



Bangladesh Govt. & UGC Approved

UNIVERSITY OF GLOBAL VILLAGE (UGV), BARISHAL.

THE FIRST SKILL BASED HI-TECH UNIVERSITY IN BANGLADESH

Advance Computer Architecture

Prepared by:

Md. Abdur Razzak

Asst. Professor.

Department of CSE

*Course
Code*

• CSE 407

Credits

• 3.00

*Exam
Hours*

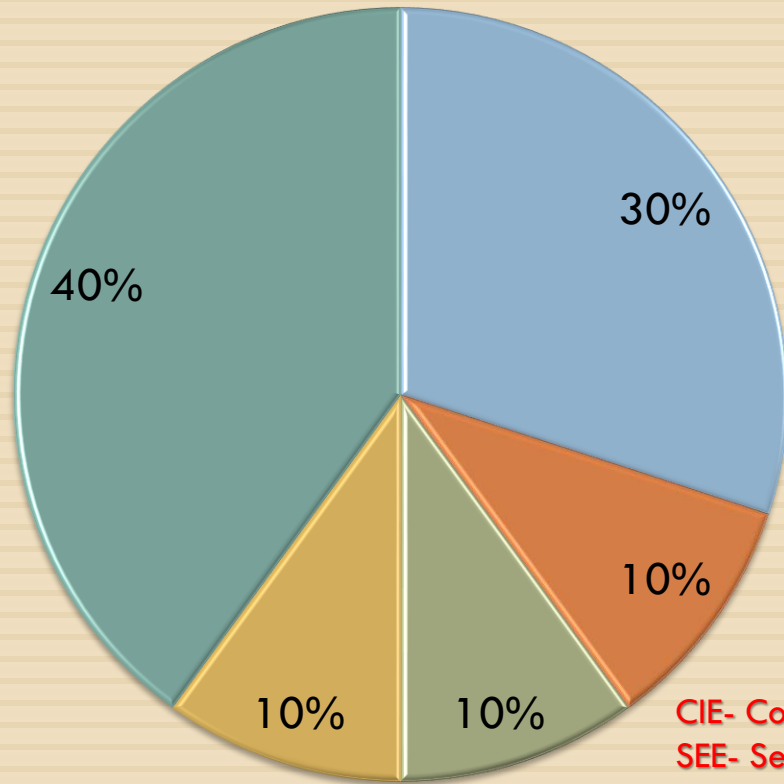
• 3.00

*Total
Marks*

150

Assessment Pattern

■ CIE Test ■ Assignments ■ Quizes ■ Attendance ■ SEE



CIE- Continues Internal Evaluation
SEE- Semester End Examination

Detailed Assessment Pattern

CIE- Continuous Internal Evaluation (90 Marks)

Bloom's Category Marks (out of 90)	Tests (45)	Assignments (15)	Quizzes (15)	Attendance (15)
Remember	5	03		
Understand	5	04	05	
Apply	15	05	05	
Analyze	10			
Evaluate	5	03	05	
Create	5			

SEE- Semester End Examination (60 Marks)

Bloom's Category	Test
Remember	7
Understand	7
Apply	20
Analyze	15
Evaluate	6
Create	5

Week No	Topic	Teaching Learning Strategy(s)	Assessment Strategy(s)
1	Basic Concepts and Computer Evolution	Quiz, Assignment	QA, Quiz, Assignment
2	Cache Memory	Lecture, Reading Assignments	QA, Quiz, Assignment
3	I/O Systems	Lecture, Case Studies	Quiz, Assignment
4	Multicore Processor		
5	Operating System Support	Lecture, Hands-on Labs	Quiz, Lab Reports
6	Operating System Support-2	Lecture, Simulation Exercises	Quiz, Simulation Reports

Week No	Topic	Teaching Learning Strategy(s)	Assessment Strategy(s)
7	Computer Arithmetic	Quiz, Assignment	QA, Quiz, Assignment
8	MIPS Instruction-Set Architecture	Lecture, Reading Assignments	QA, Quiz, Assignment
9	MIPS Instruction-Set Architecture-2	Lecture, Case Studies	Quiz, Assignment
10	MIPS Pipelining		
11	Instruction Set Design	Lecture, Hands-on Labs	Quiz, Lab Reports
12	ILP architectures with emphasis on Superscalar	Lecture, Simulation Exercises	Quiz, Simulation Reports

Week No	Topic	Teaching Learning Strategy(s)	Assessment Strategy(s)
13	Exploiting ILP with SW approaches	Quiz, Assignment	QA, Quiz, Assignment
14	SMT Simultaneously Multi-Threading	Lecture, Reading Assignments	QA, Quiz, Assignment
15	Multi-Processing -1	Lecture, Case Studies	Quiz, Assignment
16	Multi-Processing -2		
17	Multi-Processing -3	Lecture, Hands-on Labs	Quiz, Lab Reports

Week 1

Lecture overview

9

- Trends
 - ▣ Performance increase
 - ▣ Technology factors
- Computing classes
- Cost
- Performance measurement
 - ▣ Benchmarks
 - ▣ Metrics

Trends

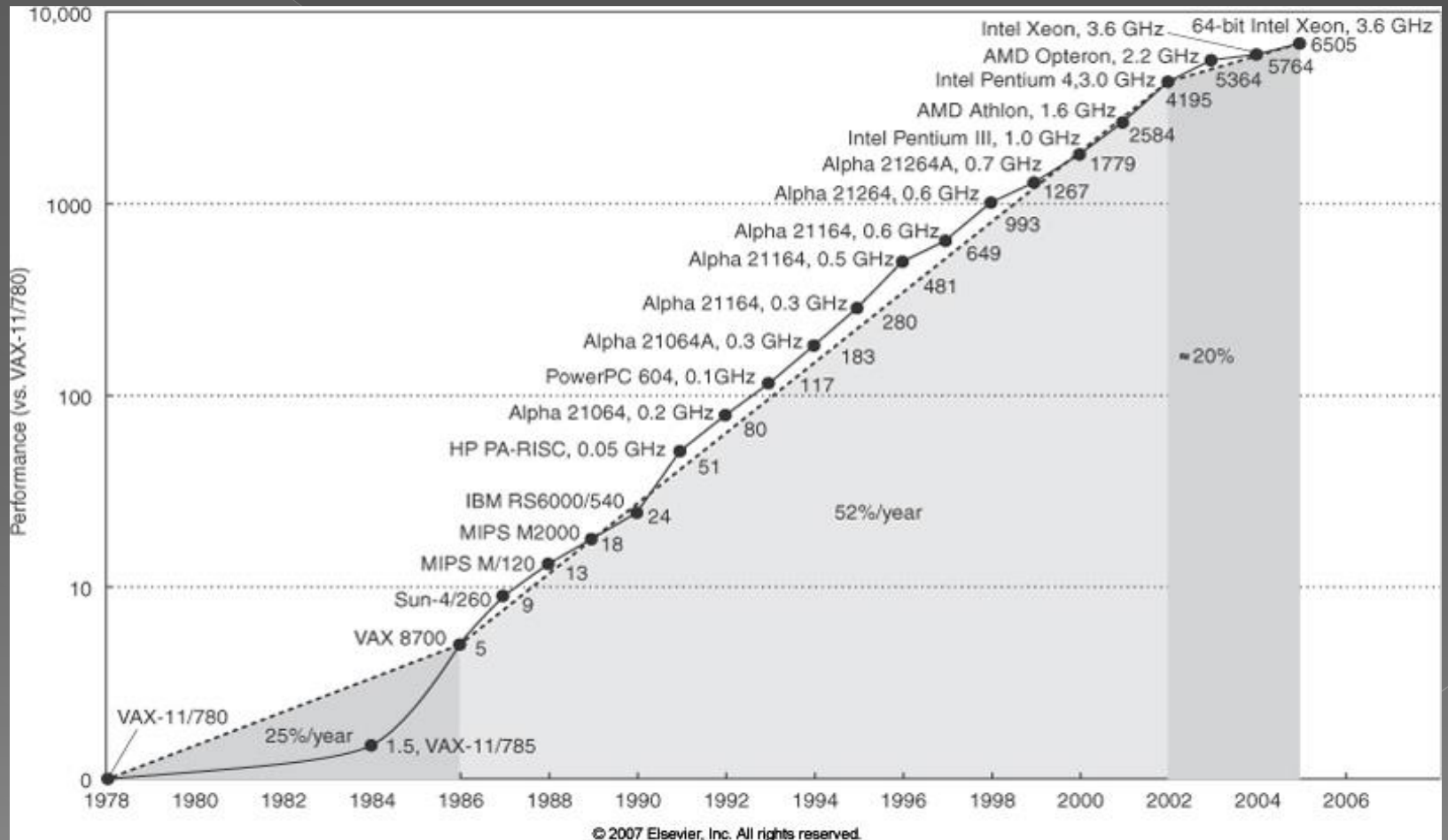
10

- See the ITRS
(International Technology Roadmap Semiconductors)
- <http://public.itrs.net/>



International Technology Roadmap for Semiconductors

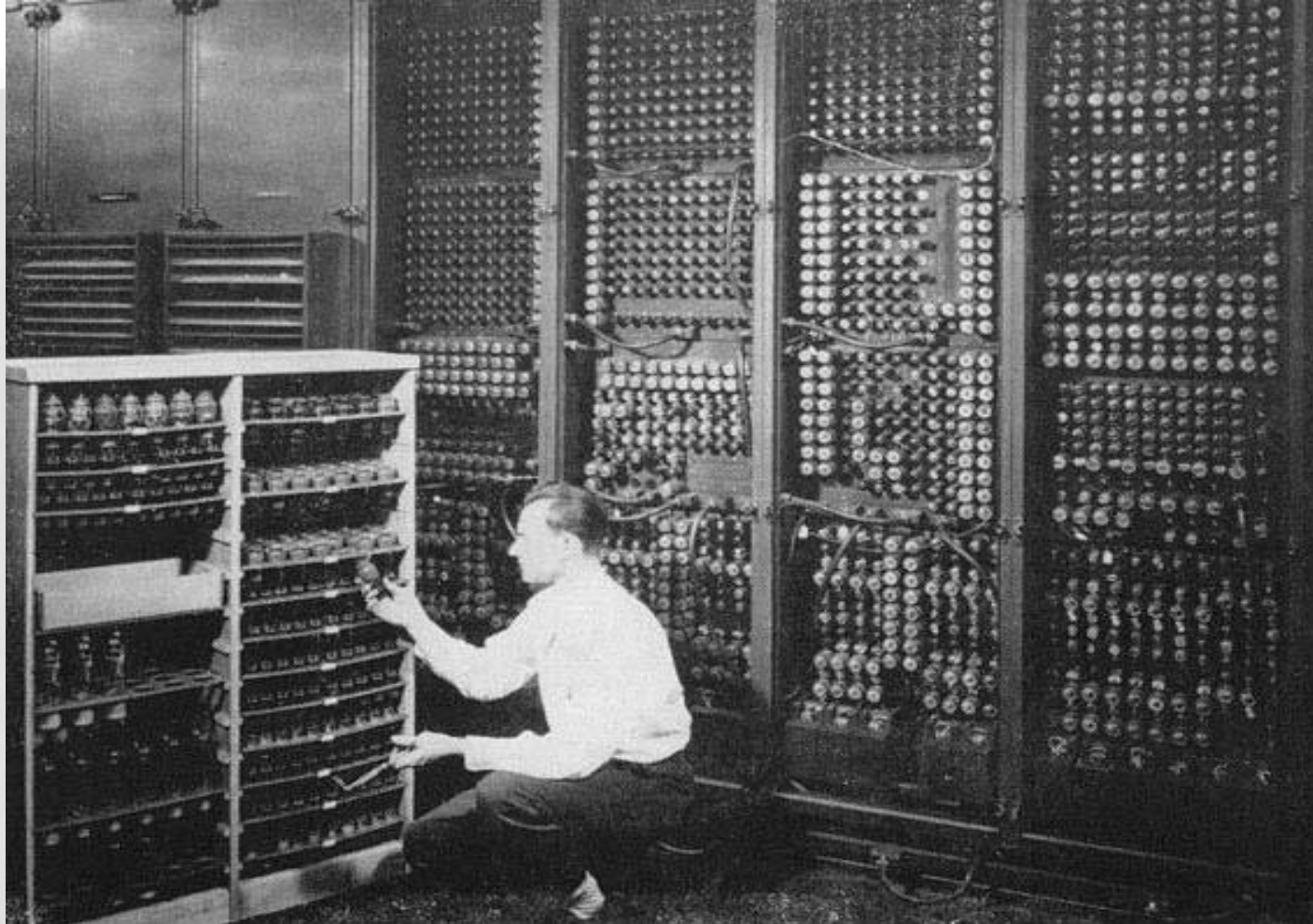
Performance of (single) processors



Where Has This Performance Improvement Come From?

- Technology
 - ▣ More transistors per chip
 - ▣ Faster logic
- Machine Organization/Implementation
 - ▣ Deeper pipelines
 - ▣ More instructions executed in parallel
- Instruction Set Architecture
 - ▣ Reduced Instruction Set Computers (RISC)
 - ▣ Multimedia extensions
 - ▣ Explicit parallelism
- Compiler technology
 - ▣ Finding more parallelism in code
 - ▣ Greater levels of optimization

ENIAC: ELECTRONIC NUMERICAL INTEGRATOR AND COMPUTER, 1946



VLSI Developments

14

1946: ENIAC electronic numerical integrator and computer

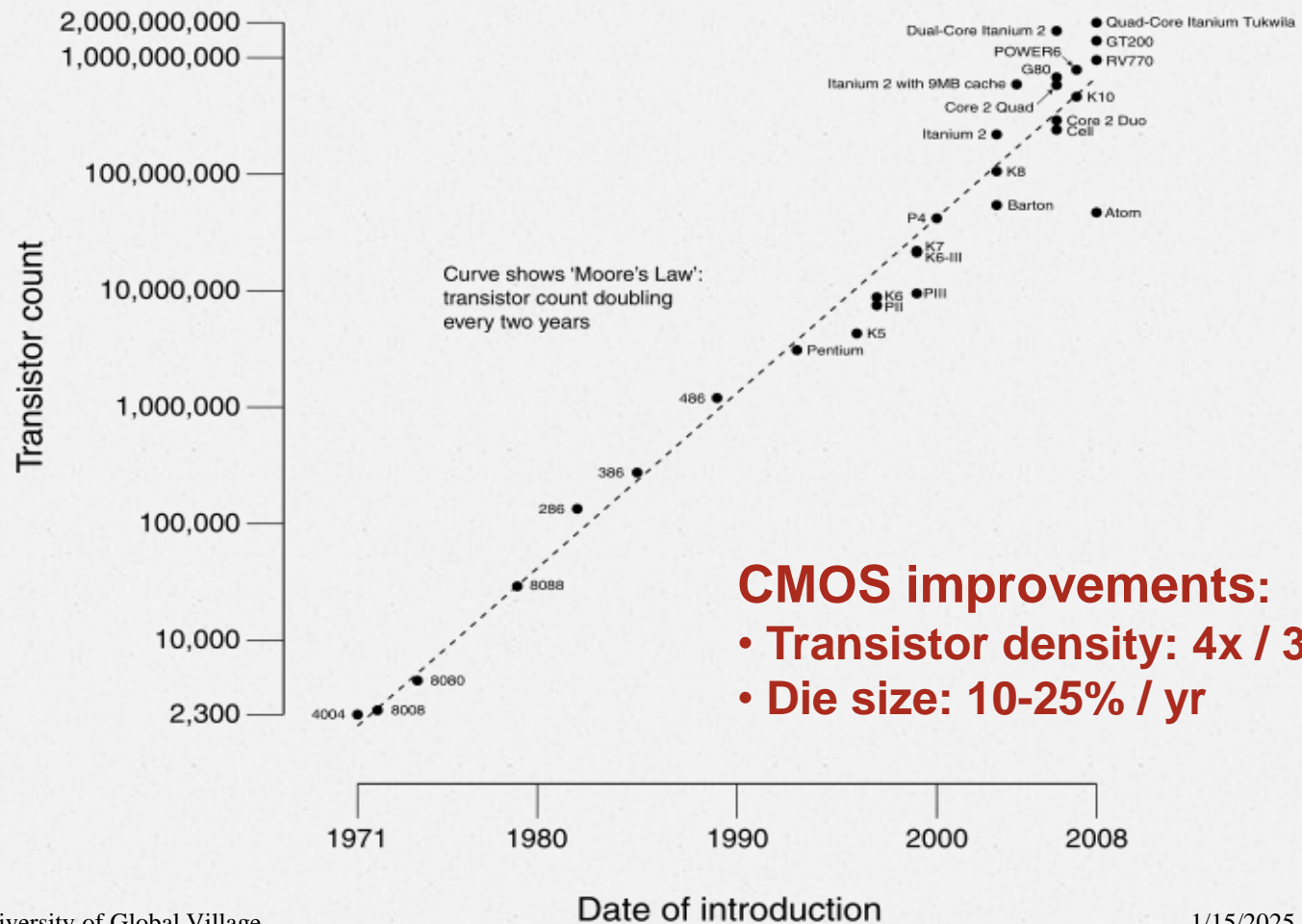
- Floor area
 - ▣ 140 m²
- Performance
 - ▣ multiplication of two 10-digit numbers in 2 ms

Technology Improvement

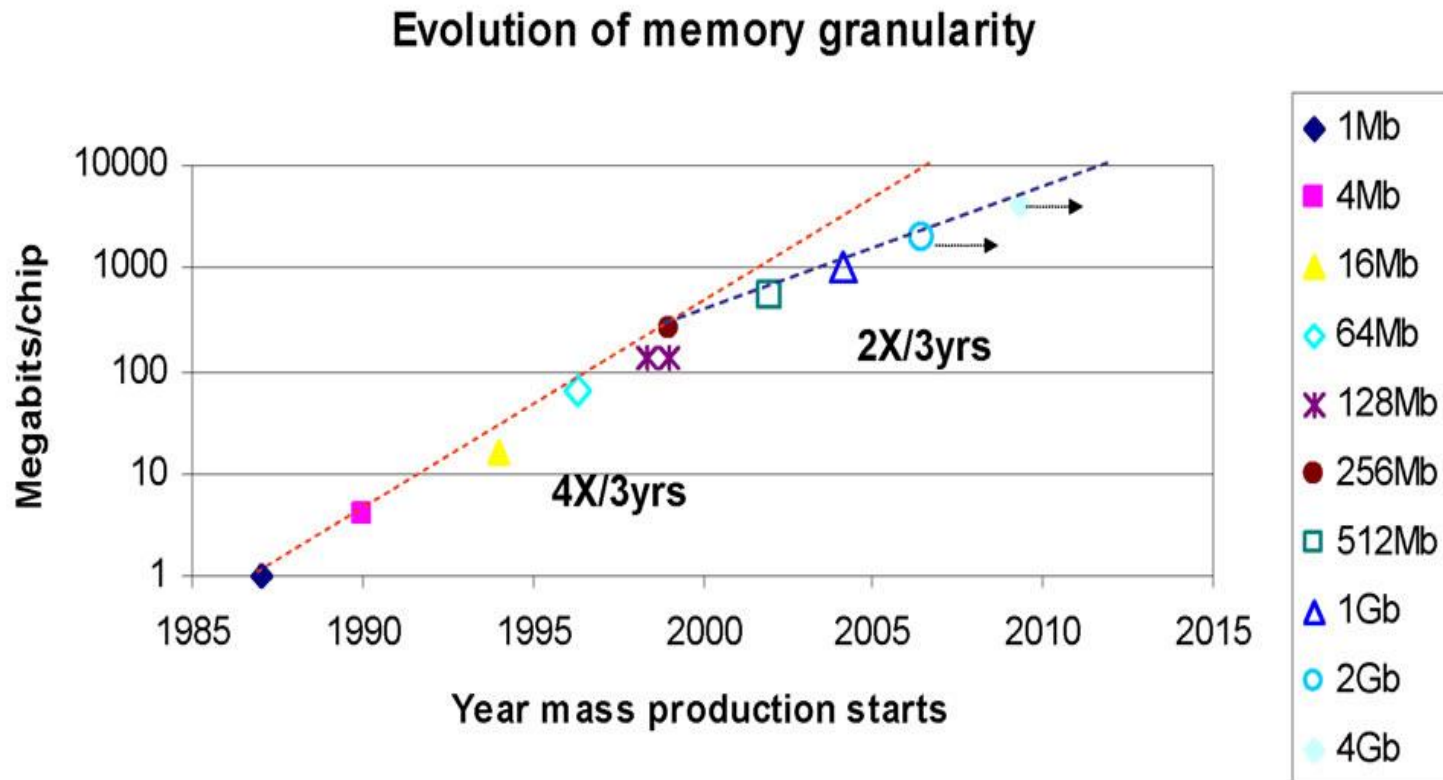
2010: High Performance microprocessor

- Chip area
 - ▣ 100-300 mm²
- Board area
 - ▣ 200 cm²; improvement of 10⁴
- Performance:
 - ▣ 64 bit multiply in O(1 ns); improvement of 10⁶
- On top
 - ▣ architectural improvements, like ILP exploitation
 - ▣ extreme cost reduction

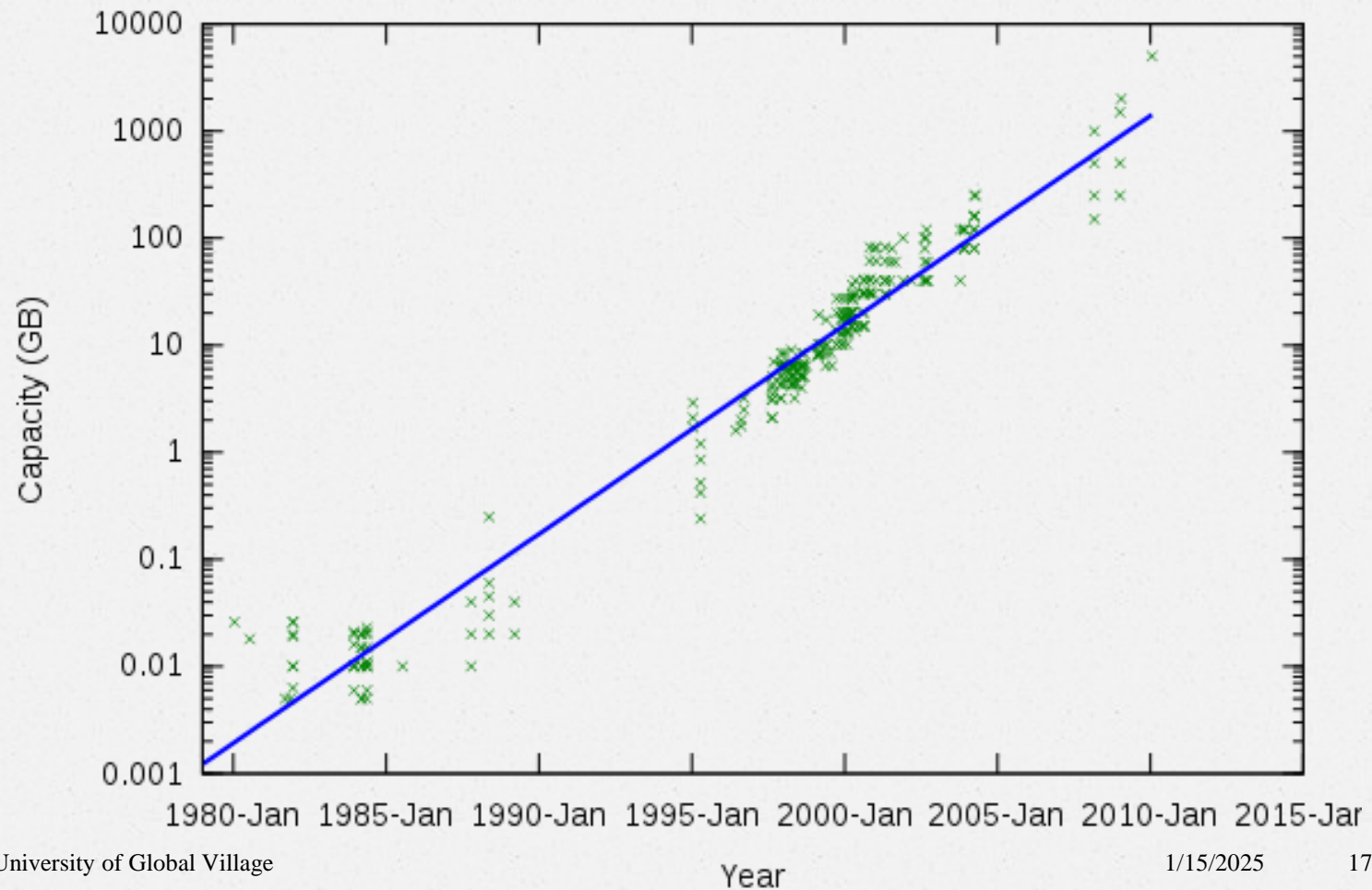
CPU Transistor Counts 1971-2008 & Moore's Law



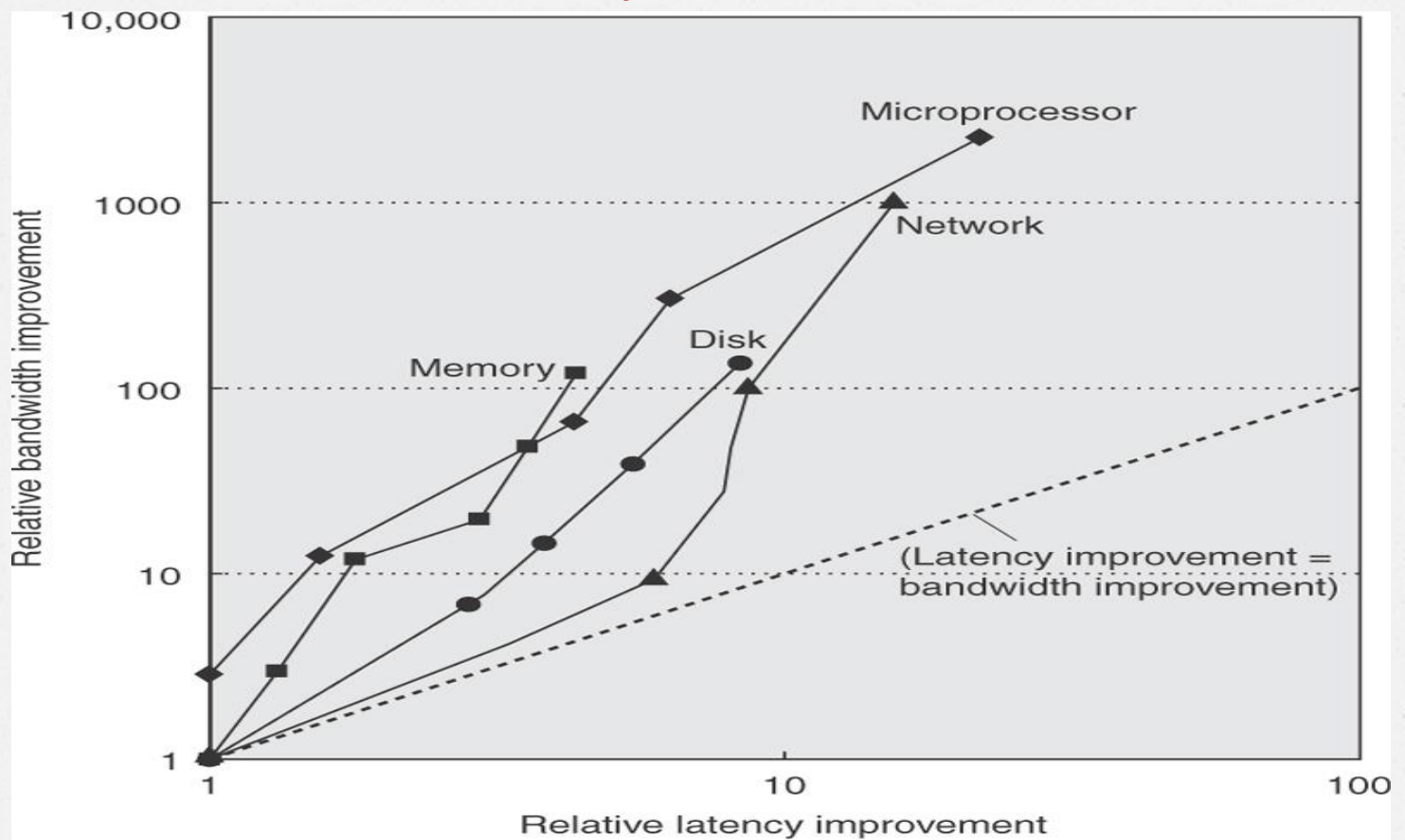
Evolution of memory



PC hard drive capacity



Bandwidth vs Latency

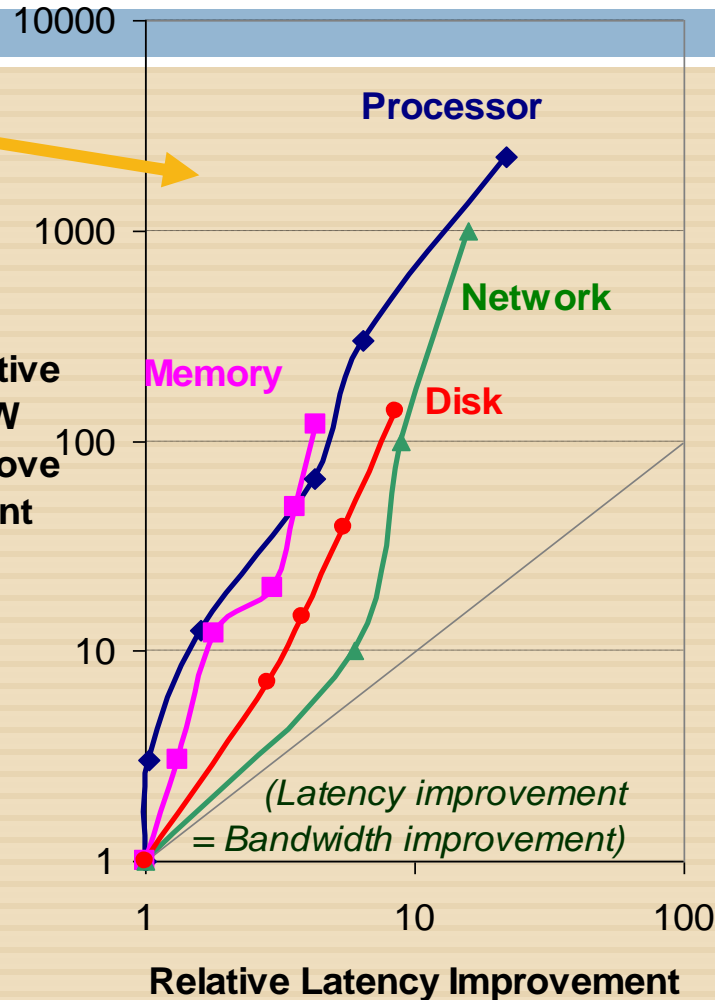


Latency Lags Bandwidth (last ~20 years)

19

**CPU high,
Memory low
("Memory
Wall")**

Relative
BW
Improve
ment



Performance Milestones

- Processor: '286, '386, '486, Pentium, Pentium Pro, Pentium 4 (21x, 2250x)
- Ethernet: 10Mb, 100Mb, 1000Mb, 10000 Mb/s (16x, 1000x)
- Memory Module: 16bit plain DRAM, Page Mode DRAM, 32b, 64b, SDRAM, DDR SDRAM (4x, 120x)
- Disk : 3600, 5400, 7200, 10000, 15000 RPM (8x, 143x)

Technology Trends (Summary)

20

	<u>Capacity</u>	<u>Speed (latency)</u>
Logic	2x in 3 years	2x in 3 years
DRAM	4x in 3 years	2x in 10 years
Disk	4x in 3 years	2x in 10 years

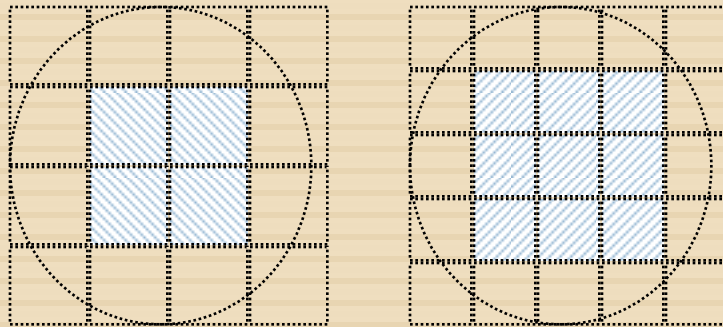
Integrated Circuits Costs

$$\text{IC cost} = \frac{\text{Die cost} + \text{Testing cost} + \text{Packaging cost}}{\text{Final test yield}}$$

$$\text{Die cost} = \frac{\text{Wafer cost}}{\text{Dies per Wafer} * \text{Die yield}}$$

Final test yield: fraction of packaged dies which pass the final testing state

Die yield: fraction of good dies on a wafer



What's the price of the final product ?

