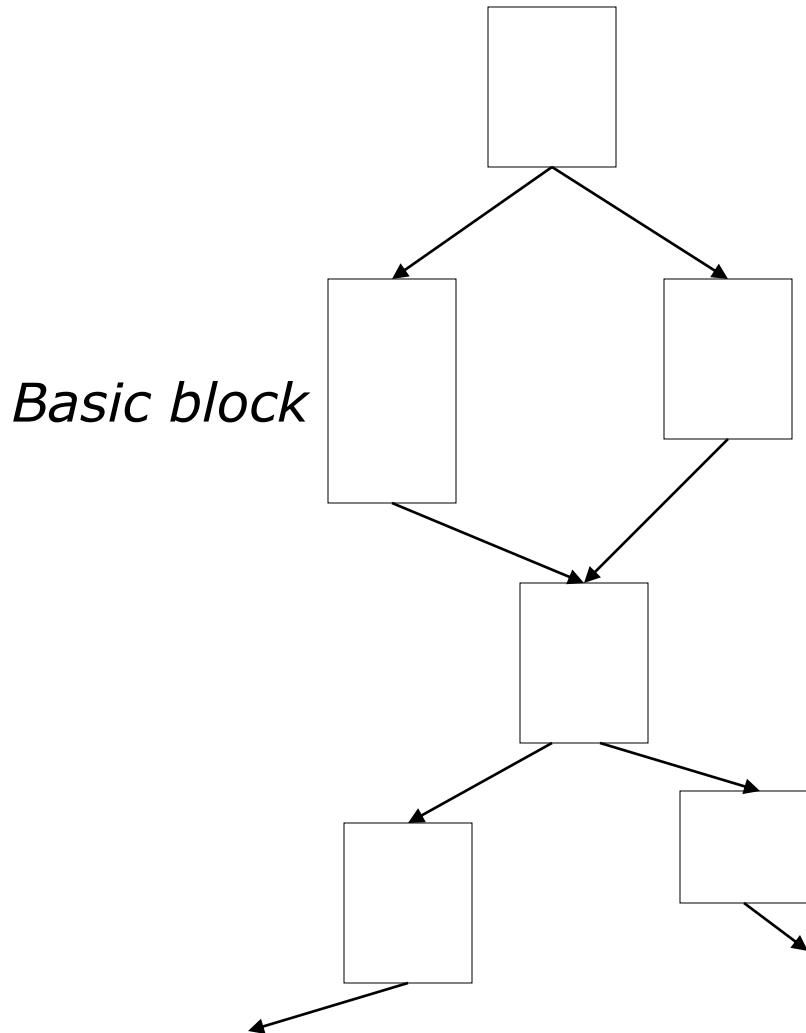
How many FLOPS/cycle?

Software Pipelining vs. Unrolling



Software pipelining pays startup/wind-down costs only once per loop, not once per iteration

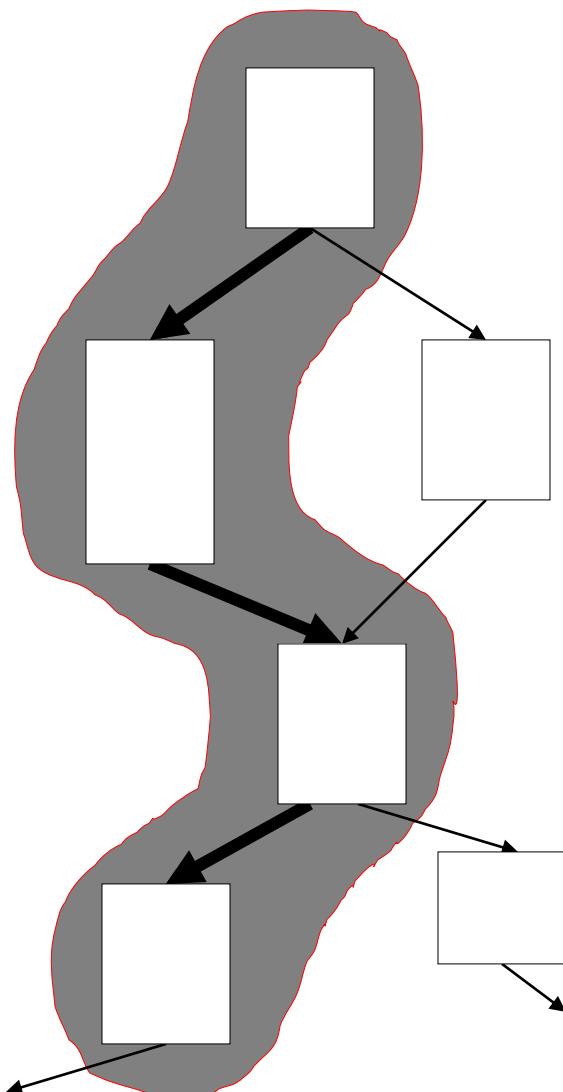
What if there are no loops?



- Branches limit basic block size in control-flow intensive irregular code
- Difficult to find ILP in individual basic blocks

Trace Scheduling

[Fisher,Ellis]



- Pick string of basic blocks, a trace, that represents most frequent branch path
- Schedule whole “trace” at once
- Add fixup code to cope with branches jumping out of trace

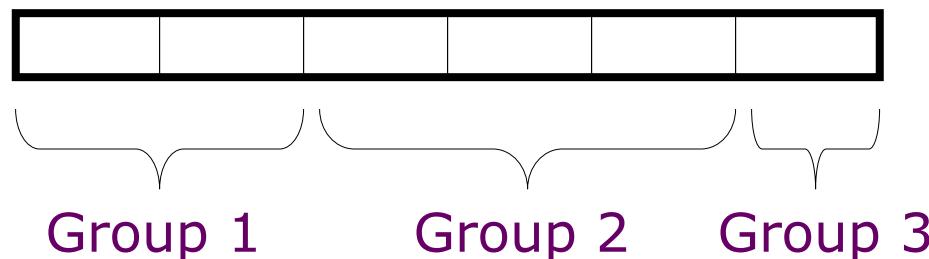
How do we know which trace to pick?

Problems with “Classic” VLIW

- Knowing branch probabilities
 - Profiling requires a significant extra step in build process
- Scheduling for statically unpredictable branches
 - Optimal schedule varies with branch path
- Object code size
 - Instruction padding wastes instruction memory/cache
 - Loop unrolling/software pipelining replicates code
- Scheduling memory operations
 - Caches and/or memory bank conflicts impose statically unpredictable variability
 - Uncertainty about addresses limit code reordering
- Object-code compatibility
 - Have to recompile all code for every machine, even for two machines in same generation

VLIW Instruction Encoding

- Schemes to reduce effect of unused fields
 - Compressed format in memory, expand on I-cache refill
 - used in Multiflow Trace
 - introduces instruction addressing challenge
 - Provide a single-op VLIW instruction
 - Cydra-5 UniOp instructions
 - Mark parallel groups
 - used in TMS320C6x DSPs, Intel IA-64



Cydra-5: Memory Latency Register (MLR)

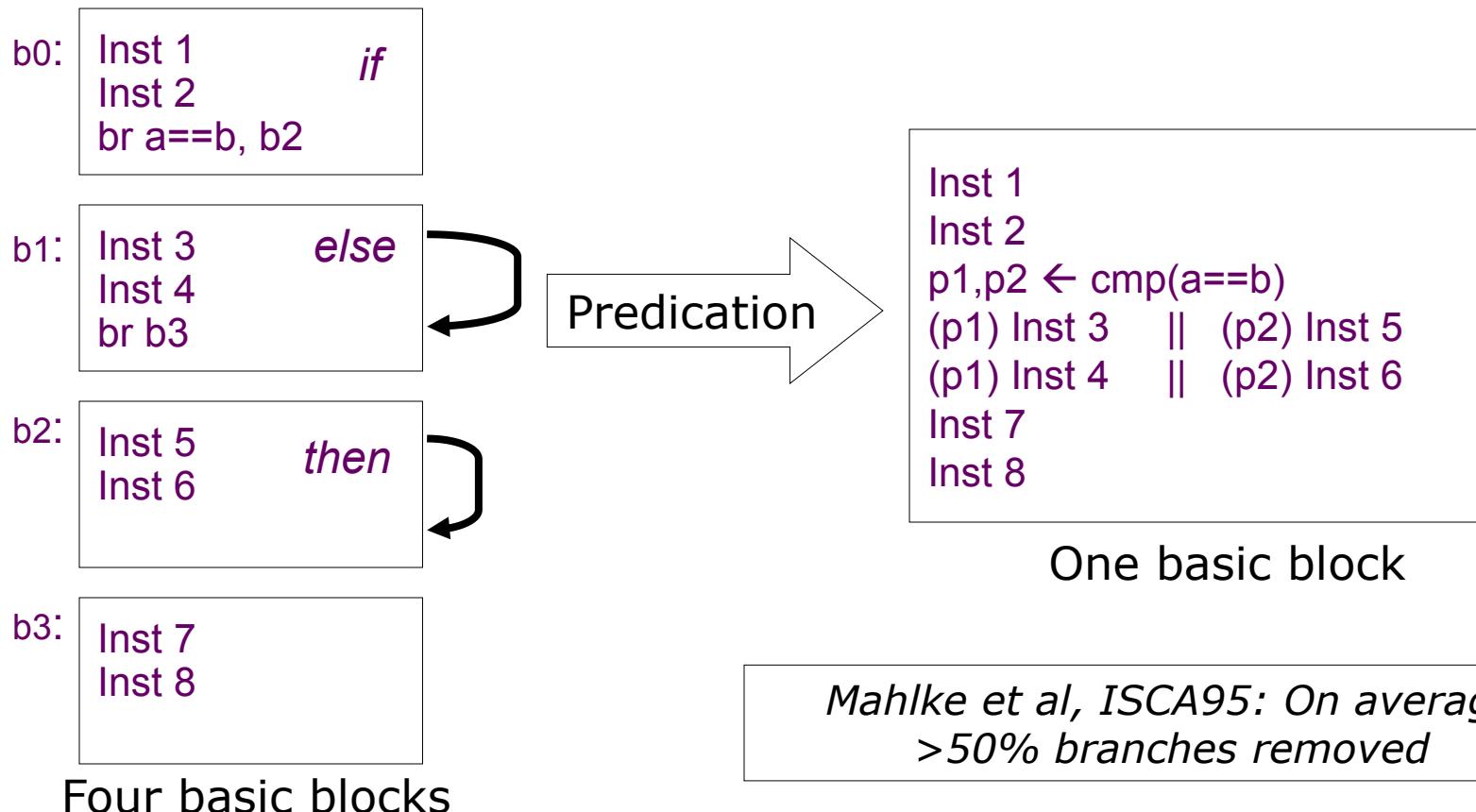
- Problem: Loads have variable latency
- Solution: Let software choose desired memory latency
- Compiler schedules code for maximum load-use distance
- Software sets MLR to latency that matches code schedule
- Hardware ensures that loads take exactly MLR cycles to return values into processor pipeline
 - Hardware buffers loads that return early
 - Hardware stalls processor if loads return late

IA-64 Predicated Execution

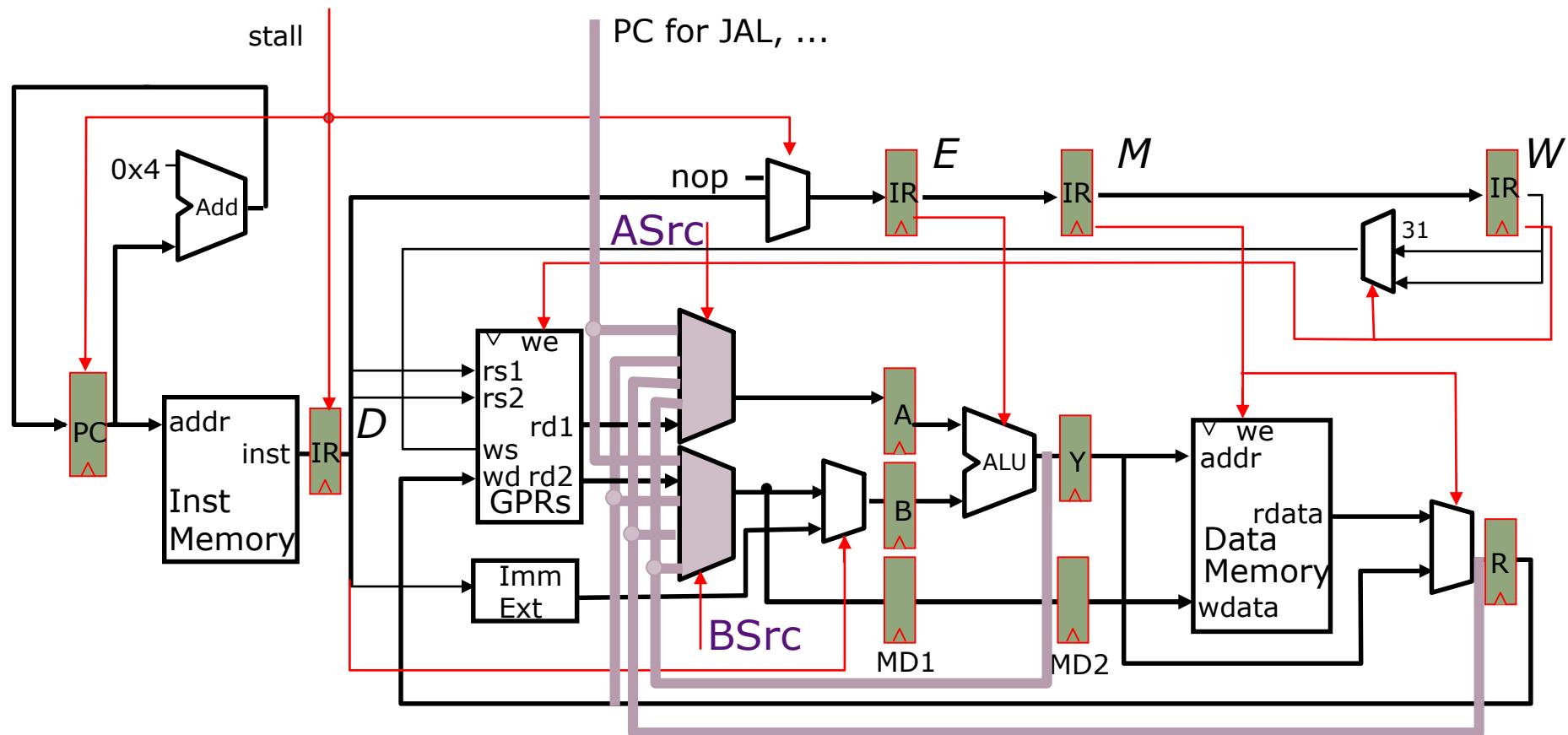
Problem: Mispredicted branches limit ILP

Solution: Eliminate hard-to-predict branches with predicated execution

- Almost all IA-64 instructions can be executed conditionally under predicate
- Instruction becomes NOP if predicate register false



Fully Bypassed Datapath

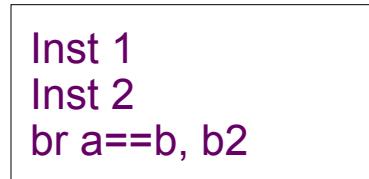


Where does predication fit in?

