ECE 552 / CPS 550
Advanced Computer Architecture I

Lecture 19
Summary

Benjamin Lee
Electrical and Computer Engineering
Duke University

www.duke.edu/~bcl15
www.duke.edu/~bcl15/class/class_ece252fall11.html

ECE552 Administrivia

7 December 2012

- Please submit project reports to Sakai by midnight

Final Exam

- Wednesday, Dec 12, 2-5pm
- Closed book, closed notes exam
- Cumulative, with emphasis on latter half.

- 6-7 Questions
- 1/3 on earlier material
- 2/3 on later material

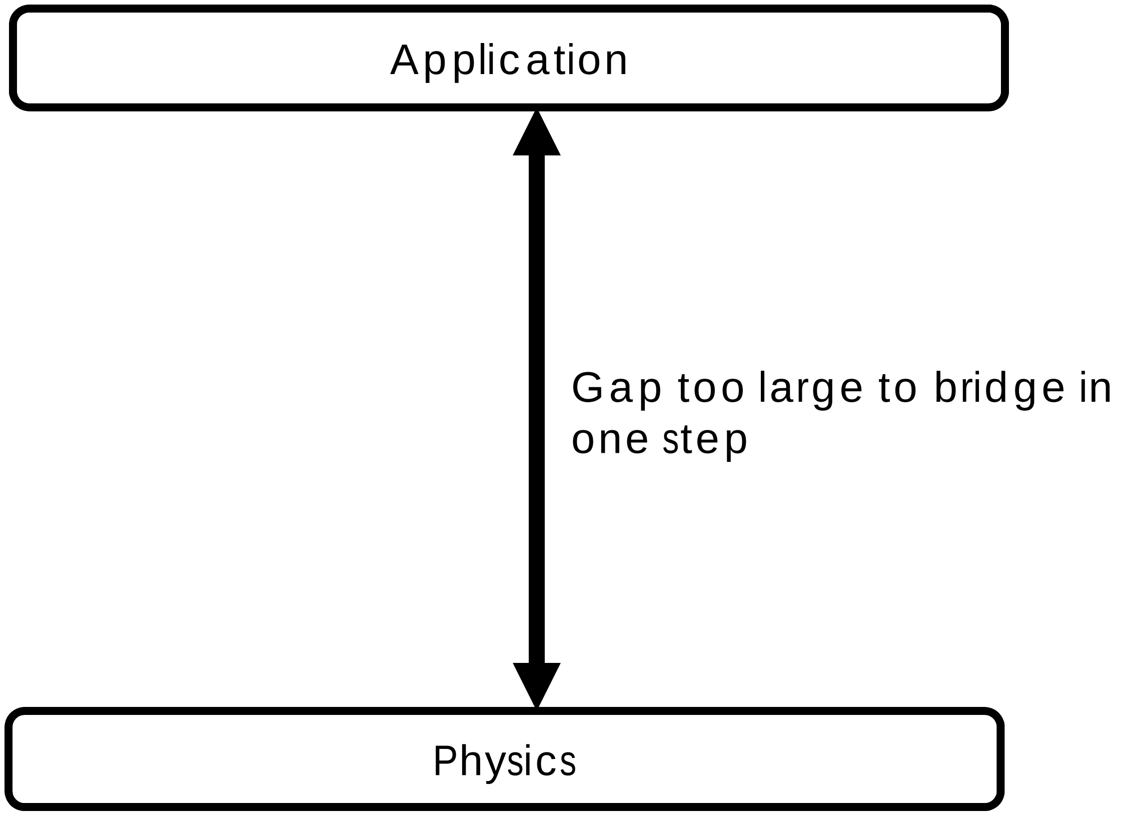
- 1/3 extended example or design questions
- 2/3 short answer

Architecture: Abstractions/Metrics

Computer architecture defines HW/SW interface

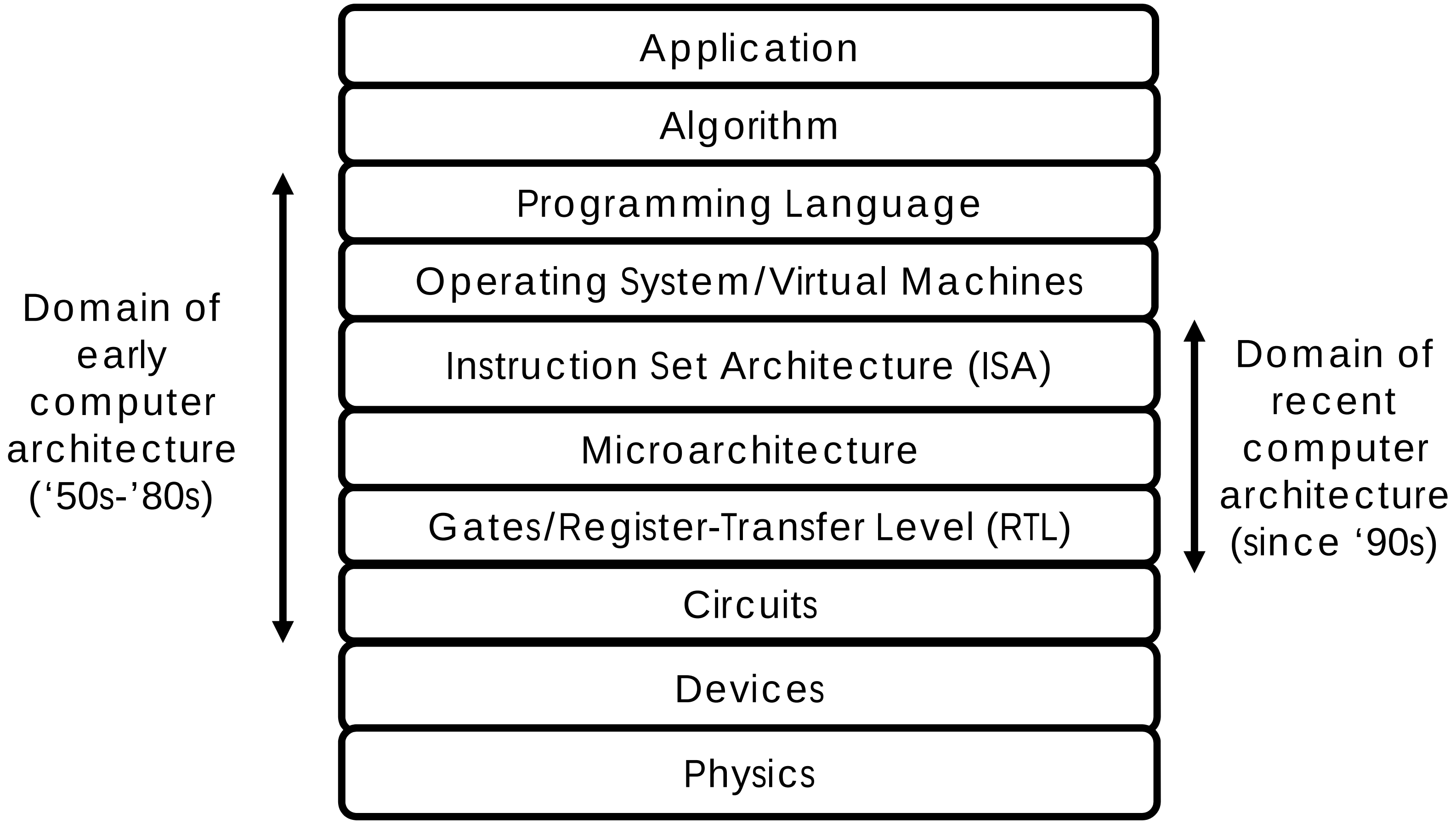
Evaluate architectures quantitatively

Computer Architecture



Computer architecture is the design of abstraction layers, which allow efficient implementations of computational applications on available technologies

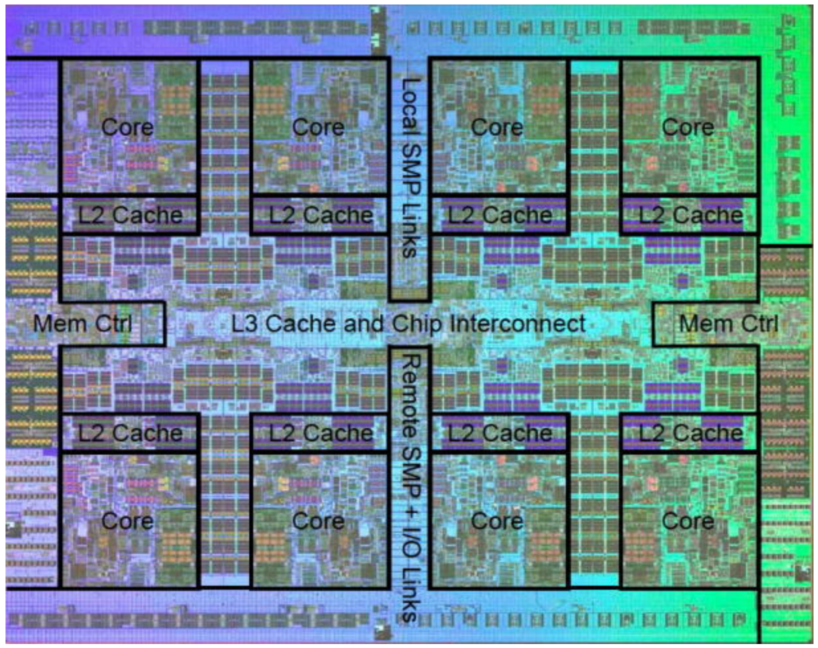
Abstraction Layers


Application
Algorithm
Programming Language
Operating System/Virtual Machines
Instruction Set Architecture (ISA)
Microarchitecture
Gates/Register-Transfer Level (RTL)
Circuits
Devices
Physics
Domain of early computer architecture ('50s-'80s)
Domain of recent computer architecture (since '90s)