- **Component Costs**
- **Direct Costs** (add 25% to 40%) recurring costs: labor, purchasing, warranty
- **Gross Margin** (add 82% to 186%) nonrecurring costs: R&D, marketing, sales, equipment maintenance, rental, financing cost, pretax profits, taxes
- **Average Discount** to get List Price (add 33% to 66%): volume discounts and/or retailer markup



Quantitative Principles of Design

23

- Take Advantage of Parallelism
- Principle of Locality
- Focus on the Common Case
 - ▣ Amdahl's Law
 - ▣ E.g. common case supported by special hardware; uncommon cases in software
- The Performance Equation

1. Parallelism

24

How to improve performance?

- (Super)-pipelining
- Powerful instructions
 - ▣ MD-technique
 - multiple data operands per operation
 - ▣ MO-technique
 - multiple operations per instruction
- Multiple instruction issue
 - ▣ single instruction-program stream
 - ▣ multiple streams (or programs, or tasks)

2. The Principle of Locality

25

- Programs access a relatively small portion of the address space at any instant of time.
- Two Different Types of Locality:
 - Temporal Locality (Locality in Time): If an item is referenced, it will tend to be referenced again soon (e.g., loops, reuse)
 - Spatial Locality (Locality in Space): If an item is referenced, items whose addresses are close by tend to be referenced soon (e.g., straight-line code, array access)
- Last 30 years, HW relied on locality for memory perf.

3. Focus on the Common Case

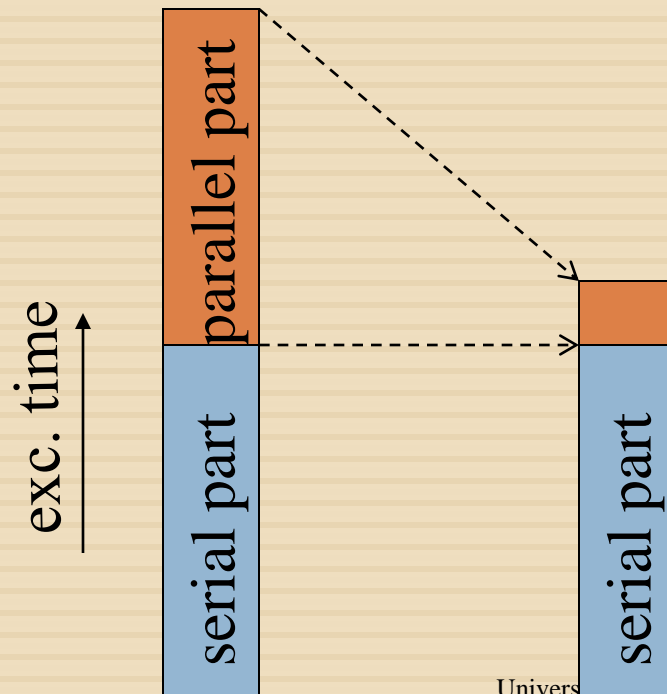
26

- Favor the frequent case over the infrequent case
 - E.g., Instruction fetch and decode unit used more frequently than multiplier, so optimize it 1st
 - E.g., If database server has 50 disks / processor, storage dependability dominates system dependability, so optimize it 1st
- Frequent case is often simpler and can be done faster than the infrequent case
 - E.g., overflow is rare when adding 2 numbers, so improve performance by optimizing more common case of no overflow
 - May slow down overflow, but overall performance improved by optimizing for the normal case
- What is frequent case? How much performance improved by making case faster? => **Amdahl's Law**

Amdahl's Law

27

$$\text{Speedup}_{\text{overall}} = \frac{T_{\text{exec,old}}}{T_{\text{exec,new}}} = \frac{1}{(1 - \text{Fraction}_{\text{enhanced}}) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}}}$$



Amdahl's Law

28

- Floating point instructions improved to run 2 times faster, but only 10% of actual instructions are FP

$$T_{\text{exec,new}} =$$

$$\text{Speedup}_{\text{overall}} =$$

Amdahl's Law

29

- Floating point instructions improved to run 2X; but only 10% of actual instructions are FP

$$T_{\text{exec,new}} = T_{\text{exec,old}} \times (0.9 + 0.1/2) = 0.95 \times T_{\text{exec,old}}$$

$$\text{Speedup}_{\text{overall}} = \frac{1}{0.95} = 1.053$$

4. The performance equation

30

- Main performance metric:

Total Execution Time

- $$\begin{aligned} T_{\text{exec}} &= N_{\text{cycles}} * T_{\text{cycle}} \\ &= N_{\text{instructions}} * \text{CPI} * T_{\text{cycle}} \end{aligned}$$

- **CPI:** Cycles Per Instruction

Example: Calculating CPI

31

Base Machine (Reg / Reg)

Op		Freq	Cycles	CPI(i)	(% Time)
ALU		50%	1	.5	(33%)
Load		20%	2	.4	(27%)
Store		10%	2	.2	(13%)
Branch		20%	2	.4	(27%)

1.5

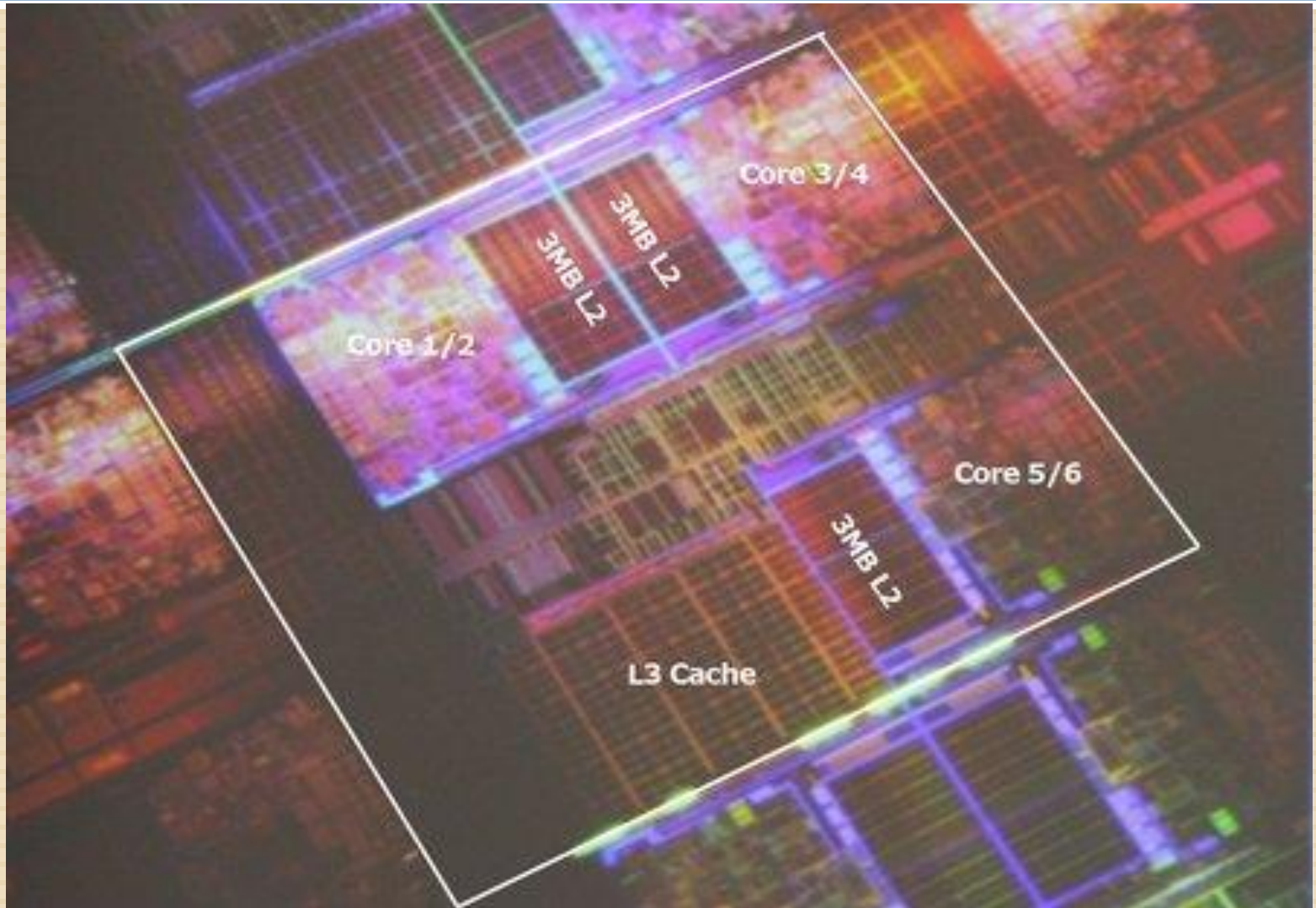
Typical Mix

What is Ahead?

- ❑ Bigger caches. More levels of cache? Software control.
- ❑ Greater **instruction level** parallelism?
- ❑ Increased exploiting **data level** parallelism:
 - ❑ Vector and Subword parallel processing
- ❑ Exploiting **task level** parallelism: Multiple processor cores per chip; how many are needed?
 - ❑ Bus based communication, or
 - ❑ Networks-on-Chip (NoC)
- ❑ Complete MP Systems on Chip: platforms
- ❑ Compute servers
- ❑ Cloud computing

Intel Dunnington 6-core

33



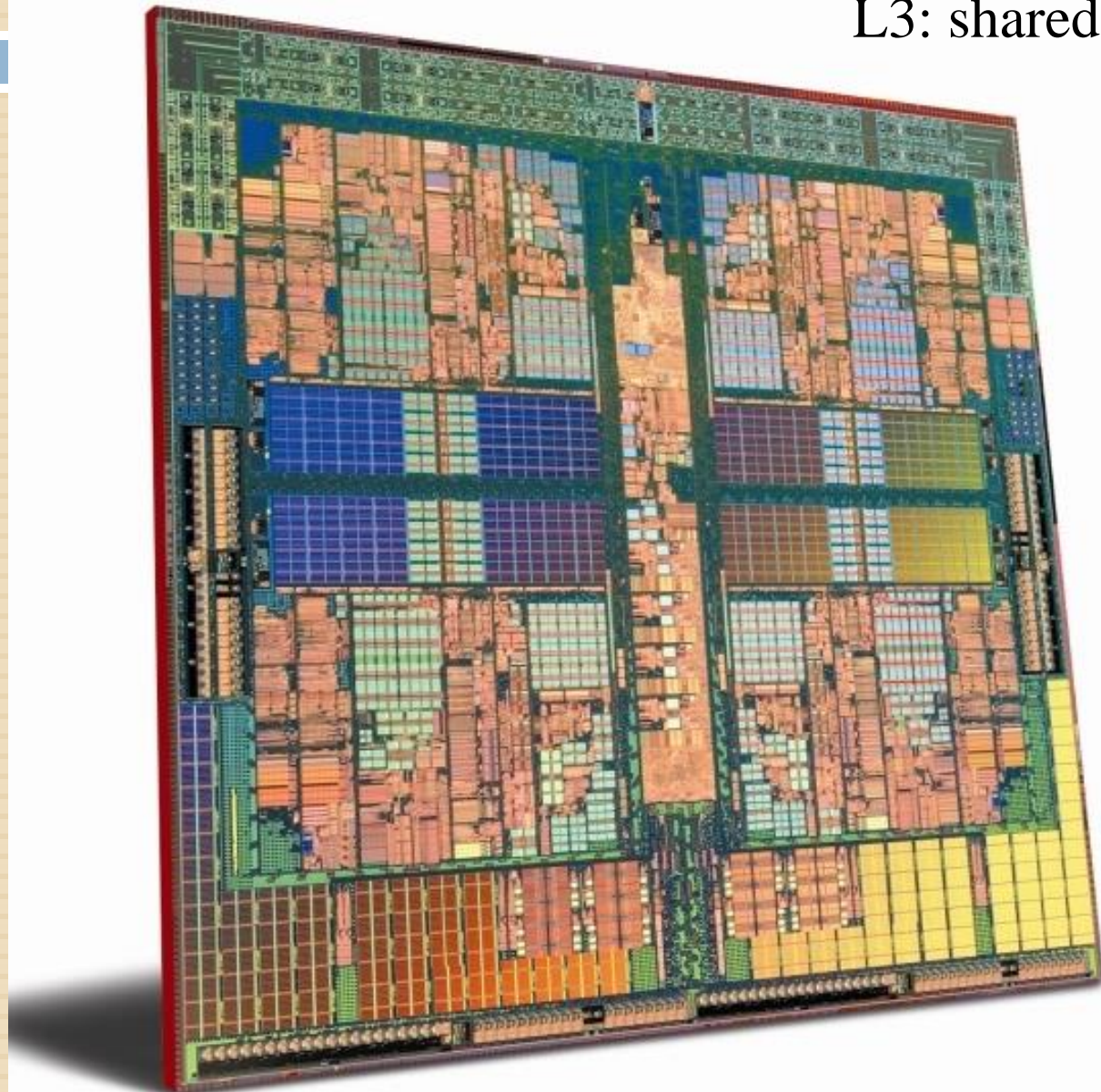
AMD Hydra 8 core

45 nm

L2: 1MByte/core

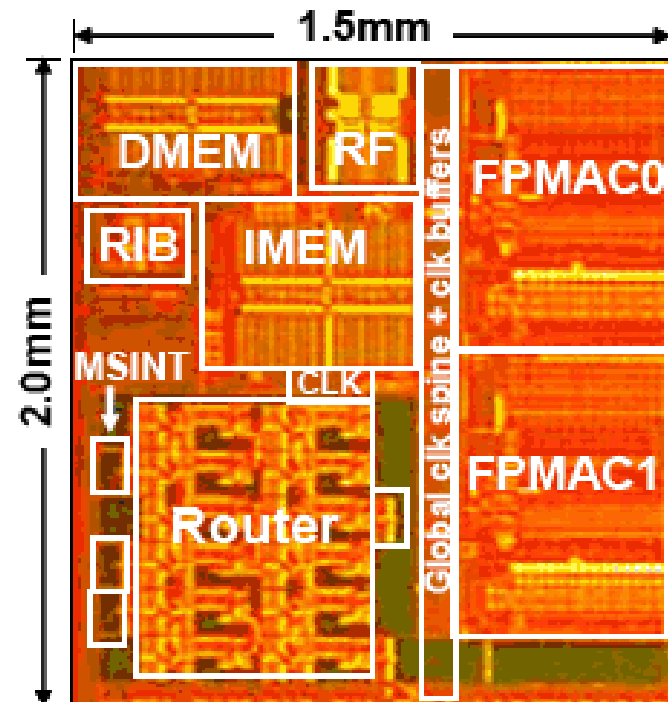
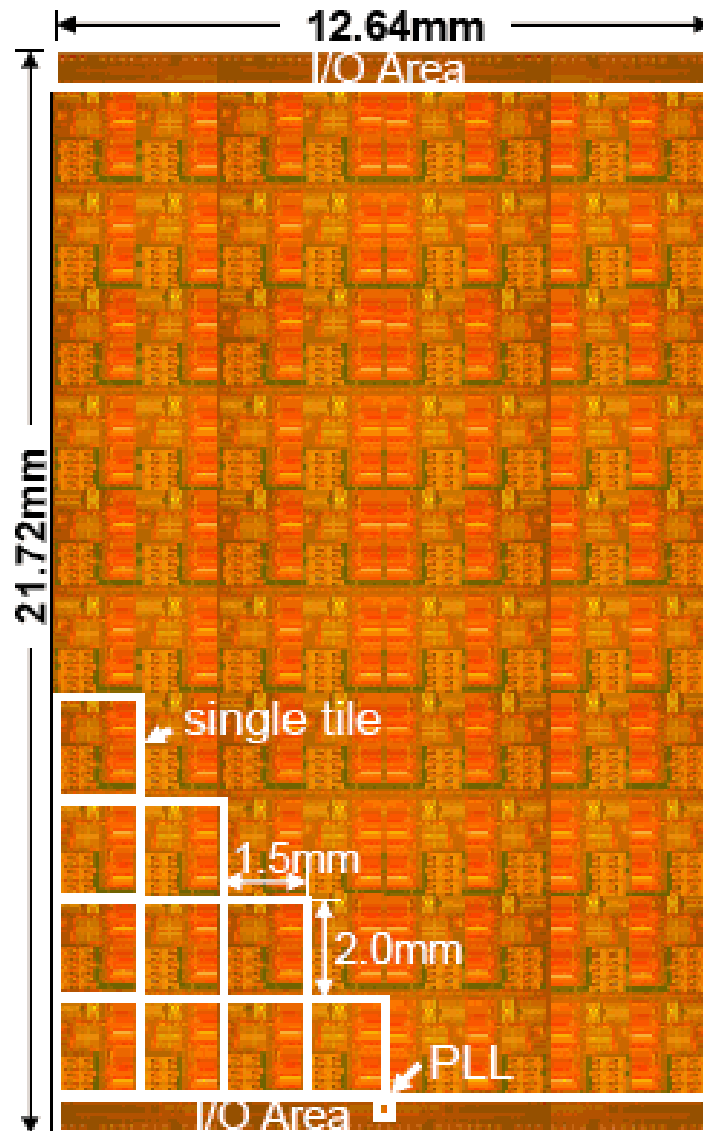
L3: shared 6MByte

34



Intel 80 processor die

35



Technology	65nm CMOS Process
Interconnect	1 poly, 8 metal (Cu)
Transistors	100 Million
Die Area	275mm ²
Tile area	3mm ²
Package	1248 pin LGA, 14 layers, 343 signal pins

Jaguar: performance nr 1 in 2009

36

- 220.000 cores
- 1.75 PetaFlop

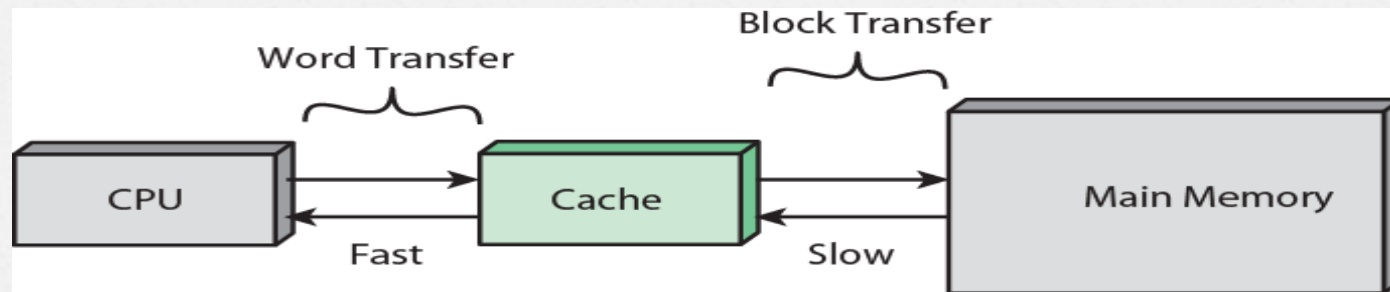


Week 2

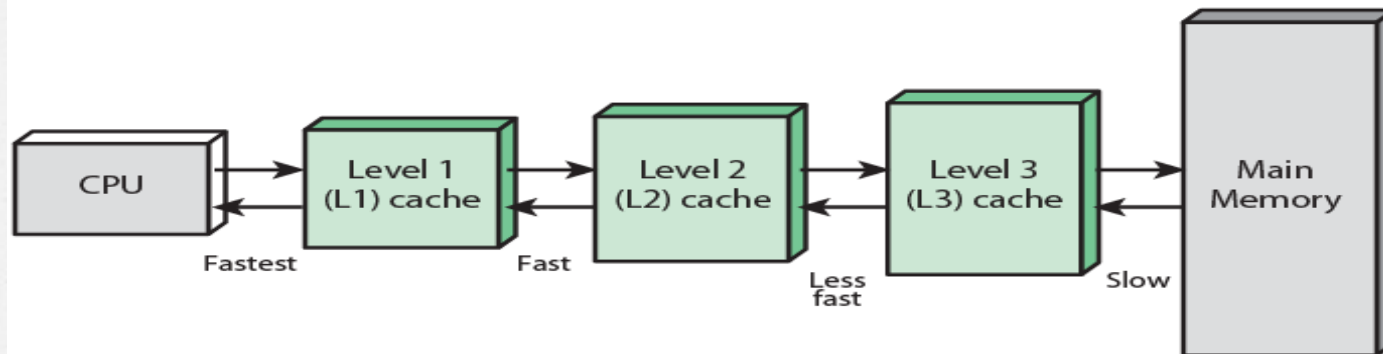
Cache Memory Design



Cache and Main Memory

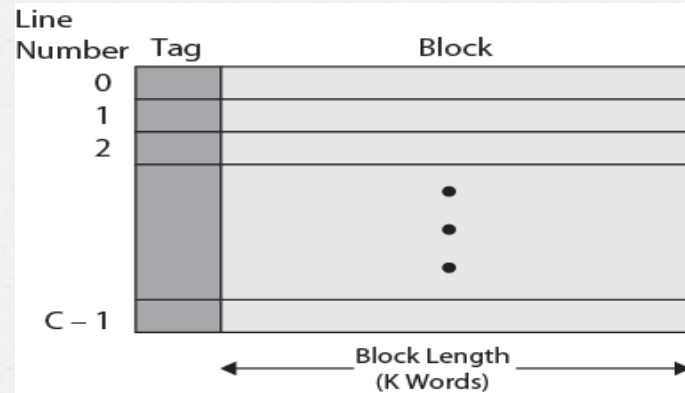


(a) Single cache

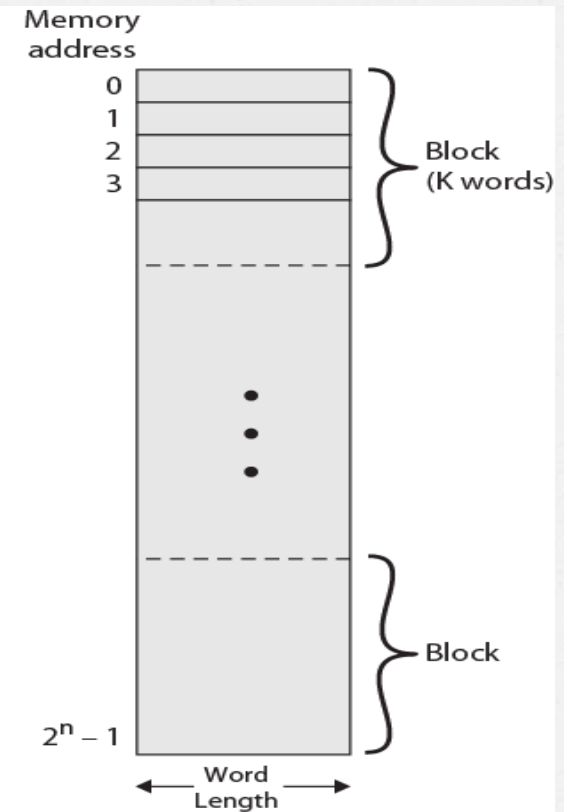


(b) Three-level cache organization

Cache/Main Memory Structure



(a) Cache

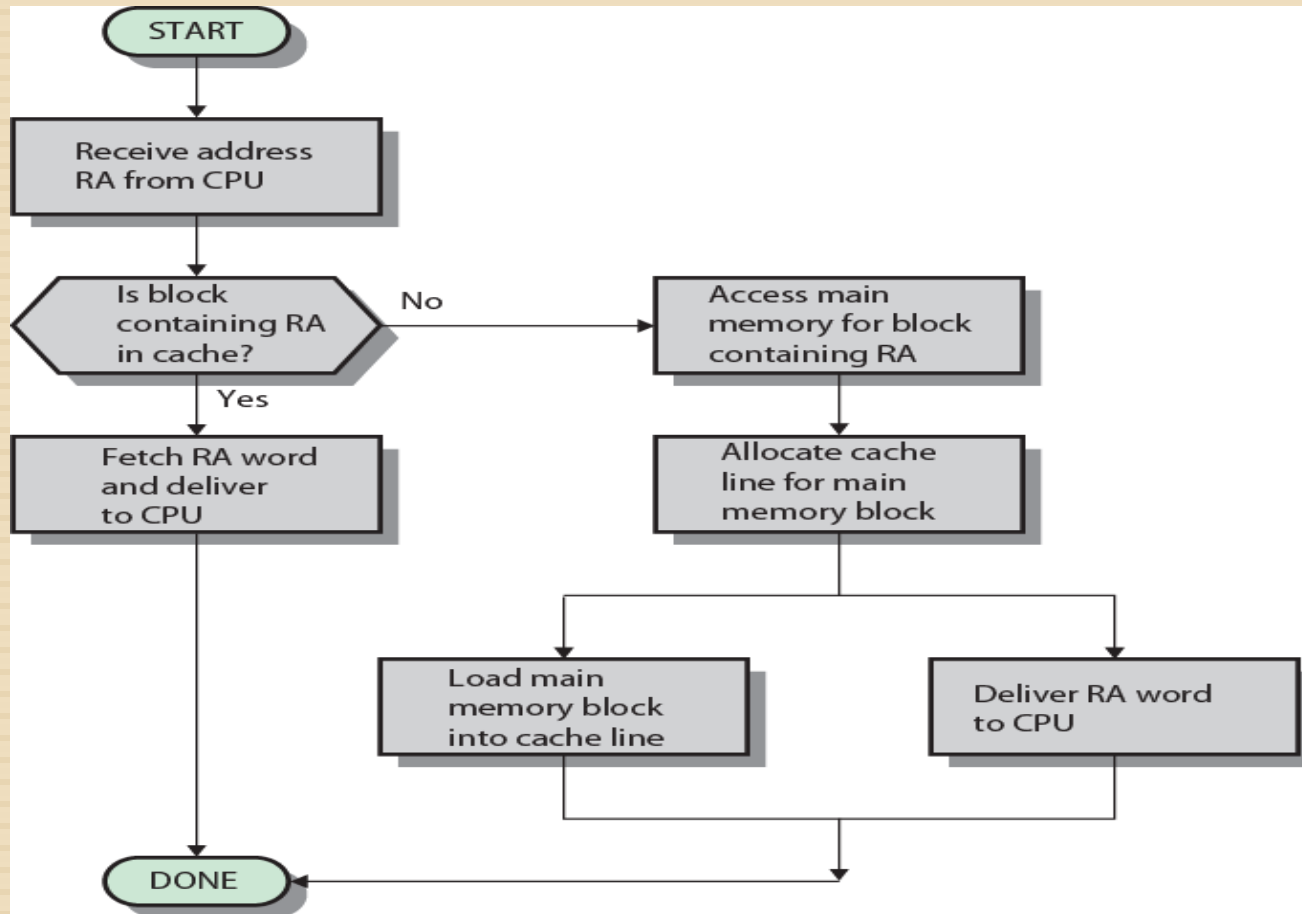


(b) Main memory

Cache operation – overview

- CPU requests contents of memory location
- Check cache for this data
- If present, get from cache (fast)
- If not present, read required block from main memory to cache
- Then deliver from cache to CPU
- Cache includes tags to identify which block of main memory is in each cache slot

Cache Read Operation - Flowchart



Cache Design

- Addressing
- Size
- Mapping Function
- Replacement Algorithm
- Write Policy
- Block Size
- Number of Caches