IA-64 Speculative Execution

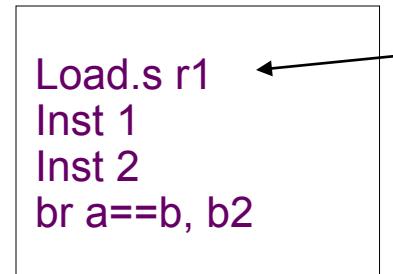
Problem: Branches restrict compiler code motion

Solution: Speculative operations that don't cause exceptions



Load r1
Use r1
Inst 3

*Can't move load above branch
because might cause spurious
exception*



Chk.s r1
Use r1
Inst 3

*Speculative load
never causes
exception, but sets
“poison” bit on
destination register*

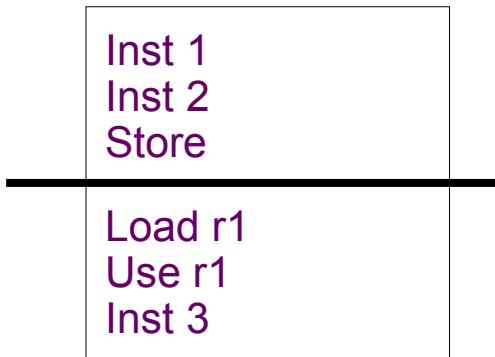
*Check for exception in
original home block
jumps to fixup code if
exception detected*

Particularly useful for scheduling long latency loads early

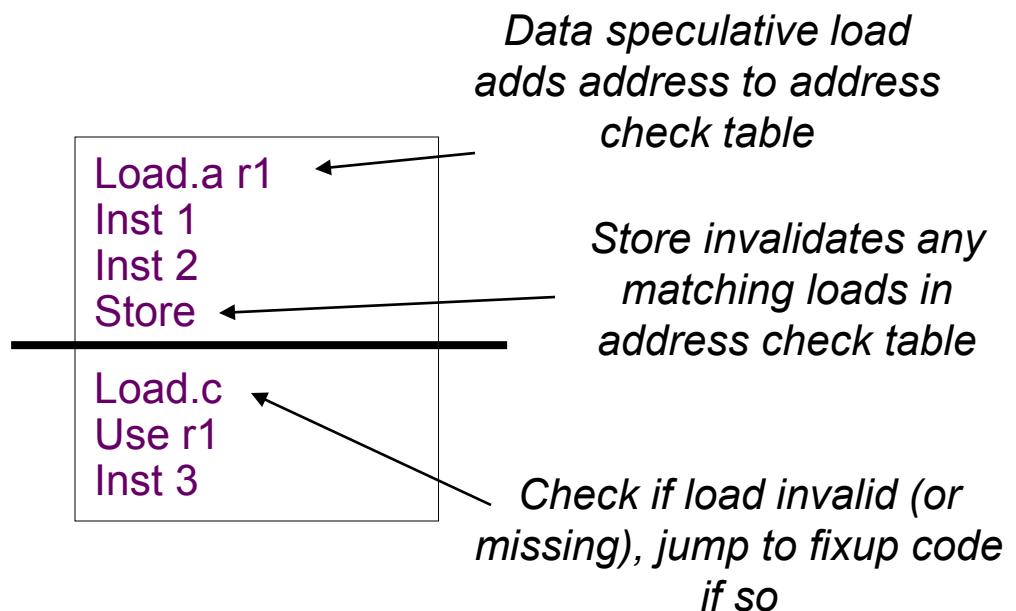
IA-64 Data Speculation

Problem: Possible memory hazards limit code scheduling

Solution: Instruction-based speculation with hardware monitor to check for pointer hazards

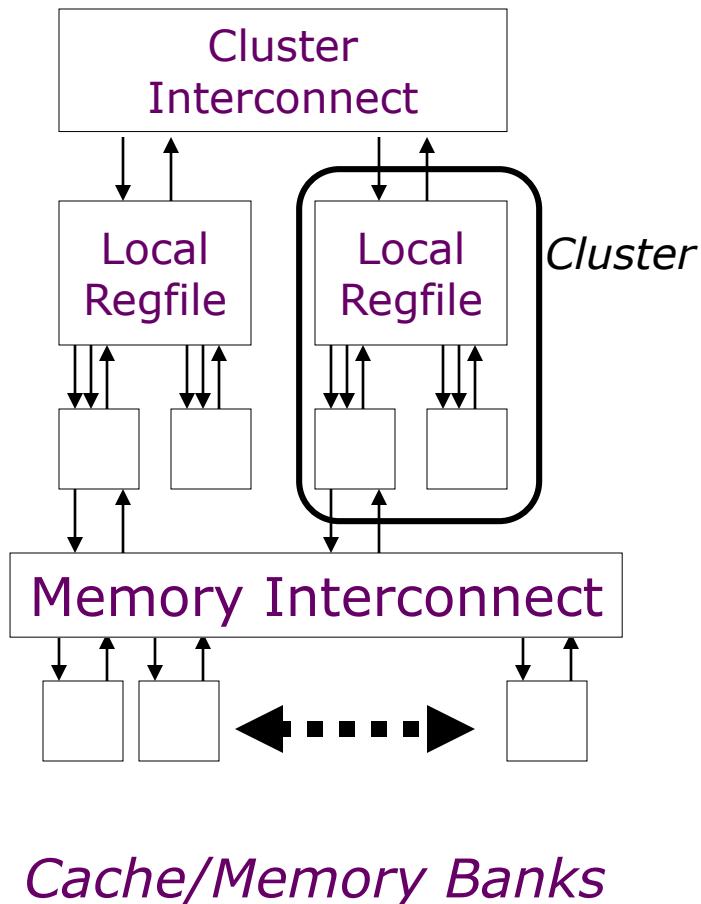


*Can't move load above store
because store might be to same
address*



Requires associative hardware in address check table

Clustered VLIW



- Divide machine into clusters of local register files and local functional units
- Lower bandwidth/higher latency interconnect between clusters
- Software responsible for mapping computations to minimize communication overhead
- Common in commercial embedded processors, examples include TI C6x series DSPs, and HP Lx processor
- Exists in some superscalar processors, e.g., Alpha 21264

Limits of Static Scheduling

- Unpredictable branches
- Unpredictable memory behavior
(cache misses and dependencies)
- Code size explosion
- Compiler complexity

Question:

How applicable are the VLIW-inspired techniques to traditional RISC/CISC processor architectures?

Thank you!

Next Lecture: Vector Processors

Vector Processors

Daniel Sanchez

Computer Science & Artificial Intelligence Lab
M.I.T.

Supercomputers

Definition of a supercomputer:

- Fastest machine in the world at given task
- A device to turn a compute-bound problem into an I/O bound problem
- Any machine costing \$30M+
- Any machine designed by Seymour Cray

CDC6600 (Cray, 1964) regarded as first supercomputer

Supercomputer Applications

Typical application areas:

- Military research (nuclear weapons, cryptography)
- Scientific research
- Weather forecasting
- Oil exploration
- Industrial design (car crash simulation)
- Bioinformatics

All involve huge computations on large data sets

In 70s-80s, Supercomputer ≡ Vector Machine

Loop Unrolled Code Schedule

```
for (i=0; i<N; i++)  
    B[i] = A[i] + C;
```

```
loop:  ld f1, 0(r1)  
        ld f2, 8(r1)  
        ld f3, 16(r1)  
        ld f4, 24(r1)  
        add r1, 32  
        fadd f5, f0, f1  
        fadd f6, f0, f2  
        fadd f7, f0, f3  
        fadd f8, f0, f4  
        sd f5, 0(r2)  
        sd f6, 8(r2)  
        sd f7, 16(r2)  
        sd f8, 24(r2)  
        add r2, 32  
        bne r1, r3, loop
```

	Int1	Int 2	M1	M2	FP+	FPx
loop:			ld f1			
			ld f2			
			ld f3			
	add r1		ld f4		fadd f5	
					fadd f6	
					fadd f7	
					fadd f8	
			sd f5			
			sd f6			
			sd f7			
	add r2	bne	sd f8			

Schedule

Vector Supercomputers

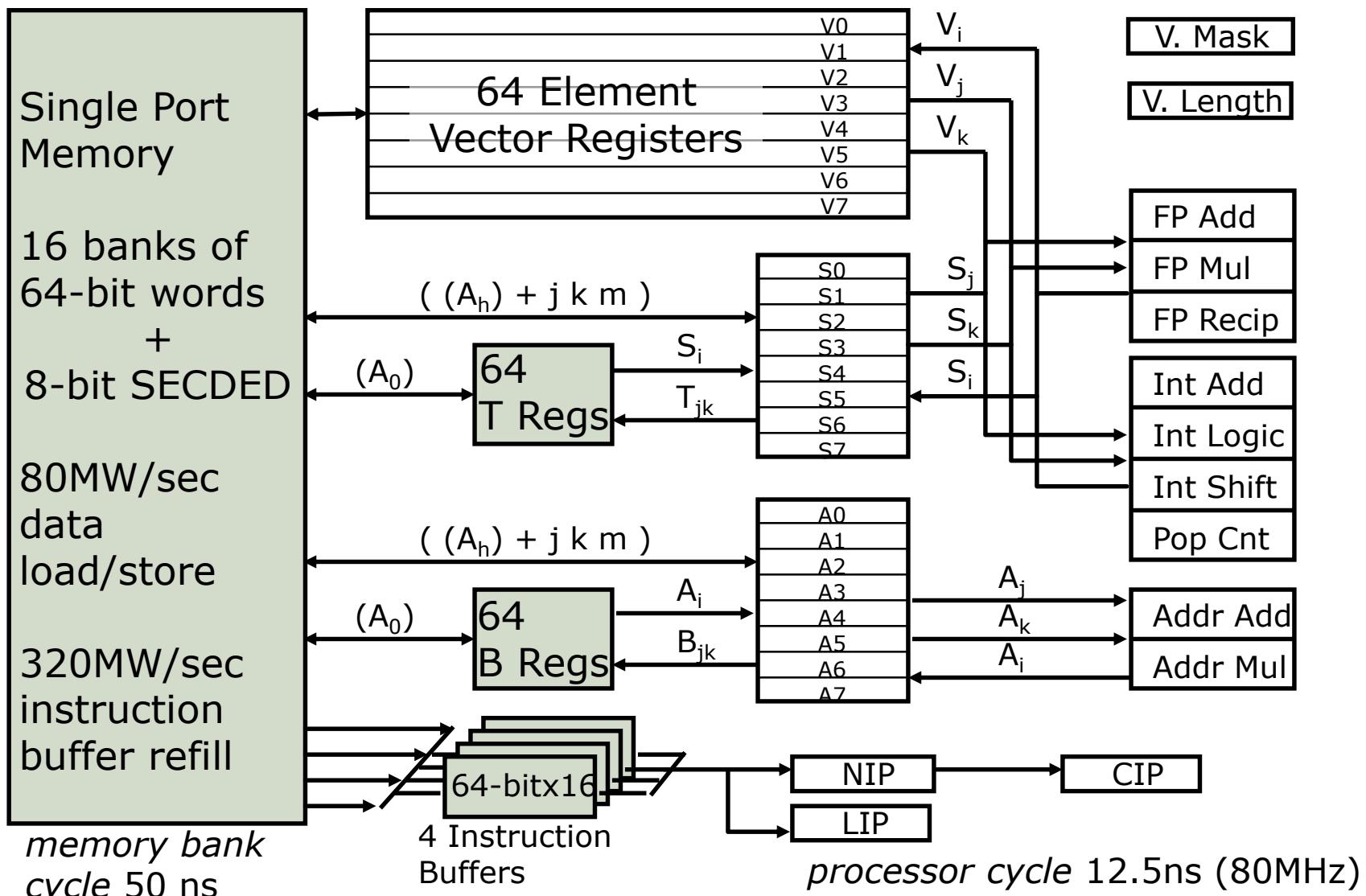
Epitomized by Cray-1, 1976:

- Scalar Unit
 - Load/Store Architecture
- Vector Extension
 - Vector Registers
 - Vector Instructions
- Implementation
 - Hardwired Control
 - Highly Pipelined Functional Units
 - No Data Caches
 - Interleaved Memory System
 - No Virtual Memory

Cray-1 (1976)

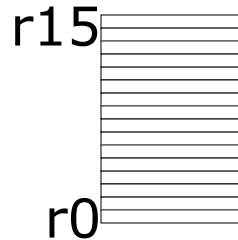


Cray-1 (1976)