ECE 552 Executive Summary

In-order Datapath
(built, ECE152)

Chip Multiprocessors
(understand, experiment ECE252)



Performance Factors

Latency = (Instructions / Program) x (Cycles / Instruction) x (Seconds / Cycle)

Seconds / Cycle

- Technology and architecture
- Transistor scaling
- Processor microarchitecture

Cycles / Instruction (CPI)

- Architecture and systems
- Processor microarchitecture
- System balance (processor, memory, network, storage)

Instructions / Program

- Algorithm and applications
- Compiler transformations, optimizations
- Instruction set architecture

Power and Energy

Definitions

- Energy (Joules) = $a \times C \times V^2$
- Power (Watts) = $a \times C \times V^2 \times f$

Power Factors and Trends

- activity (a): function of application resource usage
- capacitance (C): function of design; scales with area
- voltage (V): constrained by leakage, which increases as V falls
- frequency (f): varies with pipelining and transistor speeds

Datapath: CISC versus RISC

Complex Instruction Set Computing

- microprogramming
- motivated by technology (slow instruction fetch)

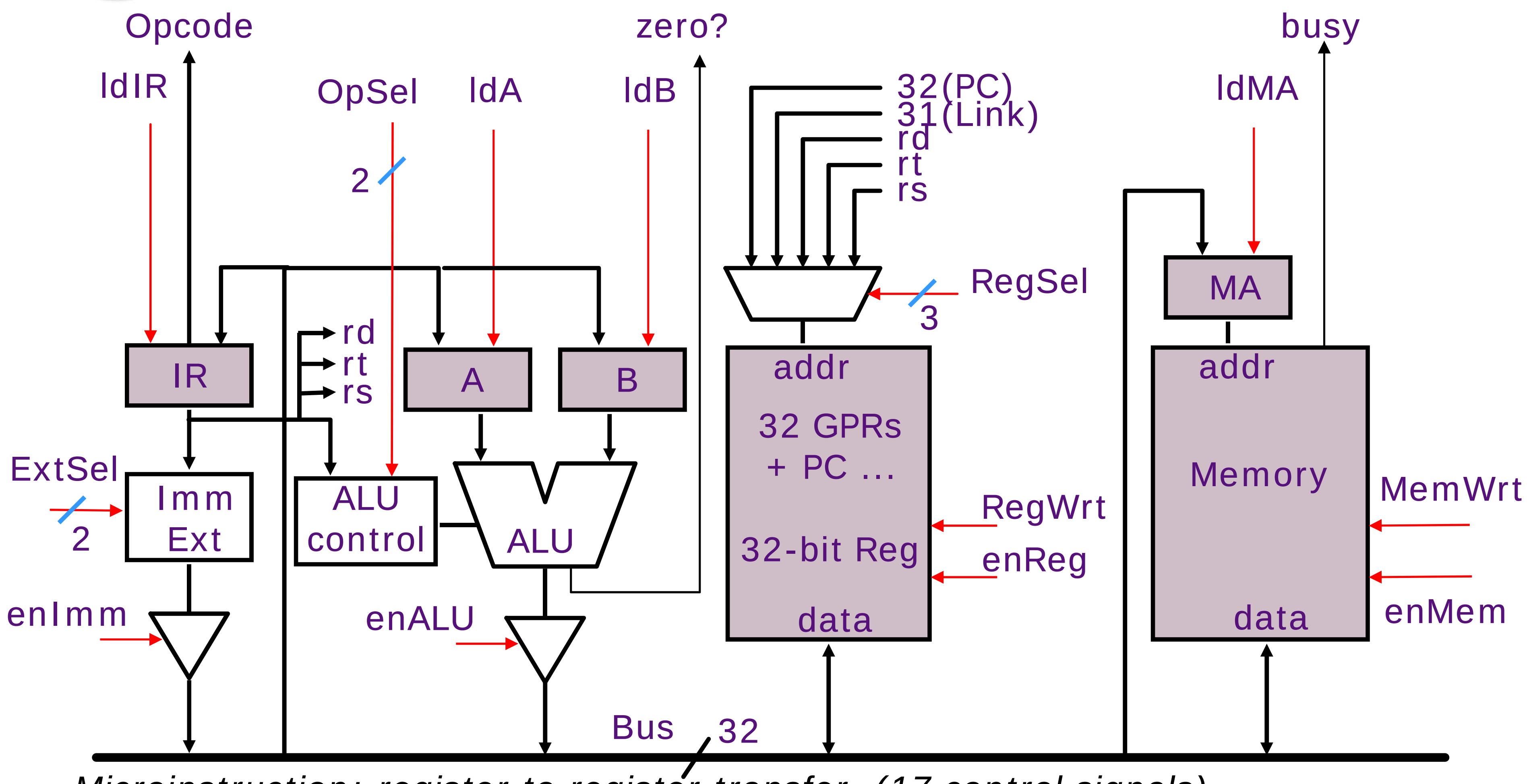
Reduced Instruction Set Computing

- hard-wired datapath
- motivated by technology (caches, fast memory)
- complex instructions rarely used

CISC Microprograms

instr fetch:	MA ← PC	# fetch current instr
	A ← PC	# next PC calculation
	IR ← Memory	
	PC ← A + 4	
	dispatch on Opcode	# start microcode
ALU:	A ← Reg[rs]	
	B ← Reg[rt]	
	Reg[rd] ← func(A,B)	
	do instruction fetch	
ALUi:	A ← Reg[rs]	
	B ← Imm	# sign extension
	Reg[rt] ← Opcode(A,B)	
	do instruction fetch	