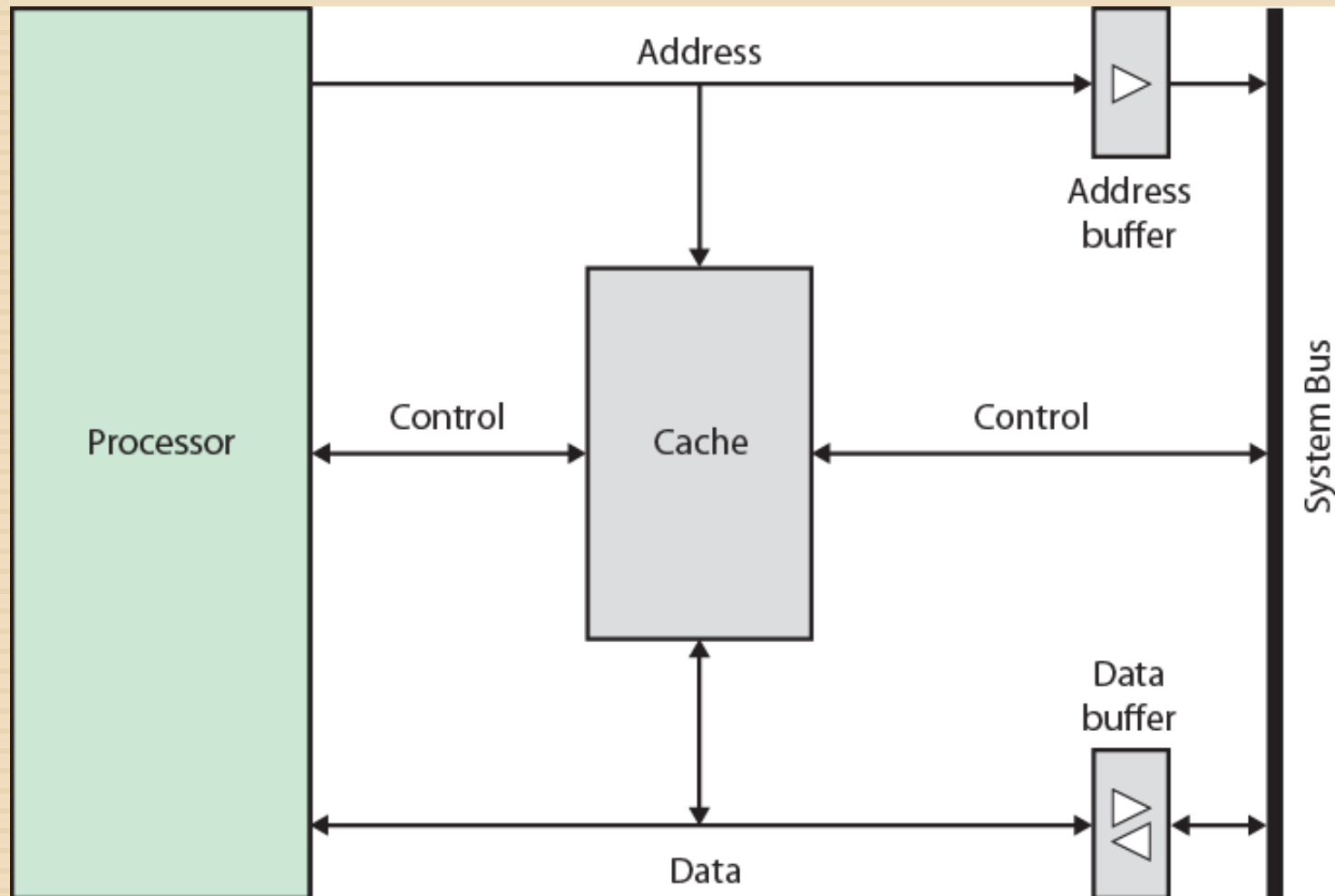
Cache Addressing

- Where does cache sit?
 - ▣ Between processor and virtual memory management unit
 - ▣ Between MMU and main memory
- Logical cache (virtual cache) stores data using virtual addresses
 - ▣ Processor accesses cache directly, not thorough physical cache
 - ▣ Cache access faster, before MMU address translation
 - ▣ Virtual addresses use same address space for different applications
 - Must flush cache on each context switch
- Physical cache stores data using main memory physical addresses

Size does matter

- Cost
 - ▣ More cache is expensive
- Speed
 - ▣ More cache is faster (up to a point)
 - ▣ Checking cache for data takes time

Typical Cache Organization



Mapping Function

- Cache of 64kByte
- Cache block of 4 bytes
 - ▣ i.e. cache is 16k (2^{14}) lines of 4 bytes
- 16MBytes main memory
- 24 bit address
 - ▣ ($2^{24}=16M$)

Direct Mapping

- Each block of main memory maps to only one cache line
 - i.e. if a block is in cache, it must be in one specific place
- Address is in two parts
- Least Significant w bits identify unique word
- Most Significant s bits specify one memory block
- The MSBs are split into a cache line field r and a tag of $s-r$ (most significant)

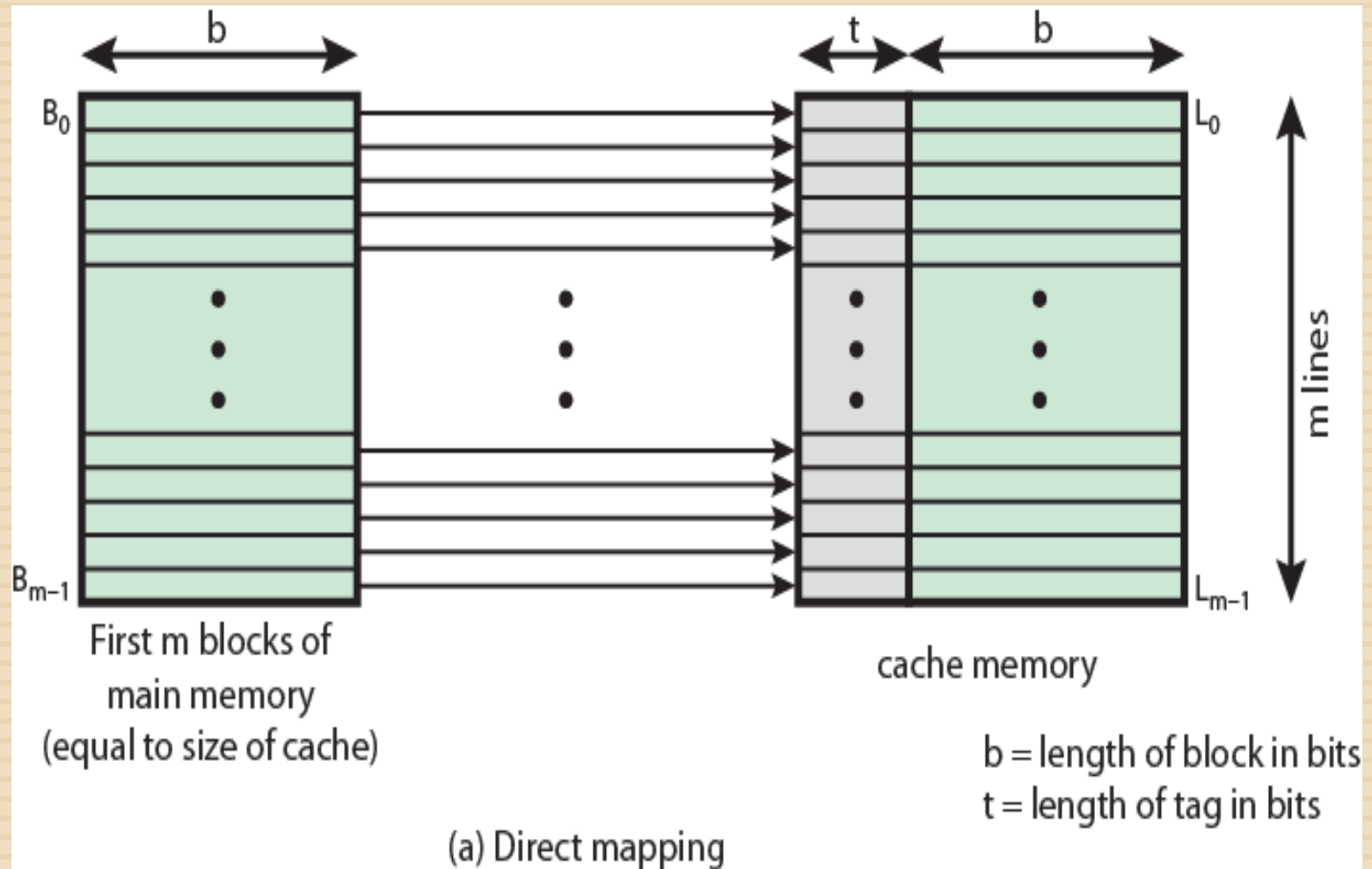
Direct Mapping

Address Structure

Tag s-r	Line or Slot r	Word w
8	14	2

- ❑ 24 bit address
- ❑ 2 bit word identifier (4 byte block)
- ❑ 22 bit block identifier
 - ▣ 8 bit tag (=22-14)
 - ▣ 14 bit slot or line
- ❑ No two blocks in the same line have the same Tag field
- ❑ Check contents of cache by finding line and checking Tag

Direct Mapping from Cache to Main Memory

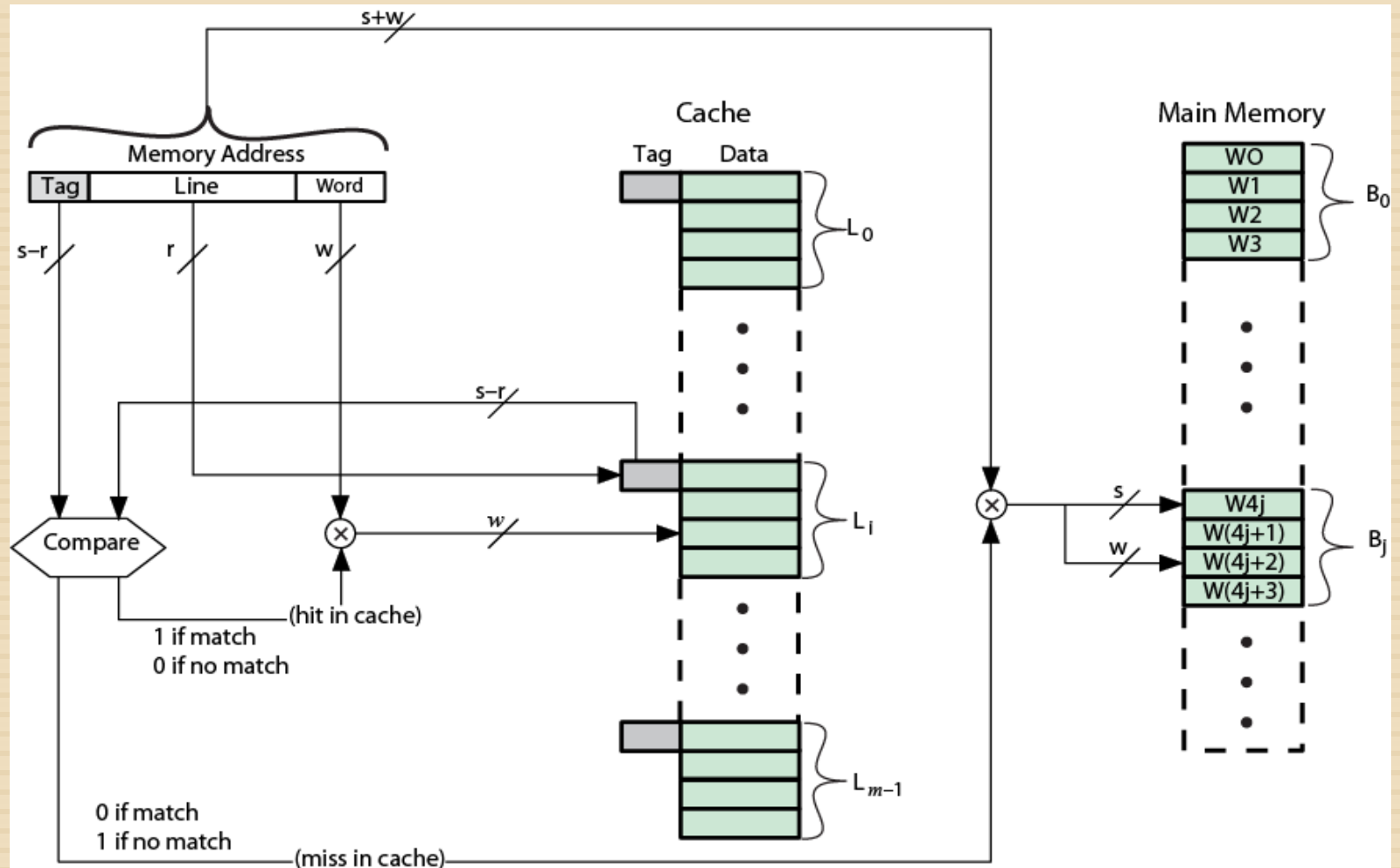


Direct Mapping

Cache Line Table

Cache line	Main Memory blocks held
0	0, m, 2m, 3m...2s-m
1	1,m+1, 2m+1...2s-m+1
...	
m-1	m-1, 2m-1,3m-1...2s-1

Direct Mapping Cache Organization



Week 3

I/O Systems

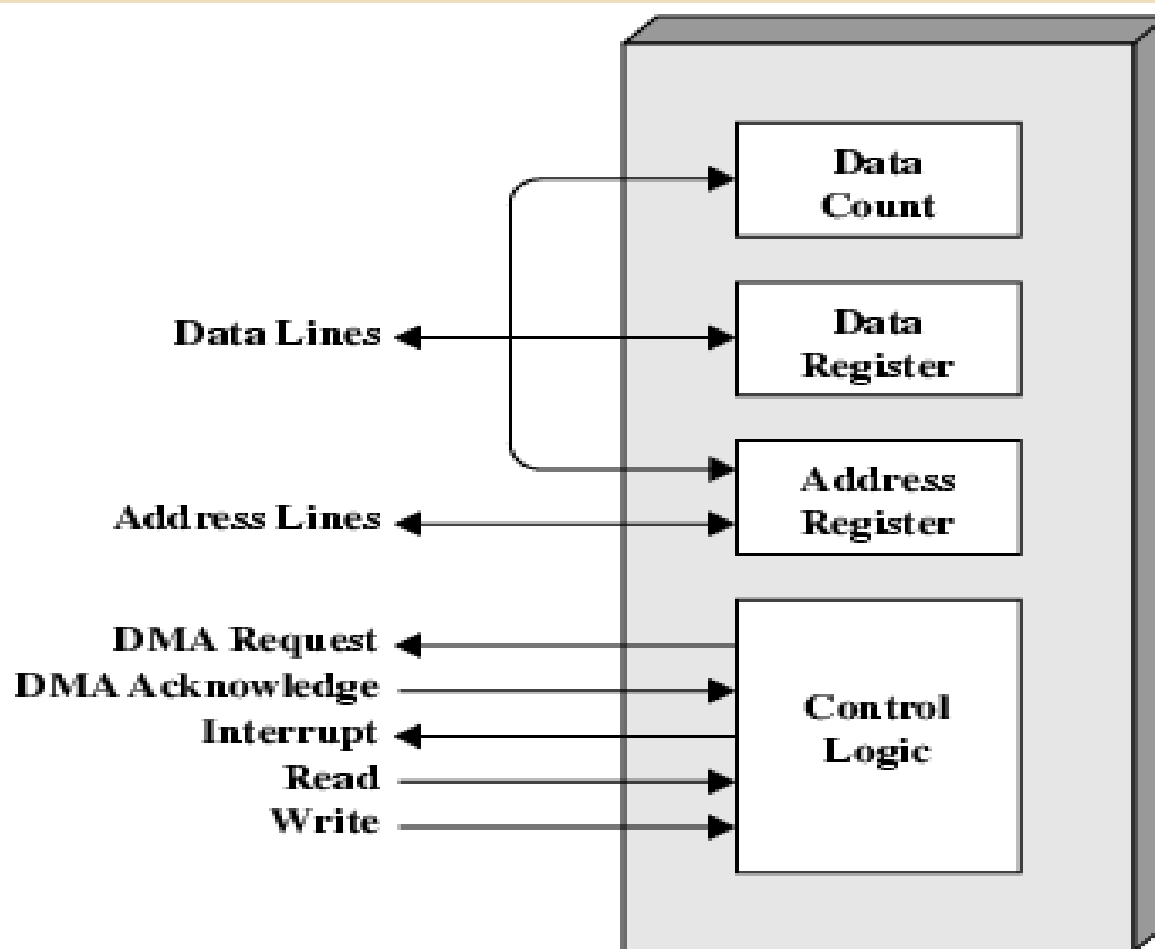
Direct Memory Access

- Interrupt driven and programmed I/O require active CPU intervention
 - ▣ Transfer rate is limited
 - ▣ CPU is tied up
- DMA is the answer

DMA Function

- Additional Module (hardware) on bus
- DMA controller takes over from CPU for I/O

Typical DMA Module Diagram



DMA Operation

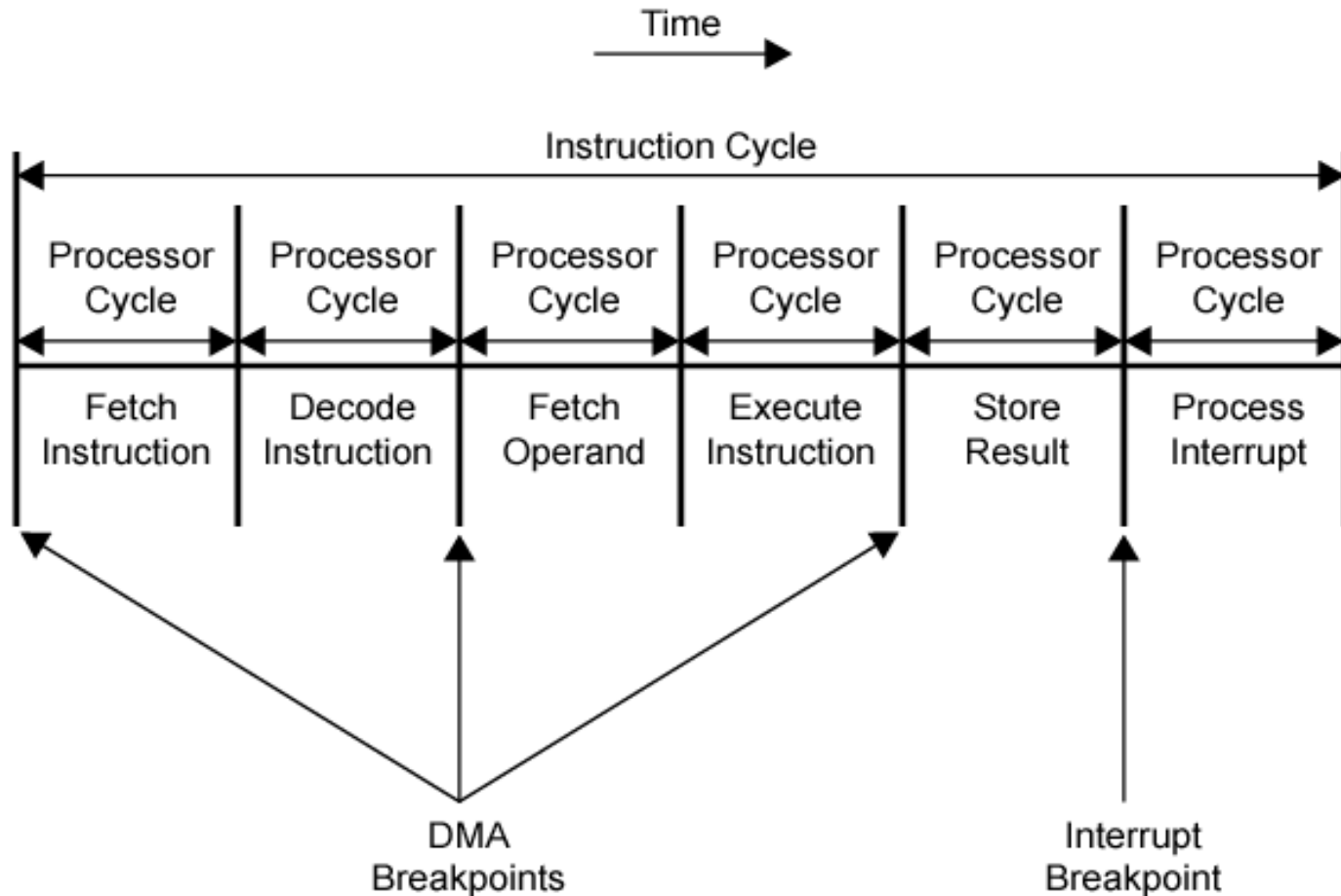
- CPU tells DMA controller:-
 - Read/Write
 - Device address
 - Starting address of memory block for data
 - Amount of data to be transferred
- CPU carries on with other work
- DMA controller deals with transfer
- DMA controller sends interrupt when finished

DMA Transfer

Cycle Stealing

- DMA controller takes over bus for a cycle
- Transfer of one word of data
- Not an interrupt
 - ▣ CPU does not switch context
- CPU suspended just before it accesses bus
 - ▣ i.e. before an operand or data fetch or a data write
- Slows down CPU but not as much as CPU doing transfer

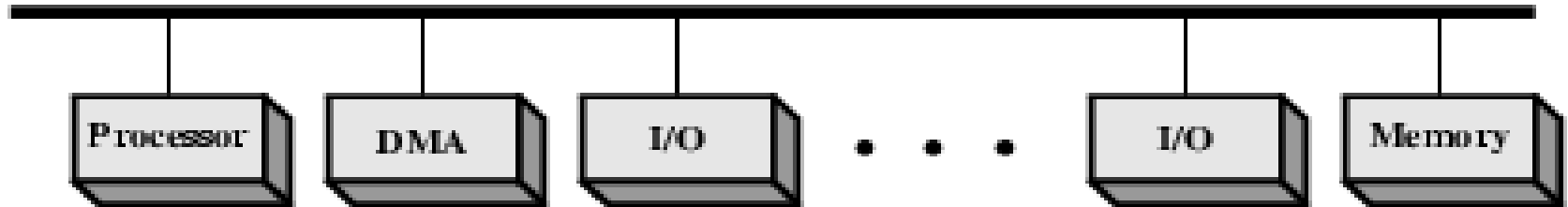
DMA and Interrupt Breakpoints During an Instruction Cycle



Aside

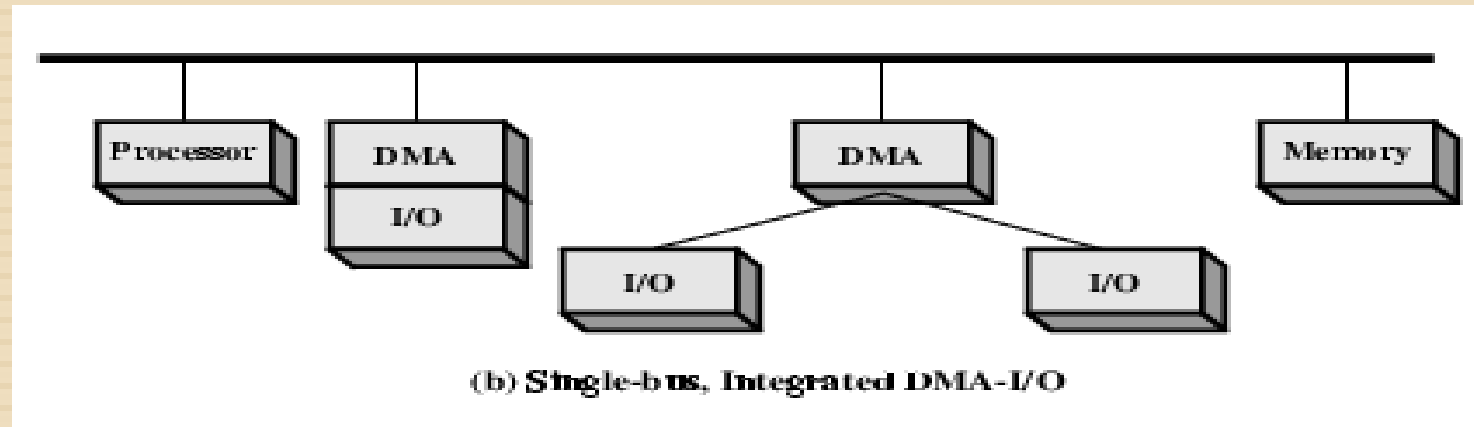
- What effect does caching memory have on DMA?
- What about on board cache?
- Hint: how much are the system buses available?

DMA Configurations (1)



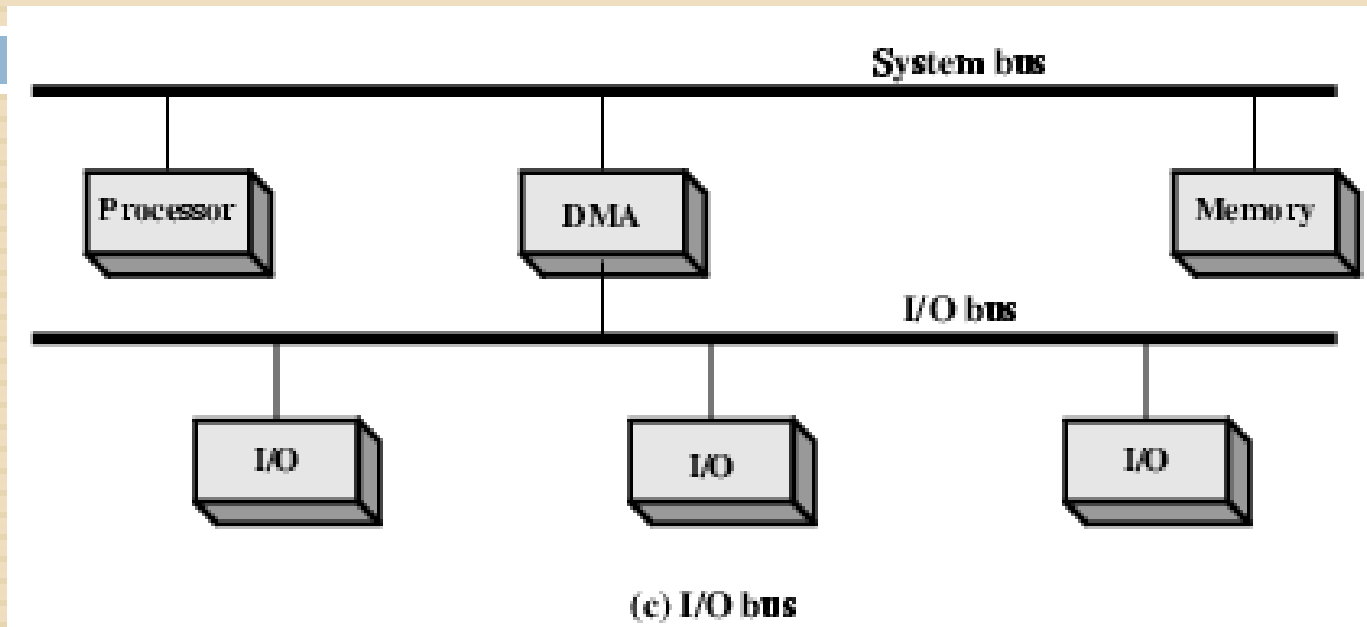
- ❑ Single Bus, Detached DMA controller
- ❑ Each transfer uses bus twice
 - ❑ I/O to DMA then DMA to memory
- ❑ CPU is suspended twice

DMA Configurations (2)



- ❑ Single Bus, Integrated DMA controller
- ❑ Controller may support >1 device
- ❑ Each transfer uses bus once
 - ❑ DMA to memory
- ❑ CPU is suspended once

DMA Configurations (3)

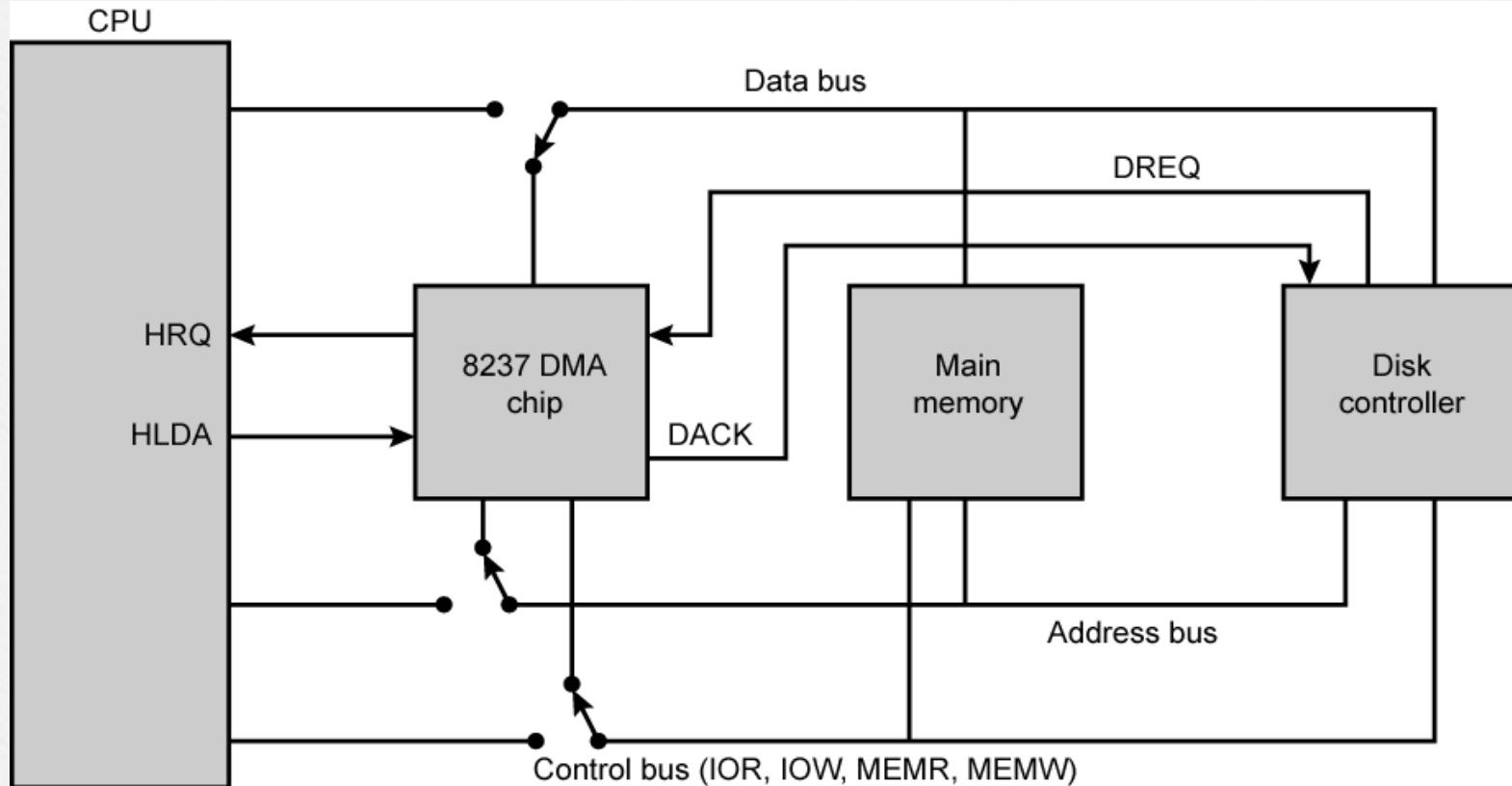


- ❑ Separate I/O Bus
- ❑ Bus supports all DMA enabled devices
- ❑ Each transfer uses bus once
 - ▣ DMA to memory
- ❑ CPU is suspended once