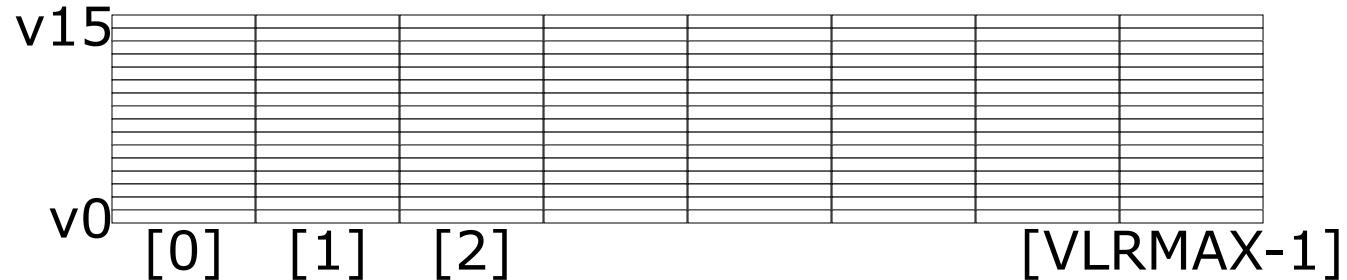


Vector Programming Model

Scalar Registers



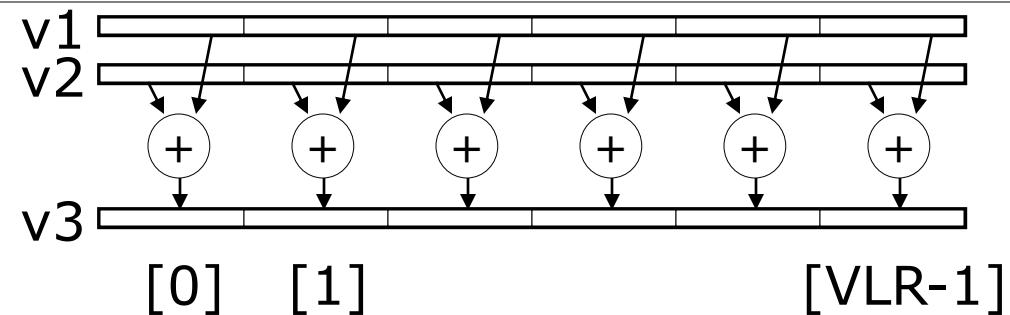
Vector Registers



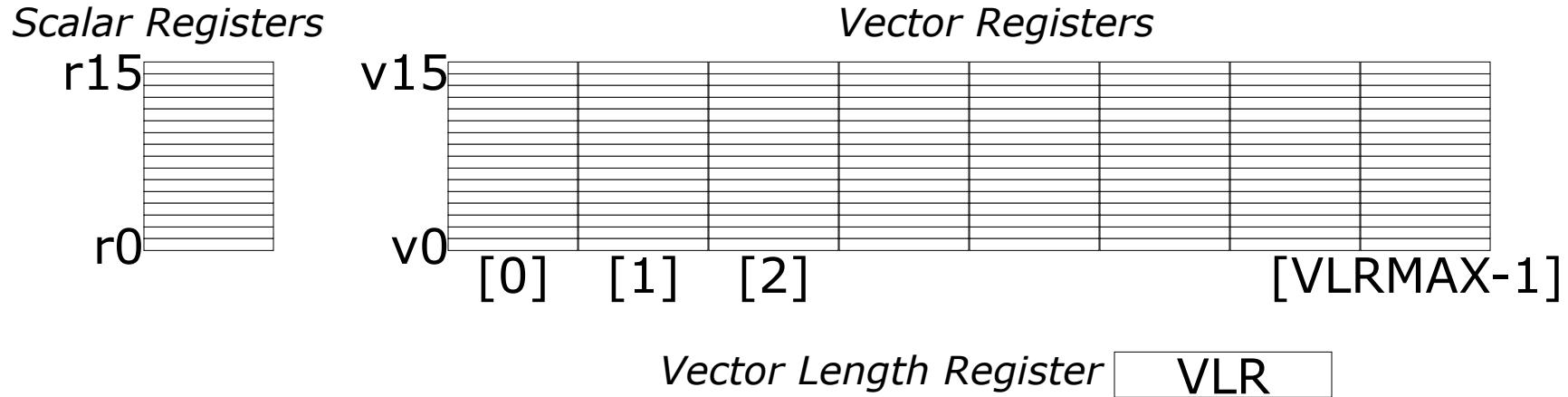
Vector Length Register VLR

Vector Arithmetic
Instructions

ADDV $v3, v1, v2$



Vector Programming Model

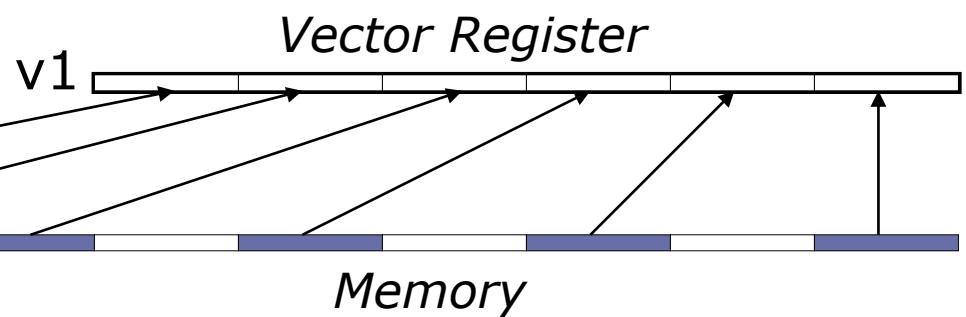


Vector Load and
Store Instructions

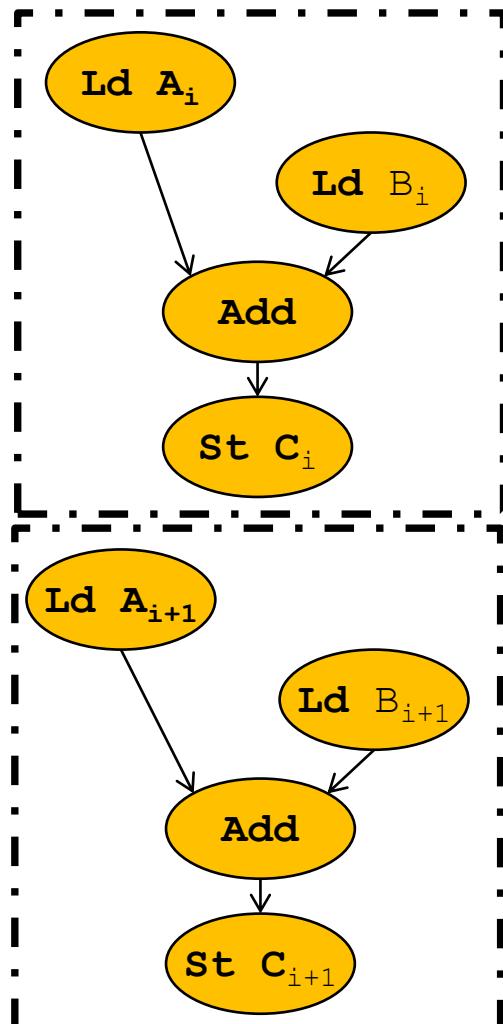
$LV\ v1, r1, r2$

Base, $r1$

Stride, $r2$



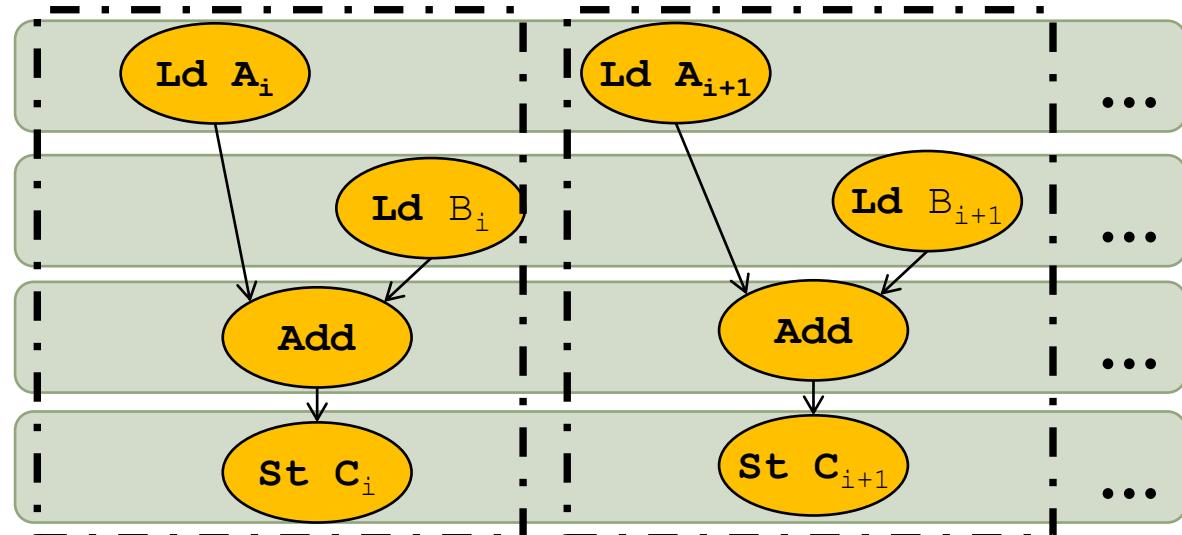
Compiler-based Vectorization



Scalar code

```
for (i=0; i<N; i++)
    C[i] = A[i] + B[i];
```

Compiler recognizes independent operations with loop dependence analysis



Vector code

Vector Code Example

```
# C code
```

```
for (i=0; i<64; i++)  
    C[i] = A[i] + B[i];
```

```
# Scalar code
```

```
LI R4, 64  
loop:  
    L.D F0, 0(R1)  
    L.D F2, 0(R2)  
    ADD.D F4, F2, F0  
    S.D F4, 0(R3)  
    DADDIU R1, 8  
    DADDIU R2, 8  
    DADDIU R3, 8  
    DSUBIU R4, 1  
    BNEZ R4, loop
```

```
# Vector code
```

```
LI VLR, 64  
LV V1, R1  
LV V2, R2  
ADDV.D V3, V1, V2  
SV V3, R3
```

Vector ISA Attributes

- Compact
 - One short instruction encodes N operations
- Expressive, tells hardware that these N operations:
 - Are independent
 - Use the same functional unit
 - Access disjoint elements in vector registers
 - Access registers in same pattern as previous instructions
 - Access a contiguous block of memory
(unit-stride load/store)
 - Access memory in a known pattern
(strided load/store)

Vector ISA Hardware Implications

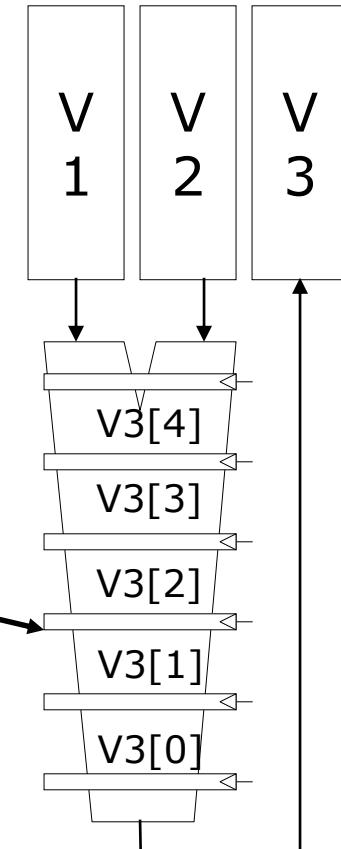
- Large amount of work per instruction
 - Less instruction fetch bandwidth requirements
 - Allows simplified instruction fetch design
- No data dependence within a vector
 - Amenable to deeply pipelined/parallel designs
- Disjoint vector element accesses
 - Banked rather than multi-ported register files
- Known regular memory access pattern
 - Allows for banked memory for higher bandwidth

Vector Arithmetic Execution

- Use deep pipeline (\Rightarrow fast clock) to execute element operations
- Simplifies control of deep pipeline because elements in vector are independent (\Rightarrow no hazards!)

Six-stage multiply pipeline

Given 64-element registers, how long does it take to compute V3?



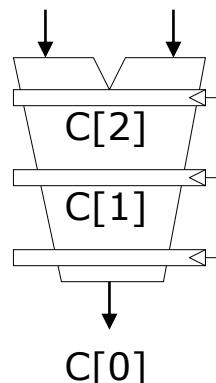
$$V3 \leftarrow V1 * V2$$

Vector Instruction Execution

ADDV C,A,B

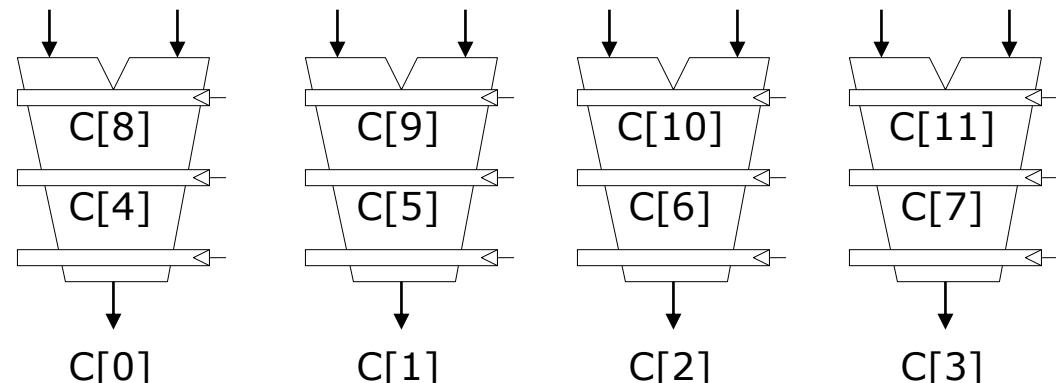
*Execution using
one pipelined
functional unit*

A[6] B[6]
A[5] B[5]
A[4] B[4]
A[3] B[3]

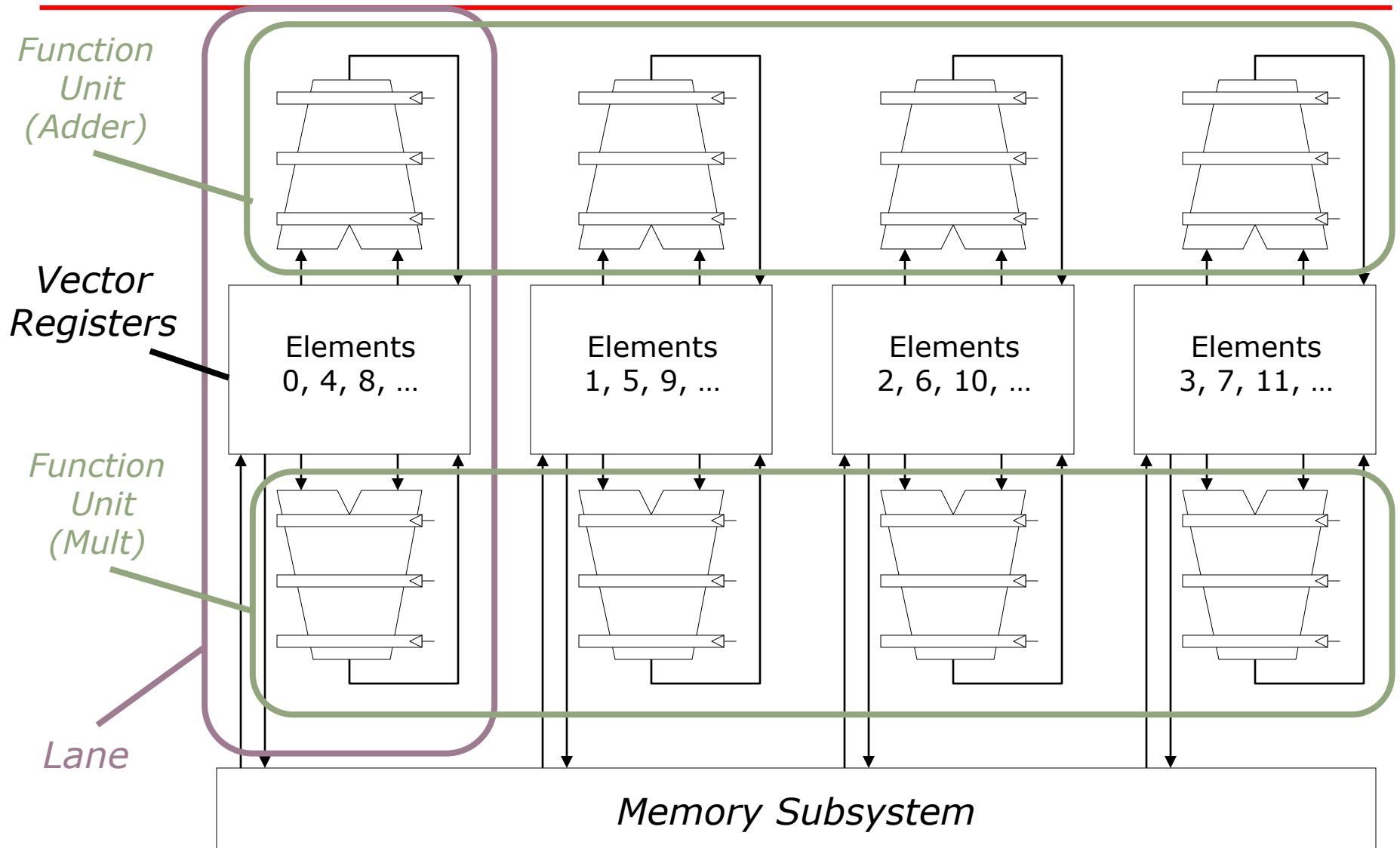


*Execution using
four pipelined
functional units*

A[24] B[24] A[25] B[25] A[26] B[26] A[27] B[27]
A[20] B[20] A[21] B[21] A[22] B[22] A[23] B[23]
A[16] B[16] A[17] B[17] A[18] B[18] A[19] B[19]
A[12] B[12] A[13] B[13] A[14] B[14] A[15] B[15]