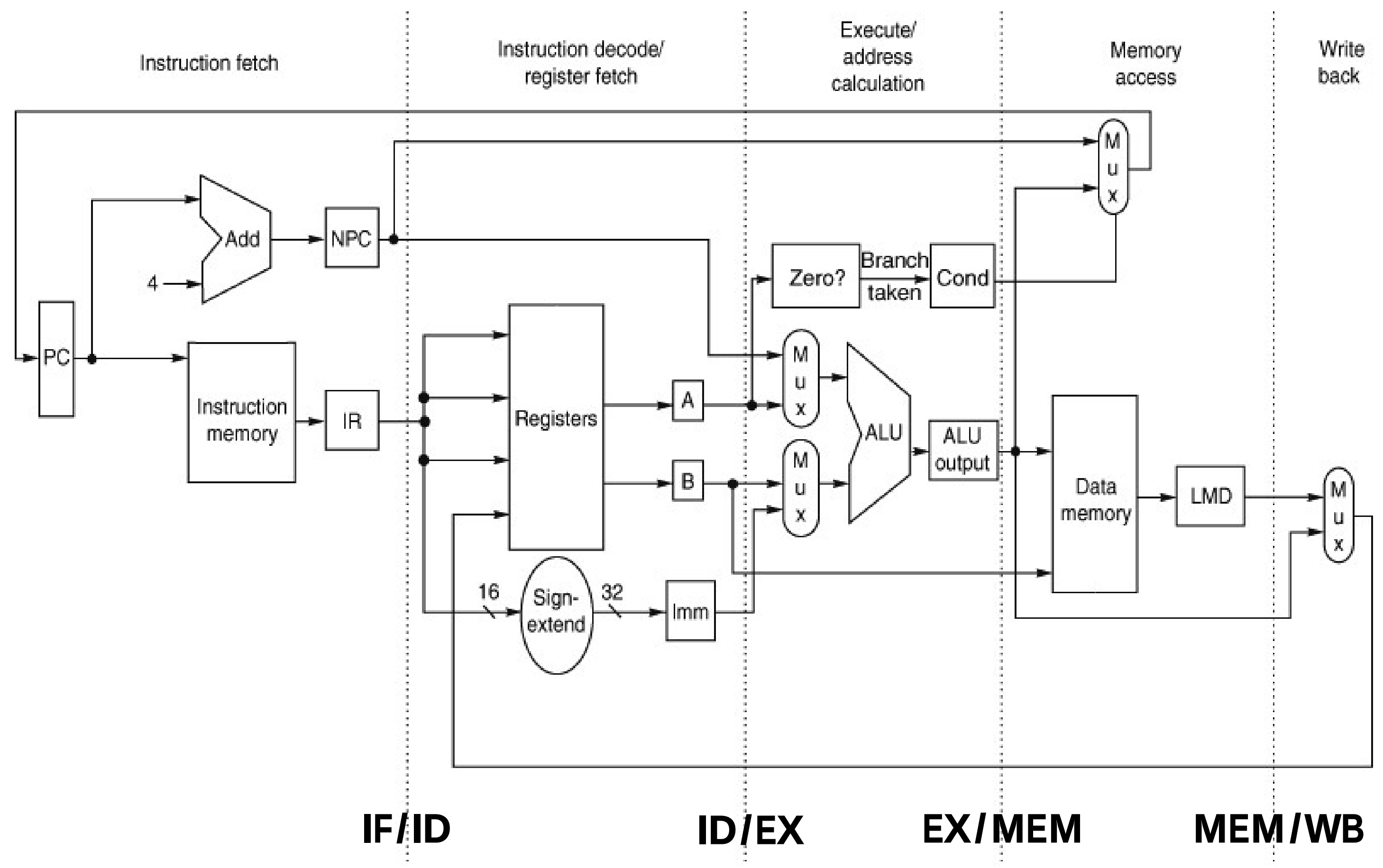
CISC Bus-Based MIPS Datapath



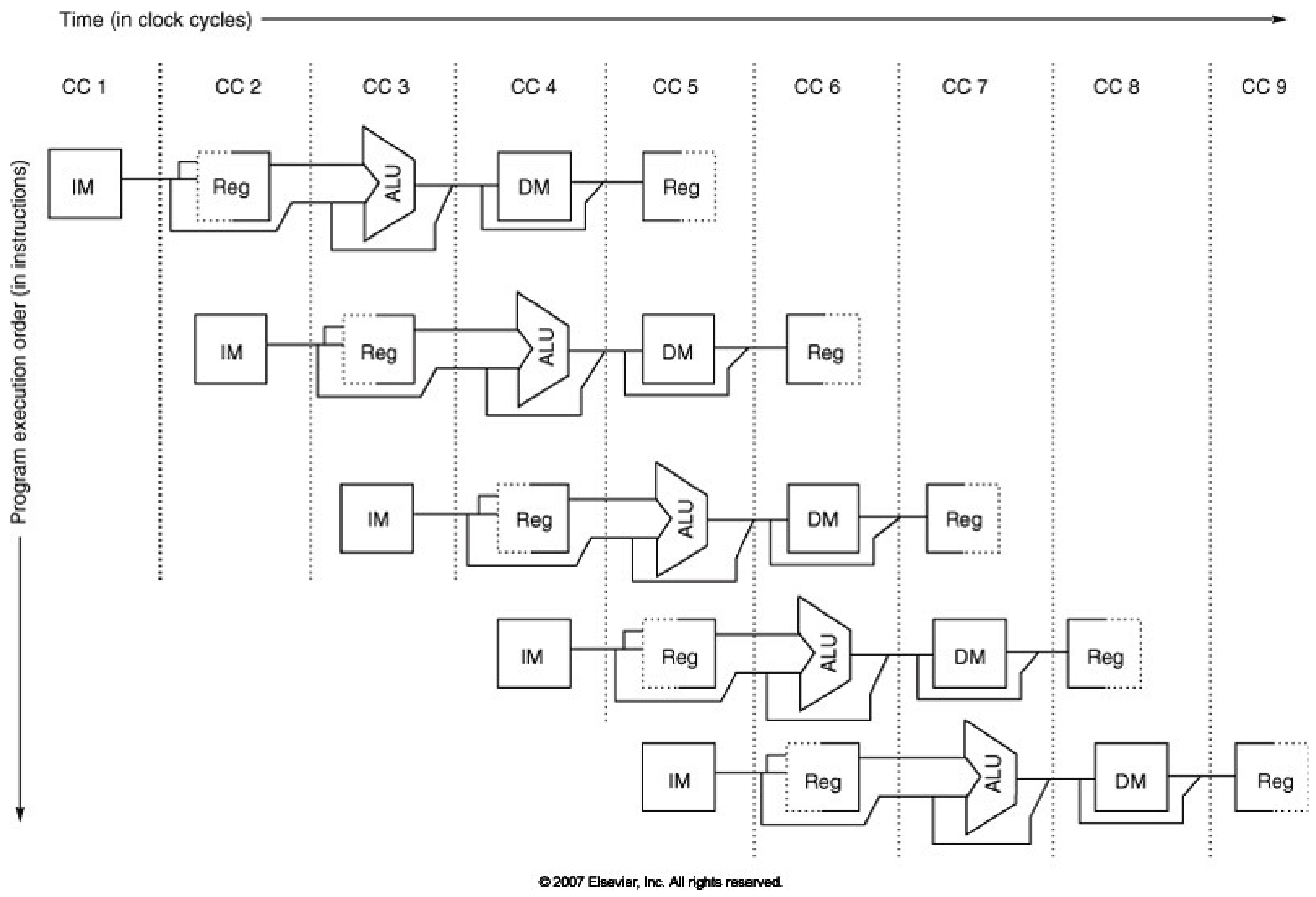
Microinstruction: register to register transfer (17 control signals)

MA ← PC	*means* RegSel = PC; enReg=yes; ldMA= yes
B ← Reg[rt]	*means* RegSel = rt; enReg=yes; ldB = yes

RISC Hard-wired MIPS Datapath



Visualizing the Pipeline


Time (in clock cycles)
CC 1
CC 2
CC 3
CC 4
CC 5
CC 6
CC 7
CC 8
CC 9
Program execution order (in instructions)
IM
Reg
ALU
DM
Reg
© 2007 Elsevier, Inc. All rights reserved.

Hazards and Limits to Pipelining

Structural Hazards

- Hardware cannot support this combination of instructions.
- Solution: stall pipeline (interlocks)

Data Hazards

- Instruction depends on result of prior instruction still in pipeline
- Solution: forward data, stall pipeline

Control Hazards

- Instruction fetch depends on decision about control flow
- Example: compute branches early in pipeline, predict branches

Tomasulo & Out-of-order

Out-of-order Execution

- Dynamically schedule instructions
- Execute instructions when dependences resolved

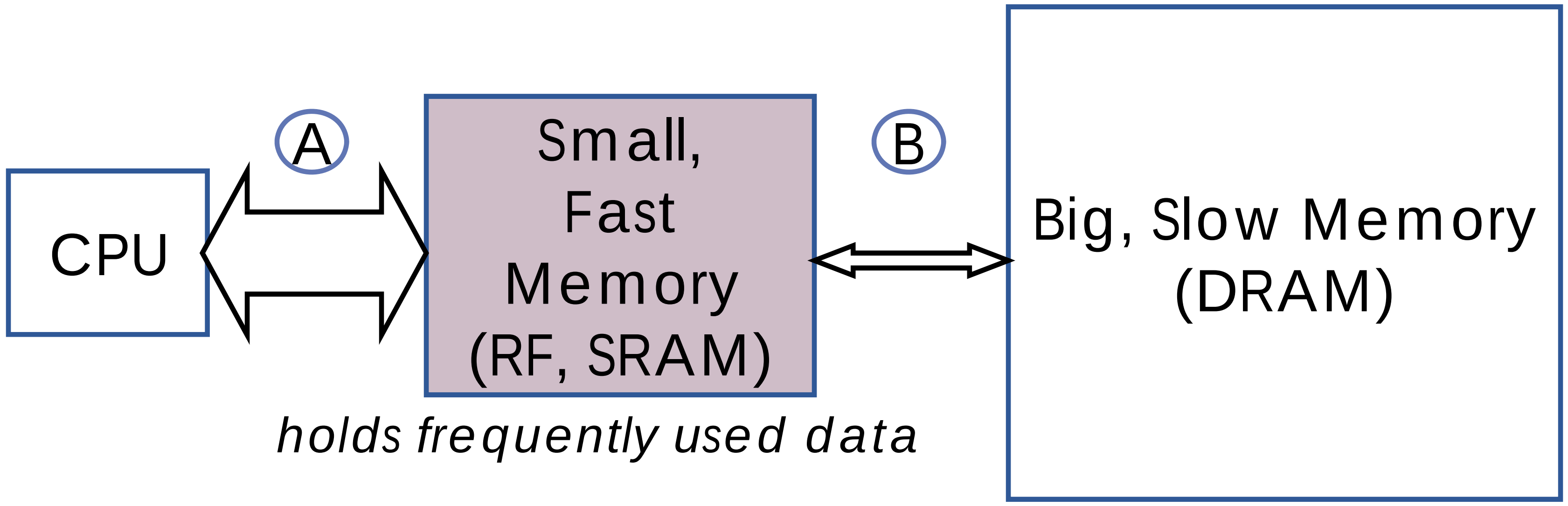
Tomasulo's Algorithm

- Queue instructions until operands ready (reservation stations, ROB)
- Rename to eliminate write hazards (rename table, physical registers)