Intel 8237A DMA Controller

- Interfaces to 80x86 family and DRAM
- When DMA module needs buses it sends HOLD signal to processor
- CPU responds HLDA (hold acknowledge)
 - DMA module can use buses
- E.g. transfer data from memory to disk
 1. Device requests service of DMA by pulling DREQ (DMA request) high
 2. DMA puts high on HRQ (hold request),
 3. CPU finishes present bus cycle (not necessarily present instruction) and puts high on HDLA (hold acknowledge). HOLD remains active for duration of DMA
 4. DMA activates DACK (DMA acknowledge), telling device to start transfer
 5. DMA starts transfer by putting address of first byte on address bus and activating MEMR; it then activates IOW to write to peripheral. DMA decrements counter and increments address pointer. Repeat until count reaches zero
 6. DMA deactivates HRQ, giving bus back to CPU

8237 DMA Usage of Systems



Week 4

Multicore Processor

Multi Core Processor Hardware Performance Issues

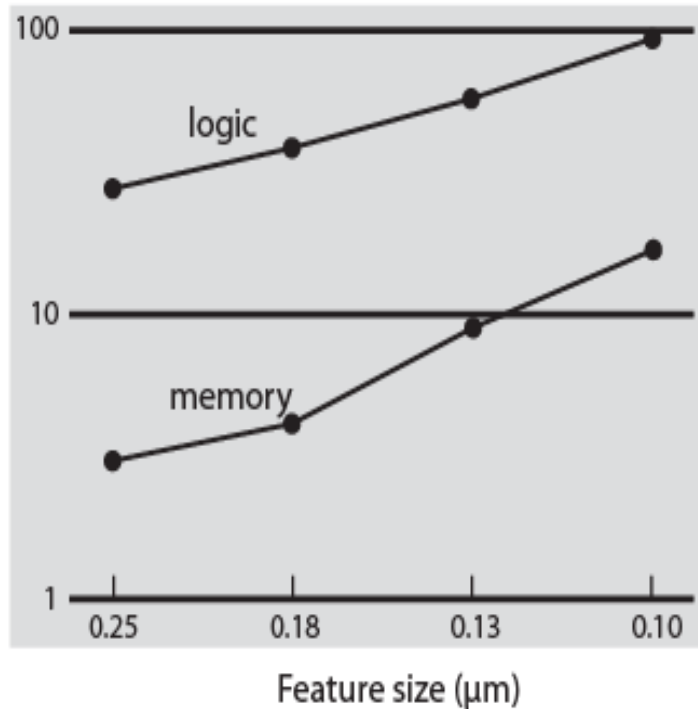
- Microprocessors have seen an exponential increase in performance
 - ▣ Improved organization
 - ▣ Increased clock frequency
- Increase in Parallelism
 - ▣ Pipelining
 - ▣ Superscalar
 - ▣ Simultaneous multithreading (SMT)
- Diminishing returns
 - ▣ More complexity requires more logic
 - ▣ Increasing chip area for coordinating and signal transfer logic
 - Harder to design, make and debug

Increased Complexity

- Power requirements grow exponentially with chip density and clock frequency
 - Can use more chip area for cache
 - Smaller
 - Order of magnitude lower power requirements
- By 2015
 - 100 billion transistors on 300mm² die
 - Cache of 100MB
 - 1 billion transistors for logic
- Pollack's rule:
 - Performance is roughly proportional to square root of increase in complexity
 - Double complexity gives 40% more performance
- Multicore has potential for near-linear improvement
- Unlikely that one core can use all cache effectively

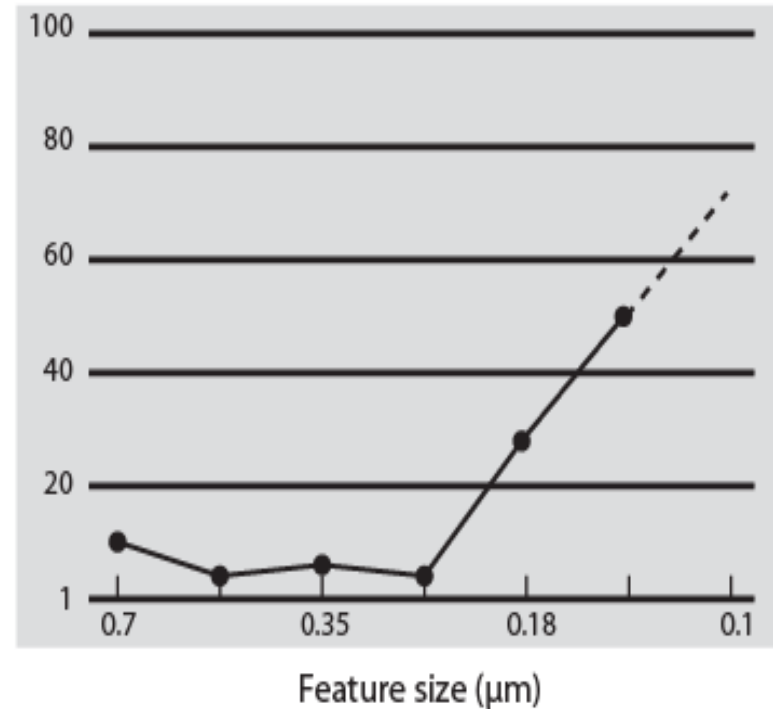
Power and Memory Considerations

Power density
(watts/cm²)



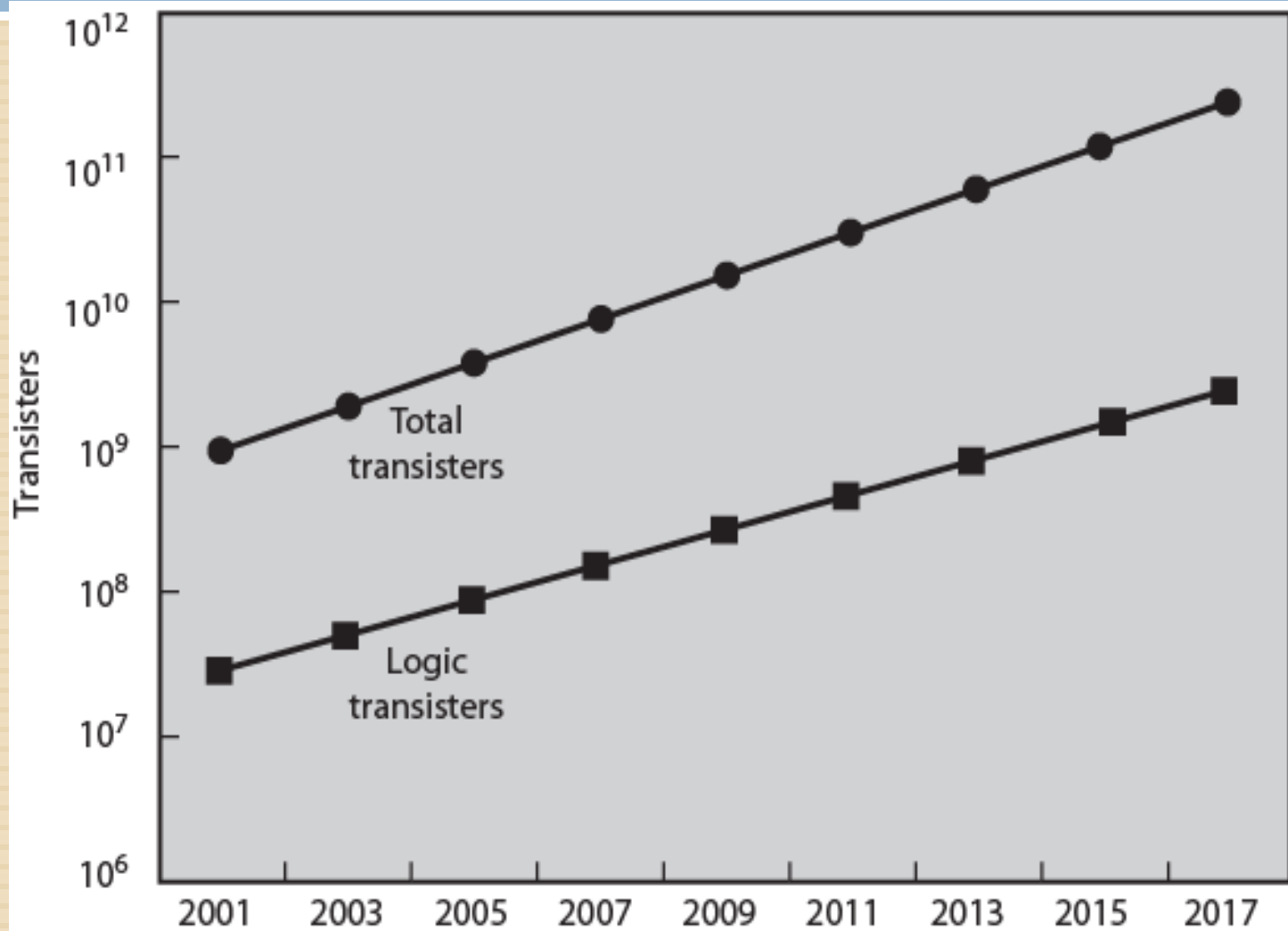
(a) Power density

cache percent
of full chip area



(b) Chip area

Chip Utilization of Transistors



Software Performance Issues

- Performance benefits dependent on effective exploitation of parallel resources
- Even small amounts of serial code impact performance
 - ▣ 10% inherently serial on 8 processor system gives only 4.7 times performance
- Communication, distribution of work and cache coherence overheads
- Some applications effectively exploit multicore processors

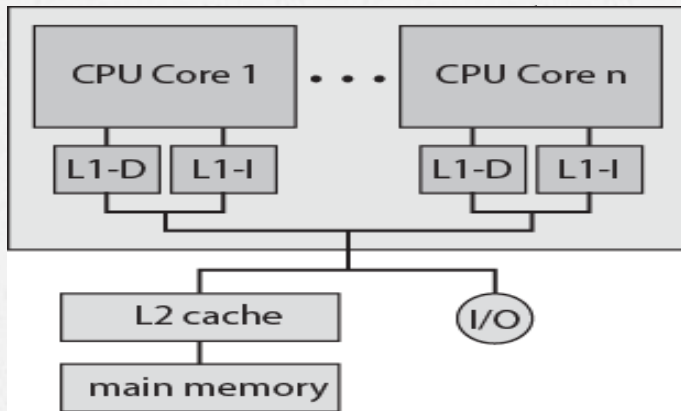
Effective Applications for Multicore Processors

- Database
- Servers handling independent transactions
- Multi-threaded native applications
 - ▣ Lotus Domino, Siebel CRM
- Multi-process applications
 - ▣ Oracle, SAP, PeopleSoft
- Java applications
 - ▣ Java VM is multi-thread with scheduling and memory management
 - ▣ Sun's Java Application Server, BEA's Weblogic, IBM Websphere, Tomcat
- Multi-instance applications
 - ▣ One application running multiple times
- E.g. Value Game Software

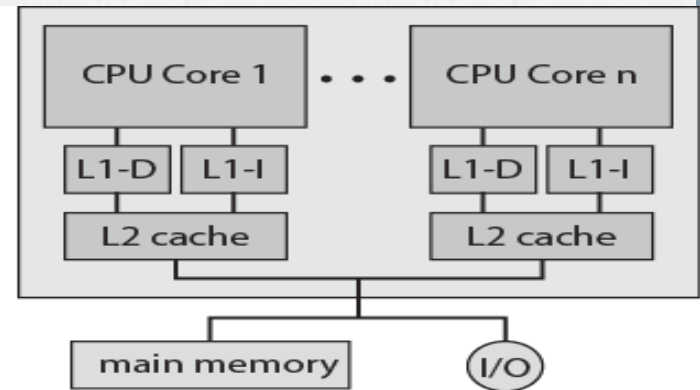
Multicore Organization

- Number of core processors on chip
- Number of levels of cache on chip
- Amount of shared cache
- Next slide examples of each organization:
 - (a) ARM11 MPCore
 - (b) AMD Opteron
 - (c) Intel Core Duo
 - (d) Intel Core i7

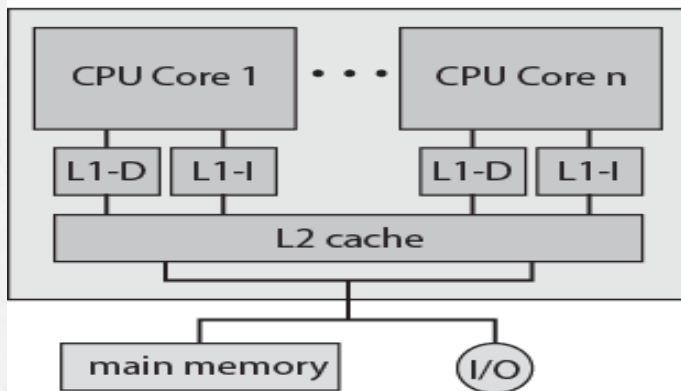
Multicore Organization



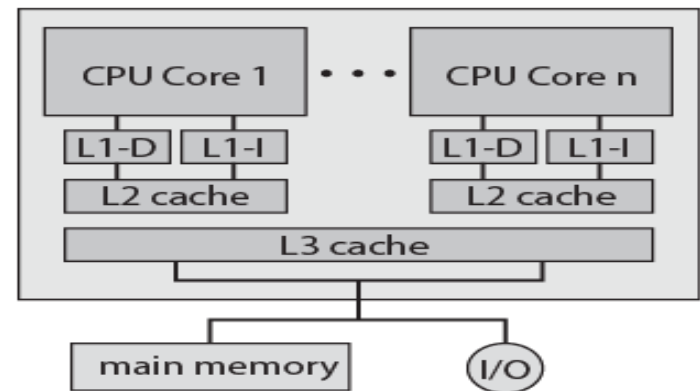
(a) Dedicated L1 cache



(b) Dedicated L2 cache



(c) Shared L2 cache



(d) Shared L3 cache

Advantages of shared L2 Cache

- Constructive interference reduces overall miss rate
- Data shared by multiple cores not replicated at cache level
- With proper frame replacement algorithms mean amount of shared cache dedicated to each core is dynamic
 - ▣ Threads with less locality can have more cache
- Easy inter-process communication through shared memory
- Cache coherency confined to L1
- Dedicated L2 cache gives each core more rapid access
 - ▣ Good for threads with strong locality
- Shared L3 cache may also improve performance

Individual Core Architecture

- Intel Core Duo uses superscalar cores
- Intel Core i7 uses simultaneous multi-threading (SMT)
 - ▣ Scales up number of threads supported
 - 4 SMT cores, each supporting 4 threads appears as 16 core

Intel x86 Multicore Organization - Core Duo (1)

- 2006
- Two x86 superscalar, shared L2 cache
- Dedicated L1 cache per core
 - ▣ 32KB instruction and 32KB data
- Thermal control unit per core
 - ▣ Manages chip heat dissipation
 - ▣ Maximize performance within constraints
 - ▣ Improved ergonomics
- Advanced Programmable Interrupt Controlled (APIC)
 - ▣ Inter-process interrupts between cores
 - ▣ Routes interrupts to appropriate core
 - ▣ Includes timer so OS can interrupt core

Intel x86 Multicore Organization -Core Duo (2)

- Power Management Logic
 - ▣ Monitors thermal conditions and CPU activity
 - ▣ Adjusts voltage and power consumption
 - ▣ Can switch individual logic subsystems
- 2MB shared L2 cache
 - ▣ Dynamic allocation
 - ▣ MESI support for L1 caches
 - ▣ Extended to support multiple Core Duo in SMP
 - L2 data shared between local cores or external
- Bus interface

Intel x86 Multicore Organization -Core i7

- ❑ November 2008
- ❑ Four x86 SMT processors
- ❑ Dedicated L2, shared L3 cache
- ❑ Speculative pre-fetch for caches
- ❑ On chip DDR3 memory controller
 - ❑ Three 8 byte channels (192 bits) giving 32GB/s
 - ❑ No front side bus
- ❑ QuickPath Interconnection
 - ❑ Cache coherent point-to-point link
 - ❑ High speed communications between processor chips
 - ❑ 6.4G transfers per second, 16 bits per transfer
 - ❑ Dedicated bi-directional pairs
 - ❑ Total bandwidth 25.6GB/s

ARM11 MPCore

- Up to 4 processors each with own L1 instruction and data cache
- Distributed interrupt controller
- Timer per CPU
- Watchdog
 - ▣ Warning alerts for software failures
 - ▣ Counts down from predetermined values
 - ▣ Issues warning at zero
- CPU interface
 - ▣ Interrupt acknowledgement, masking and completion acknowledgement
- CPU
 - ▣ Single ARM11 called MP11
- Vector floating-point unit
 - ▣ FP co-processor
- L1 cache
- Snoop control unit
 - ▣ L1 cache coherency

ARM11 MPCore Interrupt Handling

- ❑ Distributed Interrupt Controller (DIC) collates from many sources
- ❑ Masking
- ❑ Prioritization
- ❑ Distribution to target MP11 CPUs
- ❑ Status tracking
- ❑ Software interrupt generation
- ❑ Number of interrupts independent of MP11 CPU design
- ❑ Memory mapped
- ❑ Accessed by CPUs via private interface through SCU
- ❑ Can route interrupts to single or multiple CPUs
- ❑ Provides inter-process communication
 - Thread on one CPU can cause activity by thread on another CPU

DIC Routing

- ❑ Direct to specific CPU
- ❑ To defined group of CPUs
- ❑ To all CPUs
- ❑ OS can generate interrupt to:
 - ❑ All but self
 - ❑ Self
 - ❑ Other specific CPU
- ❑ Typically combined with shared memory for inter-process communication
- ❑ 16 interrupt ids available for inter-process communication

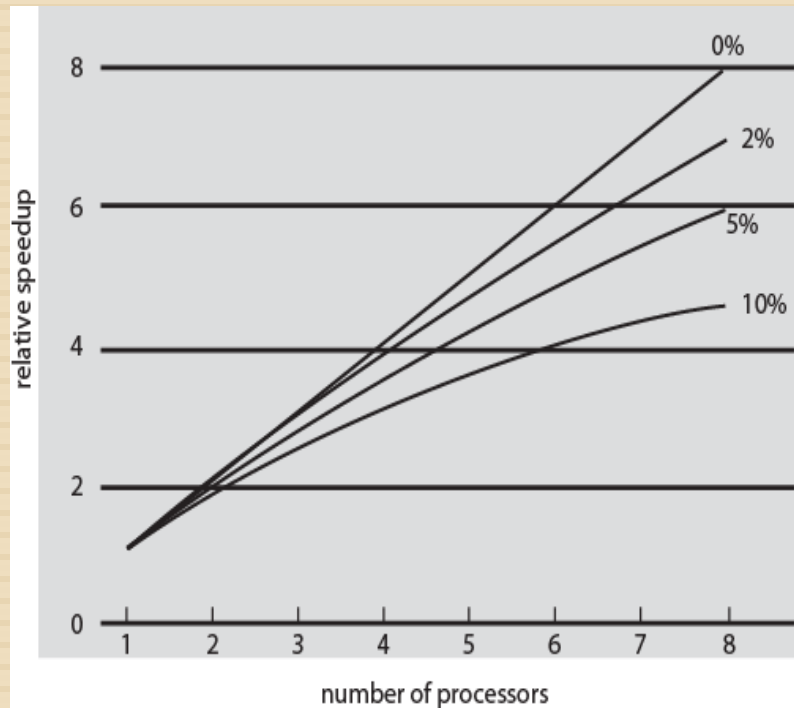
Interrupt States

- Inactive
 - ▣ Non-asserted
 - ▣ Completed by that CPU but pending or active in others
- Pending
 - ▣ Asserted
 - ▣ Processing not started on that CPU
- Active
 - ▣ Started on that CPU but not complete
 - ▣ Can be pre-empted by higher priority interrupt

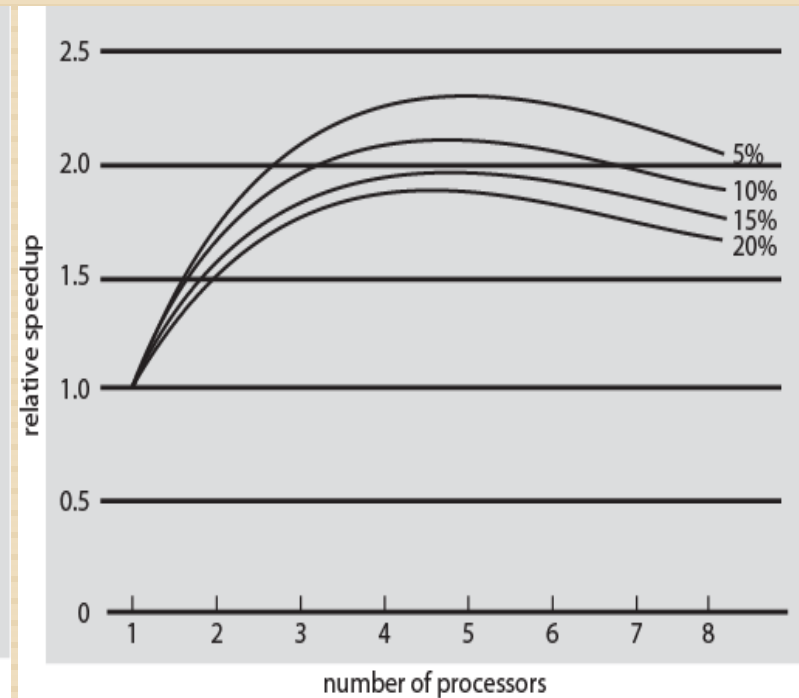
Interrupt Sources

- Inter-process Interrupts (IPI)
 - ▣ Private to CPU
 - ▣ ID0-ID15
 - ▣ Software triggered
 - ▣ Priority depends on target CPU not source
- Private timer and/or watchdog interrupt
 - ▣ ID29 and ID30
- Legacy FIQ line
 - ▣ Legacy FIQ pin, per CPU, bypasses interrupt distributor
 - ▣ Directly drives interrupts to CPU
- Hardware
 - ▣ Triggered by programmable events on associated interrupt lines
 - ▣ Up to 224 lines
 - ▣ Start at ID32

Performance Effect of Multiple Cores



(a) Speedup with 0%, 2%, 5%, and 10% sequential portions



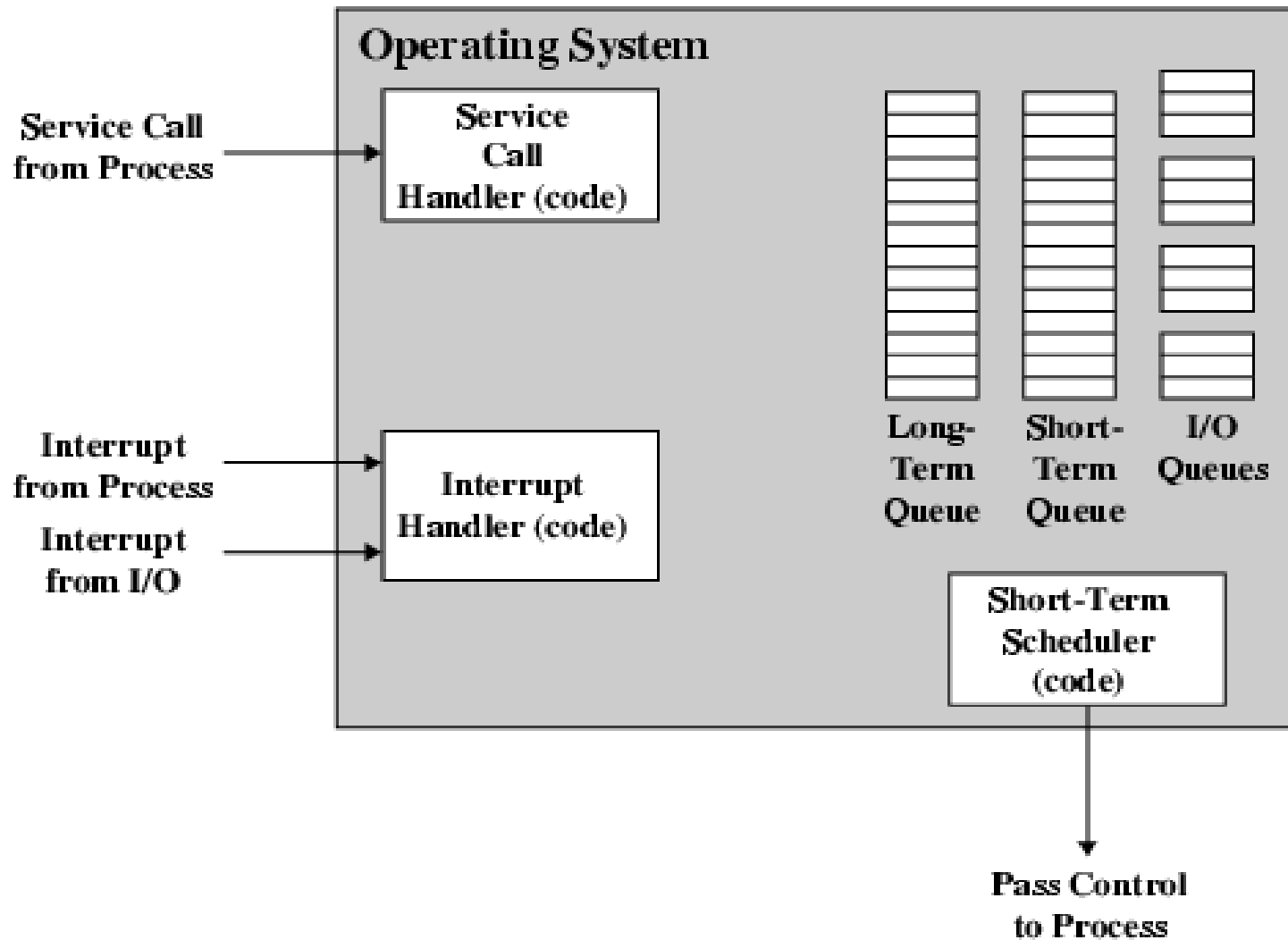
(b) Speedup with overheads

Week 5

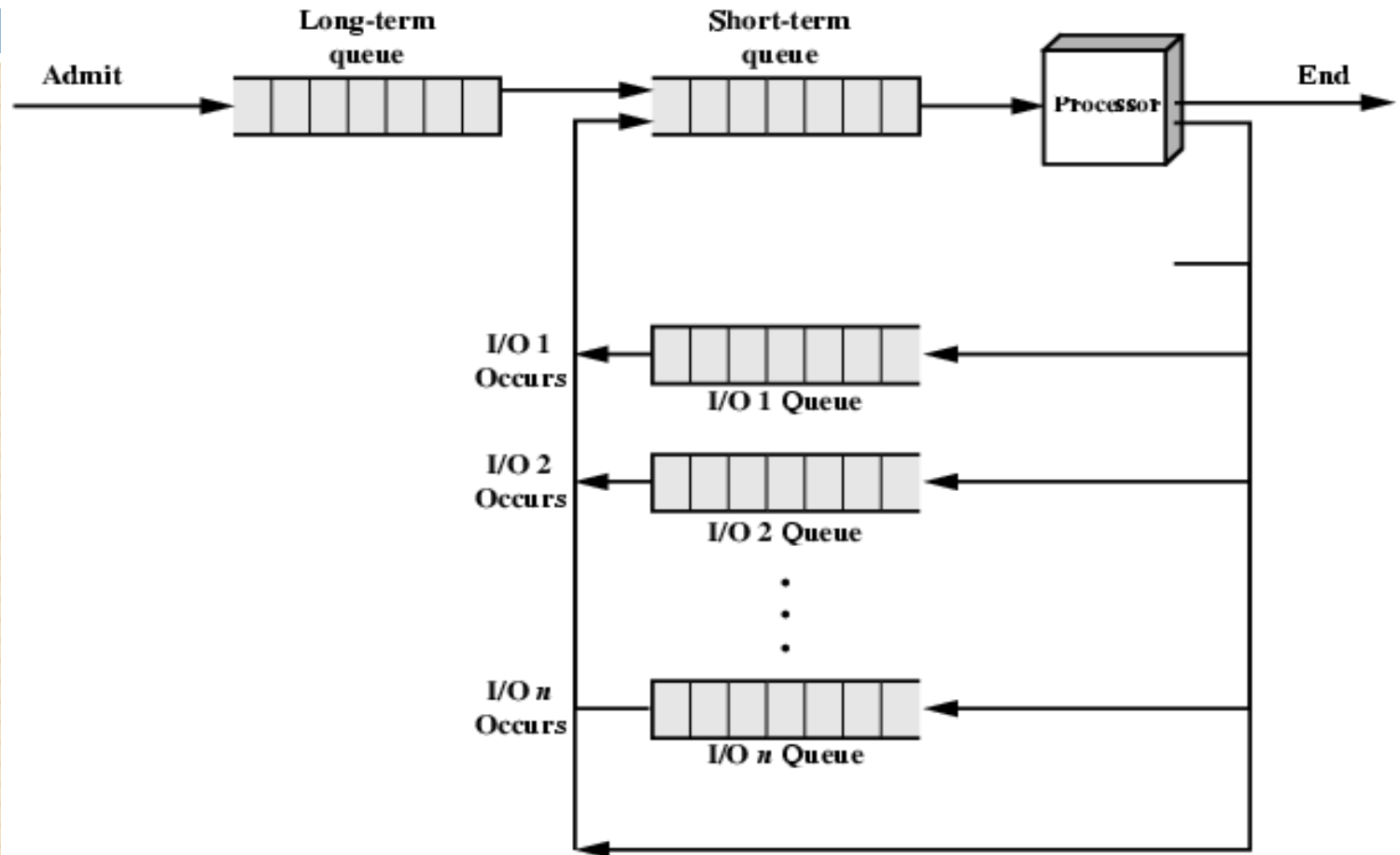
86

Operating System Support

Key Elements of O/S



Process Scheduling



Memory Management

- Uni-program
 - ▣ Memory split into two
 - ▣ One for Operating System (monitor)
 - ▣ One for currently executing program
- Multi-program
 - ▣ “User” part is sub-divided and shared among active processes