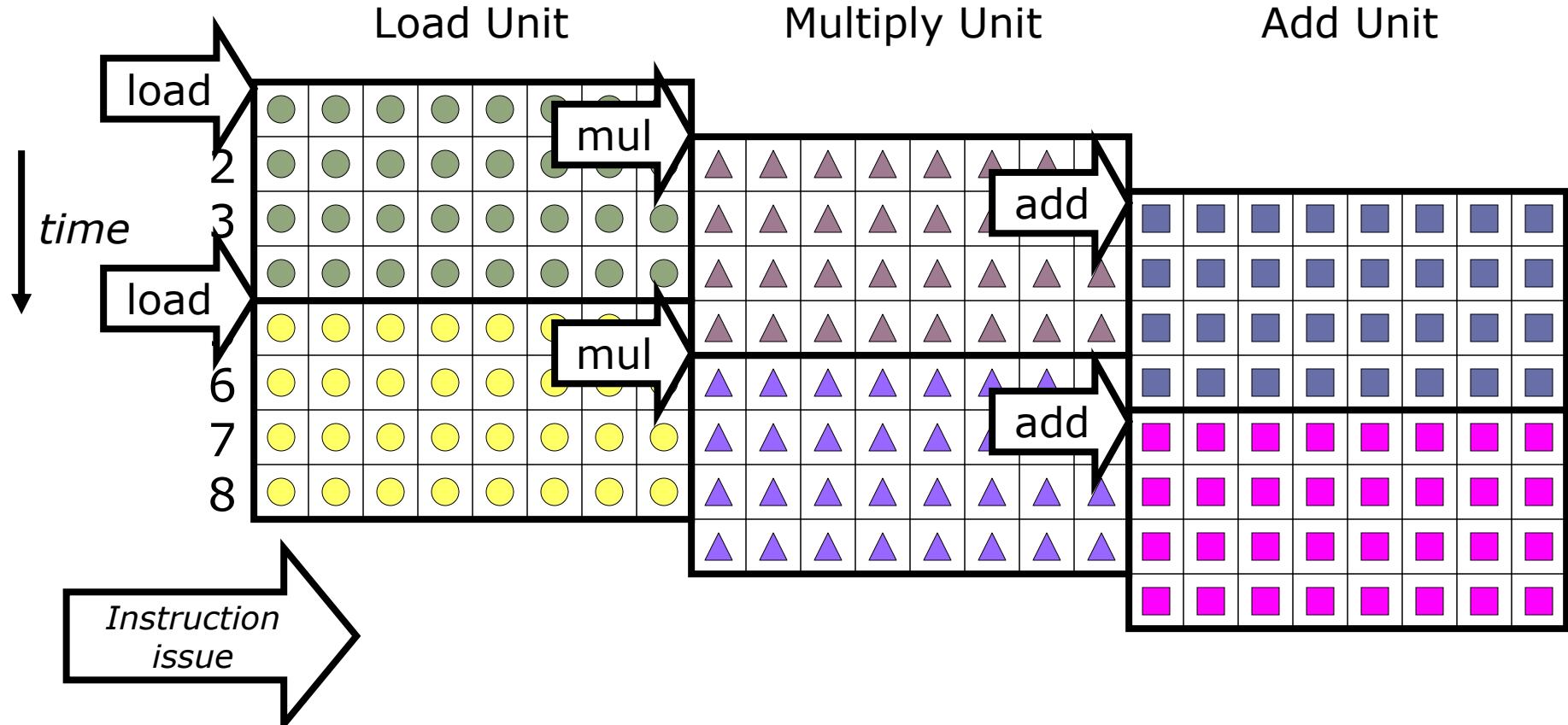
Vector Unit Structure



Vector Instruction Parallelism

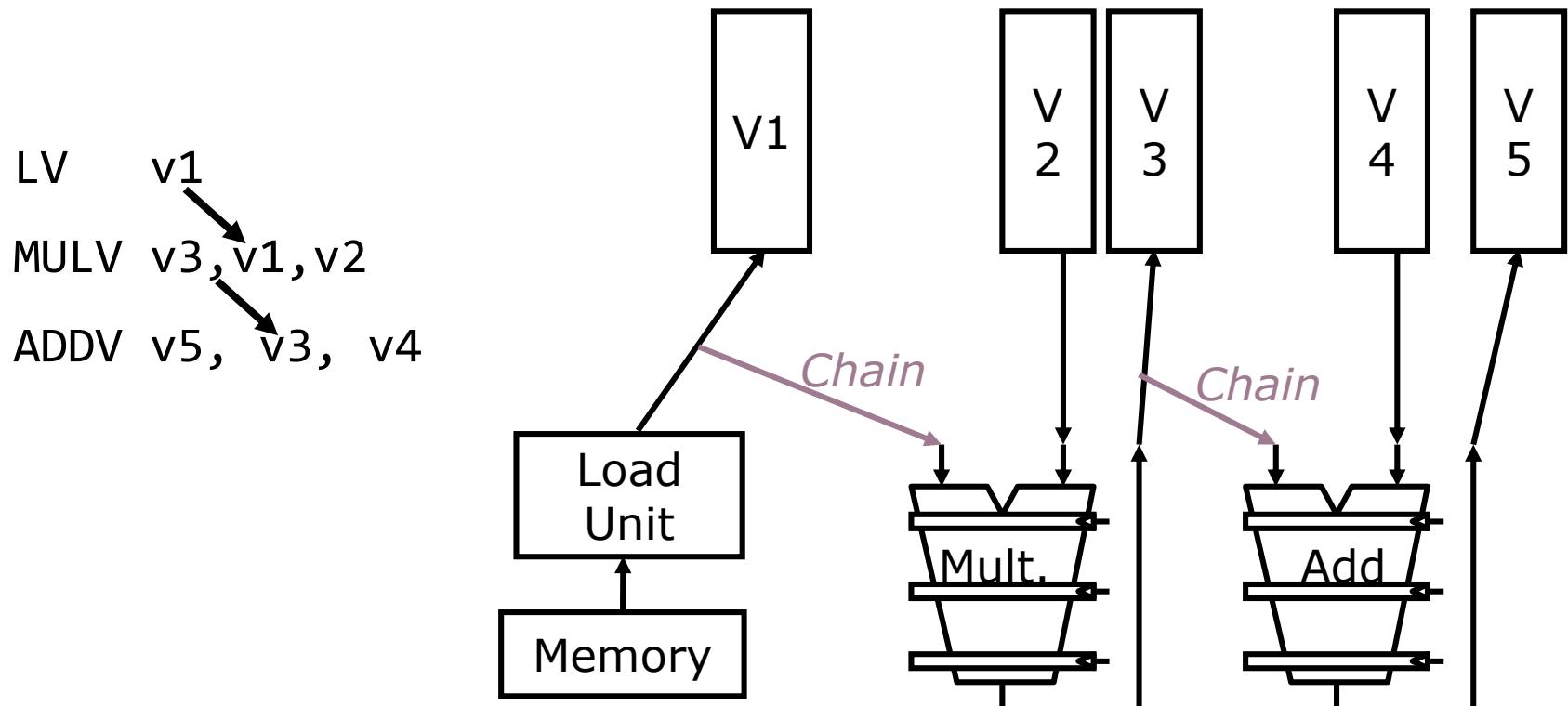
Can overlap execution of multiple vector instructions

- example machine has 32 elements per vector register and 8 lanes



Vector Chaining

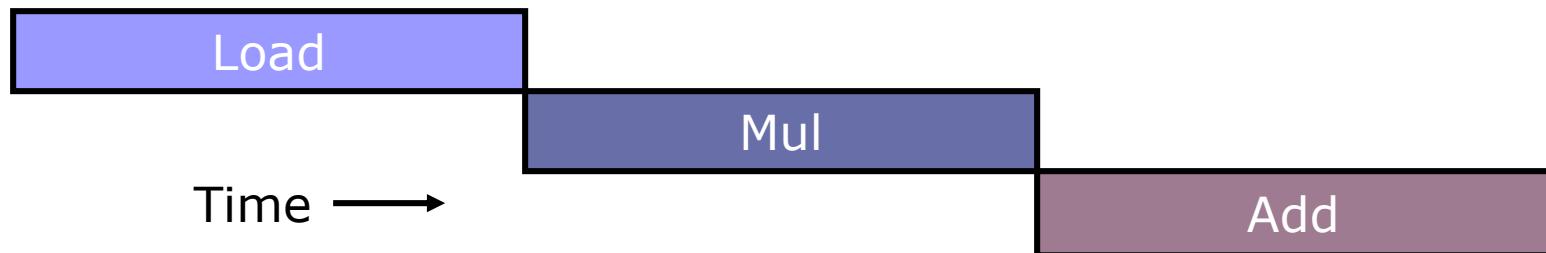
Problem: Long latency for RAW register dependencies



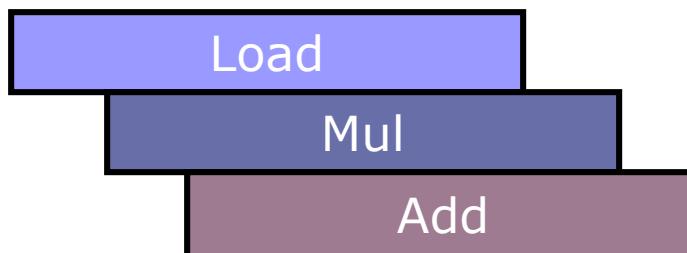
- Vector version of register bypassing
 - introduced with Cray-1

Vector Chaining Advantage

- Without chaining, must wait for last element of result to be written before starting dependent instruction



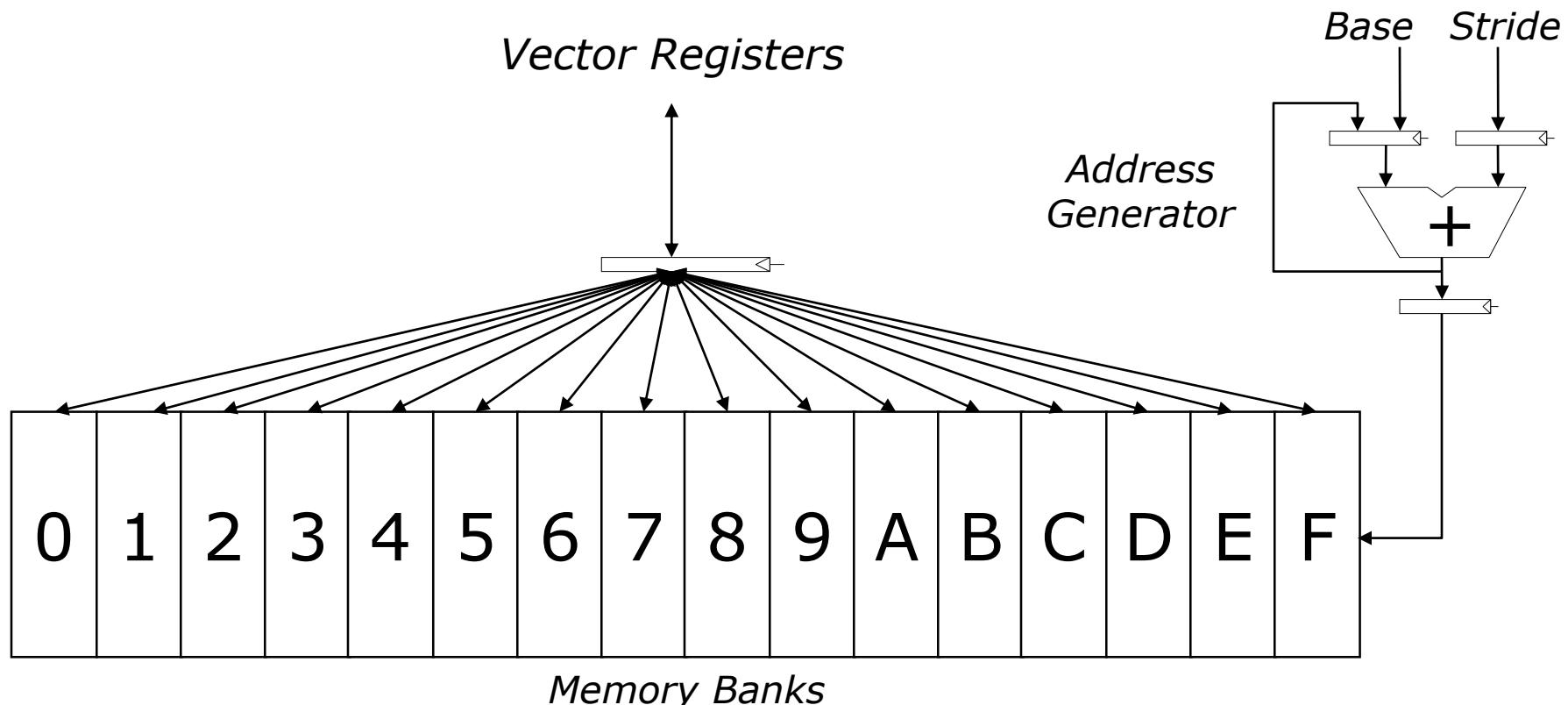
- With chaining, can start dependent instruction as soon as first result appears



Vector Memory System

Cray-1: 16 banks, 4 cycle bank busy time, 12 cycle latency

- *Bank busy time*: Cycles between accesses to same bank
- Allows 16 parallel accesses (if data in different banks)

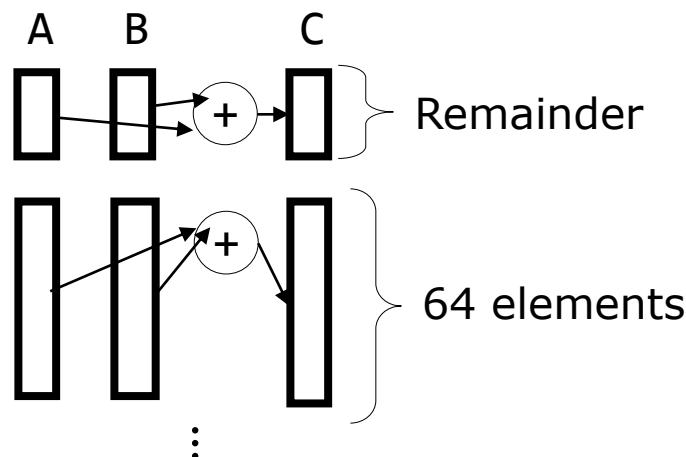


Vector Stripmining

Problem: Vector registers have finite length

Solution: Break loops into pieces that fit in registers, "*Strip mining*"

```
for (i = 0; i < N; i++)  
    C[i] = A[i] + B[i];
```



```
ANDI R1, N, 63      # N mod 64  
MTC1 VLR, R1        # Do remainder  
loop:  
    LV V1, RA  
    DSLL R2, R1, 3    # Multiply by 8  
    DADDU RA, RA, R2  # Bump pointer  
    LV V2, RB  
    DADDU RB, RB, R2  
    ADDV.D V3, V1, V2  
    SV V3, RC  
    DADDU RC, RC, R2  
    DSUBU N, N, R1    # Subtract elements  
    LI R1, 64  
    MTC1 VLR, R1        # Reset full length  
    BGTZ N, loop      # Any more to do?
```

Vector Conditional Execution

Problem: Want to vectorize loops with conditional code:

```
for (i = 0; i < N; i++)
    if (A[i] > 0) then
        A[i] = B[i];
```

Solution: Add vector *mask* (or *flag*) registers

- vector version of predicate registers, 1 bit per element

...and *maskable* vector instructions

- vector operation becomes NOP at elements where mask bit is clear

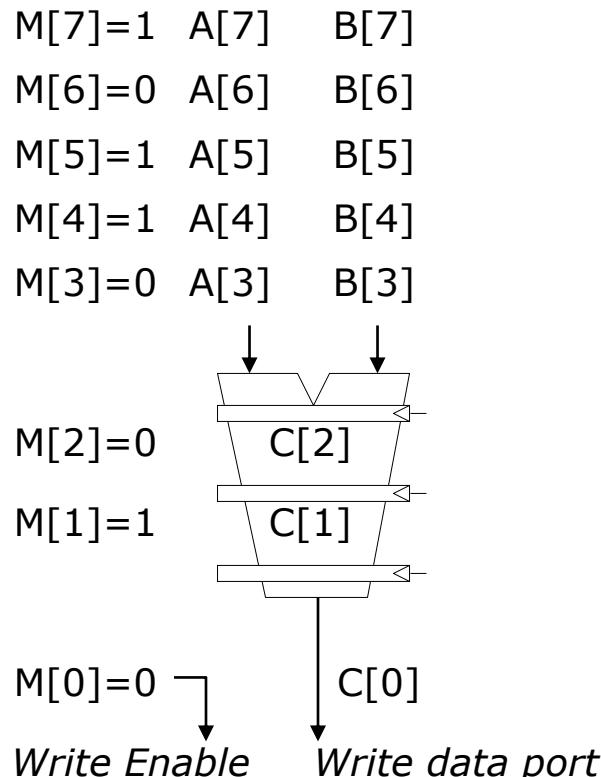
Code example:

```
CVM                      # Turn on all elements
LV vA, rA                 # Load entire A vector
SGTVS.D vA, F0            # Set bits in mask register where A>0
LV vA, rB                 # Load B vector into A under mask
SV vA, rA                 # Store A back to memory under mask
```

Masked Vector Instructions

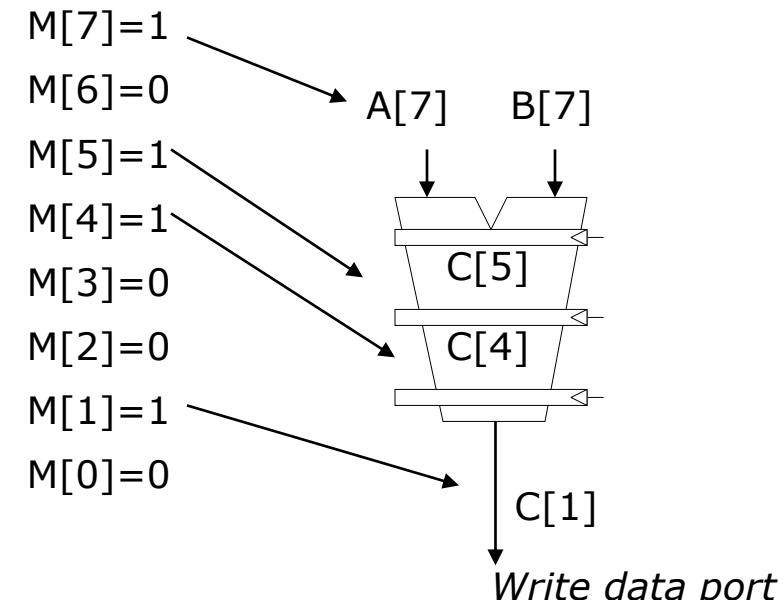
Simple implementation

- execute all N operations, turn off result writeback according to mask



Density-time implementation

- scan mask vector and only execute elements with non-zero masks



$$C[i] = A[i] + B[i]$$

Vector Scatter/Gather

Want to vectorize loops with indirect accesses:

```
for (i = 0; i < N; i++)
    A[i] = B[i] + C[D[i]]
```

Indexed load instruction (Gather)

```
LV vD, rD          # Load indices in D vector
LVI vC, rC, vD    # Load indirect from rC base
LV vB, rB          # Load B vector
ADDV.D vA, vB, vC # Do add
SV vA, rA          # Store result
```

Is this a correct translation?