Precise Interrupts/Exceptions

- Instructions execute/complete out-of-order
- Instructions commit in-order via reorder buffer
- Check for exceptions when committing instruction

Memory



DRAM – access dense array of slow memory with a command protocol

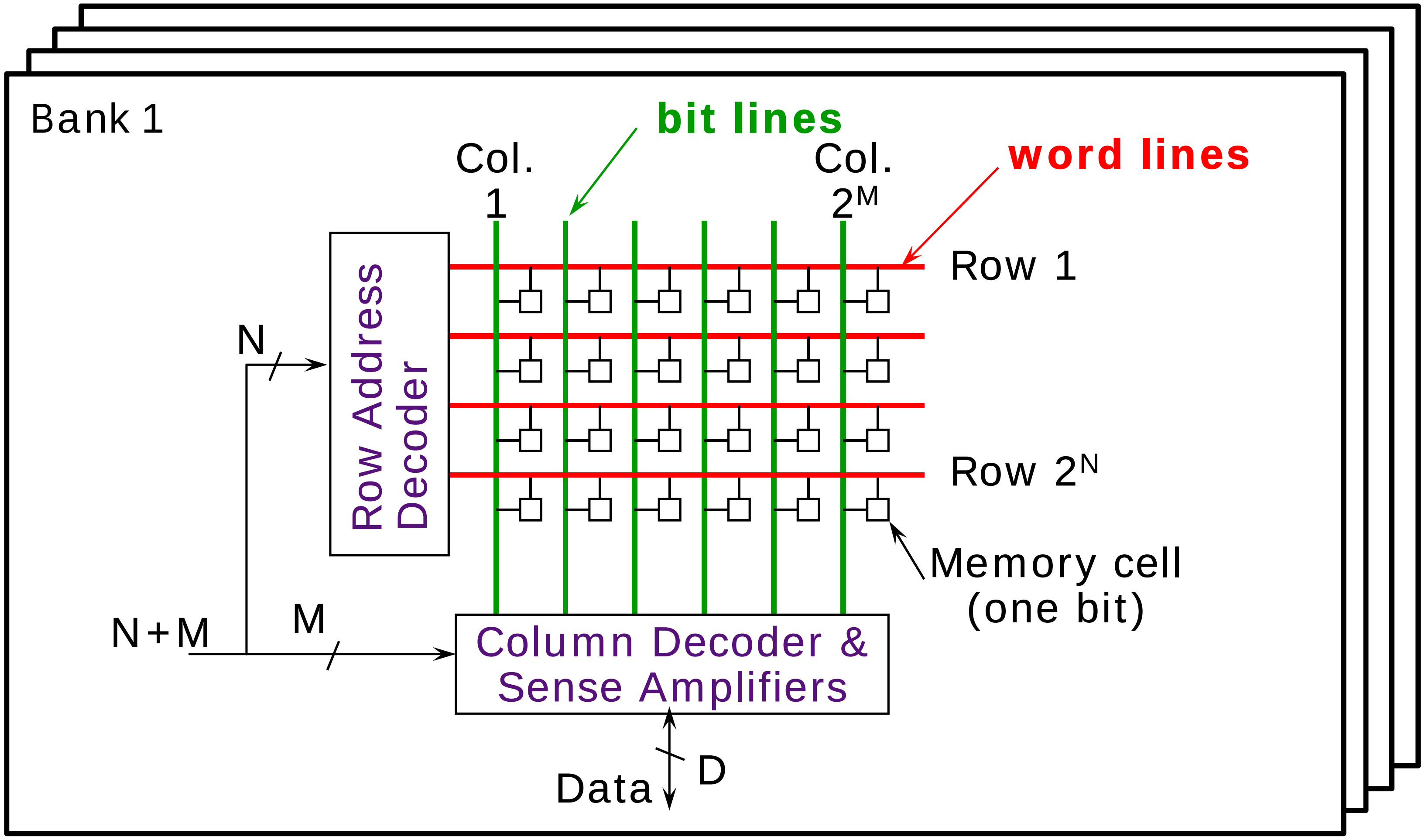
SRAM – access smaller array of fast memory on processor die

Virtual Memory – translate applications' virtual addresses into physical addresses, providing better memory management and protection