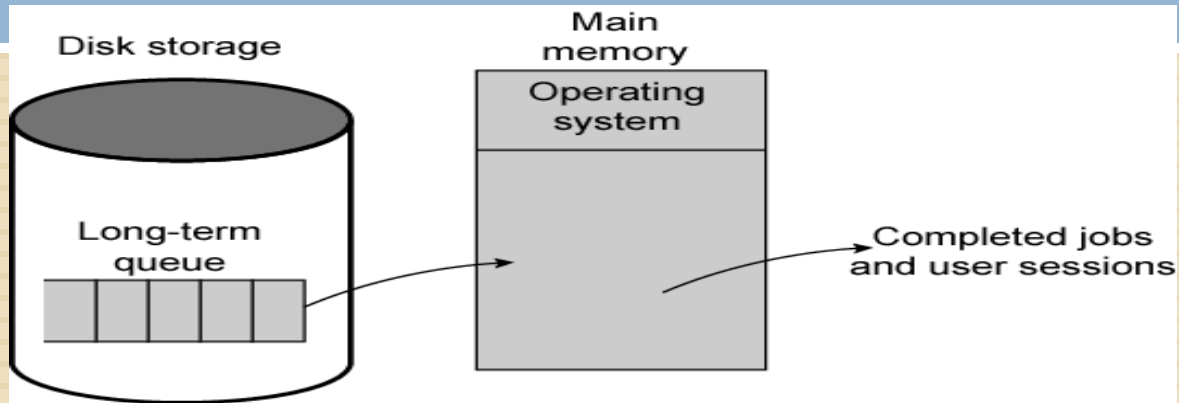
Swapping

- Problem: I/O is so slow compared with CPU that even in multi-programming system, CPU can be idle most of the time
- Solutions:
 - ▣ Increase main memory
 - Expensive
 - Leads to larger programs
 - ▣ Swapping

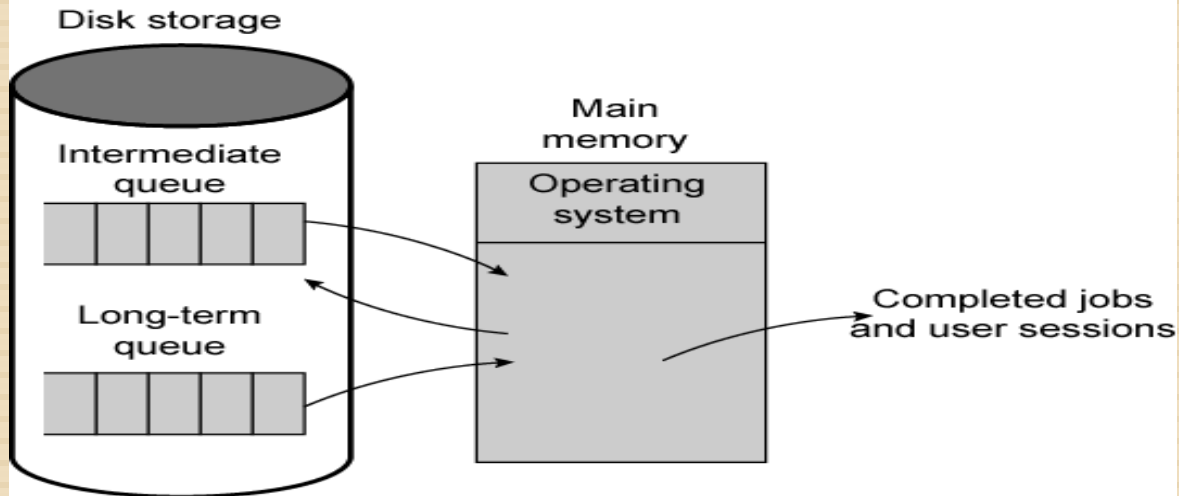
What is Swapping?

- Long term queue of processes stored on disk
- Processes “swapped” in as space becomes available
- As a process completes it is moved out of main memory
- If none of the processes in memory are ready (i.e. all I/O blocked)
 - ▣ Swap out a blocked process to intermediate queue
 - ▣ Swap in a ready process or a new process
 - ▣ But swapping is an I/O process...

Use of Swapping



(a) Simple job scheduling

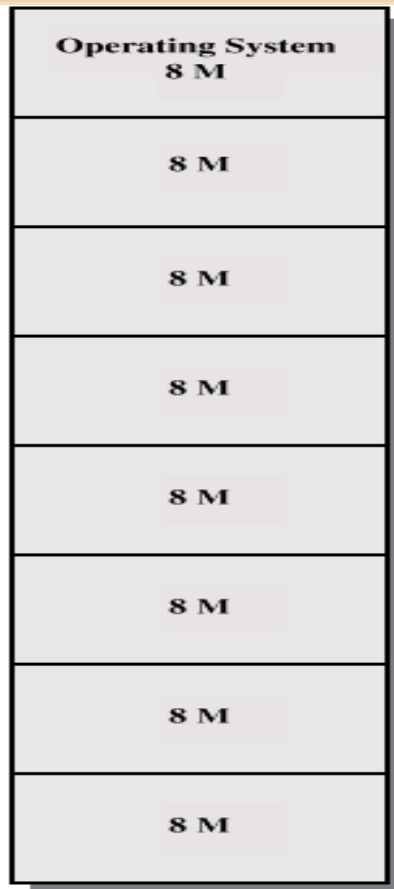


(b) Swapping

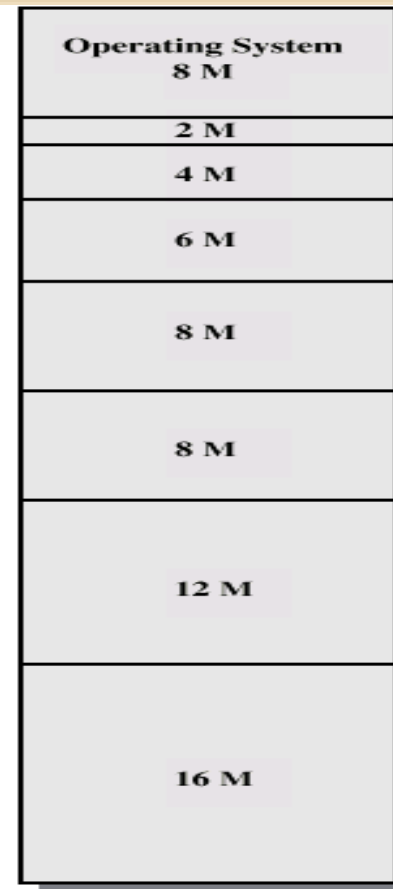
Partitioning

- Splitting memory into sections to allocate to processes (including Operating System)
- Fixed-sized partitions
 - ▣ May not be equal size
 - ▣ Process is fitted into smallest hole that will take it (best fit)
 - ▣ Some wasted memory
 - ▣ Leads to variable sized partitions

Fixed Partitioning



(a) Equal-size partitions



(b) Unequal-size partitions

Variable Sized Partitions (1)

- Allocate exactly the required memory to a process
- This leads to a hole at the end of memory, too small to use
 - ▣ Only one small hole - less waste
- When all processes are blocked, swap out a process and bring in another
- New process may be smaller than swapped out process
- Another hole

Variable Sized Partitions (2)

- Eventually have lots of holes (fragmentation)
- Solutions:
 - Coalesce - Join adjacent holes into one large hole
 - Compaction - From time to time go through memory and move all hole into one free block (c.f. disk de-fragmentation)

Week 6

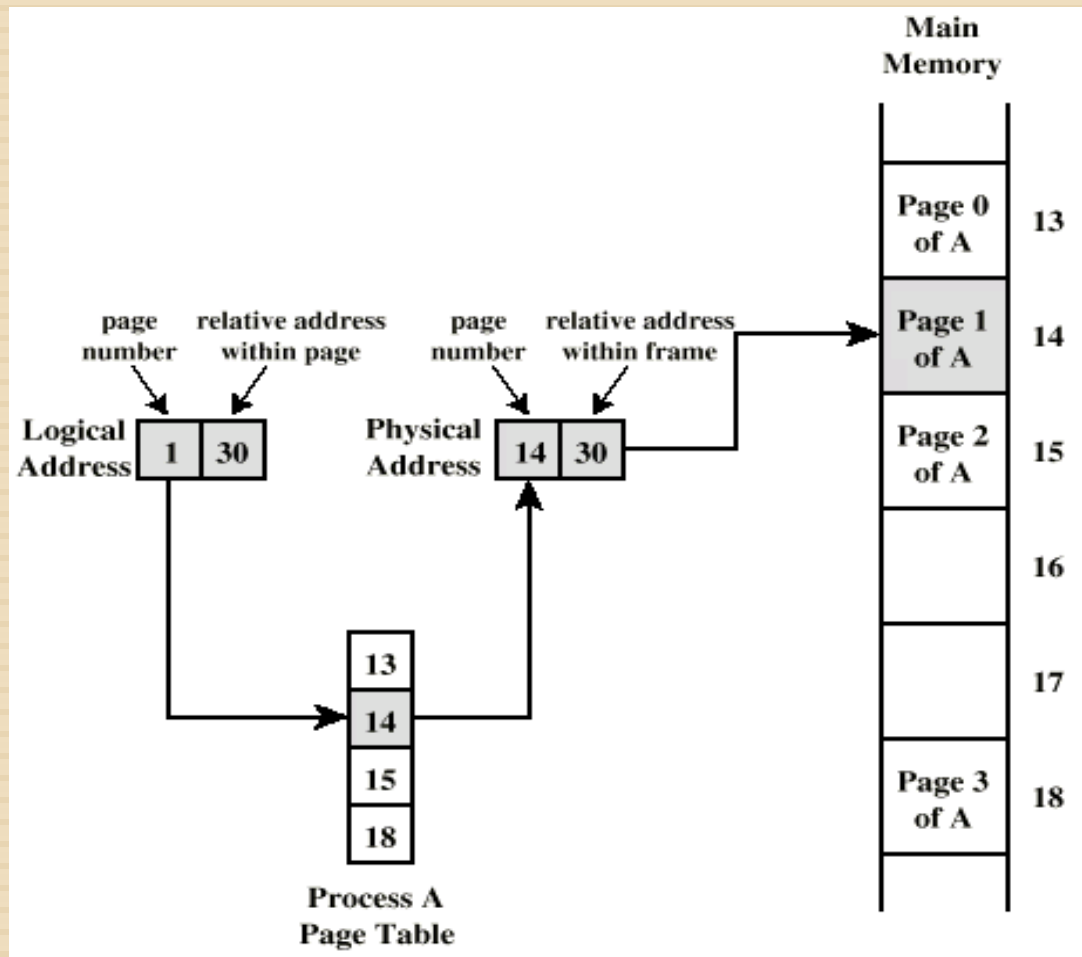
97

Operating System Support -2

Paging

- Split memory into equal sized, small chunks -page frames
- Split programs (processes) into equal sized small chunks - pages
- Allocate the required number page frames to a process
- Operating System maintains list of free frames
- A process does not require contiguous page frames
- Use page table to keep track

Logical and Physical Addresses - Paging



Virtual Memory

- Demand paging
 - ▣ Do not require all pages of a process in memory
 - ▣ Bring in pages as required
- Page fault
 - ▣ Required page is not in memory
 - ▣ Operating System must swap in required page
 - ▣ May need to swap out a page to make space
 - ▣ Select page to throw out based on recent history

Thrashing

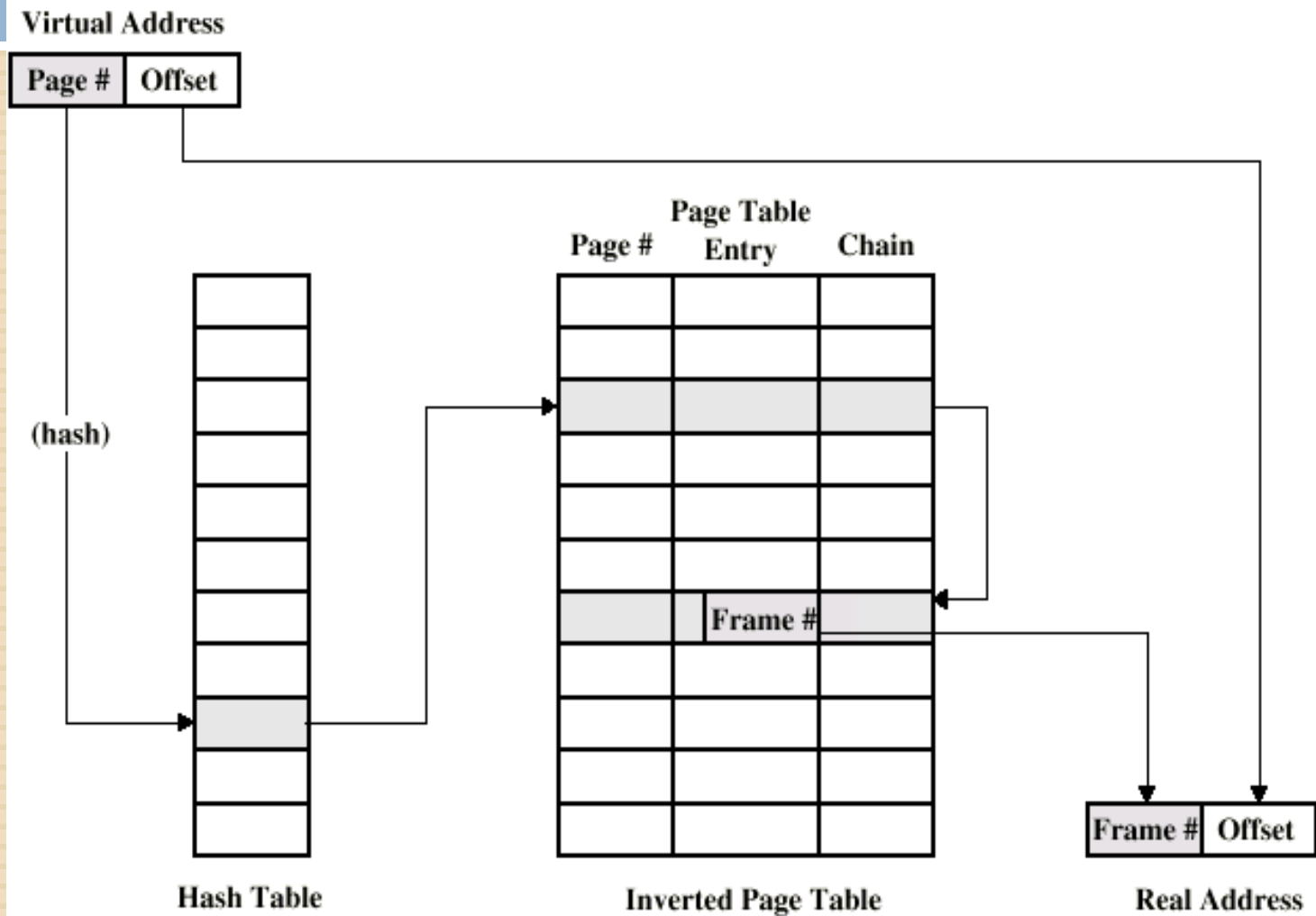
- ❑ Too many processes in too little memory
- ❑ Operating System spends all its time swapping
- ❑ Little or no real work is done
- ❑ Disk light is on all the time

- ❑ Solutions
 - ▣ Good page replacement algorithms
 - ▣ Reduce number of processes running
 - ▣ Fit more memory

Bonus

- We do not need all of a process in memory for it to run
 - We can swap in pages as required
 - So - we can now run processes that are bigger than total memory available!
-
- Main memory is called real memory
 - User/programmer sees much bigger memory - virtual memory

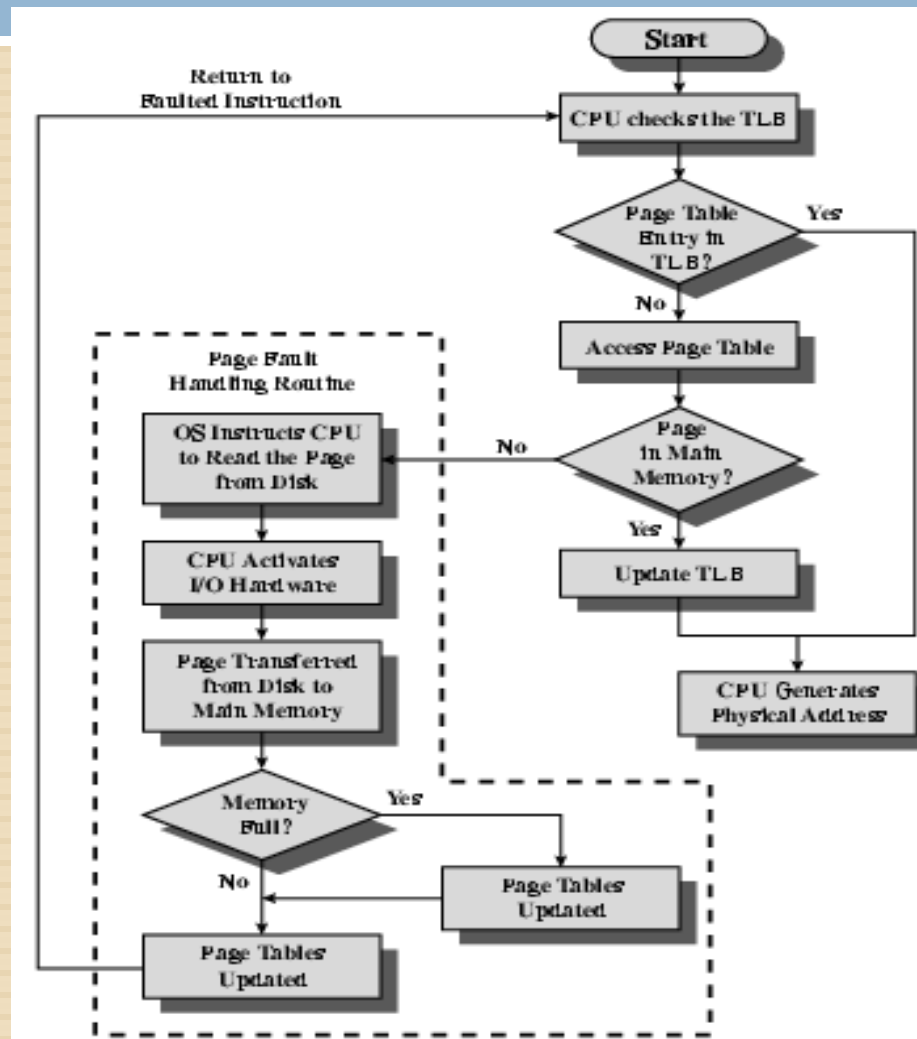
Inverted Page Table Structure



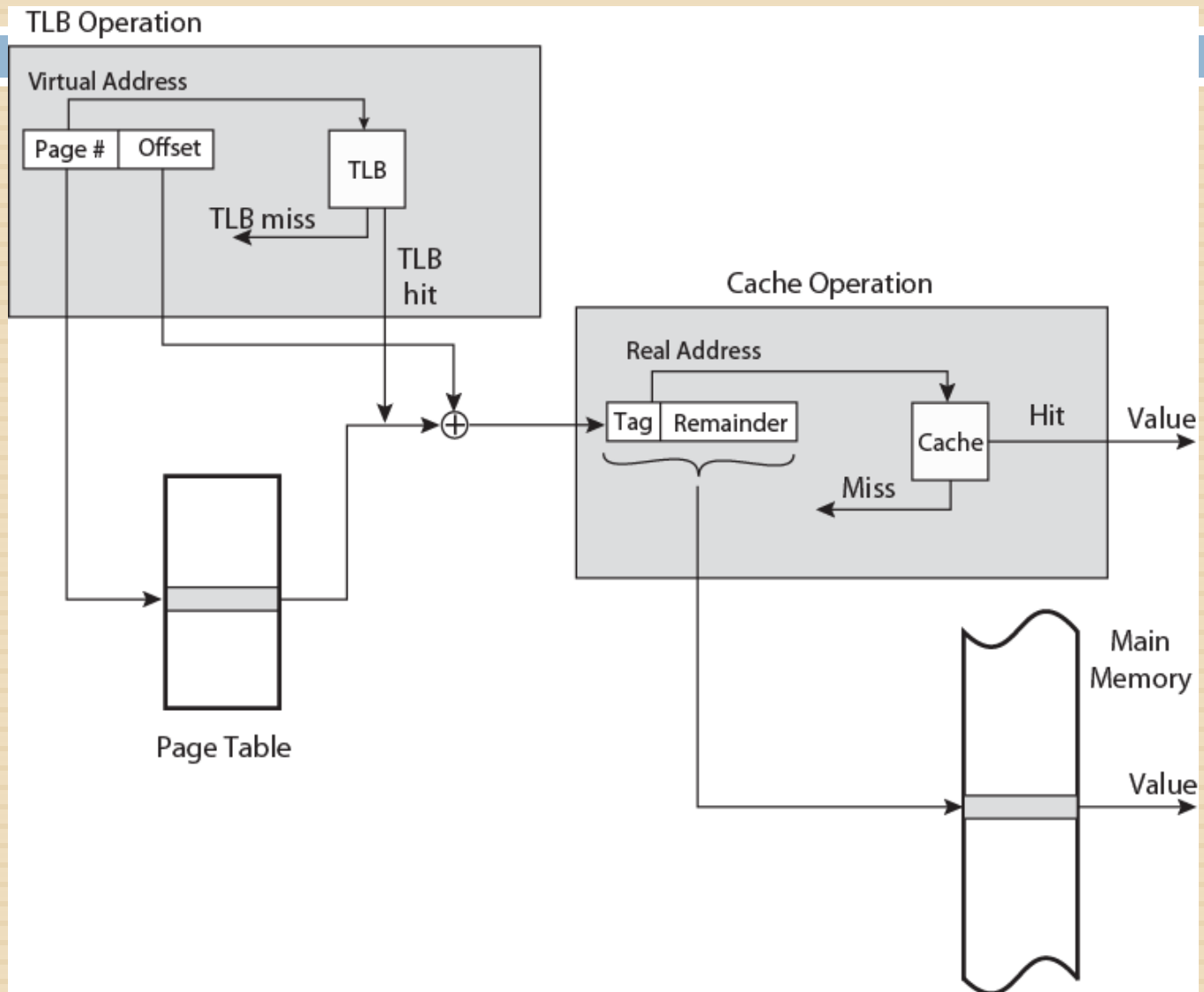
Translation Lookaside Buffer

- Every virtual memory reference causes two physical memory access
 - ▣ Fetch page table entry
 - ▣ Fetch data
- Use special cache for page table
 - ▣ TLB

TLB Operation



TLB and Cache Operation



Segmentation

- Paging is not (usually) visible to the programmer
- Segmentation is visible to the programmer
- Usually different segments allocated to program and data
- May be a number of program and data segments

Week 7

108

Computer Arithmetic

Chapter Organization

- Representing negative numbers
- Integer addition and subtraction
- Integer multiplication and division
- Floating point operations

A warning



- Binary addition, subtraction, multiplication and division are ***very easy***

ADDITION AND SUBTRACTION



General concept

□ Decimal addition

$$\begin{array}{r} \text{(carry)} \quad 1 \\ 19 \\ + 7 \\ \hline 26 \end{array}$$

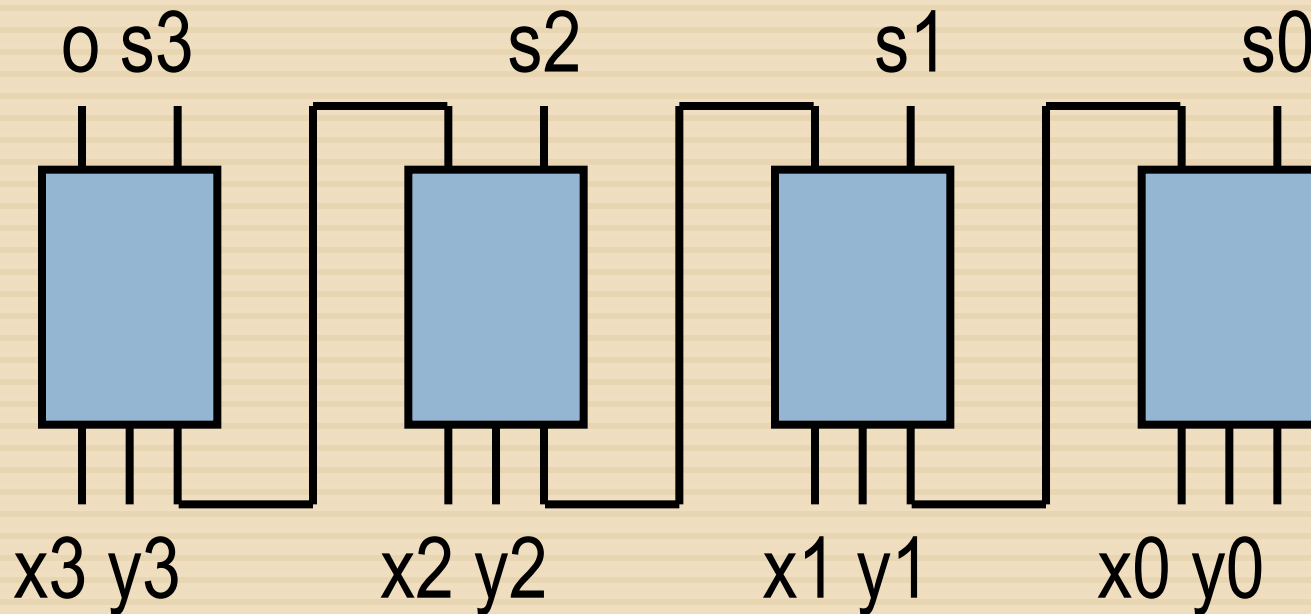
□ Binary addition

$$\begin{array}{r} \text{(carry)} \quad 111 \\ 10011 \\ + 111 \\ \hline 11010 \end{array}$$

$$\square \quad 16 + 8 + 2 = 26$$

Realization

- Simplest solution is a battery of full adders



Observations

- Adder add four-bit values
- Output **o** indicates if there is an **overflow**
 - A result that cannot be represented using 4 bits
 - Happens when **$x + y > 15$**

Signed and unsigned additions

□ Unsigned addition in 4-bit arithmetic

$$\begin{array}{r} \text{(carry) } \quad 11_ \\ \quad 1011 \\ + \quad 0011 \\ \hline \quad 1110 \end{array}$$

$$\begin{array}{l} \square \quad 11 + 3 = 14 \\ \quad (8 + 4 + 2) \end{array}$$

- Signed addition in 4-bit arithmetic

$$\begin{array}{r} \text{(carry) } \quad 11_ \\ \quad 1011 \\ + \quad 0011 \\ \hline \quad 1110 \end{array}$$

$$\bullet \quad -5 + 3 = -2$$

Signed and unsigned additions

- **Same rules** apply even though bit strings represent **different values**
- Sole difference is **overflow handling**

Overflow handling (I)

- No overflow in signed arithmetic

$$\begin{array}{r} \text{(carry)} \quad 111_ \\ 1110 \\ + 0011 \\ \hline 0001 \end{array}$$

- Signed addition in 4-bit arithmetic

$$\begin{array}{r} \text{(carry)} \quad 1_ \\ 0110 \\ + 0011 \\ \hline 1001 \end{array}$$

- $6 + 3 \neq -7$
(false)

- $-2 + 3 = 1$
(correct)

Overflow handling (II)

- In signed arithmetic an overflow happens when
 - ▣ The *sum of two positive numbers* exceeds the maximum positive value that can be represented using n bits: $2^n - 1$
 - ▣ The *sum of two negative numbers* falls below the minimum negative value that can be represented using n bits: -2^{n-1}

Example

- Four-bit arithmetic:
 - Sixteen possible values
 - Positive overflow happens when result > 7
 - Negative overflow happens when result < -8
- Eight-bit arithmetic:
 - 256 possible values
 - Positive overflow happens when result > 127
 - Negative overflow happens when result < -128

Overflow handling (III)

- MIPS architecture handles signed and unsigned overflows in a very different fashion:
 - ▣ **Ignores unsigned overflows**
 - ▣ Generates an **interrupt** whenever it detects a **signed overflows**
 - Lets the OS handled the condition

Why?