Vector Scatter/Gather

Scatter example:

```
for (i = 0; i < N; i++)
    A[B[i]]++;
```

Is the following a correct translation?

```
LV vB, rB          # Load indices in B vector
LVI vA, rA, vB    # Gather initial A values
ADDV vA, vA, 1    # Increment
SVI vA, rA, vB    # Scatter incremented values
```

A Later-Generation Vector Super: NEC SX-6 (2003)

- CMOS Technology
 - 500 MHz CPU, fits on single chip
 - SDRAM main memory (up to 64GB)
- Scalar unit
 - 4-way superscalar
 - with out-of-order and speculative execution
 - 64KB I-cache and 64KB data cache
- Vector unit
 - 8 foreground VRegs + 64 background VRegs (256x64-bit elements/VReg)
 - 1 multiply unit, 1 divide unit, 1 addshift unit, 1 logical unit, 1 mask unit
 - 8 lanes (8 GFLOPS peak, 16 FLOPS/cycle)
 - 1 load & store unit (32x8 byte accesses/cycle)
 - 32 GB/s memory bandwidth per processor
- SMP structure
 - 8 CPUs connected to memory through crossbar
 - 256 GB/s shared memory bandwidth (4096 interleaved banks)



Multimedia/SIMD Extensions

- Short vectors added to existing general-purpose ISAs
- Idea first used on Lincoln Labs TX-2 computer in 1957, with 36b datapath split into 2x18b or 4x9b
- Recent incarnations initially reused existing registers
 - e.g., 64-bit registers split into 2x32bits or 4x16bits or 8x8bits
- Trend towards larger vector support in microprocessors
 - e.g. x86:
 - MMX (64 bits)
 - SSE (128 bits)
 - AVX (256 bits)
 - AVX-512 (512 bits/masks)

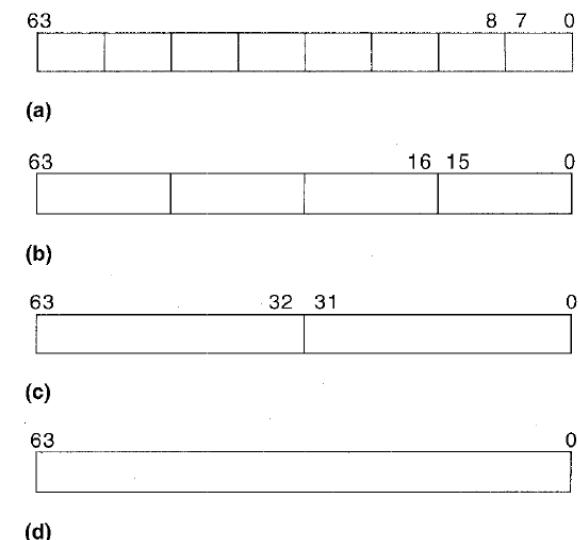


Figure 1. MMX technology data types: packed byte (a), packed word (b), packed doubleword (c), and quadword (d).

Intel SIMD Evolution

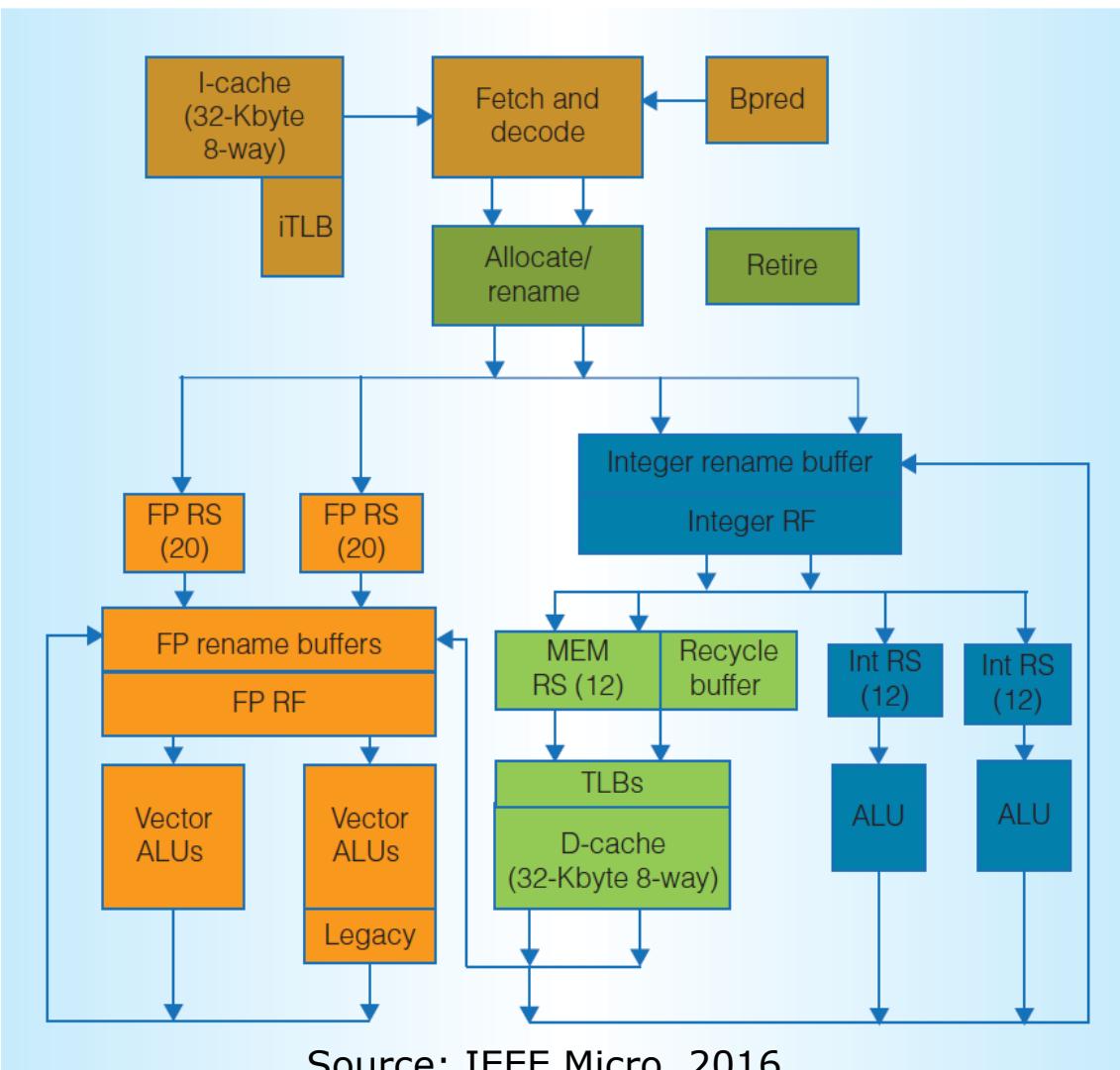
Implementations:

- Intel MMX (1996) – 64bits
 - Eight 8-bit integer ops, or
 - Four 16-bit integer ops
 - Two 32-bit integer ops
- Streaming SIMD Extensions (SSE) (1999) – 128bits
 - Four 32-bit integer ops (and smaller integer types)
 - Four 32-bit integer/fp ops, or
 - Two 64-bit integer/fp ops
- Advanced Vector Extensions (2010) – 256bits
 - Four 64-bit integer/fp ops (and smaller fp types)
- AVX-512 (2017) – 512bits
 - New instructions: scatter/gather, mask registers

Multimedia Extensions vs Vectors

- Limited instruction set
 - No vector length control
 - Usually no masks
 - Up until recently, no strided load/store or scatter/gather
 - Unit-stride loads must be aligned to 64/128-bits
- Limited vector register length
 - requires superscalar dispatch to keep units busy
 - loop unrolling to hide latencies increases register pressure
- Trend towards fuller vector support
 - Better support for misaligned memory accesses
 - Support of double-precision (64-bit floating-point)
 - Support for masked operations

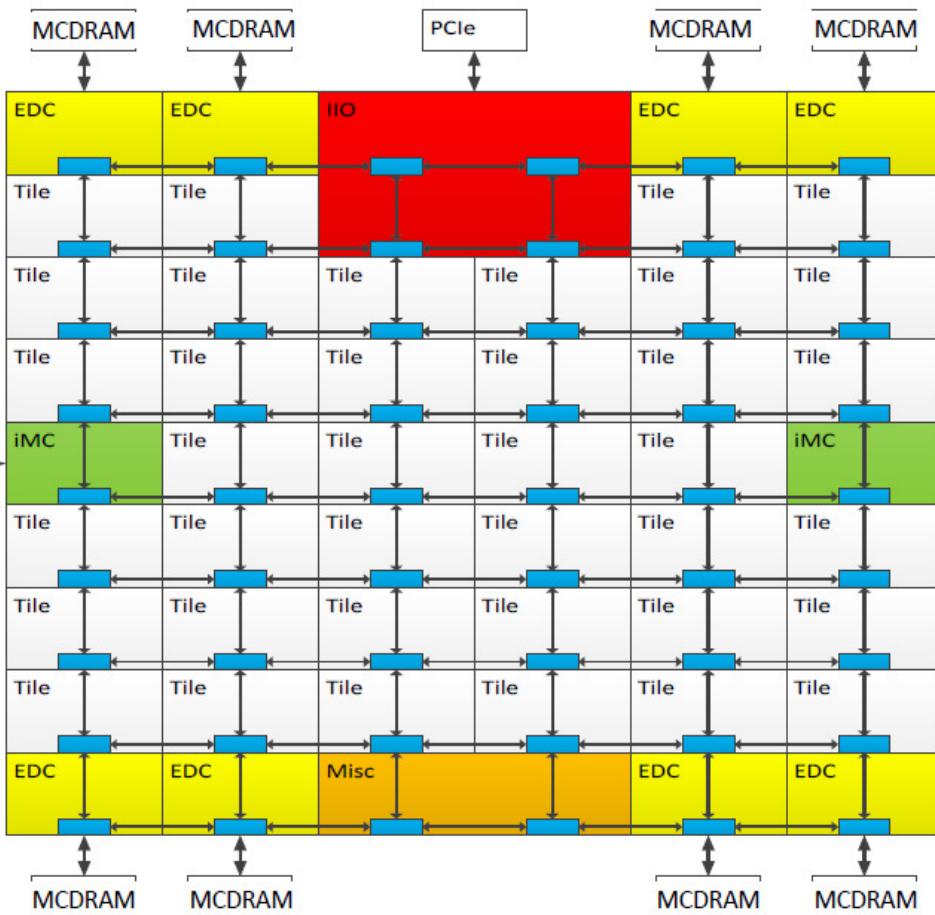
Knights Landing (KNL) CPU



Source: IEEE Micro, 2016

- 2-wide decode/retire
- 6-wide execute
- 72-entry ROB
- 64B cache ports
- 2 load/1 store ports
- Fast unaligned access
- Fast scatter/gather
- OoO int/fp RS
- In-order mem RS
- 4 thread SMT
- Many shared resources
 - ROB, rename buffer, RS: dynamically partitioned
- Several thread choosers

Knights Landing (KNL) Mesh



Mesh of Rings

- Rows/columns (half) ring
- YX routing
- Message arbitration on:
 - Injection
 - Turns

Cache Coherent Interconnect

- MESIF protocol
- Distributed directory
 - to filter snoops

Partitioning modes

- All-to-all
- Quadrant
- Sub-NUMA

Source: IEEE Micro, 2016

Thank you!

Next Lecture: GPUs

Graphics Processing Units (GPUs)

Daniel Sanchez
Computer Science & Artificial Intelligence Lab
M.I.T.

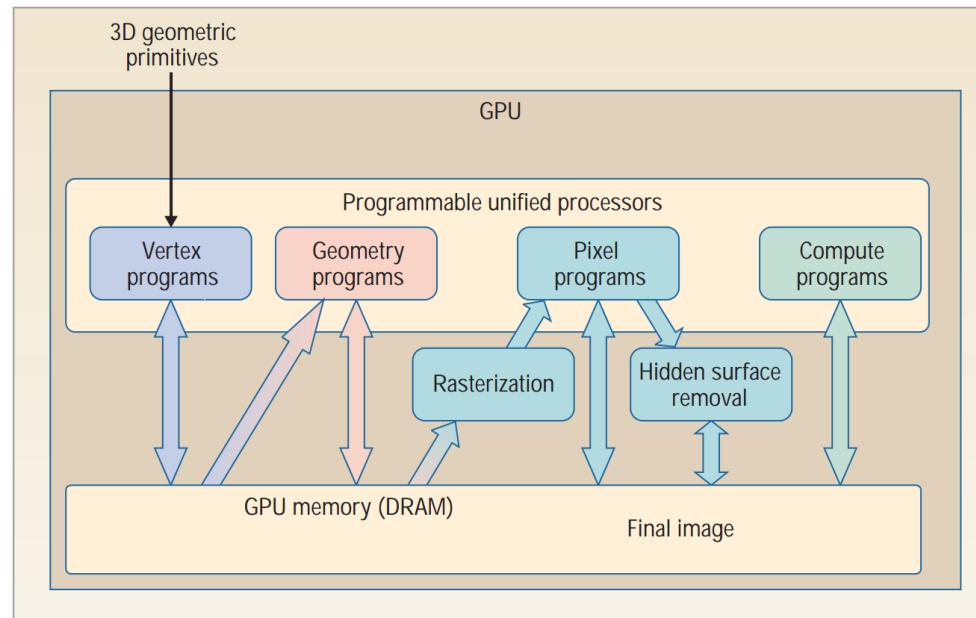
Why Study GPUs?

- Very successful commodity accelerator/co-processor
- GPUs combine two strategies to increase efficiency
 - Massive parallelism
 - Specialization
- Illustrates tension between performance and programmability in accelerators

Graphics Processors Timeline

- Until mid-90s
 - Most graphics processing in CPU
 - VGA controllers used to accelerate some display functions
- Mid-90s to mid-2000s
 - Fixed-function accelerators for 2D and 3D graphics
 - triangle setup & rasterization, texture mapping & shading
 - Programming:
 - OpenGL and DirectX APIs

Contemporary GPUs



Luebke and Humphreys, 2007

- Modern GPUs
 - Some fixed-function hardware (texture, raster ops, ...)
 - Plus programmable data-parallel multiprocessors
 - Programming:
 - OpenGL/DirectX
 - Plus more general purpose languages (CUDA, OpenCL, ...)