DRAM

-- Chip organized into 4-8 logical banks, which can be accessed in parallel

-- Access DRAM with activate , read/write, precharge commands



Caches

Caches exploit predictable patterns

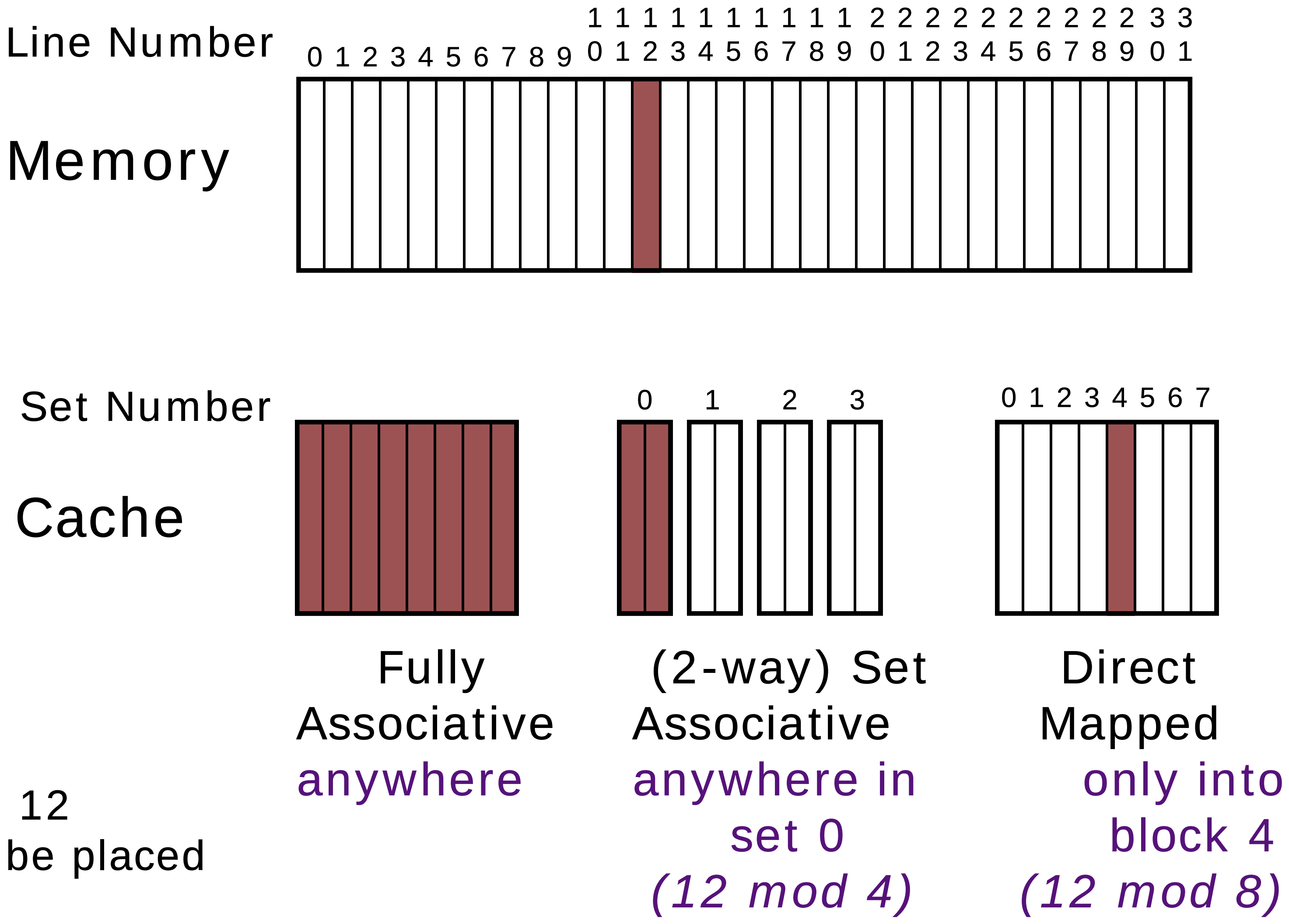
Temporal Locality

Caches remember the contents of recently accessed locations

Spatial Locality

Caches fetch blocks of data nearby recently accessed locations

Placement Policy



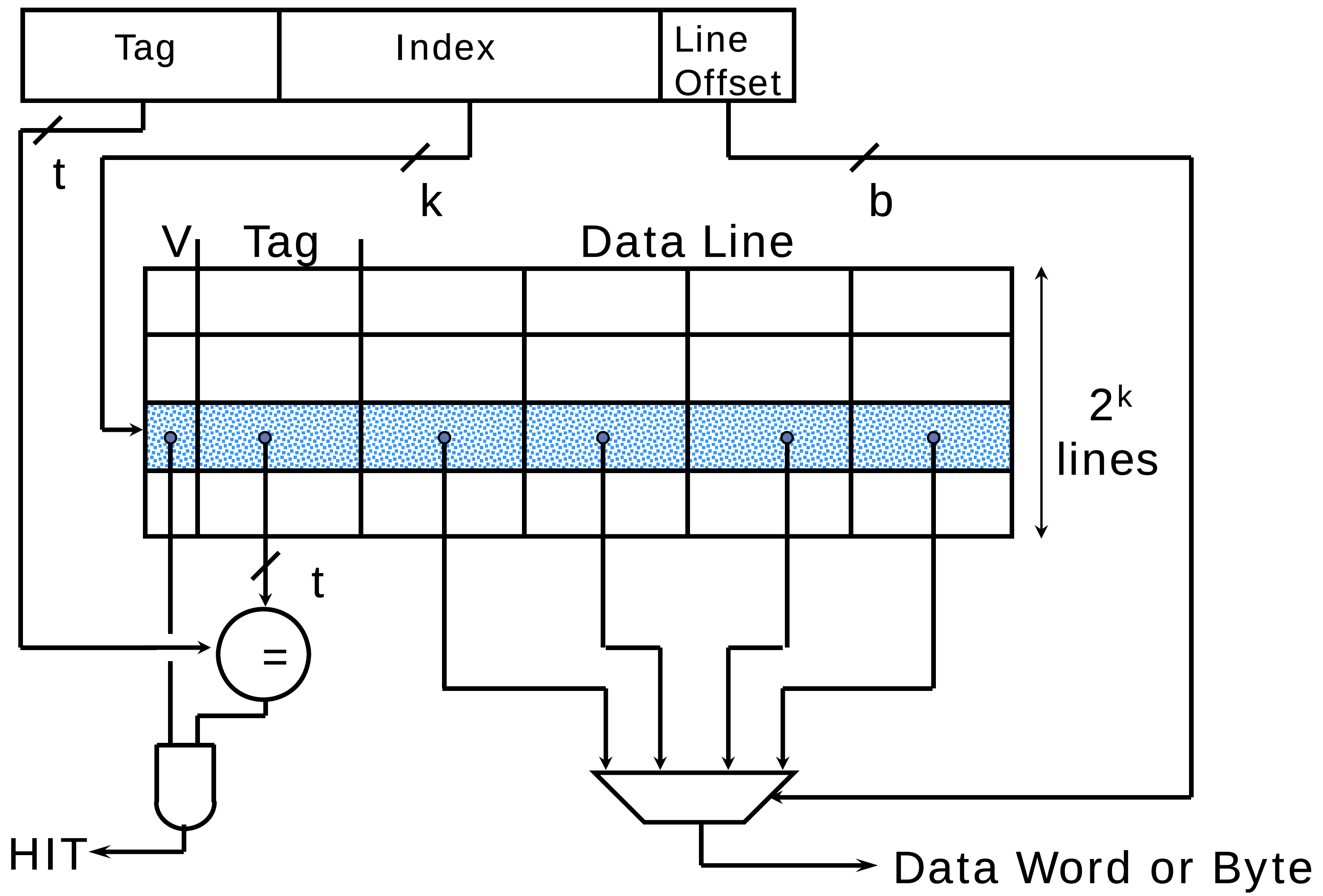
Line 12 can be placed

Fully Associative
anywhere

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

Direct Mapped
only into block 4
(12 mod 8)

Direct-Mapped Cache



Average Memory Access Time

AMAT = [Hit Time] + [Miss Prob.] x [Miss Penalty]

- Miss Penalty equals AMAT of next cache/memory/storage level.
- AMAT is recursively defined

To improve performance

- Reduce the hit time (e.g., smaller cache)
- Reduce the miss rate (e.g., larger cache)
- Reduce the miss penalty (e.g., optimize the next level)

Caches and Code

Restructuring code affects data access sequences

- Group data accesses together to improve spatial locality
- Re-order data accesses to improve temporal locality

Prevent data from entering the cache

- Useful for variables that are only accessed once
- Requires SW to communicate hints to HW.
- Example: “no-allocate” instruction hints

Kill data that will never be used again

- Streaming data provides spatial locality but not temporal locality
- If particular lines contain dead data, use them in replacement policy.
- Toward software-managed caches

Caches and Code

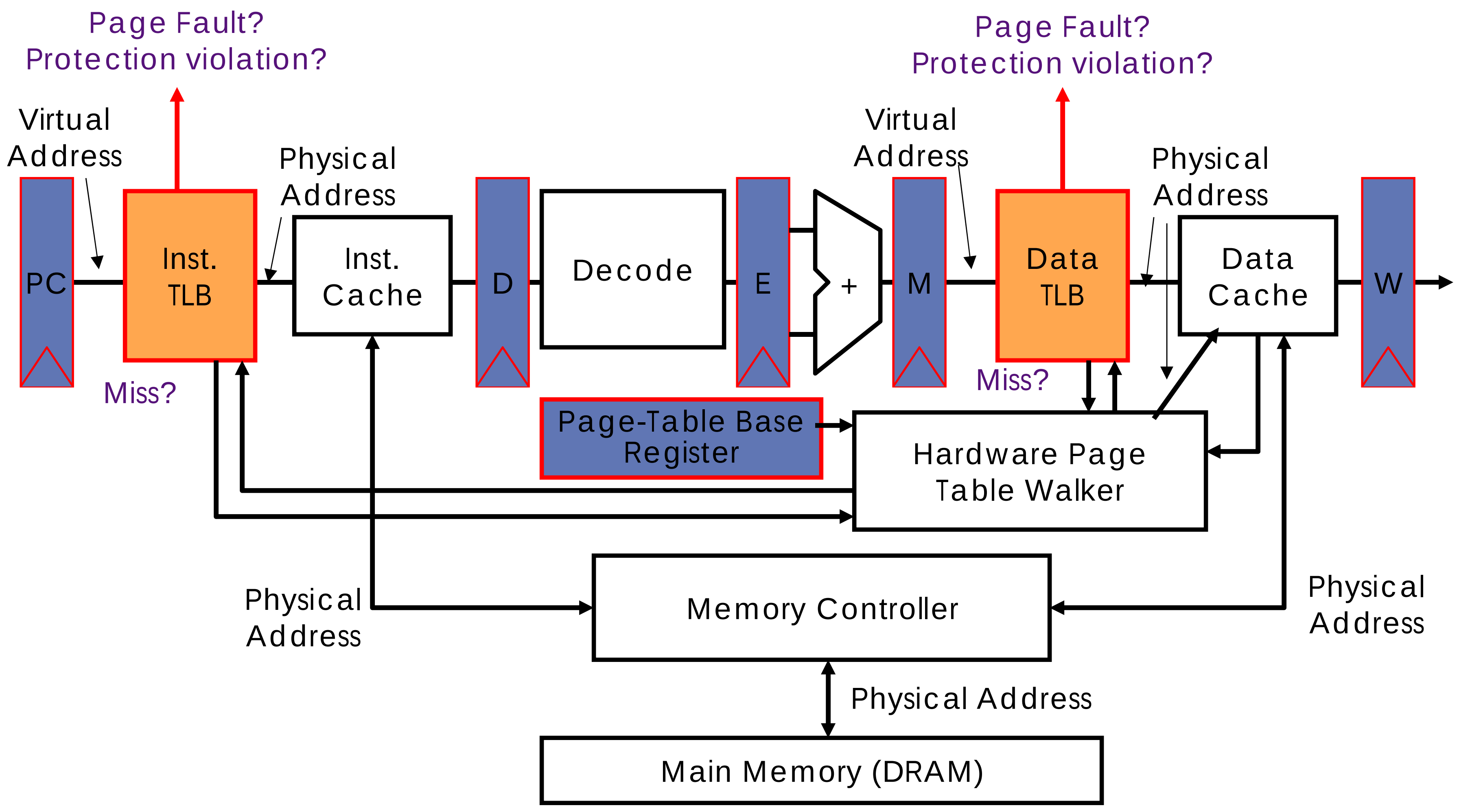
```
for(i=0; i < N; i++)
    a[i] = b[i] * c[i];

for(i=0; i < N; i++)
     d[i] = a[i] * c[i];
```

```
 for(i=0; i < N; i++)
{
       a[i] = b[i] * c[i];
       d[i] = a[i] * c[i];
 }
```

What type of locality does this improve?