- To keep the CPU as simple and regular as possible

An interesting consequence

- Most C compilers ignore overflows
 - ▣ C compilers must use unsigned arithmetic for their integer operations
- Fortran compilers expect overflow conditions to be detected
 - ▣ Fortran compilers must use signed arithmetic for their integer operations

Subtraction

- Can be implementing by
 - ▣ Specific hardware
 - ▣ Negating the subtrahend

Negating a number



- Toggle all bits then add one

In 4-bit arithmetic (I)

0000	0	1111	+1 = 0000	0
0001	1	1110	+1 = 1111	-1
0010	2	1101	+1 = 1110	-2
0011	3	1100	+1 = 1101	-3
0100	4	1011	+1 = 1100	-4
0101	5	1010	+1 = 1011	-5
0110	6	1001	+1 = 1010	-6
0111	7	1000	+1 = 1001	-7

In 4-bit arithmetic (II)

1000	-8	0111	+1 = 1000	?
1001	-7	0110	+1 = 0111	7
1010	-6	0101	+1 = 0110	6
1011	-5	0100	+1 = 0101	5
1100	-4	0011	+1 = 0100	4
1101	-3	0010	+1 = 0011	3
1110	-2	0001	+1 = 0010	2
1111	-1	0000	+1 = 0001	1

MULTIPLICATION



Decimal multiplication

$$\begin{array}{r} \text{(carry) } 1 _ \\ 37 \\ \times 12 \\ \hline 74 \\ 370 \\ \hline 444 \end{array}$$

- What are the rules?
 - Successively multiply the multiplicand by each digit of the multiplier starting at the right shifting the result left by an extra left position each time each time but the first
 - Sum all partial results

Binary multiplication

$$\begin{array}{r} \text{(carry)} \\ 111 \\ 1101 \\ \times 101 \\ \hline 1101 \\ 00 \\ \hline 110100 \\ \hline 1000001 \end{array}$$

- What are the rules?
 - Successively multiply the multiplicand by each digit of the multiplier starting at the right shifting the result left by an extra left position each time each time but the first
 - Sum all partial results
- **Binary multiplication is easy!**

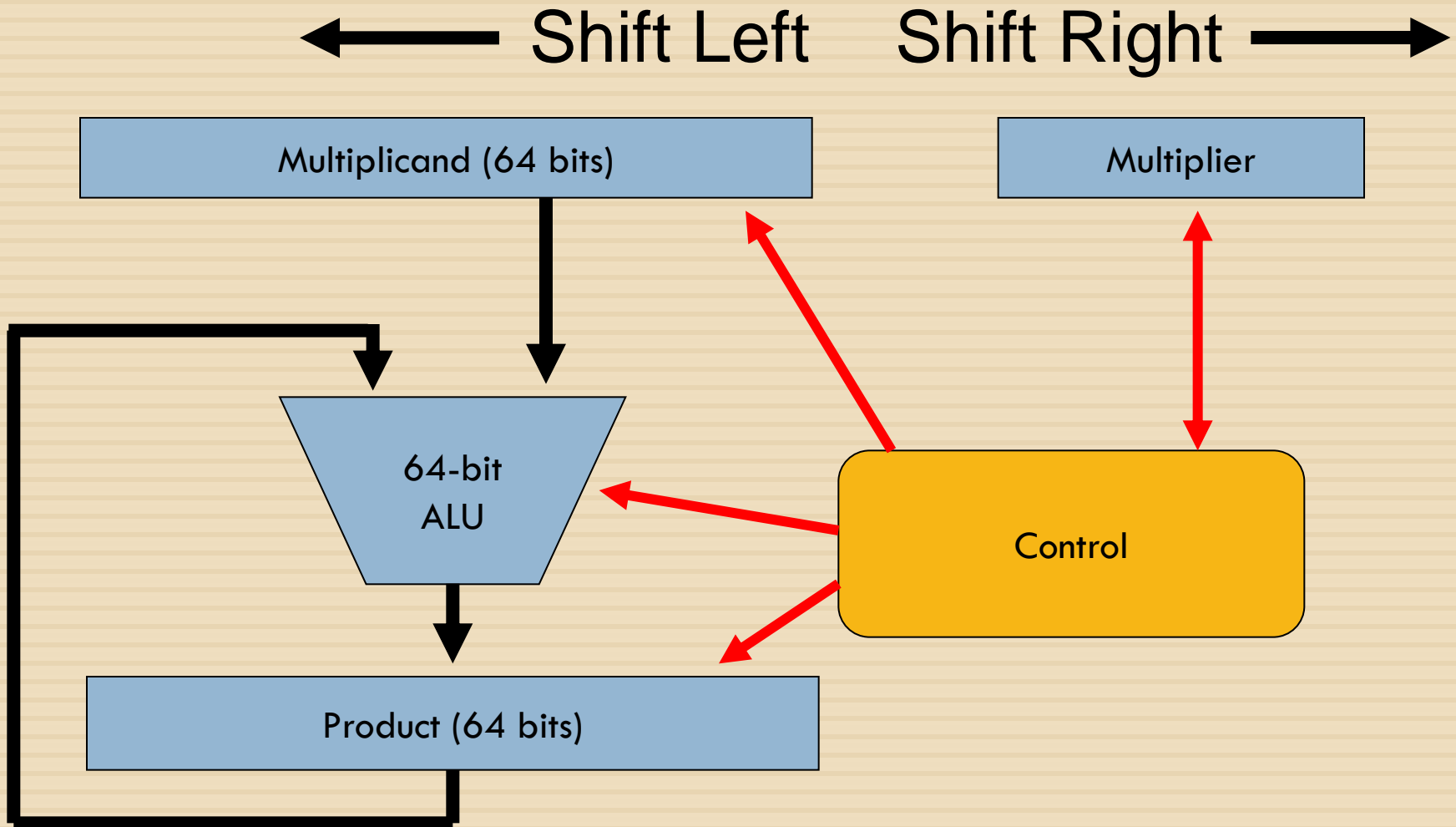
Binary multiplication table

X	0	1
0	0	0
1	0	1

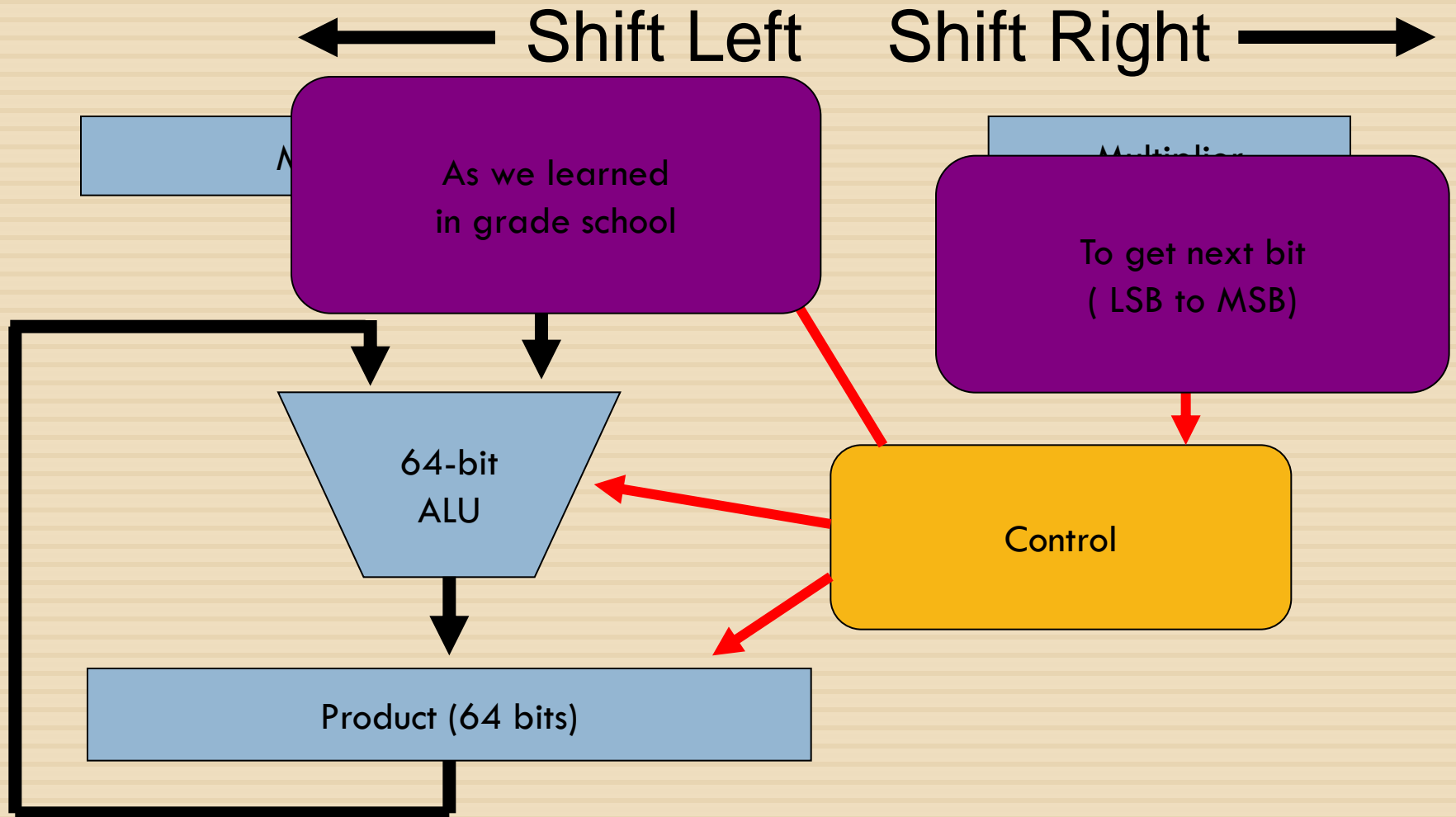
Algorithm

- Clear contents of 64-bit product register
- For $(i = 0; i < 32; i++)$ {
 - If (LSB of multiplier_register == 1)
 - Add contents of multiplicand register to product register
 - Shift **right** one position multiplier register
 - Shift **left** one position multiplicand register
- } // for loop

Multiplier: First version



Multiplier: First version



Explanations

- Multiplicand register must be 64-bit wide because 32-bit multiplicand will be shifted 32 times to the left
 - ▣ Requires a 64-bit ALU
- Product register must be 64-bit wide to accommodate the result
- Contents of multiplier register is shifted 32 times to the right so that each bit successively becomes its least significant bit (LSB)

Example (I)

□ Multiply **0011** by **0011**

□ **Start**

Multiplicand	Multiplier	Product
0011	0011	0000

□ **First addition**

Multiplicand	Multiplier	Product
0011	001<u>1</u>	0011

Example (II)

□ Shift right and left

Multiplicand	Multiplier	Product
0110	0001	0011

□ Second addition

Multiplicand	Multiplier	Product
0110	000<u>1</u>	1001

□ **$0110 + 011 = 1001$**

Example (III)

- **Shift right and left**

Multiplicand	Multiplier	Product
1100	0000	1001

- **Multiplier is all zeroes:** we are done

First Optimization

- Must have a 64-bit ALU
 - ▣ More complex than a 32-bit ALU
- Solution is not to shift the multiplicand
 - ▣ After each cycle, the LSB being added remains unchanged
 - ▣ Will save that bit elsewhere and shift the product register one position to the left after each iteration

Binary multiplication

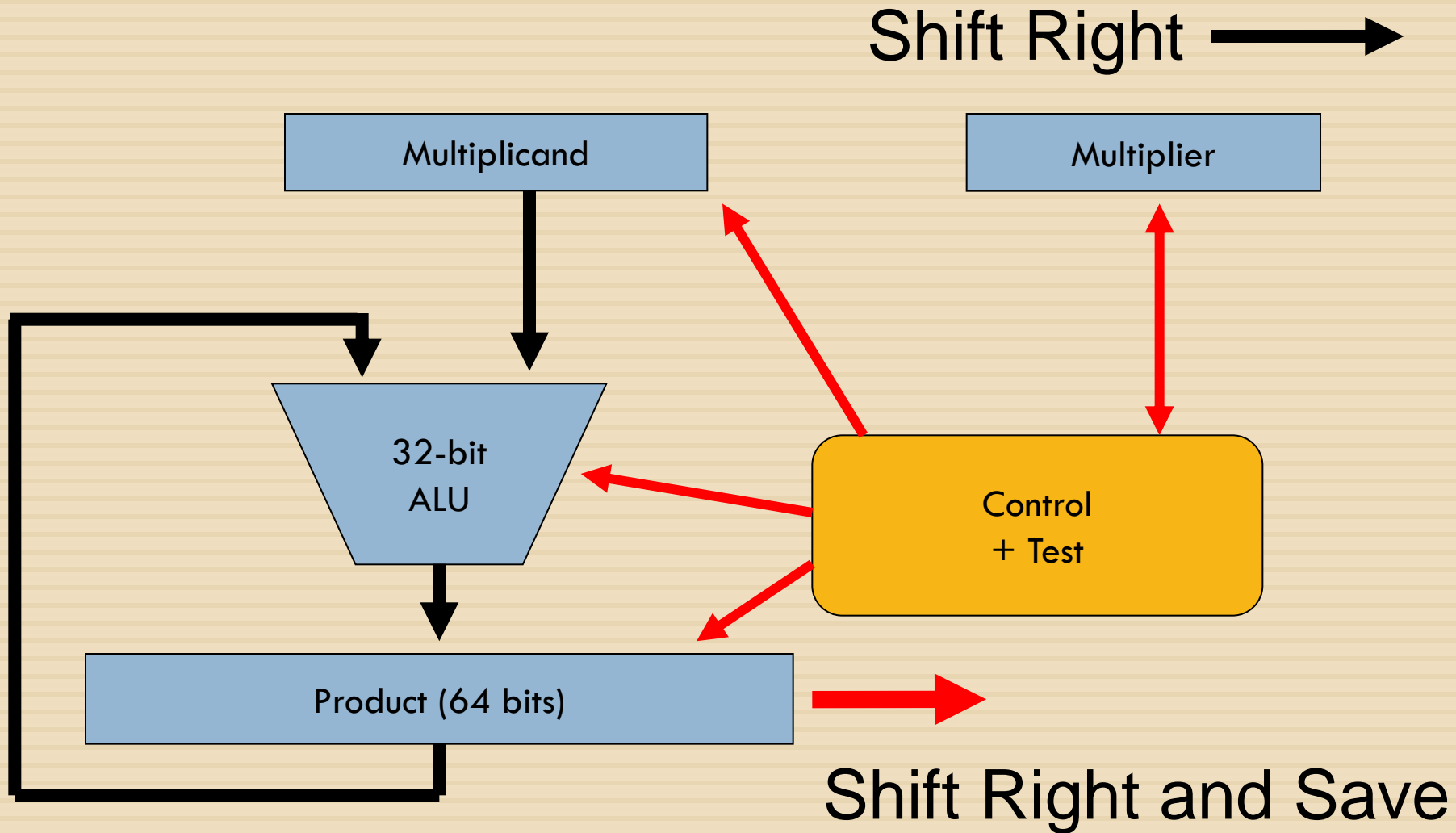
$$\begin{array}{r} 1101 \\ \times 101 \\ \hline 1101 \\ 00 \\ 1100 \\ \hline 1000101 \end{array}$$

- Observe that the least significant bit added during each cycle remains unchanged

Algorithm

- Clear contents of 64-bit product register
- For (i = 0; i < 32; i++) {
 - If (LSB of multiplier_register == 1)
 - Add contents of multiplicand register to product register
 - Save LSB of product register
 - Shift **right** one position **both** multiplier register and product register
- } // for loop

Multiplier: Second version



Decimal Example (I)

□ Multiply **27** by **12**

□ **Start**

Multiplicand	Multiplier	Product	Result
27	12	--	--

□ **First digit**

Multiplicand	Multiplier	Product	Result
27	1<u>2</u>	54	--

Decimal Example (II)

□ Shift right multiplier and product

Multiplicand	Multiplier	Product	Result
27	1	5	4

□ Second digit

Multiplicand	Multiplier	Product	Result
27	1	32	4

Decimal Example (III)

□ Shift right multiplier and product

Multiplicand	Multiplier	Product	Result
27	0	3	24

□ Multiplier equals zero

Result is obtained by concatenating contents of product and result registers

□ **324**

How did it work?

- We learned

- ▣ $27 \times 12 = 27 \times 10 + 27 \times 2$
 $= 27 \times 10 + 54$
 $= 270 + 54$

- Algorithm uses another decomposition

- ▣ $27 \times 12 = 27 \times 10 + 27 \times 2$
 $= 27 \times 10 + 50 + 4$
 $= (27 \times 10 + 50) + 4$
 $= 320 + 4$

Example (I)

□ Multiply **0011** by **0011**

□ **Start**

Multiplicand	Multiplier	Product	Result
0011	0011	--	--

□ **First bit**

Multiplicand	Multiplier	Product	Result
0011	001<u>1</u>	0011	--

Example (II)

□ Shift right multiplier and product

Multiplicand	Multiplier	Product	Result
0011	0001	0001	1-

□ Second bit

Multiplicand	Multiplier	Product	Result
0011	0001	0100	1-

Product register contains **0011 + 001 = 0100**

Example (III)

□ Shift right multiplier and product

Multiplicand	Multiplier	Product	Result
0011	0000	010	01-

□ Multiplier equals zero

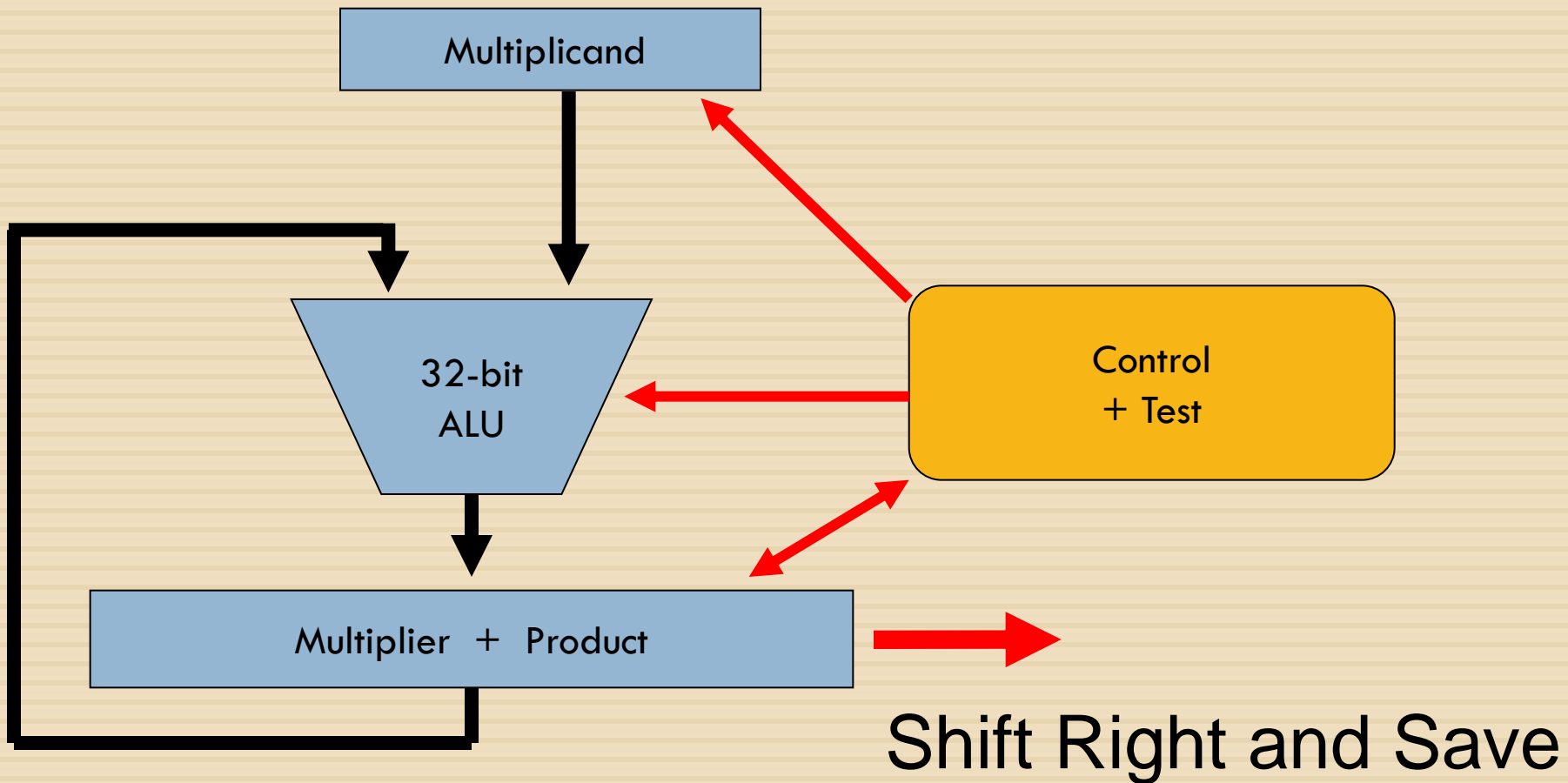
Result is obtained by concatenating contents of product and result registers

▣ $1001 = 9$

Second Optimization

- Both multiplier and product must be shifted to one position to the right after each iteration
- Both are now 32-bit quantities
- Can store both quantities in the product register

Multiplier: Third version



Third Optimization

- Multiplication requires 32 additions and 32 shift operations
- Can have two or more ***partial multiplications***
 - ▣ One using ***bits 0-15*** of multiplier
 - ▣ A second using ***bits 16-31******then add together the partial results***

Multiplying negative numbers

- Can use the same algorithm as before but we must ***extend the sign*** bit of the product

Related MIPS instructions (I)

- Integer multiplication uses a ***separate pair*** of registers (***hi*** and ***lo***)
- **mult \$s0, \$s1**
 - ▣ multiply contents of register \$s0 by contents of register \$s1 and store results in register pair hi-lo
- **multu \$s0, \$s1**
 - ▣ same but **unsigned**

Related MIPS instructions (II)

- **mflo \$s9**
 - ▣ Move contents of register lo to register \$s0
- **mfhi \$s9**
 - ▣ Move contents of register hi to register \$s0

DIVISION



Division

- Implemented by successive subtractions
- Result must verify the equality

$$\text{Dividend} = \text{Multiplier} \times \text{Quotient} + \text{Remainder}$$

Decimal division (long division)

$$\begin{array}{r} 303 \\ 7 \overline{) 2126} \\ \underline{-210} \\ 26 \\ \underline{-21} \\ 5 \end{array}$$

□ What are the rules?

- Repeatedly try to subtract smaller multiple of divisor from dividend
- Record multiple (or zero)
- At each step, repeat with a lower power of ten
- Stop when remainder is smaller than divisor

Binary division

$$\begin{array}{r}
 011 \\
 11 \overline{) 1011} \\
 \underline{-11} \quad \text{X} \\
 1011 \\
 \underline{>-11} \\
 101 \\
 \underline{>>-11} \\
 10
 \end{array}$$

X

□ What are the rules?

- Repeatedly try to subtract powers of two of divisor from dividend
- Mark 1 for success, 0 for failure
- At each step, shift divisor one position to the right
- Stop when remainder is smaller than divisor

Same division in decimal

$$\begin{array}{r} 2+1=3 \\ 3 \overline{) 11} \\ \underline{-12} \\ 11 \\ \underline{>-6} \\ 5 \\ \underline{>-3} \\ 2 \end{array}$$

□ What are the rules?

- Repeatedly try to subtract powers of two of divisor from dividend
- Mark 1 for success, 0 for failure
- At each step, shift divisor one position to the right
- Stop when remainder is smaller than divisor

X

Observations

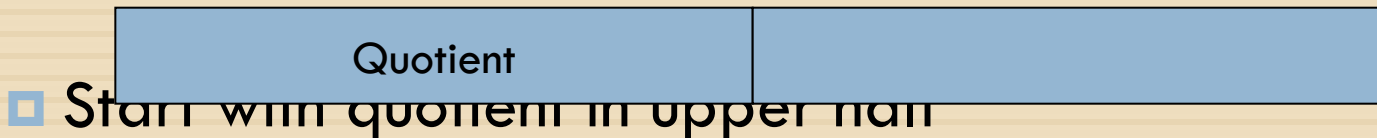
- Binary division is actually simpler
 - ▣ We start with a left-shifted version of divisor
 - ▣ We try to subtract it from dividend
 - *No need to find out which multiple to subtract*
 - ▣ We mark 1 for success, 0 for failure
 - ▣ We shift divisor one position left after every attempt

How to start the division

- One 64-bit register for successive remainders

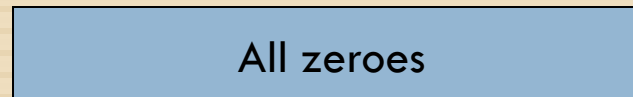


- One 64-bit register for divisor



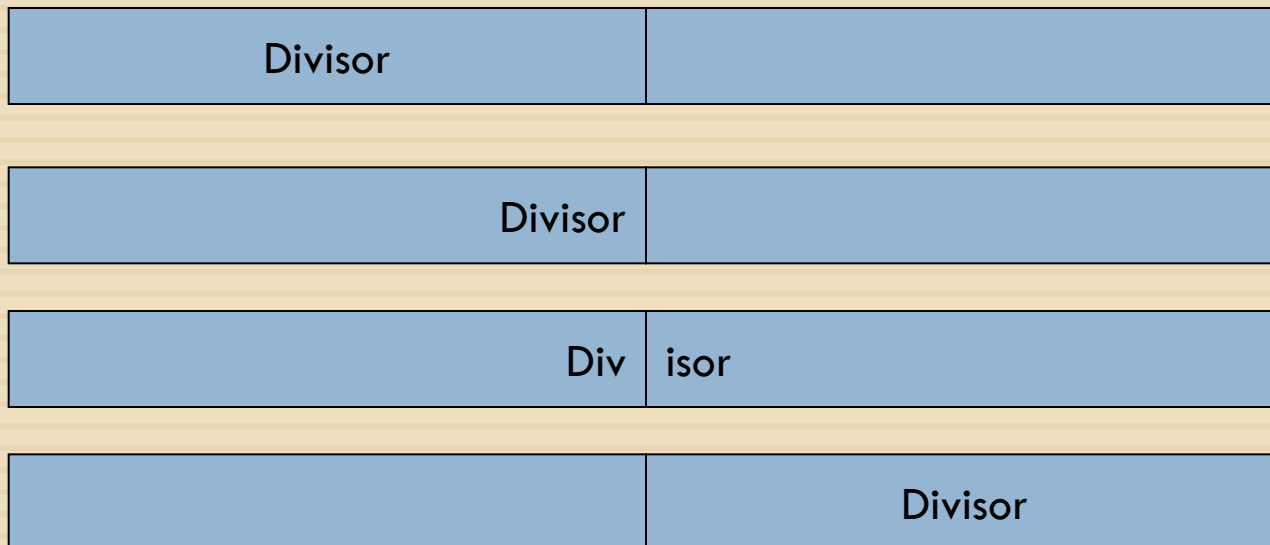
- Start with quotient in upper half

- One 32-bit register for the quotient



How we proceed (I)-

- After each step we shift the quotient *to the right* one position at a time



How we proceed (II)

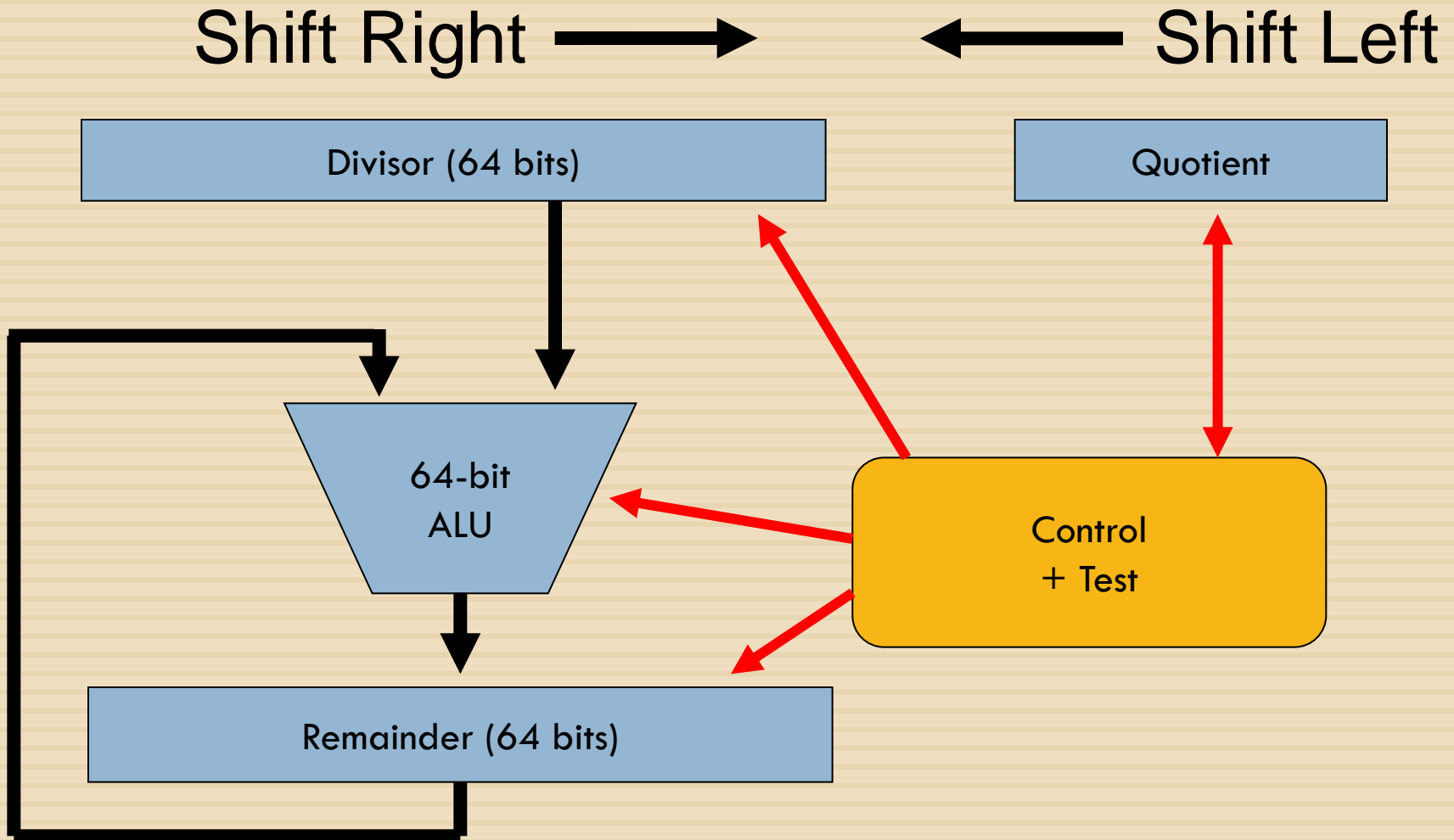
- After each step we shift the contents of the quotient register one position to the left
 - ▣ To make space for the new 0 or 1 being inserted

0
01
011
0110

Division Algorithm

- **For** i in range(0,33) : # from 0 to 32
 - Subtract contents of ***divisor register*** from ***remainder register***
 - **If** remainder ≥ 0 :
 - Shift ***quotient register to the left***
 - Set new rightmost bit to 1
 - Else :**
 - ***Undo*** subtraction
 - Shift ***quotient register to the left***
 - Set new rightmost bit to 0
 - Shift ***right*** one position contents of divisor register

A simple divider



Signed division

- Easiest solution is to remember the sign of the operands and adjust the sign of the quotient and remainder accordingly
 - A little problem:
 - $5 \div 2 = 2$ and the remainder is 1
 - $-5 \div 2 = -2$ and the remainder is -1
- The sign of the remainder must match the sign of the quotient*

Related MIPS instructions

- Integer division uses the *same pair* of registers (*hi* and *lo*) as integer multiplication
- **div \$s0, \$s1**
 - ▣ divide contents of register \$s0 by contents of register \$s, leave the quotient in register lo and the remainder in register hi
- **divu \$s0, \$s1**
 - ▣ same but **unsigned**

TRANSITION SLIDE

- Here end the materials that were on the first fall 2012 midterm
- Here start the materials that will be on the fall 2012 midterm

**To be moved to
the right place**

FLOATING POINT OPERATIONS

Floating point numbers

- Used to represent *real numbers*

- Very similar to *scientific notation*

3.5×10^6 , 0.82×10^{-5} , 75×10^6 , ...