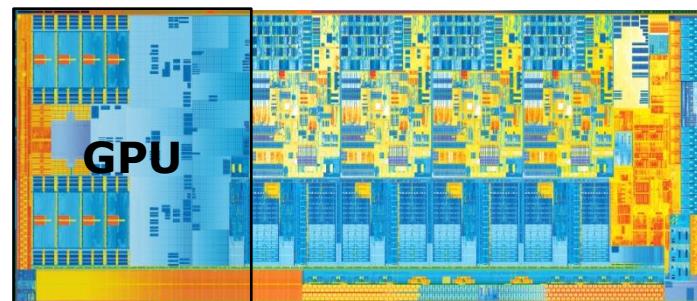
GPUs in Modern Systems

- Discrete GPUs
 - PCIe-based accelerator
 - Separate GPU memory
- Integrated GPUs
 - CPU and GPU on same die
 - Shared main memory and last-level cache

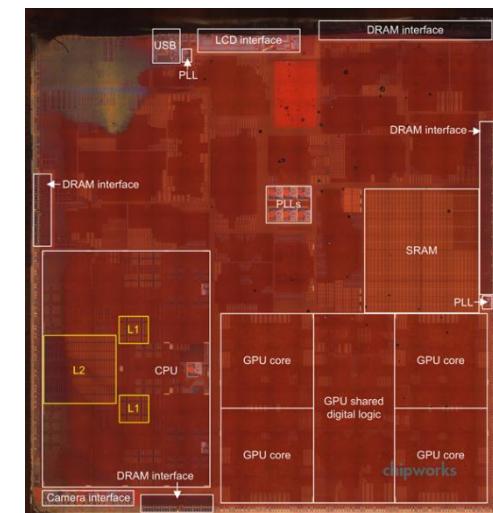


Nvidia Kepler

- Pros/cons?



Intel Ivy Bridge, 22nm 160mm²

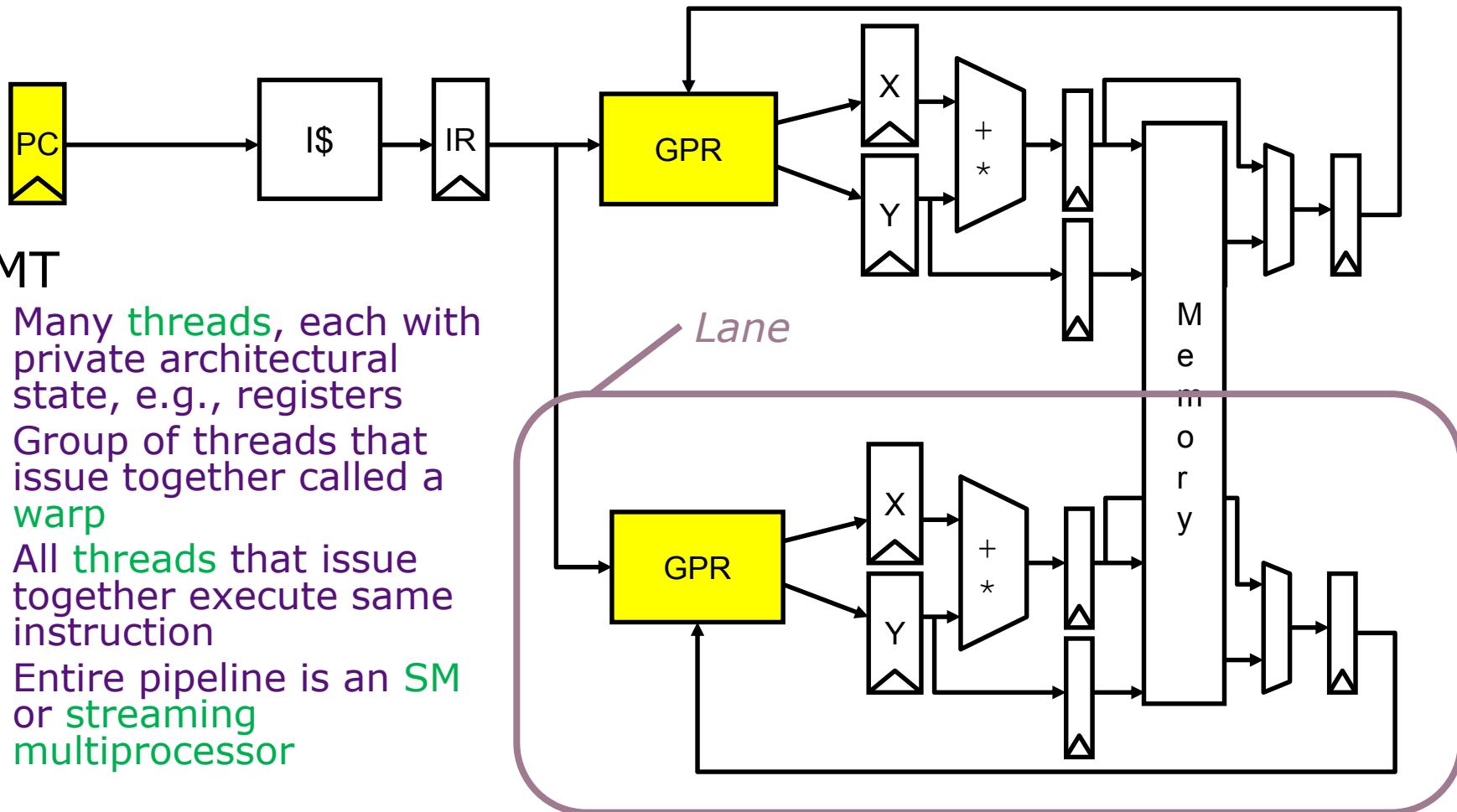


Apple A7, 28nm
TSMC, 102mm²

Single Instruction Multiple Thread

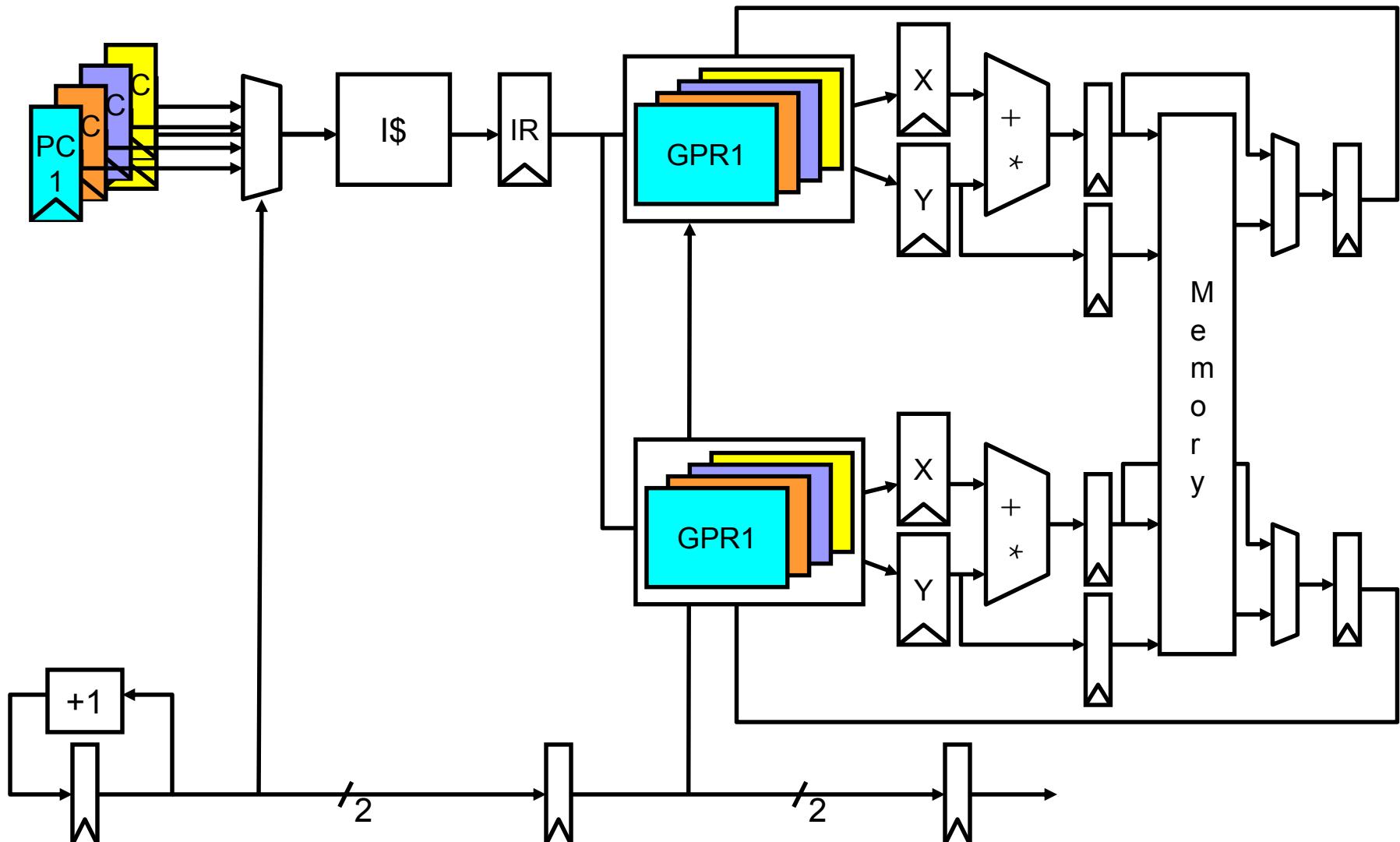
SIMT

- Many **threads**, each with private architectural state, e.g., registers
- Group of threads that issue together called a **warp**
- All **threads** that issue together execute same instruction
- Entire pipeline is an **SM** or **streaming multiprocessor**

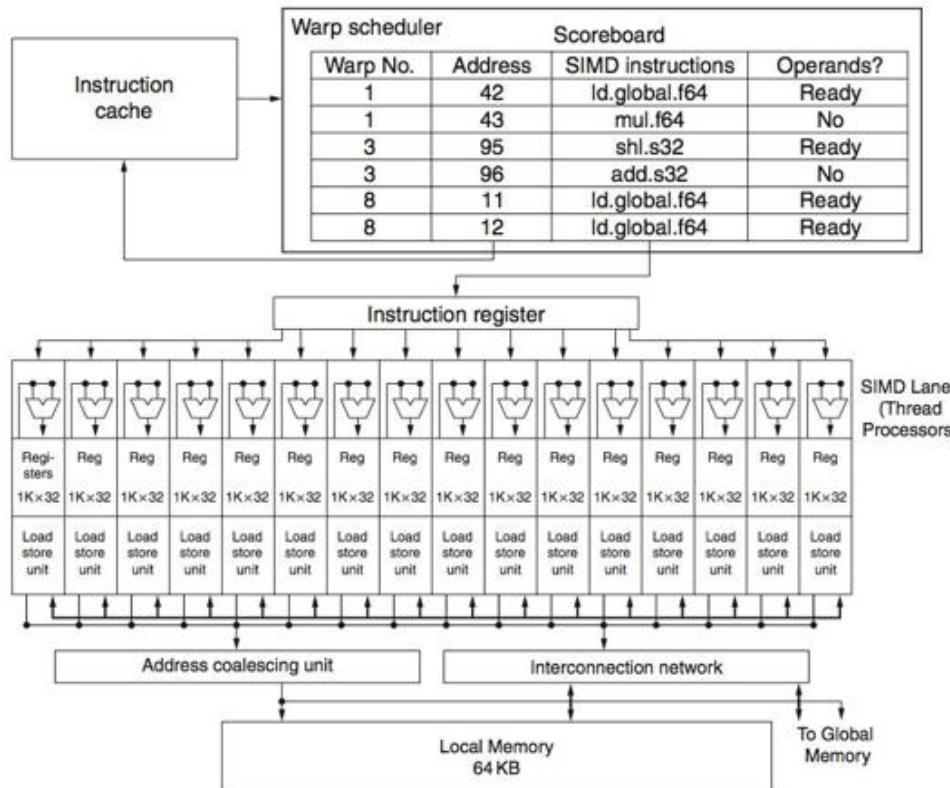


green-> Nvidia terminology

Multithreading + Single Instruction Multiple Thread



Streaming Multiprocessor Overview



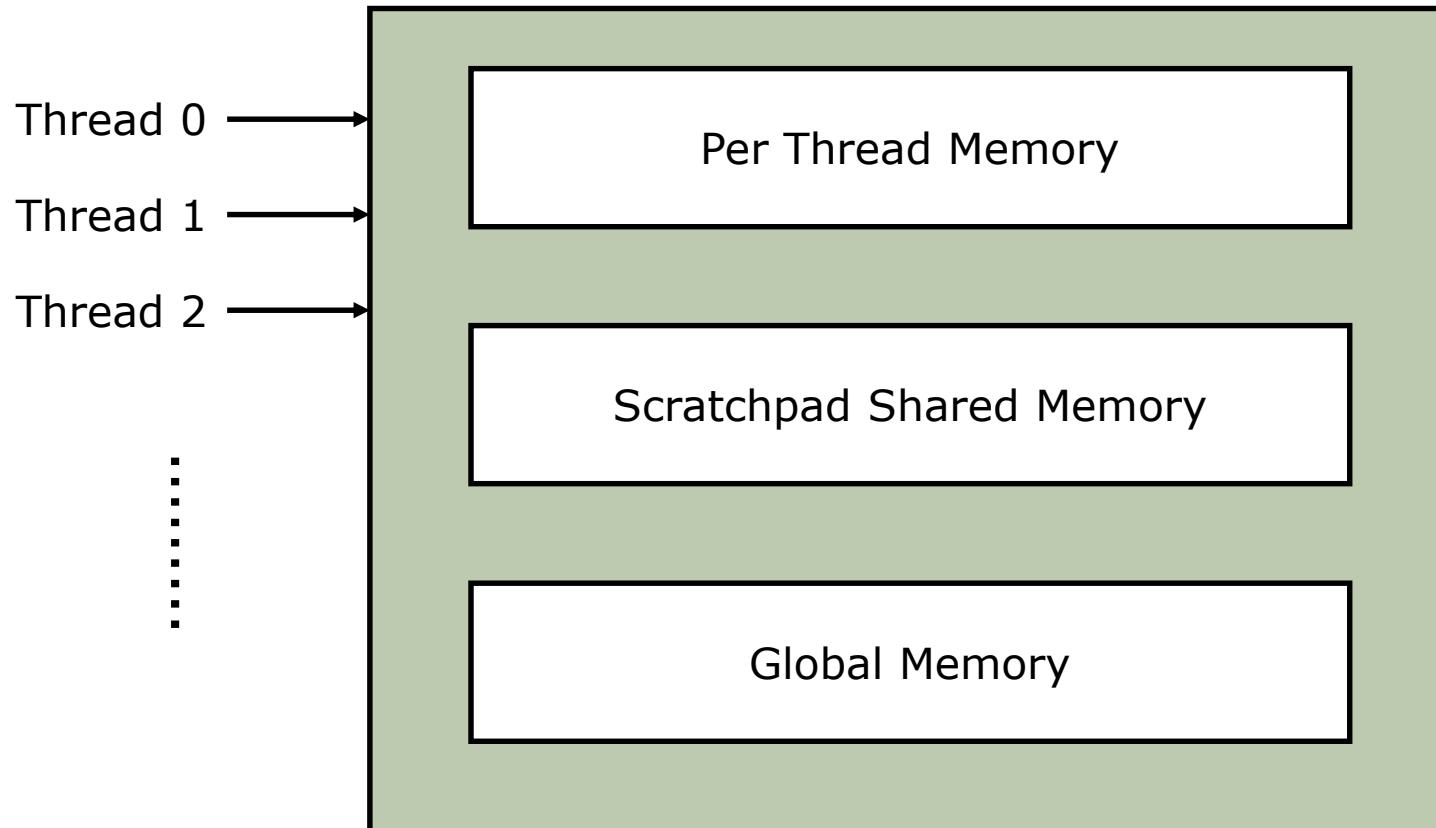
- Each SM supports 10s of warps (e.g., 64 in Kepler) with 32 threads/warp
- Fetch 1 instr/cycle
- Issue 1 ready instr/cycle
 - Simple scoreboard: all warp elements must be ready
- Instruction broadcast to all lanes
- Multithreading is the main latency-hiding mechanism

What average latency is needed to keep busy?