Virtual Memory



Parallelism

Instruction-level Parallelism (ILP)

- multiple instructions in-flight
- hardware-scheduled: (1) pipelining, (2) out-of-order execution
- software-scheduled: (3) VLIW

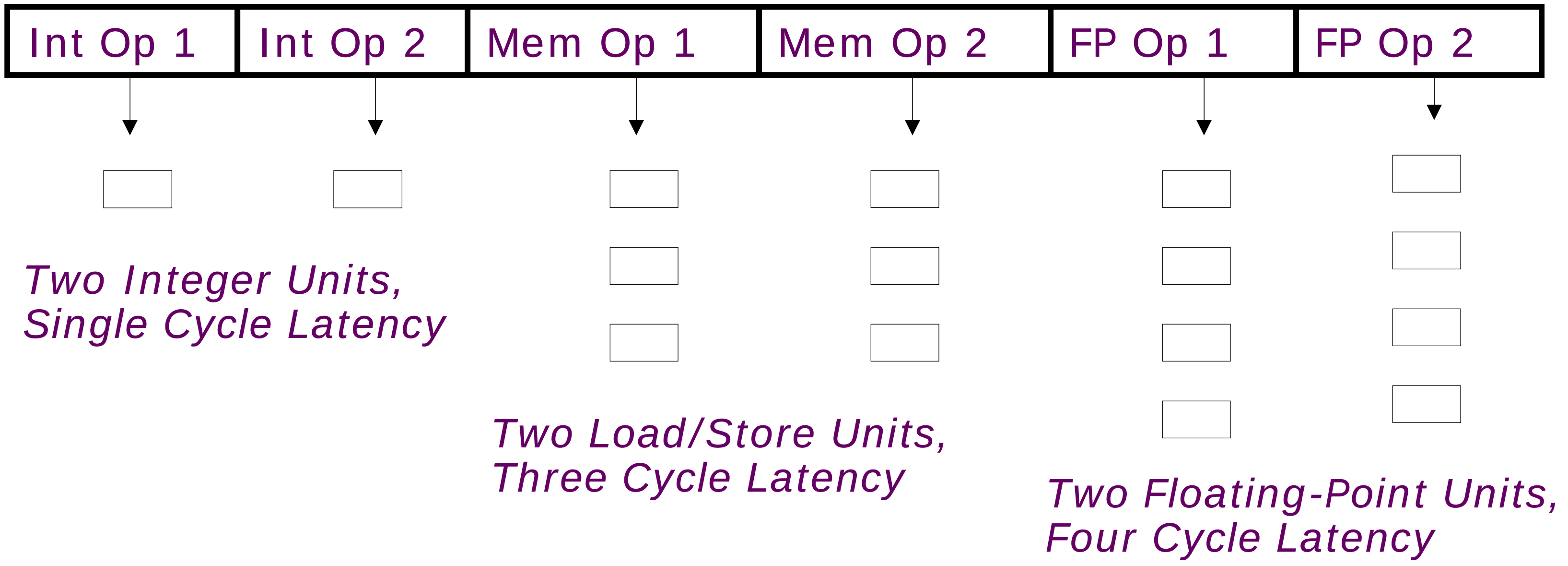
Data-level Parallelism (DLP)

- multiple, identical operations on data arrrays/streams
- (1) vector processors, (2) GPUs
- (3) single-instruction, multiple-data (SIMD) extensions

Thread-level Parallelism (TLP)

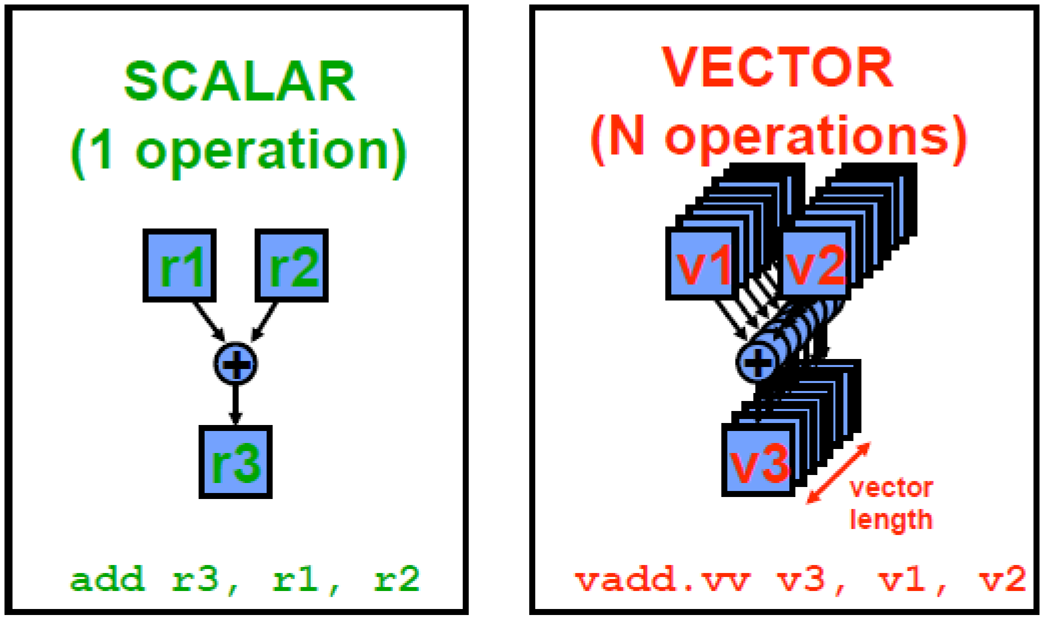
- multiple threads of control
- if a thread stalls, issue instructions from other threads
- (1) multi-threading, (2) multiprocessors

VLIW and ILP (SW-managed)

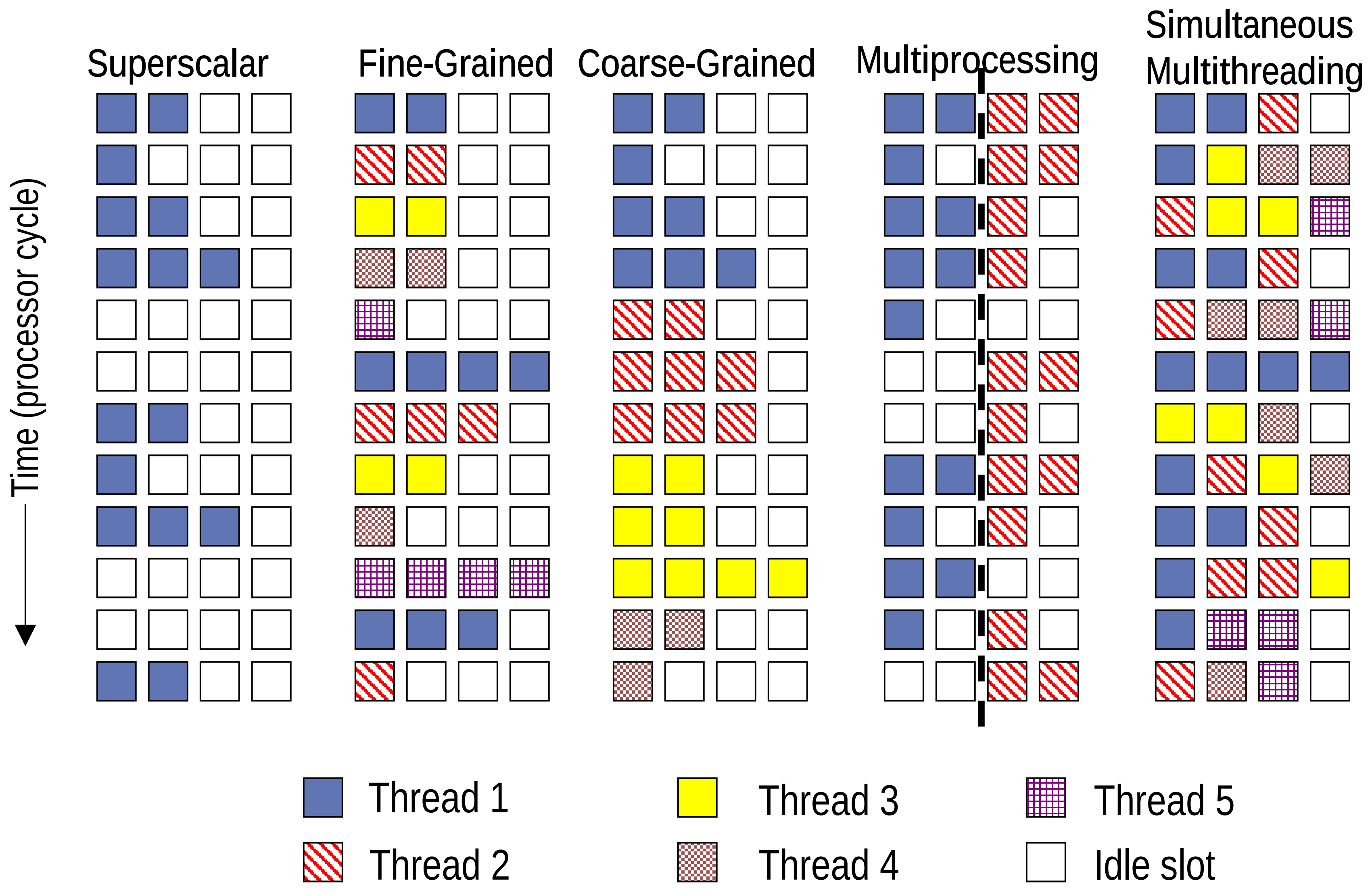

Int Op 1
Int Op 2
Mem Op 1
Mem Op 2
FP Op 1
FP Op 2
Two Integer Units, Single Cycle Latency
Two Load/Store Units, Three Cycle Latency
Two Floating-Point Units, Four Cycle Latency