- Both decimal numbers in scientific notation and floating point numbers can be normalized:

3.5×10^6 , 8.2×10^{-6} , 7.5×10^7 , ...

Fractional binary numbers

- 0.1 is $\frac{1}{2}$ or 0.5_{ten}
- 0.01 is 0.1 is $\frac{1}{4}$ or 0.25_{ten}
- 0.11 is $\frac{1}{2} + \frac{1}{4} = \frac{3}{4}$ or 0.75_{ten}
- 1.1 is $1\frac{1}{2}$ or 1.5_{ten}
- 10.01 is $2 + \frac{1}{4}$ or 2.5_{ten}
- 11.11 is _____ or _____

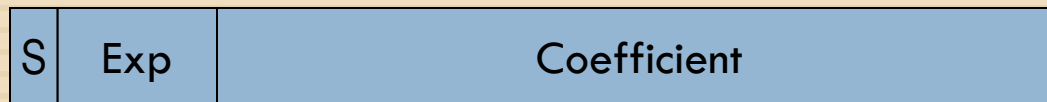
Normalizing binary numbers

- **0.1** becomes **1.0×2^{-1}**
- **0.01** becomes **1.0×2^{-2}**
- **0.11** becomes **1.1×2^{-1}**

- **1.1** is already normalized and equal to **1.0×2^0**
- **10.01** becomes **1.001×2^1**
- **11.11** becomes **1×2^1**

Representation

- Sign + exponent + coefficient



- IEEE Standard 754
 - ▣ $1 + 8 + 23 = 32$ bits
 - ▣ $1 + 11 + 52 = 64$ bits (double precision)

Week 8

174

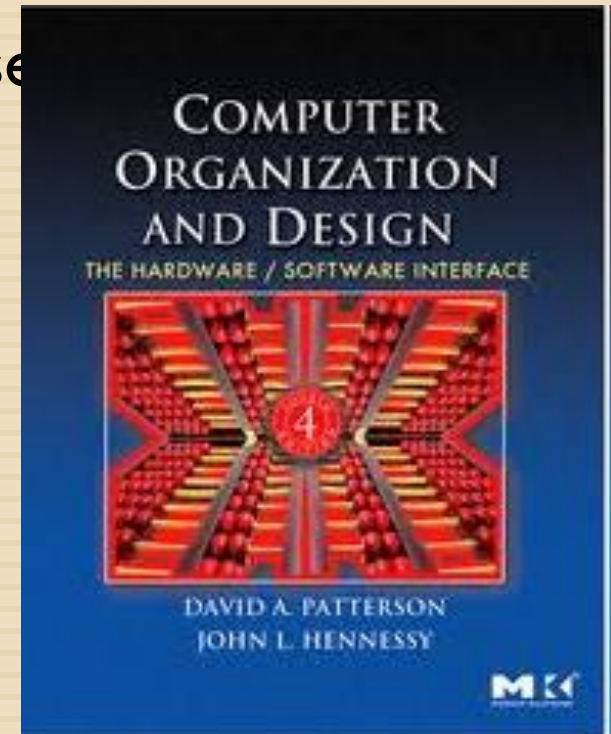
MIPS

Instruction-Set Architecture

Topics

175

- Instructions & MIPS instruction set
- Where are the operands ?
- Machine language
- Assembler
- Translating C statements into Asse
- For details see the book (ch 2):



Main Types of Instructions

176

- Arithmetic
 - Integer
 - Floating Point
- Memory access instructions
 - Load & Store
- Control flow
 - Jump
 - Conditional Branch
 - Call & Return

MIPS arithmetic

177

- Most instructions have 3 operands
- Operand order is fixed (destination first)

Example:

C code: $A = B + C$

MIPS code: `add $s0, $s1, $s2`

(\$s0, \$s1 and \$s2 are associated with variables by compiler)

MIPS arithmetic

178

C code: $A = B + C + D;$
 $E = F - A;$

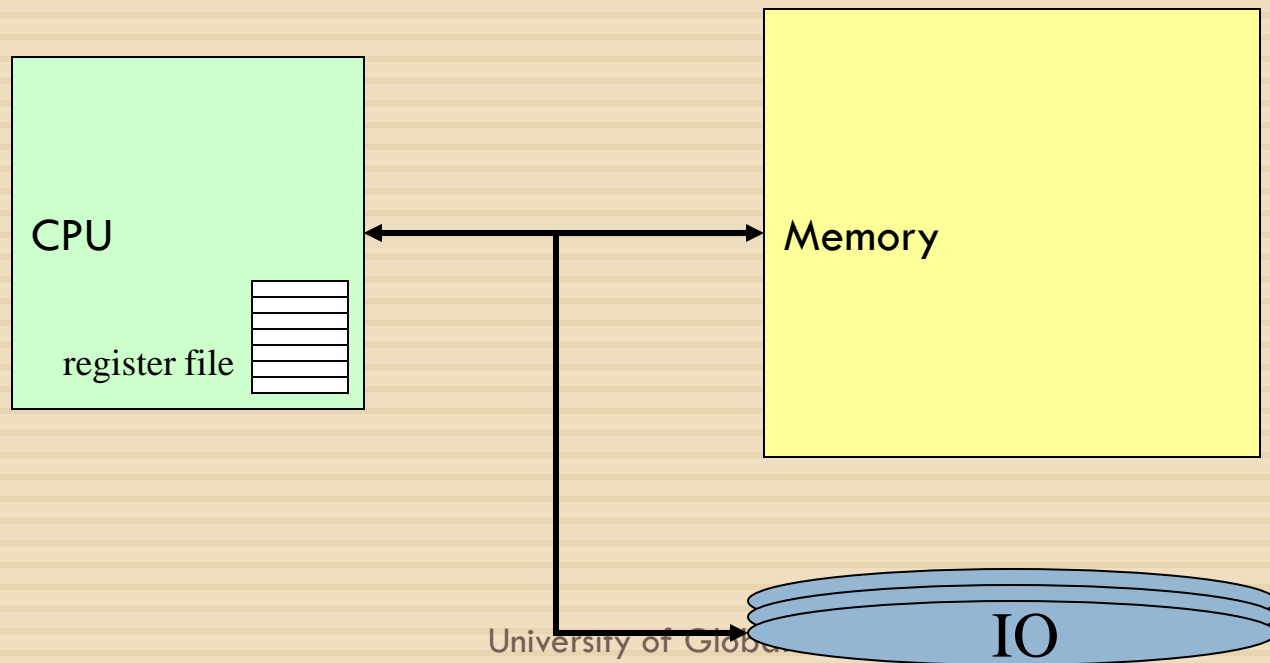
MIPS code: `add $t0, $s1, $s2`
 `add $s0, $t0, $s3`
 `sub $s4, $s5, $s0`

- Operands must be registers, only 32 registers provided
- Design Principle: *smaller is faster*. Why?

Registers vs. Memory

179

- Arithmetic instruction operands must be registers,
— only 32 registers provided
- Compiler associates variables with registers
- What about programs with lots of variables ?



Register allocation

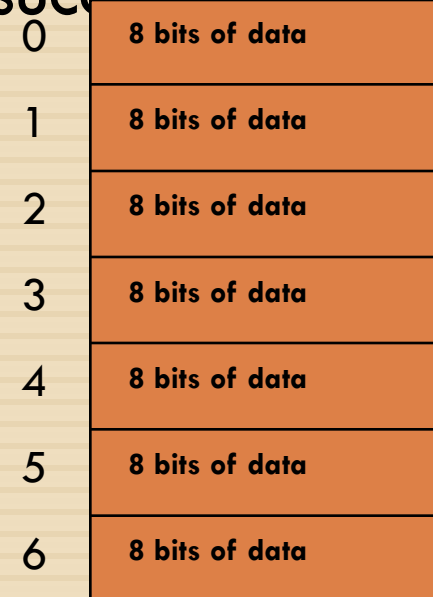
180

- Compiler tries to keep as many variables in registers as possible
- Some variables can not be allocated
 - ▣ large arrays (too few registers)
 - ▣ aliased variables (variables accessible through pointers in C)
 - ▣ dynamic allocated variables
 - heap
 - stack
- Compiler may run out of registers => **spilling**

Memory Organization

181

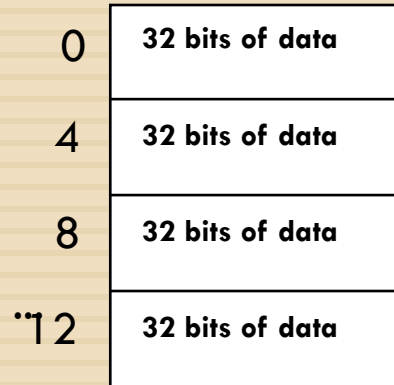
- Viewed as a large, single-dimension array, with an address
- A memory address is an index into the array
- "Byte addressing" means that successive addresses are one byte apart



Memory Organization

182

- Bytes are nice, but most data items use larger "words"
- For MIPS, a word is 32 bits or 4 bytes.

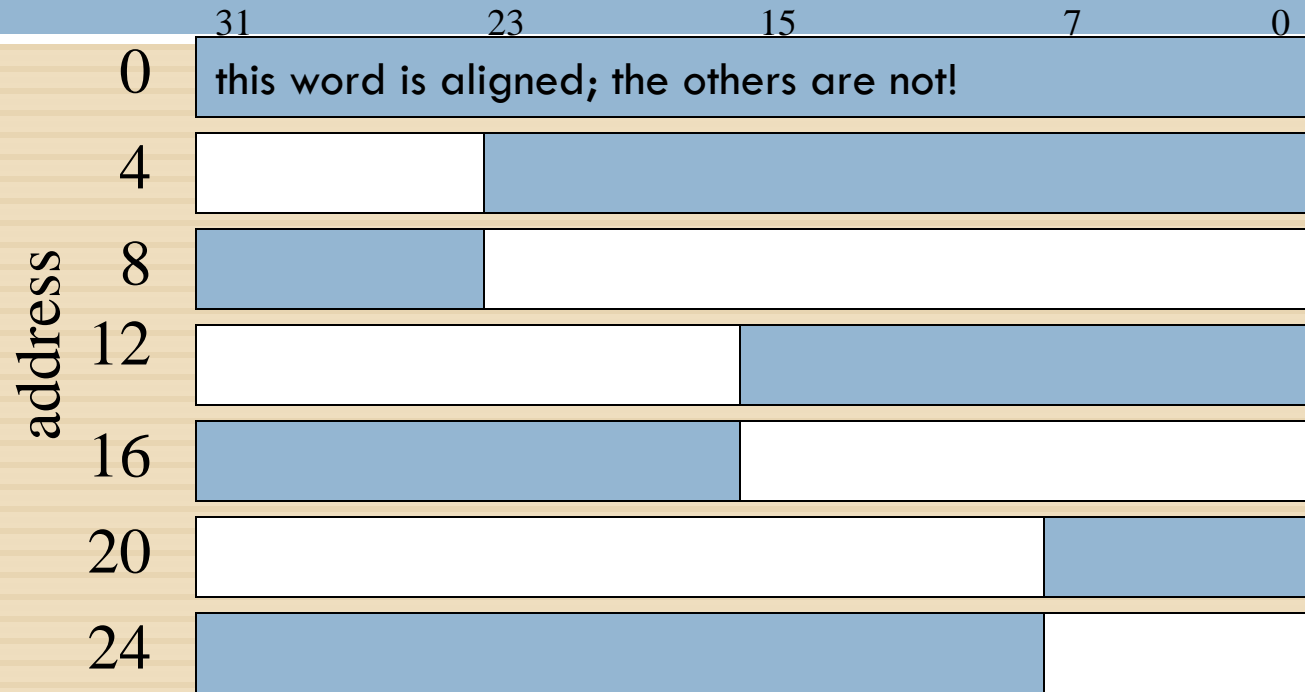


Registers hold 32 bits of data

- 2^{32} bytes with byte addresses from 0 to $2^{32}-1$
- 2^{30} words with byte addresses 0, 4, 8, ... $2^{32}-4$

Memory layout: Alignment

183



Words are aligned

- What are the least 2 significant bits of a word address?

Week 9

184

MIPS

Instruction-Set Architecture-2

Instructions: load and store

185

Example:

C code: `A[8] = h + A[8];`

MIPS code: `lw $t0, 32($s3)`
 `add $t0, $s2, $t0`
 `sw $t0, 32($s3)`

- Store word operation has no destination (reg) operand
- Remember arithmetic operands are registers, not memory!

Our First C code translated

186

- Can we figure out the code?

```
swap(int v[], int k);  
{ int temp;  
  temp = v[k]  
  v[k] = v[k+1];  
  v[k+1] = temp;  
}
```



```
swap:  
    muli    $2, $5, 4  
    add     $2, $4, $2  
    lw      $15, 0($2)  
    lw      $16, 4($2)  
    sw      $16, 0($2)  
    sw      $15, 4($2)  
    jr      $31
```

Explanation:

index k : \$5

base address of v: \$4

address of v[k] is $\$4 + 4 \cdot \5

So far we've learned:

187

□ MIPS

- loading words but addressing bytes
- arithmetic on registers only

□ Instruction

Meaning

add \$s1, \$s2, \$s3

$\$s1 = \$s2 + \$s3$

sub \$s1, \$s2, \$s3

$\$s1 = \$s2 - \$s3$

lw \$s1, 100(\$s2)

$\$s1 = \text{Memory}[\$s2 + 100]$

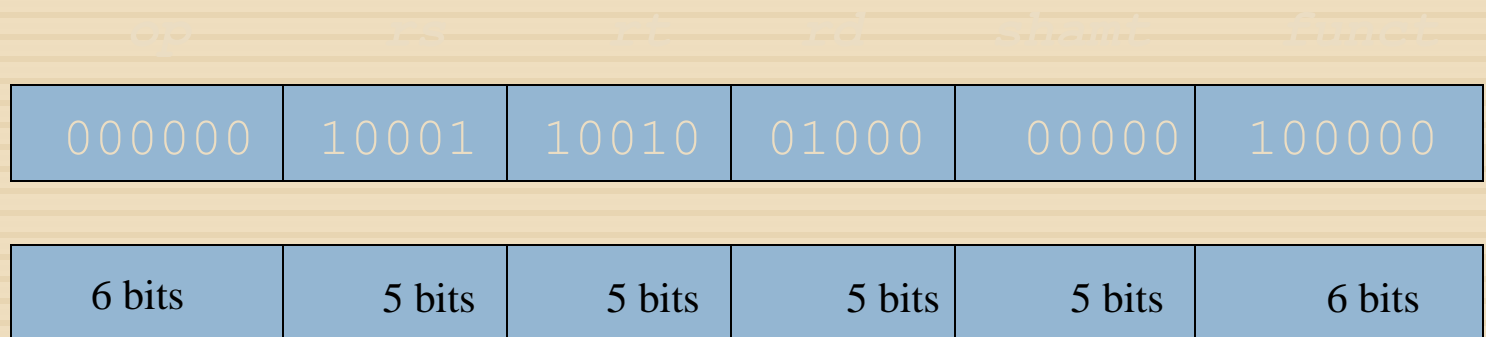
sw \$s1, 100(\$s2)

$\text{Memory}[\$s2 + 100] = \$s1$

Machine Language: R-type instr

188

- Instructions, like registers and words of data, are also 32 bits long
 - Example: `add $t0, $s1, $s2`
 - Registers have numbers: `$t0=9, $s1=17, $s2=18`
- Instruction Format:

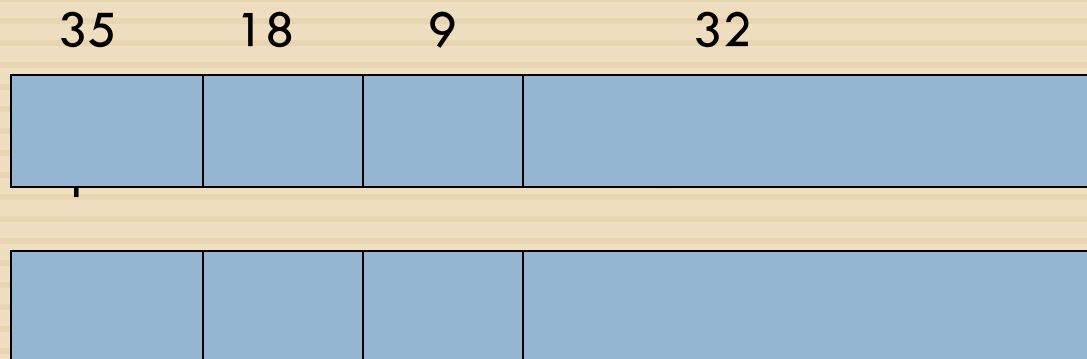


Can you guess what the field names stand for?

Machine Language: I-type instr

189

- Consider the load-word and store-word instructions,
 - What would the **regularity principle** have us do?
 - New principle: *Good design demands a compromise*
- Introduce a new type of instruction format
 - I-type for data transfer instructions
 - other format was R-type for register
- **Example:** `lw $t0, 32($s2)`



Control

190

- Decision making instructions
 - ▣ alter the control flow,
 - ▣ i.e., change the "next" instruction to be executed

- MIPS conditional branch instructions:

```
bne $t0, $t1, Label  
beq $t0, $t1, Label
```

- Example: if (i==j) h = i + j;

```
        bne $s0, $s1, Label  
        add $s3, $s0, $s1  
Label:      . . . .
```

Control

191

- MIPS unconditional branch instructions:

```
j    label
```

- Example:

```
if (i!=j)
    h=i+j;
else
    h=i-j;
```

```
    beq $s4, $s5, Lab1
    add $s3, $s4, $s5
    j   Lab2
Lab1: sub $s3, $s4, $s5
Lab2: ...
```

- *Can you build a simple for loop?*

So far (including J-type instr):

192

□ Instruction Meaning

add \$s1,\$s2,\$s3	\$s1 = \$s2 + \$s3
sub \$s1,\$s2,\$s3	\$s1 = \$s2 - \$s3
lw \$s1,100(\$s2)	\$s1 = Memory[\$s2+100]
sw \$s1,100(\$s2)	Memory[\$s2+100] = \$s1
bne \$s4,\$s5,L	Next instr. is at Label if \$s4 \neq \$s5
beq \$s4,\$s5,L	Next instr. is at Label if \$s4 = \$s5
j Label	Next instr. is at Label

□ **Formats:**

R	op	rs	rt	rd	shamt	funct
I	op	rs	rt	16 bit address		
J	op	26 bit address				

Control Flow

193

- We have: beq, bne, what about Branch-if-less-than?
- New instruction:

```
slt $t0, $s1, $s2
```



- Can use this instruction to build "blt \$s1, \$s2, Label"
— can now build general control structures
- Note that the assembler needs a register to do this,
— use conventions for registers

Used MIPS compiler conventions

194

Name	Register number	Usage
\$zero	0	the constant value 0
\$v0-\$v1	2-3	values for results and expression evaluation
\$a0-\$a3	4-7	arguments
\$t0-\$t7	8-15	temporaries
\$s0-\$s7	16-23	saved (by callee)
\$t8-\$t9	24-25	more temporaries
\$gp	28	global pointer
\$sp	29	stack pointer
\$fp	30	frame pointer
\$ra	31	return address

Small Constants: immediates

195

- Small constants are used quite frequently (50% of operands) e.g., $A = A + 5;$

$B = B + 1;$

$C = C - 18;$

- MIPS Instructions:

```
addi $29, $29, 4
```

```
slti $8, $18, 10
```

```
andi $29, $29, 6
```

```
ori $29, $29, 4
```

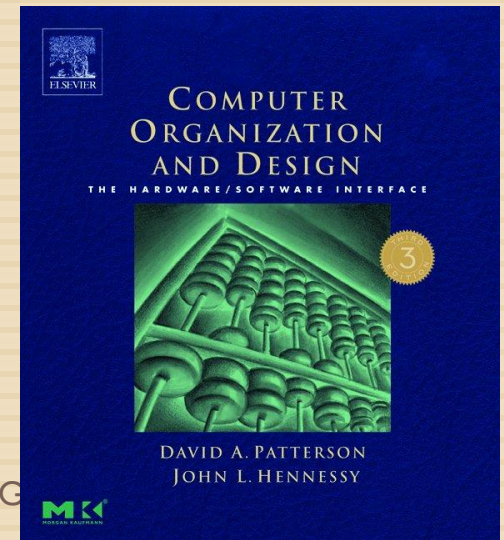
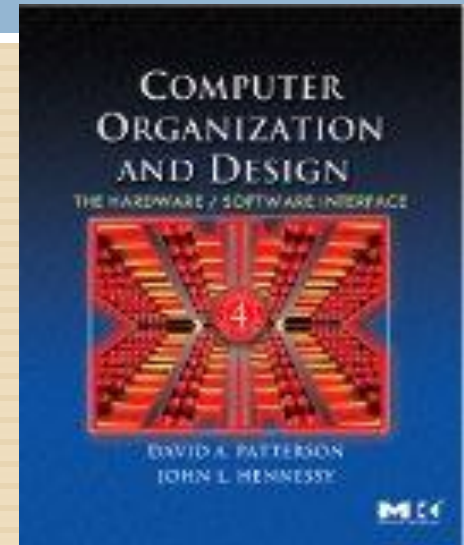
Week 10

MIPS Pipelining

Topics

197

- Pipelining
- Pipelined datapath
- Pipelined control
- Hazards:
 - Structural
 - Data
 - Control
 - Exceptions
- Scheduling
- For details see the book (3rd ed chapter 6 / 4th ed ch 4):

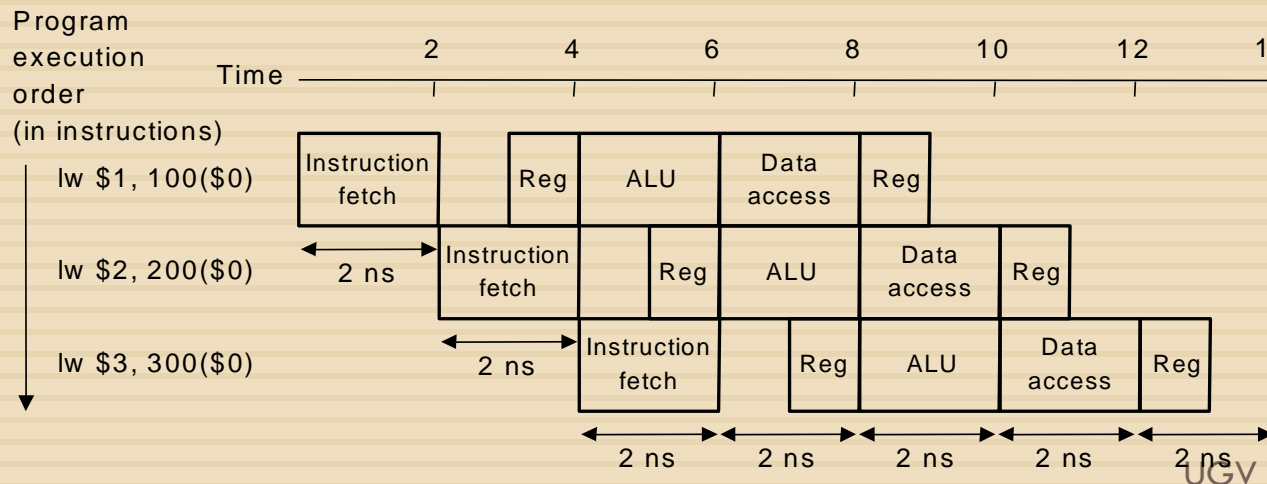
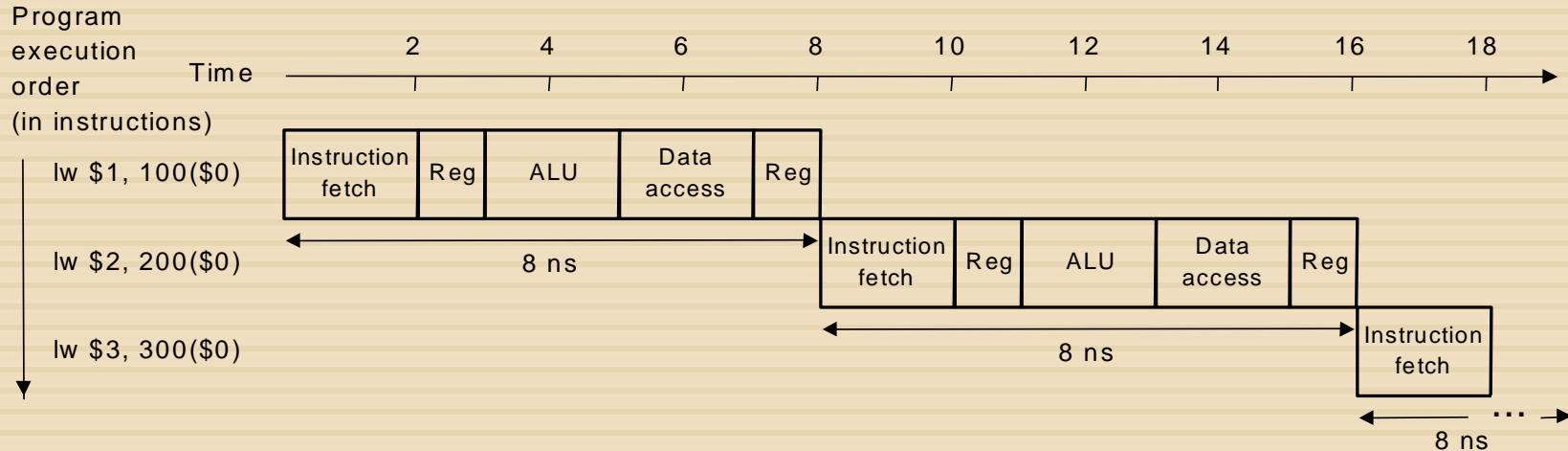


UG

Pipelining principle

Improve performance by increasing instruction throughput

198



Pipelining speedup?

199

- Ideal speedup = number of stages
- Do we achieve this?

Pipelining

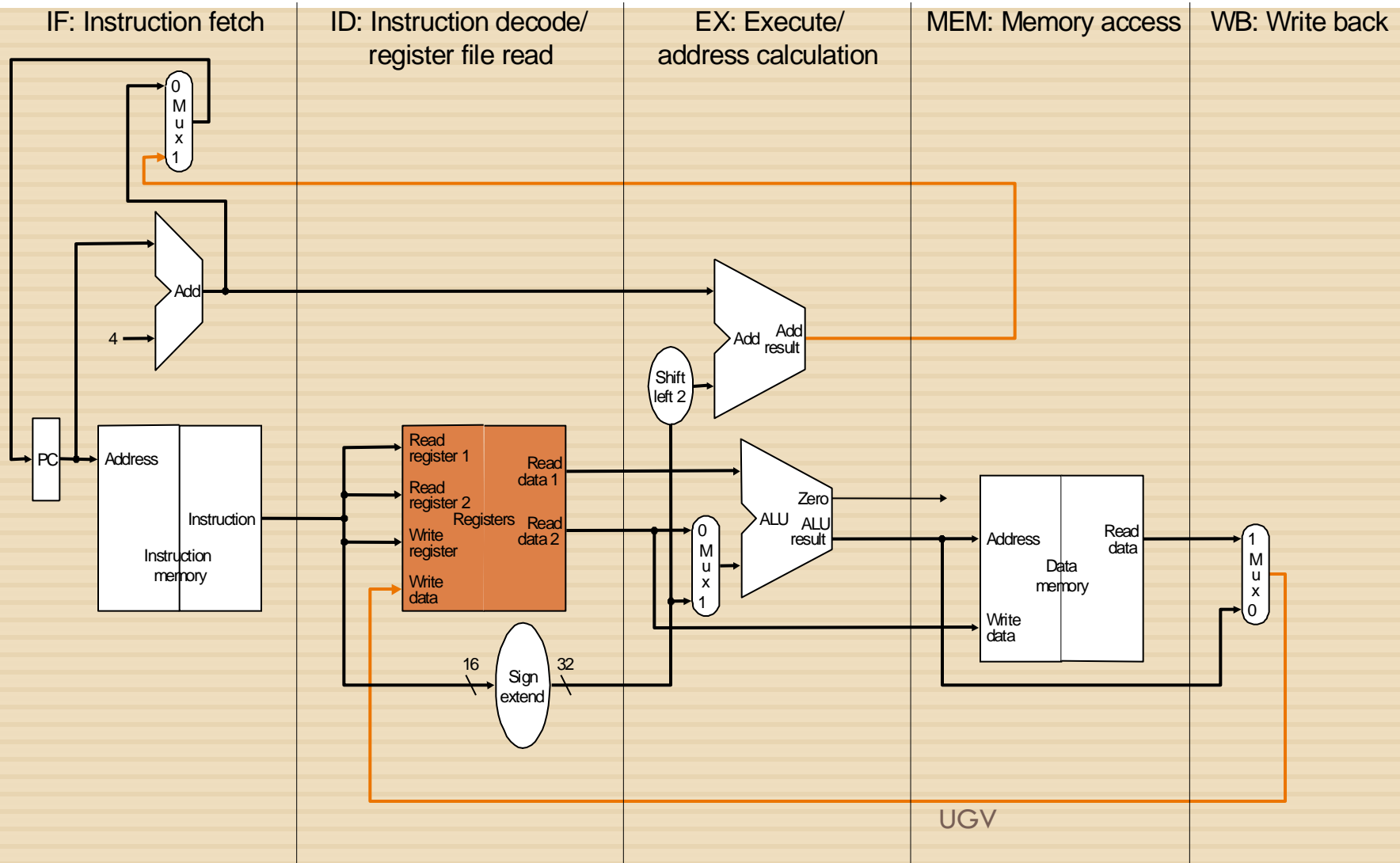
200

- What makes it easy
 - ▣ all instructions are the same length
 - ▣ just a few instruction formats
 - ▣ memory operands appear only in loads and stores
- What makes it hard?
 - ▣ structural hazards: suppose we had only one memory
 - ▣ control hazards: need to worry about branch instructions
 - ▣ data hazards: an instruction depends on a previous instruction
 - ▣ exceptions
- We'll build a simple pipeline and look at these issues
- We'll talk about modern processors and what really makes it hard:
 - ▣ exception handling
 - ▣ trying to improve performance with out-of-order execution, etc.