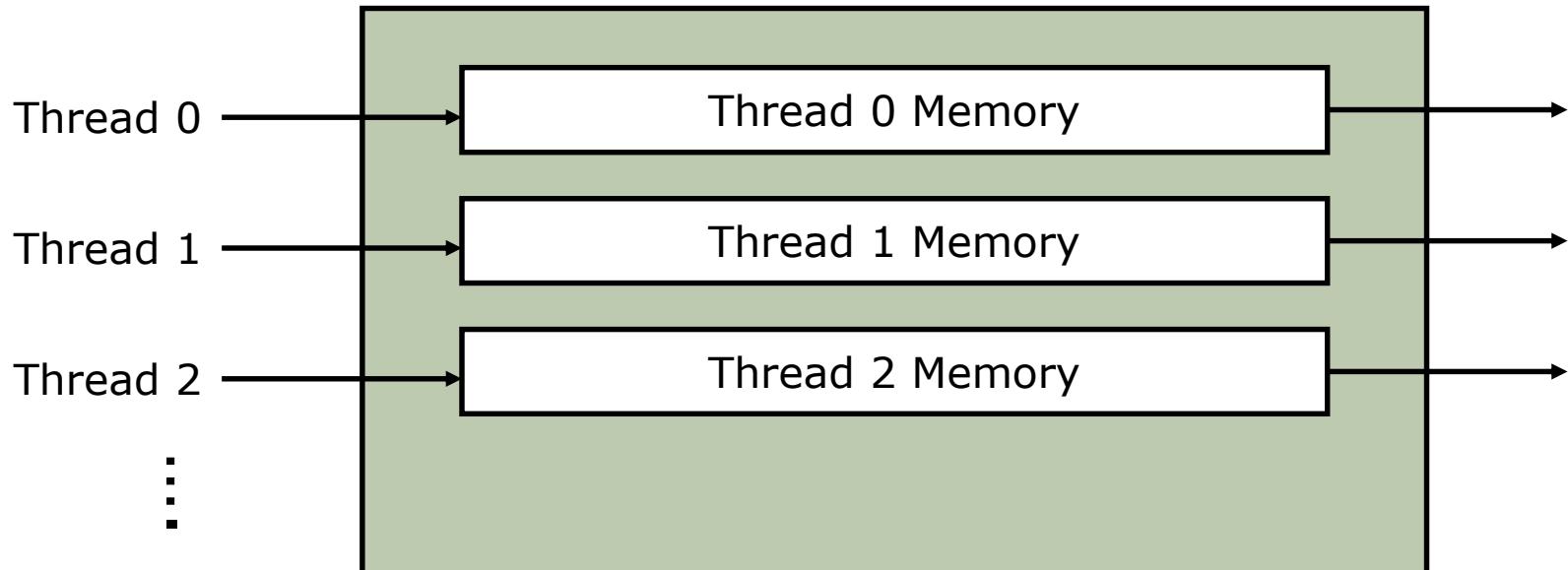
Context Size vs Number of Contexts

- SMs support a variable number of contexts based on required registers (and shared memory)
 - Few large contexts → Fewer register spills
 - Many small contexts → More latency tolerance
 - Choice left to the compiler
- Example: Kepler supports up to 64 warps
 - Max: 64 warps @ ≤ 32 registers/thread
 - Min: 8 warps @ 256 registers/thread

Many Memory Types

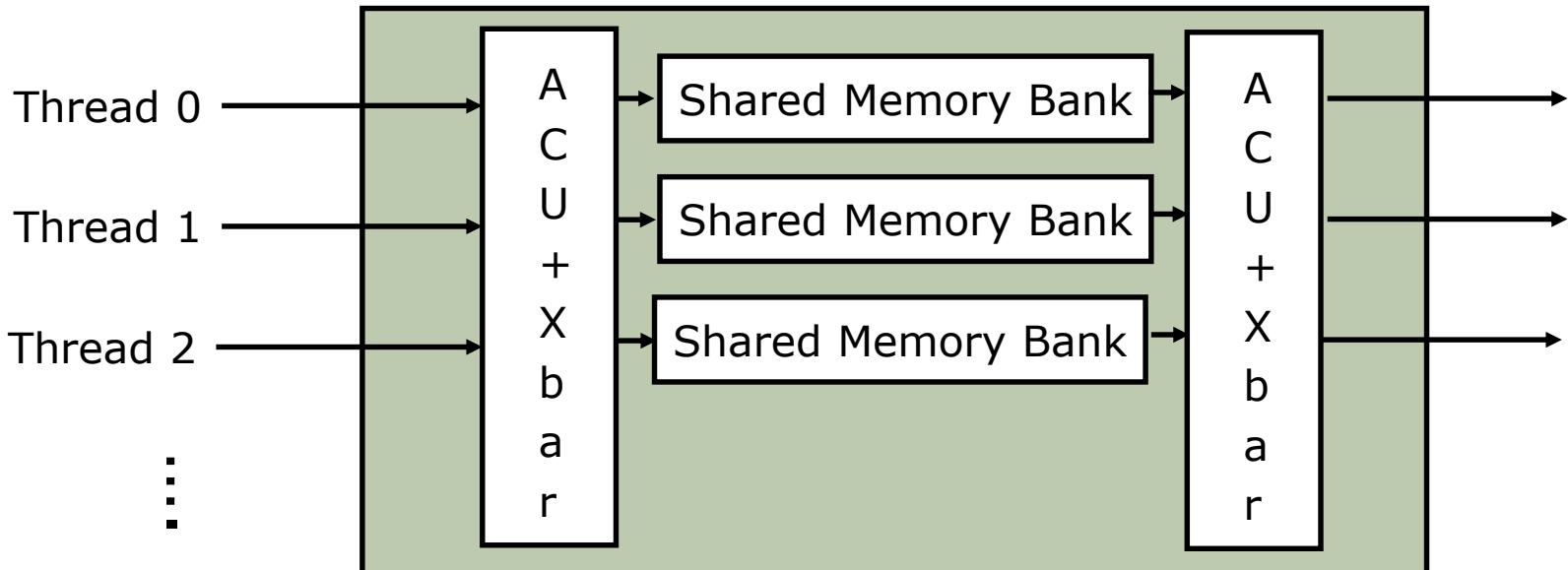


Private Per Thread Memory



- Private memory
 - No cross-thread sharing
 - Small, fixed size memory
 - Can be used for constants
 - Multi-bank implementation (can be in global memory)

Shared Scratchpad Memory

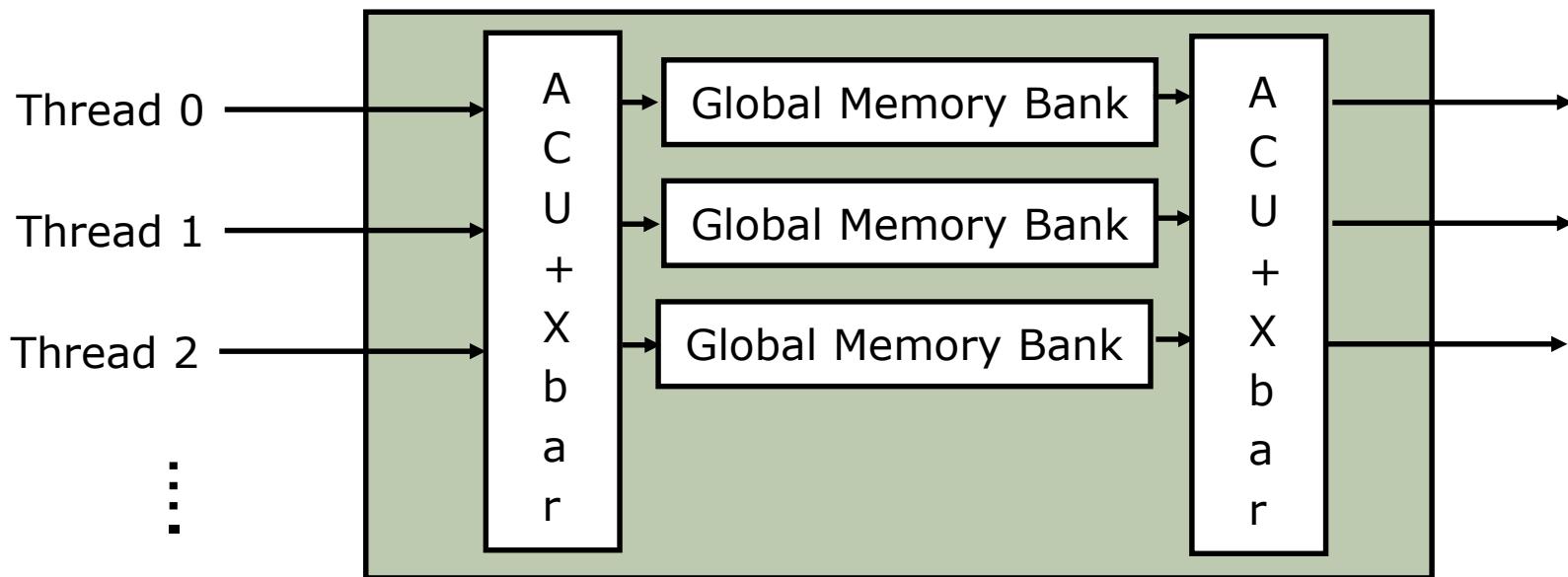


- Shared scratchpad memory (threads share data)
 - Small, fixed size memory (16K-64K per SM = 'core')
 - Banked for high bandwidth
 - Fed with address coalescing unit (ACU) + crossbar
 - ACU can buffer/coalesce requests

Memory Access Divergence

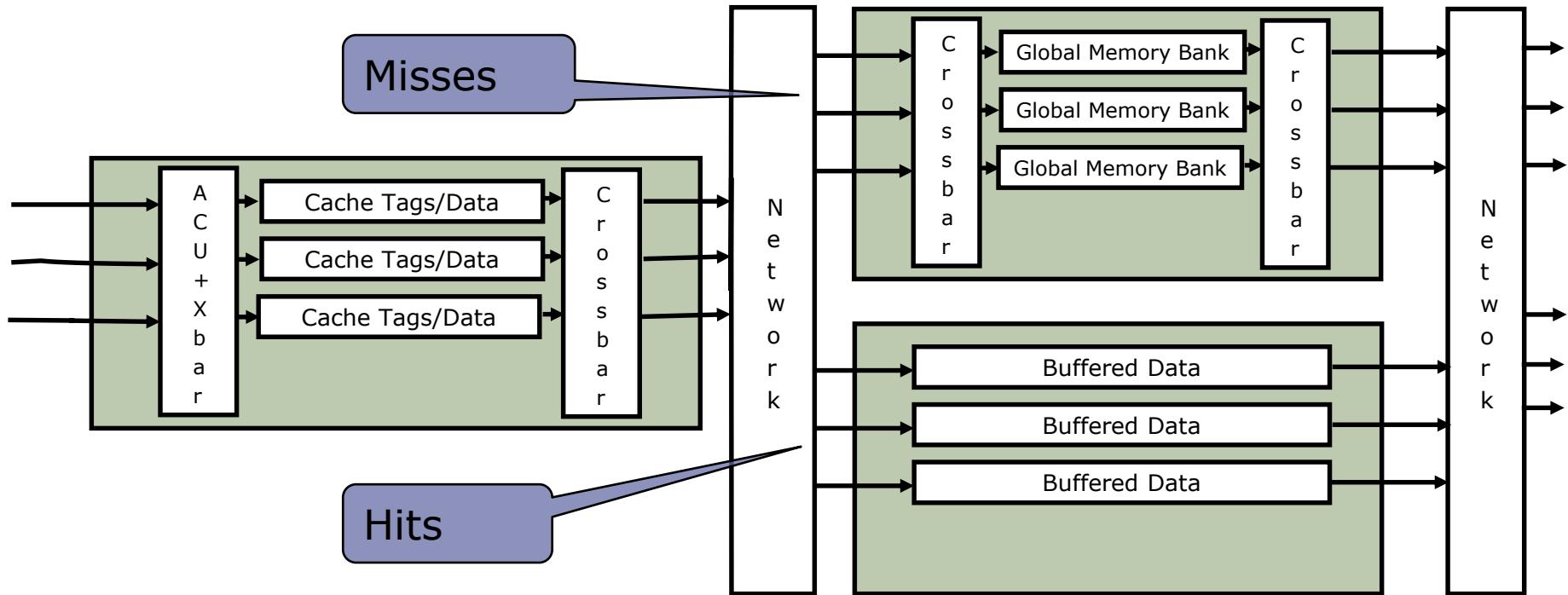
- All loads are gathers, all stores are scatters
- Address coalescing unit detects sequential and strided patterns, coalesces memory requests, but complex patterns can result in multiple lower bandwidth requests (memory divergence)
- Writing efficient GPU code requires most accesses to not conflict, even though programming model allows arbitrary patterns!

Shared Global Memory



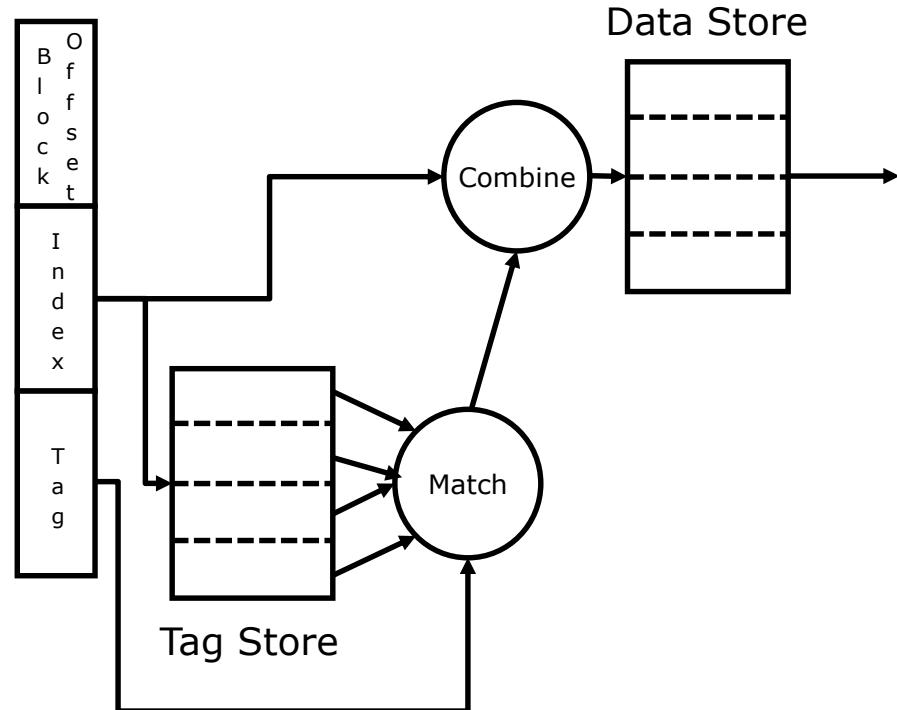
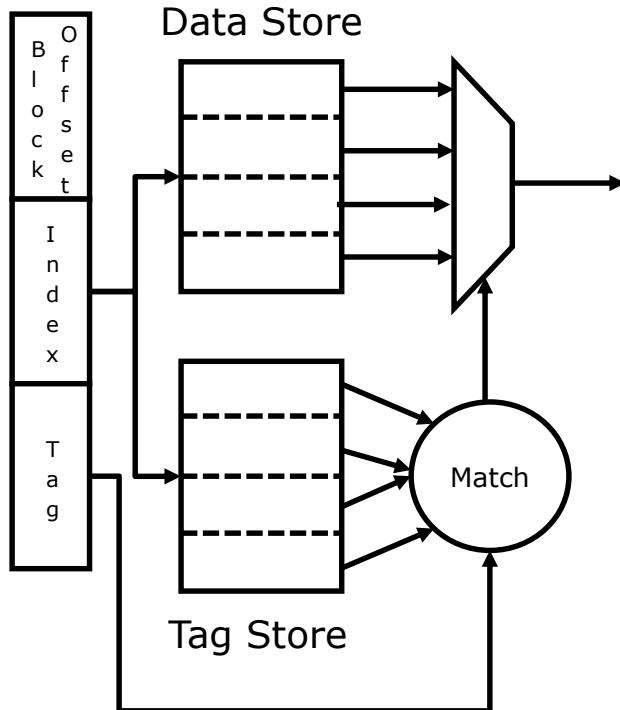
- Shared global memory
 - Large shared memory
 - Will suffer also from memory divergence

Shared Global Memory



- Memory hierarchy with caches
 - Cache to save memory bandwidth
 - Caches also enable compression/decompression of data

Serialized cache access



- Trade latency for power/flexibility
 - Only access data bank that contains data
 - Facilitate more sophisticated cache organizations
 - e.g., greater associativity

Handling Branch Divergence

- Similar to vector processors, but masks are handled internally
 - Per-warp stack stores PCs and masks of non-taken paths
- On a conditional branch
 - Push the current mask onto the stack
 - Push the mask and PC for the non-taken path
 - Set the mask for the taken path
- At the end of the taken path
 - Pop mask and PC for the non-taken path and execute
- At the end of the non-taken path
 - Pop the original mask before the branch instruction
- If a mask is all zeros, skip the block

Example: Branch Divergence

Assume 4 threads/warp,
initial mask 1111

```
if (m[i] != 0) {  
    if (a[i] > b[i]) {  
        y[i] = a[i] - b[i];  
    } else {  
        y[i] = b[i] - a[i];  
    }  
} else {  
    y[i] = 0;  
}
```

- 1
- 2
- 3
- 4
- 5
- 6

- 1 Push mask 1111
Push mask 0011
Set mask 1100
- 2 Push mask 1100
Push mask 0100
Set mask 1000
- 3 Pop mask 0100
- 4 Pop mask 1100
- 5 Pop mask 0011
- 6 Pop mask 1111

Optimization for branches that all go same way?