- Multiple operations packed into one instruction format
- Instruction format is fixed, each slot supports particular instruction type
- Constant operation latencies are specified (e.g., 1 cycle integer op)
- Software schedules operations into instruction format, guaranteeing
  (1) Parallelism within an instruction – no RAW checks between ops
  (2) No data use before ready – no data interlocks/stalls

Vectors and DLP



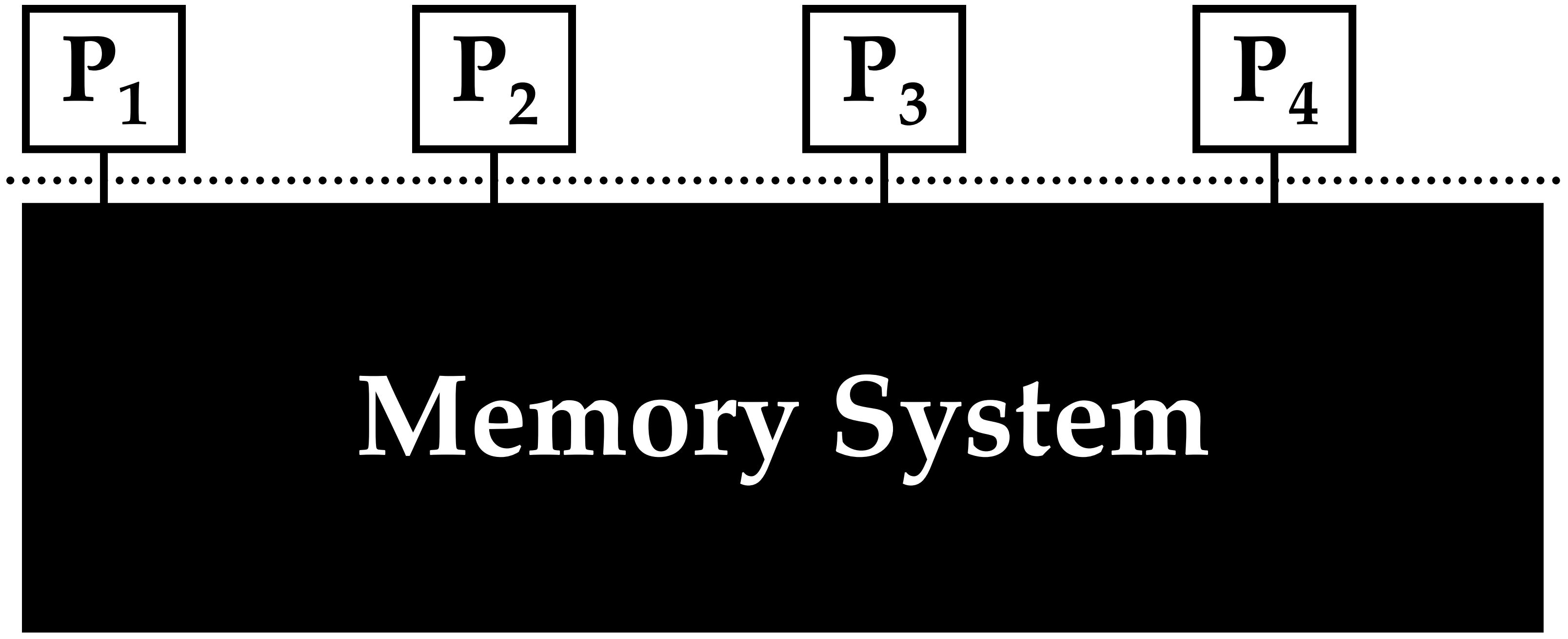
Multithreading and TLP



Multiprocessors

Shared-memory Multiprocessors

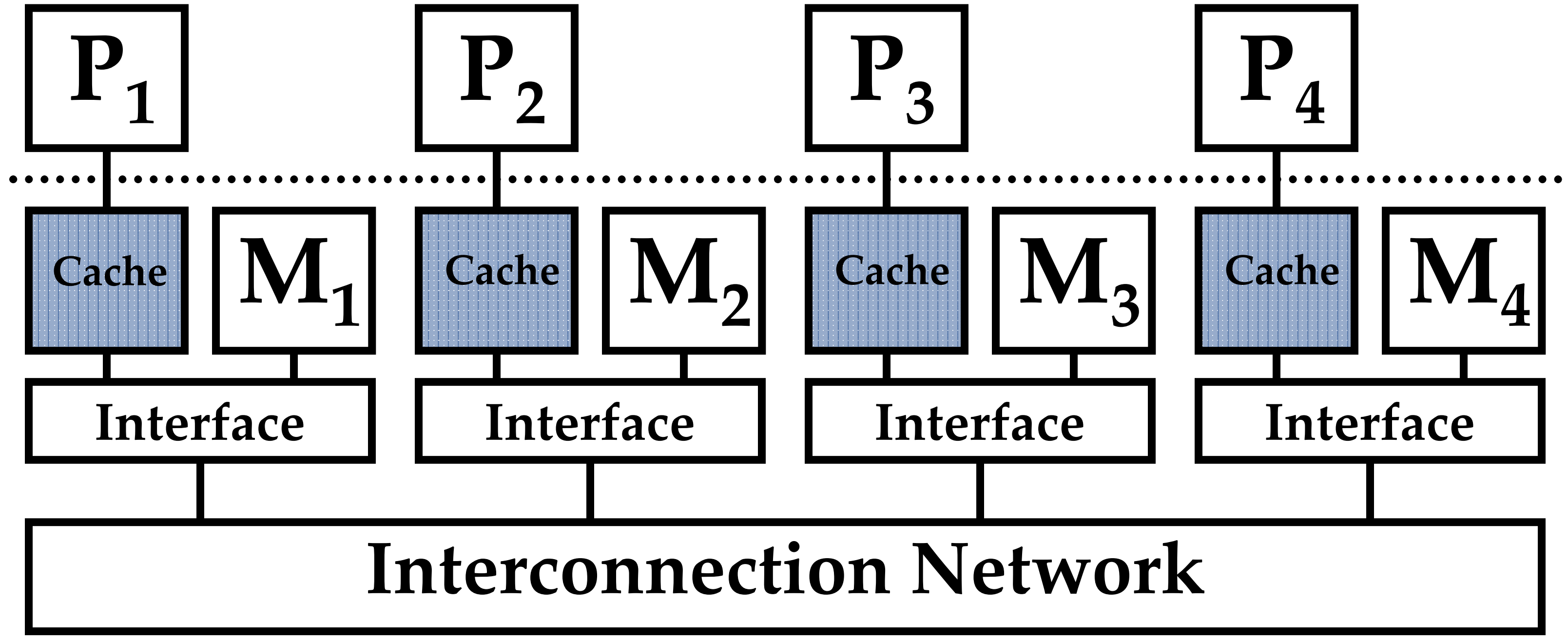
- Provide a shared-memory abstraction
- Enables familiar and efficient programmer interface