Branch divergence and locking

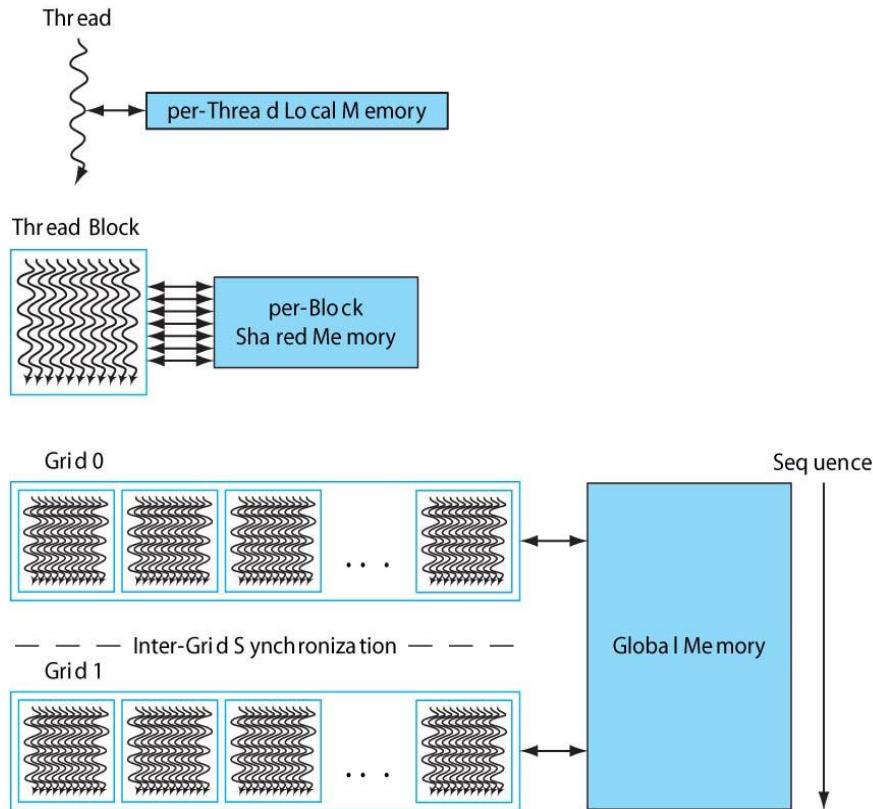
- Consider the following executing in multiple threads in a warp:

```
if (condition[i]) {  
    while (locked(map0[i])){}  
    lock(locks[map0[i]]);  
} else {  
    unlock(locks[map1[i]]);  
}
```

where i is a thread id and $\text{map0}[]$, $\text{map1}[]$ are permutations of thread ids.

What can go wrong here?

CUDA GPU Thread Model



- Single-program multiple data (SPMD) model
- Each context is a thread
 - Threads have registers
 - Threads have local memory
- Parallel threads packed in blocks
 - Blocks have shared memory
 - Threads synchronize with barrier
 - Blocks run to completion (or abort)
- Grids include independent blocks
 - May execute concurrently
 - Share global memory, but
 - Have limited inter-block synchronization

Code Example: DAXPY

C Code

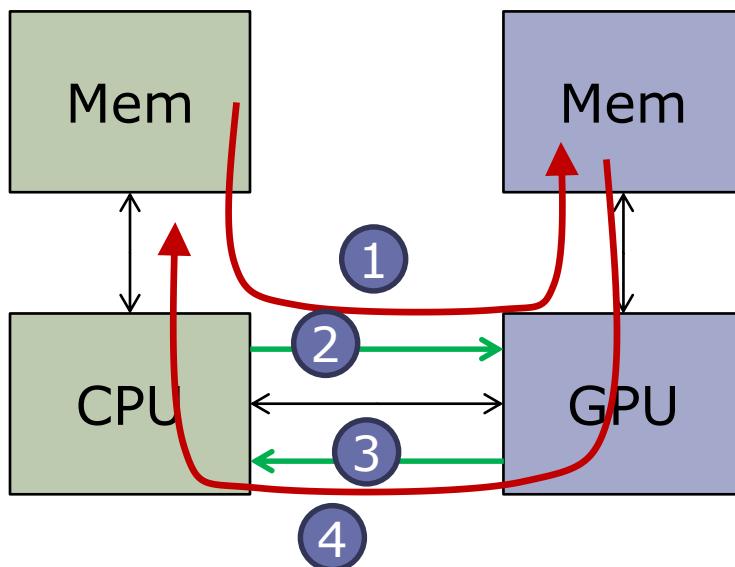
```
// Invoke DAXPY
daxpy(n, 2.0, x, y);
// DAXPY in C
void daxpy(int n, double a, double *x, double *y)
{
    for (int i = 0; i < n; ++i)
        y[i] = a*x[i] + y[i];
}
```

CUDA Code

```
// Invoke DAXPY with 256 threads per block
__host__
int nbblocks = (n+ 255) / 256;
daxpy<<<nbblocks, 256>>>(n, 2.0, x, y);
// DAXPY in CUDA
__device__
void daxpy(int n, double a, double *x, double *y)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n) y[i] = a*x[i] + y[i];
}
```

- CUDA code launches 256 threads per block
- CUDA vs vector terminology:
 - Thread = 1 iteration of scalar loop (1 element in vector loop)
 - Block = Body of vectorized loop (VL=256 in this example)
 - Grid = Vectorizable loop

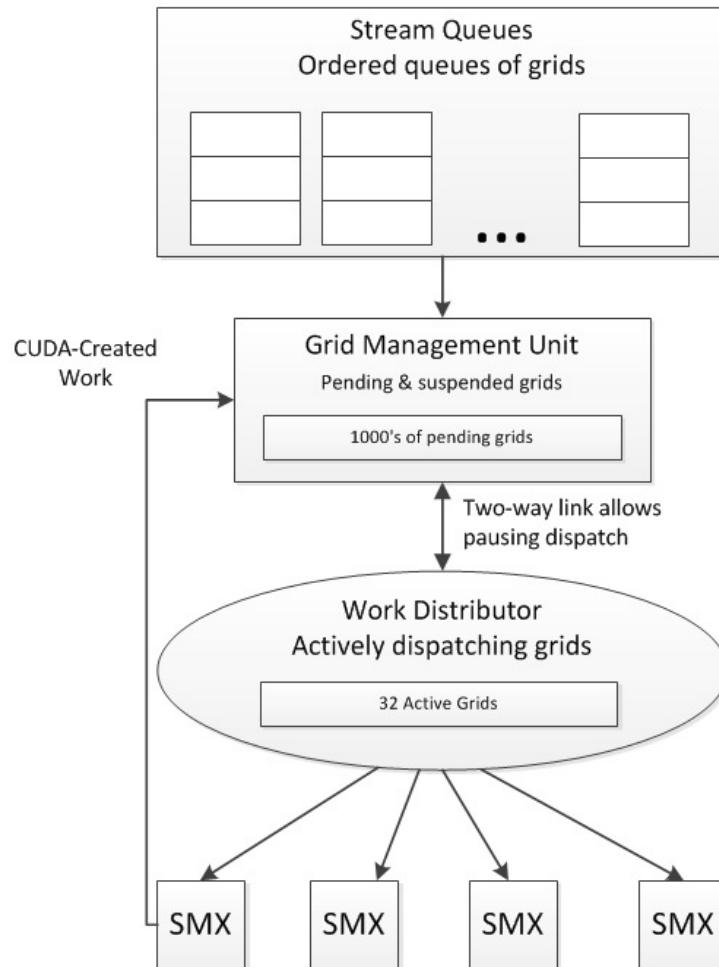
GPU Kernel Execution



- ① Transfer input data from CPU to GPU memory
- ② Launch kernel (grid)
- ③ Wait for kernel to finish (if synchronous)
- ④ Transfer results to CPU memory

- Data transfers can dominate execution time
- Integrated GPUs with unified address space
→ no copies, but CPU & GPU contend for memory

Hardware Scheduling



- Grids can be launched by CPU or GPU
 - Work from multiple CPU threads and processes
- HW unit schedules grids on SMs
 - Priority-based scheduling
- Multi-level scheduling
 - Limited number of active grids
 - More queued/paused

Synchronization

- Barrier synchronization within a thread block (`__syncthreads()`)
 - Tracking simplified by grouping threads into warps
 - Counter tracks number of warps that have arrived to barrier
- Atomic operations to global memory
 - Read-modify-write operations (add, exchange, compare-and-swap, ...)
 - Performed at the memory controller or at the L2
- Limited inter-block synchronization!
 - Can't wait for other blocks to finish

GPU ISA and Compilation

- GPU microarchitecture and instruction set change very frequently
- To achieve compatibility:
 - Compiler produces intermediate pseudo-assembler language (e.g., Nvidia PTX)
 - GPU driver JITs kernel, tailoring it to specific microarchitecture
- In practice, little performance portability
 - Code is often tuned to specific GPU architecture

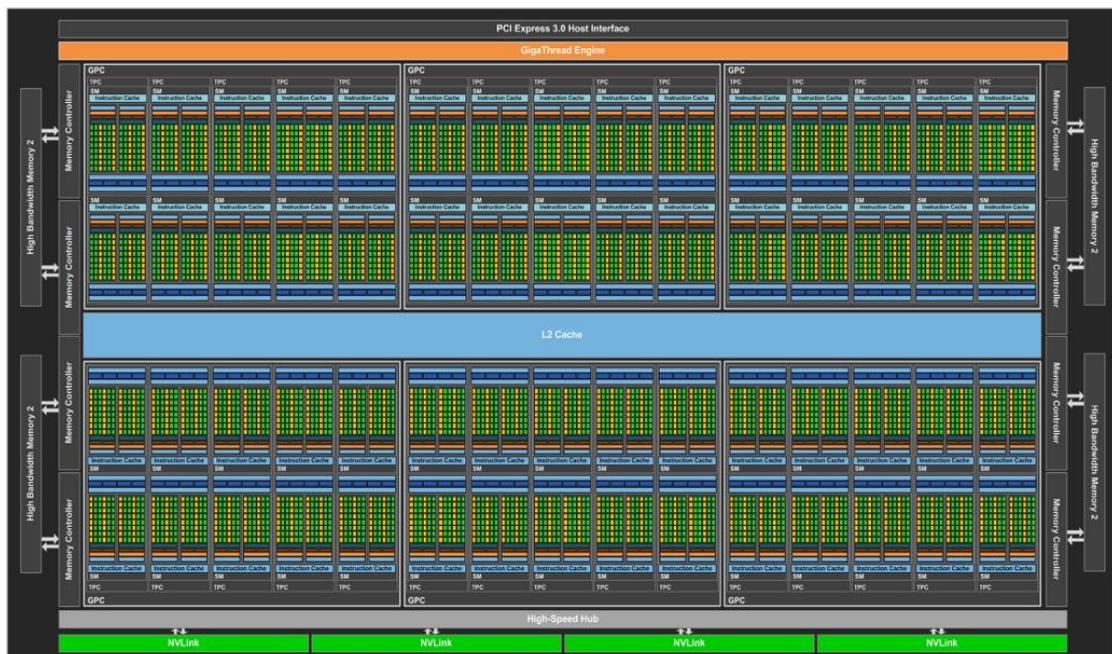
System-Level Issues

- Instruction semantics
 - Exceptions
- Scheduling
 - Each kernel is non-preemptive (but can be aborted)
 - Resource management and scheduling left to GPU driver, opaque to OS
- Memory management
 - First GPUs had no virtual memory
 - Recent support for basic virtual memory (protection among grids, no paging)
 - Host-to-device copies with separate memories (discrete GPUs)

GPU: Multithreaded Multicore Chip

- Example: Nvidia Pascal GP100 (2016)

- 60 streaming multiprocessors (SMs)
- 4MB Shared L2 cache
- 8 memory controllers
 - 720 GB/s (HBM2)
- Fixed-function logic for graphics (texture units, raster ops, ...)
- Scalability → change number of cores and memory channels
- Scheduling mostly controlled by hardware



Pascal Streaming Multiprocessor (SM)



- Execution units
 - 64 FUs (int and FP)
 - 16 load-store FUs
 - 16 special FUs (e.g., sqrt, sin, cos, ...)
- Memory structures
 - 64K 32-bit registers
 - 64KB shared memory
- Contexts
 - 2048 threads
 - 32 blocks

Vector vs GPU Terminology

Type	More descriptive name	Closest old term outside of GPUs	Official CUDA/NVIDIA GPU term	Book definition
Program abstractions	Vectorizable Loop	Vectorizable Loop	Grid	A vectorizable loop, executed on the GPU, made up of one or more Thread Blocks (bodies of vectorized loop) that can execute in parallel.
	Body of Vectorized Loop	Body of a (Strip-Mined) Vectorized Loop	Thread Block	A vectorized loop executed on a multithreaded SIMD Processor, made up of one or more threads of SIMD instructions. They can communicate via Local Memory.
	Sequence of SIMD Lane Operations	One iteration of a Scalar Loop	CUDA Thread	A vertical cut of a thread of SIMD instructions corresponding to one element executed by one SIMD Lane. Result is stored depending on mask and predicate register.
Machine object	A Thread of SIMD Instructions	Thread of Vector Instructions	Warp	A traditional thread, but it contains just SIMD instructions that are executed on a multithreaded SIMD Processor. Results stored depending on a per-element mask.
	SIMD Instruction	Vector Instruction	PTX Instruction	A single SIMD instruction executed across SIMD Lanes.
	Multithreaded SIMD Processor	(Multithreaded) Vector Processor	Streaming Multiprocessor	A multithreaded SIMD Processor executes threads of SIMD instructions, independent of other SIMD Processors.
	Thread Block Scheduler	Scalar Processor	Giga Thread Engine	Assigns multiple Thread Blocks (bodies of vectorized loop) to multithreaded SIMD Processors.
	SIMD Thread Scheduler	Thread scheduler in a Multithreaded CPU	Warp Scheduler	Hardware unit that schedules and issues threads of SIMD instructions when they are ready to execute; includes a scoreboard to track SIMD Thread execution.
Processing hardware	SIMD Lane	Vector Lane	Thread Processor	A SIMD Lane executes the operations in a thread of SIMD instructions on a single element. Results stored depending on mask.
	GPU Memory	Main Memory	Global Memory	DRAM memory accessible by all multithreaded SIMD Processors in a GPU.
	Private Memory	Stack or Thread Local Storage (OS)	Local Memory	Portion of DRAM memory private to each SIMD Lane.
	Local Memory	Local Memory	Shared Memory	Fast local SRAM for one multithreaded SIMD Processor, unavailable to other SIMD Processors.
Memory hardware	SIMD Lane Registers	Vector Lane Registers	Thread Processor Registers	Registers in a single SIMD Lane allocated across a full thread block (body of vectorized loop).

[H&P5, Fig 4.25]

Thank you!

*Next Lecture:
Transactional Memory*