Multiprocessors

Shared-memory Multiprocessors

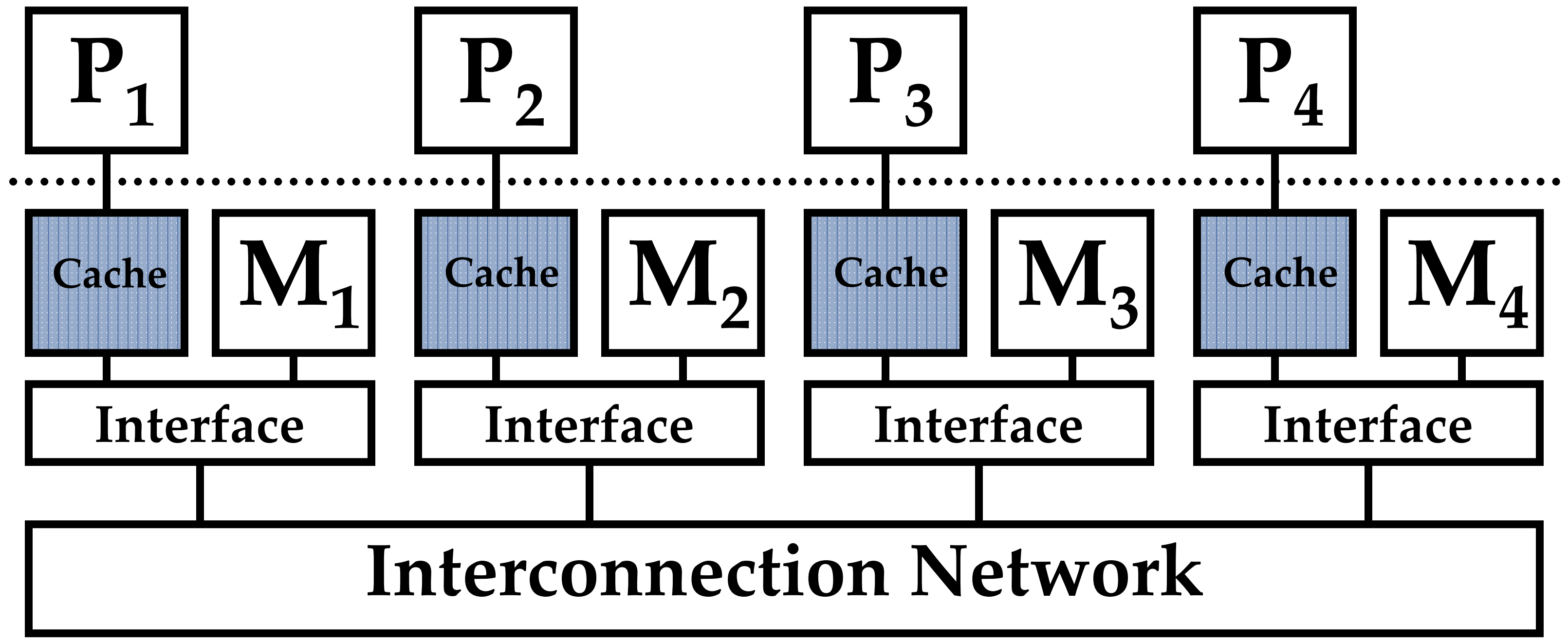
- Provide a shared-memory abstraction
- Enables familiar and efficient programmer interface



Multiprocessors

Shared-memory Multiprocessors

- Provide a shared-memory abstraction
- Enables familiar and efficient programmer interface



Challenges in Shared Memory

Cache Coherence

- **"Common Sense"**	
- **P1-Read[X] → P1-Write[X] → P1-Read[X]**	**Read returns X**
- **P1-Write[X] → P2-Read[X]**	**Read returns value written by P1**
- **P1-Write[X] → P2-Write[X]**	**Writes serialized**
	All P's see writes in same order

Synchronization

- Atomic read/write operations

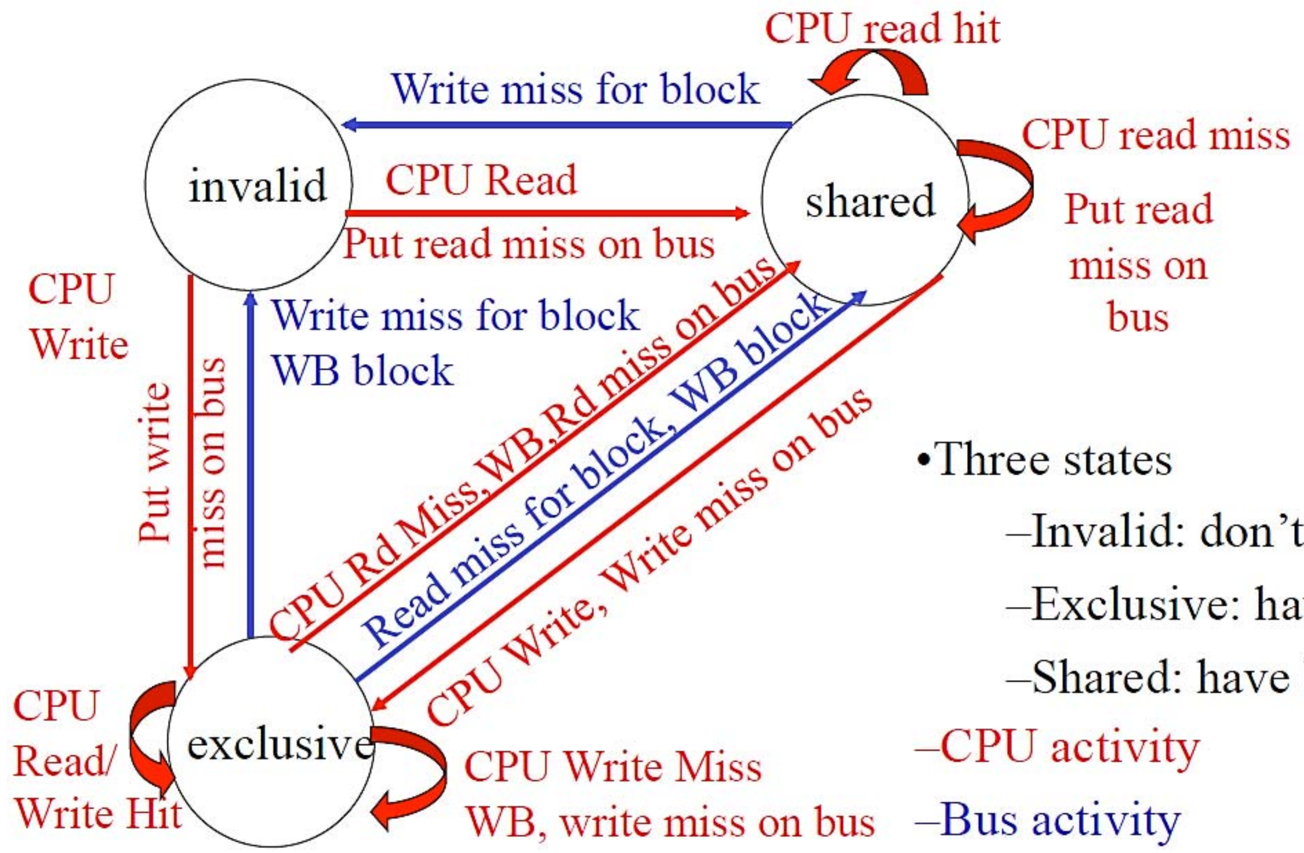
Memory Consistency

- What behavior should programmers expect from shared memory?
- Provide a formal definition of memory behavior to programmer
- Example: When will a written value be seen?
- Example: P1-Write[X] <<10ps>> P2-Read[X]. What happens?

Coherence Protocols

Implement protocol for every cache line.

Compare, contrast snoopy and directory protocols [[Stanford Dash]]



- Three states
  - –Invalid: don't have block
  - –Exclusive: have block and wrote it
  - –Shared: have block but only read it
- –CPU activity
- –Bus activity

Synchronization and Atomicity

Solution: Test-and-set instruction

- Add single instruction for load-test-store (t&s R1, lock)
- Test-and-set atomically executes

```
ld R1, lock;        # load previous lock value
st 1, lock;         # store 1 to set/acquire
```

- If lock initially free (0), t&s acquires lock (sets to 1)
- If lock initially busy (1), t&s does not change it
- Instruction is un-interruptible/atomic by definition

```
Inst-0     t&s R1, lock     # atomically load, check, and set lock=1
Inst-1     bnez R1          # if previous value of R1 not 0,
....                        acquire unsuccessful
Inst-n     stw R1, 0        # atomically release lock
```

Sequential Consistency (SC)

Definition of Sequential Consistency

Formal definition of programmers' expected view of memory

(1) Each processor P sees its own loads/stores in program order

(2) Each processor P sees !P loads/stores in program order

(3) All processors see same global load/store ordering.

P and !P loads/stores may be interleaved into some order.

But all processors see the same interleaving/ordering.

Definition of Multiprocessor Ordering [Lamport]

Multi-processor ordering corresponds to some sequential interleaving of uni-processor orderings. Multiprocessor ordering should be indistinguishable from multi-programmed uni-purocessor

For More

ECE 652

- Advanced Computer Architecture II
- Parallel computer architecture design and evaluation
- Parallel programming, coherence, synchronization, consistency

ECE 590

- Energy Efficient Computer Systems
- Technology, architecture, application strategies for energy efficiency
- Datacenter computing

ECE 554

- Fault-Tolerant and Testable Computer Systems
- Fault models, redundancy, recovery, testing

Computer architecture is HW/SW interface.

Consider classes on both sides of this interface.

Looking Forward

Energy-efficiency

- Technology limitations motivate new architectures for efficiency
- Ex: specialization, heterogeneity, management

Technology

- Emerging technologies motivate new architectures for capability
- Ex: memory (phase change), networks (optical),

Reliability and Security

- Variations in fabrication, design process motivate new safeguards
- Ex: tunable structures, trusted bases

Multiprocessors

- Abundant transistors, performance goals motivate parallel computing
- Ex: parallel programming, coherence/consistency, management