Basic idea: start from single cycle impl.

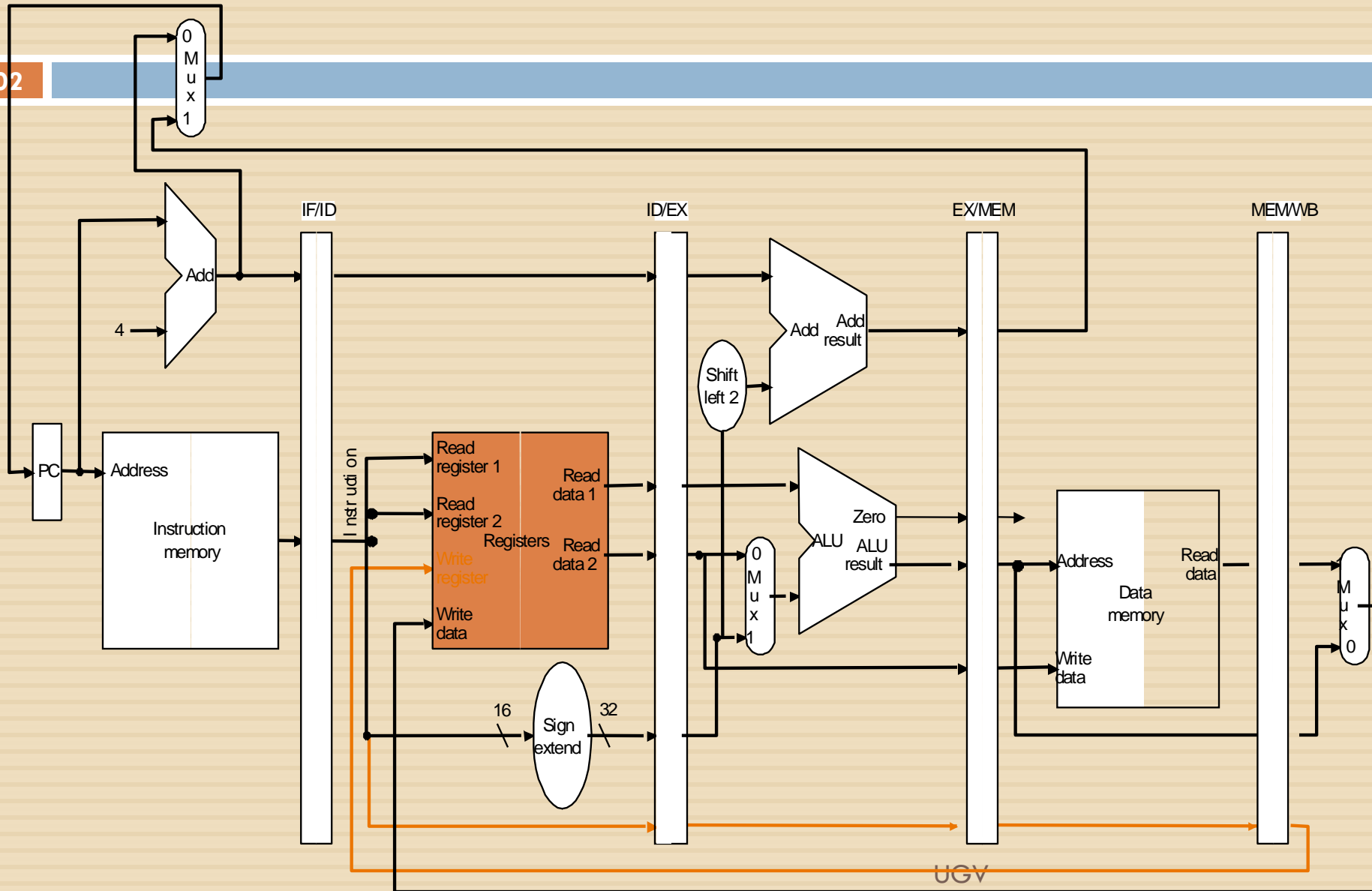
What do we need to add to actually split the datapath into stages?

201



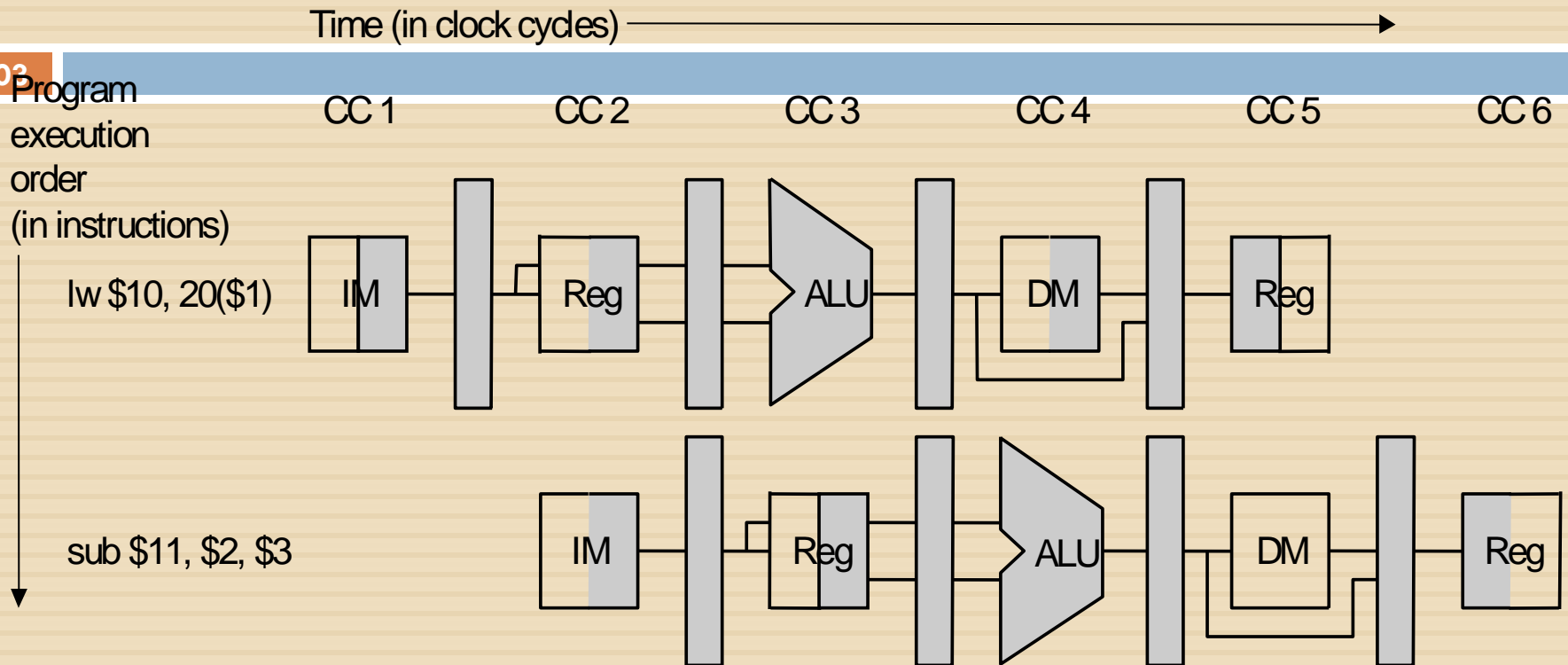
Corrected Datapath

202



Graphically Representing Pipelines

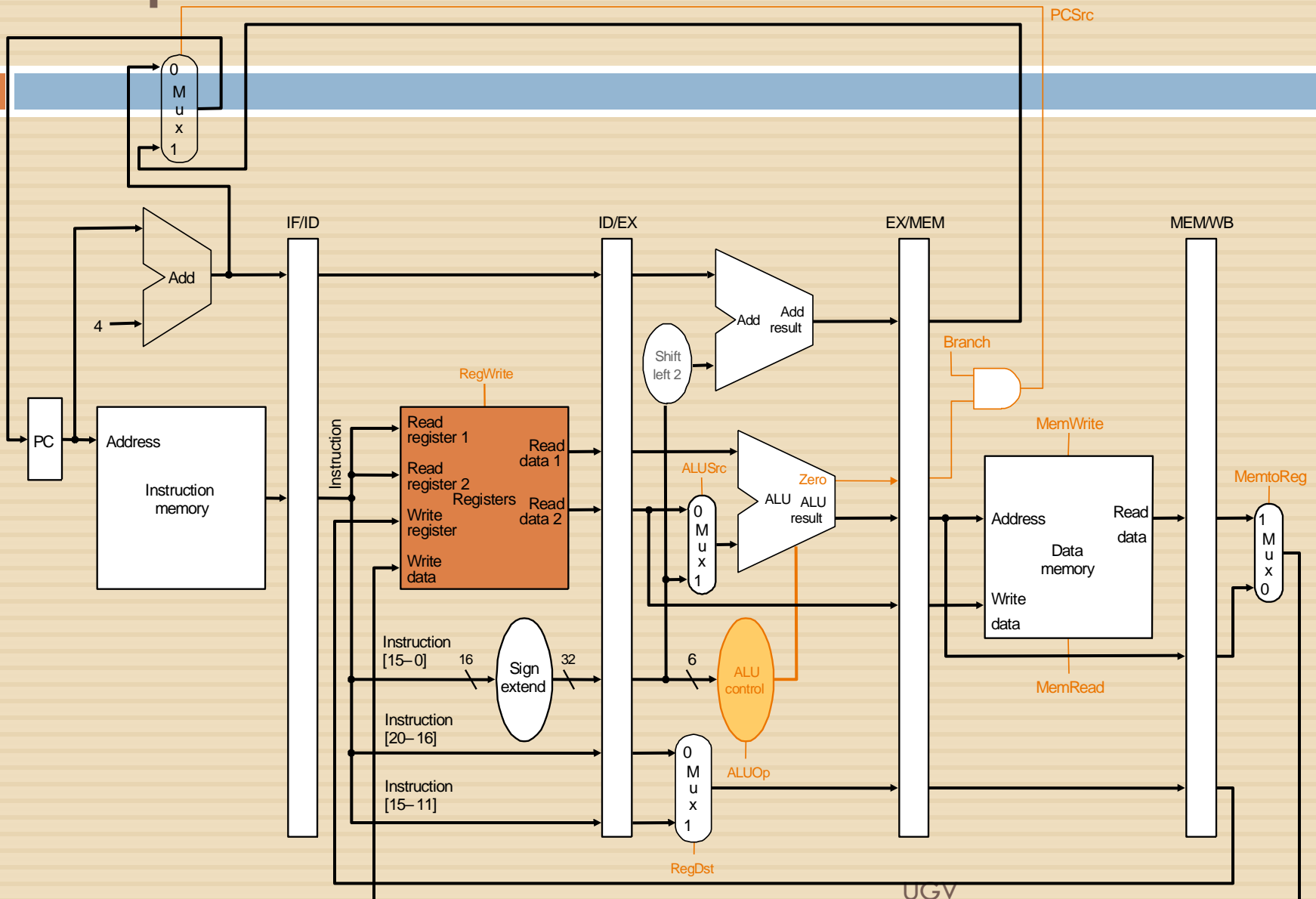
203



- Can help with answering questions like:
 - how many cycles does it take to execute this code?
 - what is the ALU doing during cycle 4?
 - use this representation to help understand datapaths

Pipeline Control

204



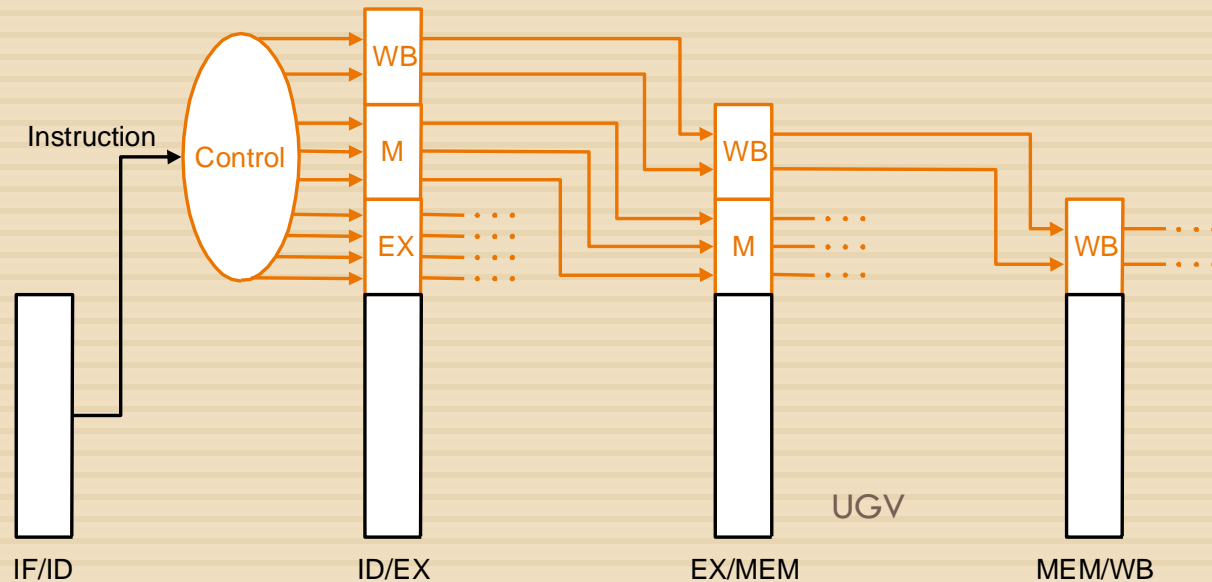
Pipeline Control

205

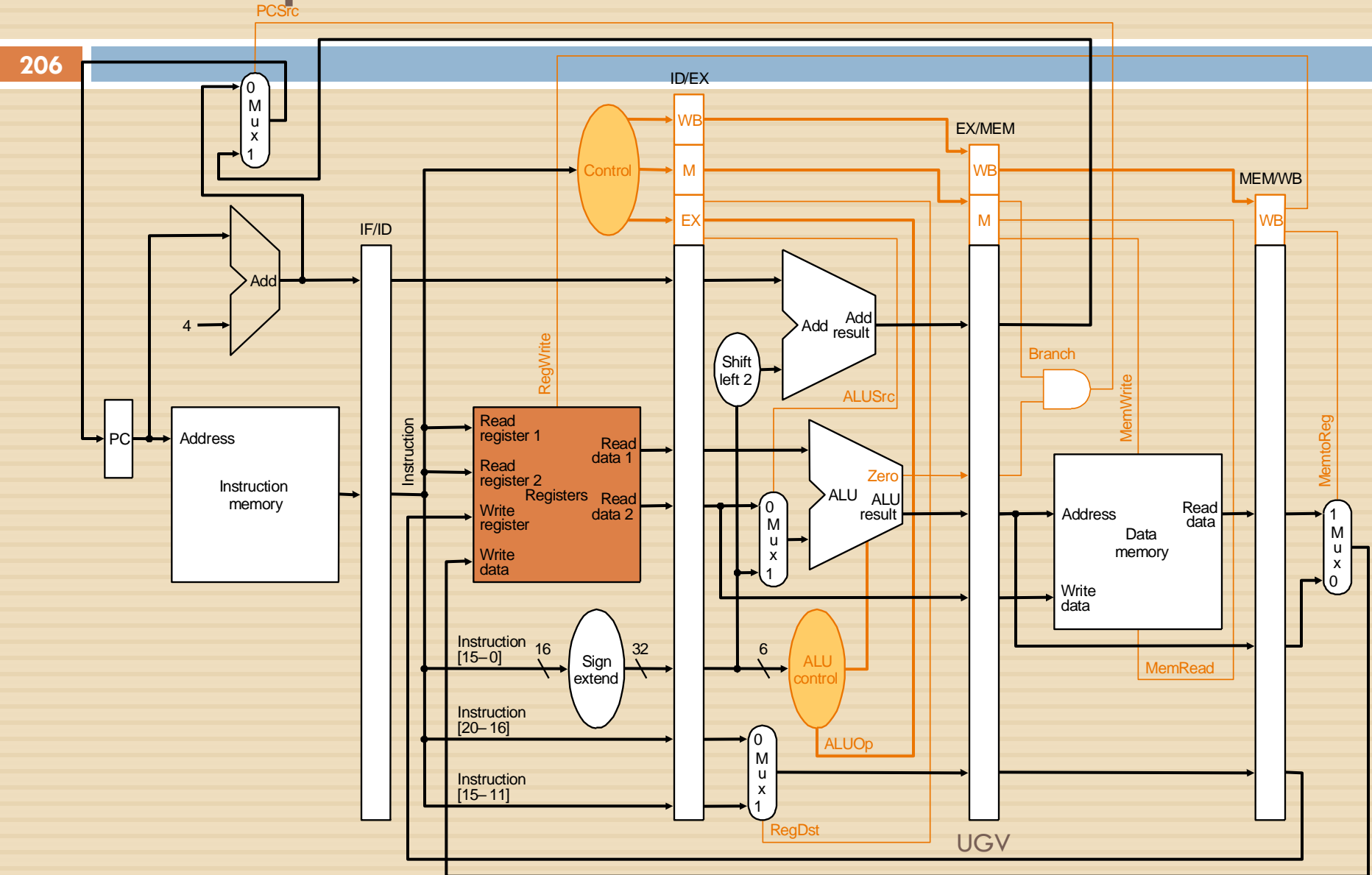
Instruction	Execution/Address Calculation stage control lines				Memory access stage control lines			Write-back stage control lines	
	Reg Dst	ALU Op1	ALU Op0	ALU Src	Branch	Mem Read	Mem Write	Reg write	Mem to Reg
R-format	1	1	0	0	0	0	0	1	0
lw	0	0	0	1	0	1	0	1	1
sw	X	0	0	1	0	0	1	0	X
beq	X	0	1	0	1	0	0	0	X

Pass control signals along

just like the data:



Datapath with Control



Hazards

Hazards

208

Hazards: problems due to pipelining

Hazard types:

- **Structural**
 - ▣ same resource is needed multiple times in the **same** cycle
- **Data**
 - ▣ data **dependencies** limit pipelining
- **Control**
 - ▣ next executed instruction may not be the next specified instruction

Structural hazards

209

Examples:

- Two accesses to a single ported memory
- Two operations need the same function unit at the same time
- Two operations need the same function unit in successive cycles, but the unit is not pipelined

Solutions:

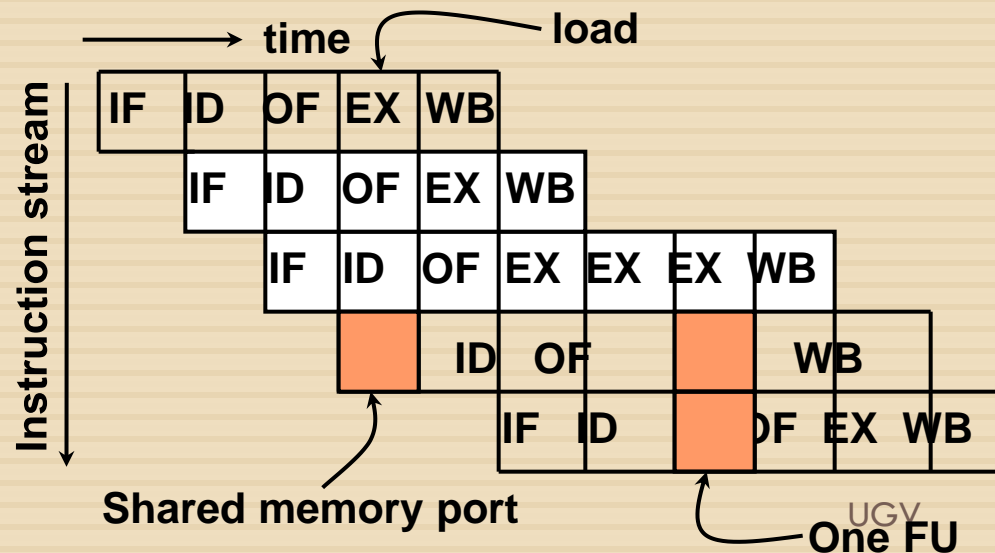
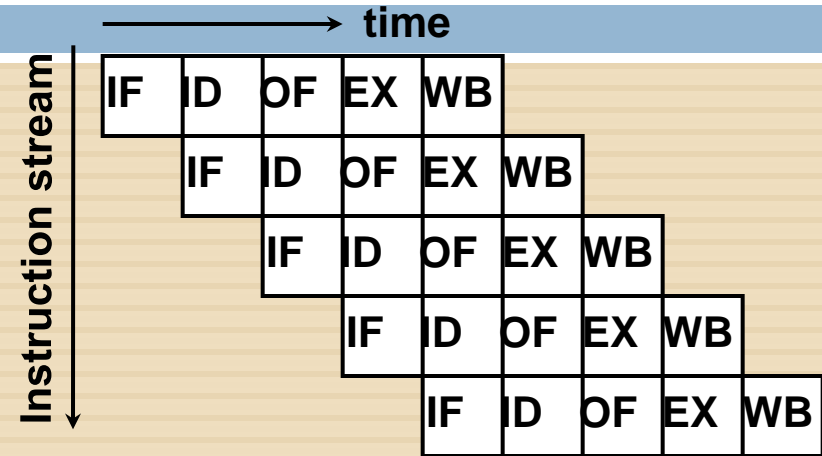
- stalling
- add more hardware

Structural hazards

210

Simple pipelining diagram (but not MIPS!):

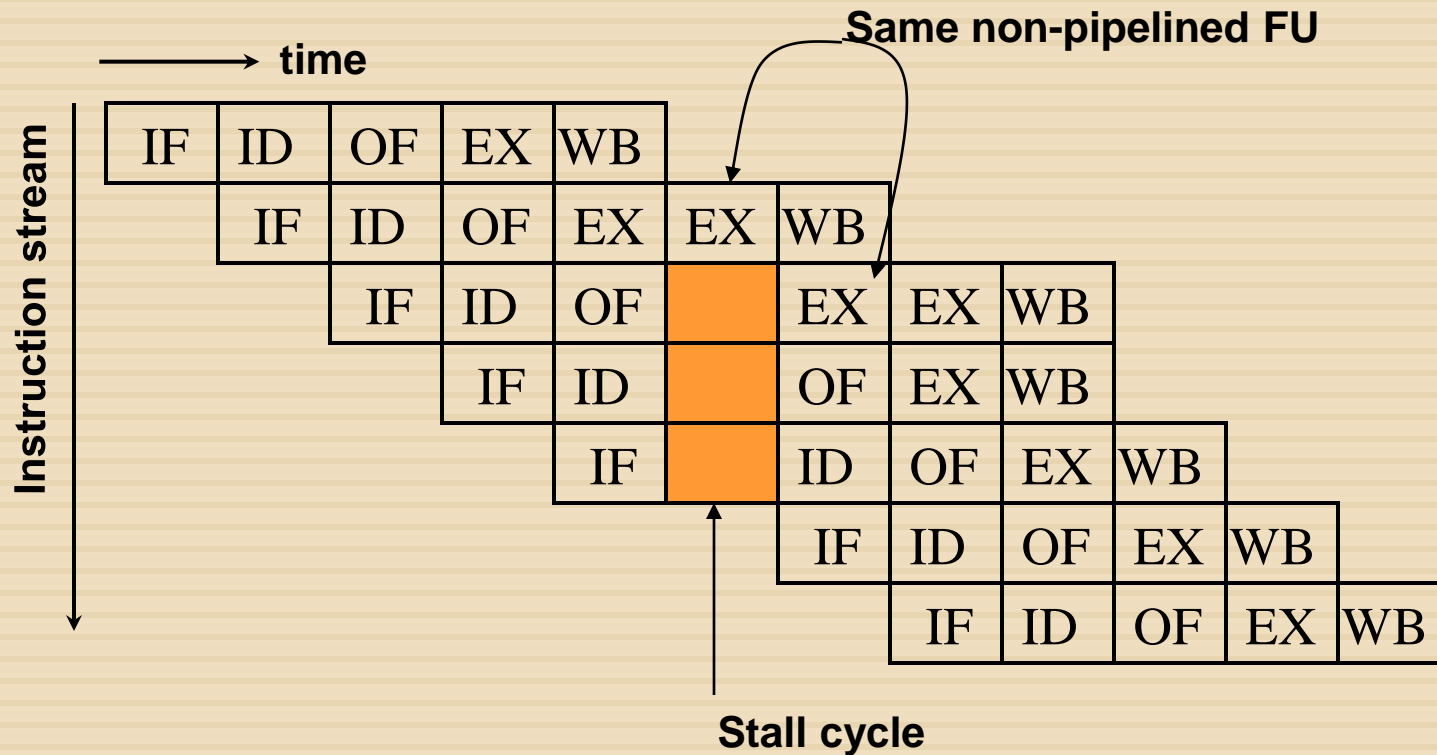
- IF: instruction fetch
- ID: instruction decode
- OF: operand fetch
- EX: execute stage(s)
- WB: write back



Structural hazards

211

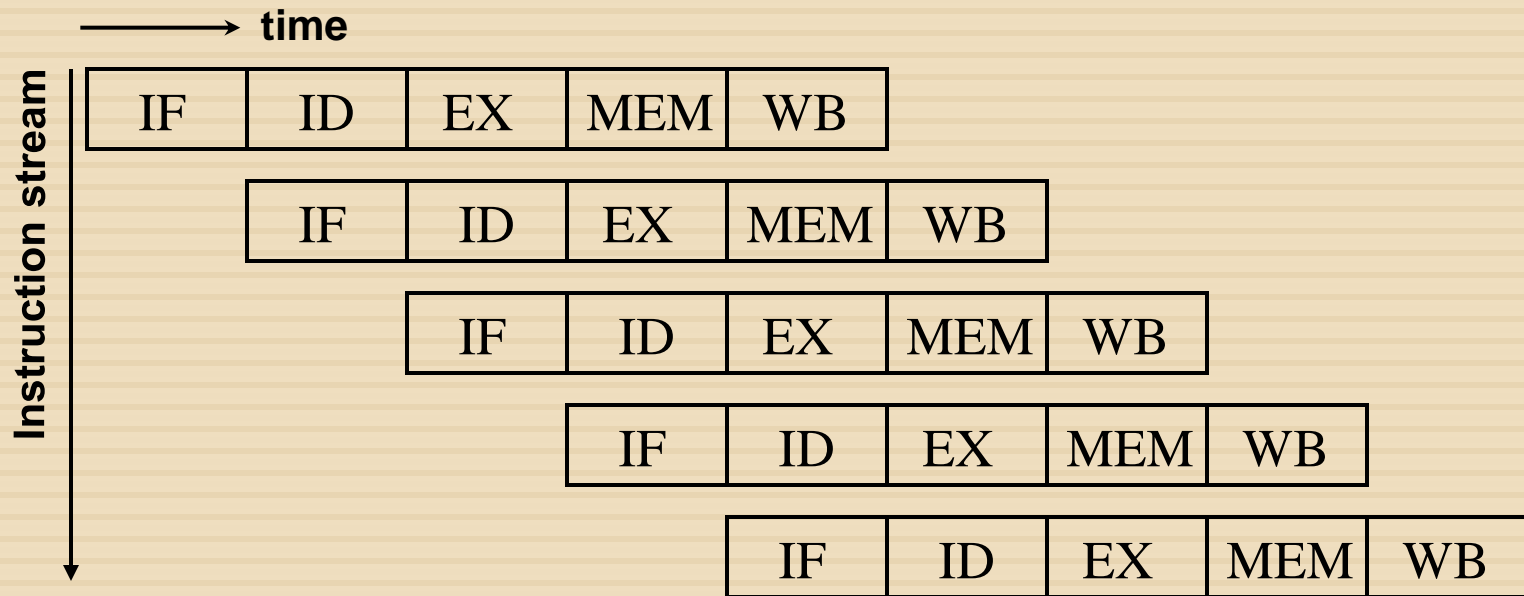
Non-pipelined units



Structural hazards on MIPS

212

Q: Do we have structural hazards on our MIPS pipeline?



Data hazards

213

- Data dependencies:
 - RaW (read-after-write)
 - WaW (write-after-write)
 - WaR (write-after-read)
- Hardware solution:
 - Forwarding / Bypassing
 - Detection logic
 - Stalling
- Software solution: Scheduling

Data dependences

Three types: **RaW**, **WaR** and **WaW**

214

```
add r1, r2, 5          ; r1 := r2+5
sub r4, r1, r3          ; RaW of r1
```

```
add r1, r2, 5
sub r2, r4, 1           ; WaR of r2
```

```
add r1, r2, 5
sub r1, r1, 1 ; WaW of r1
```

```
st  r1, 5(r2) ; M[r2+5] := r1
ld  r5, 0(r4) ; RaW if 5+r2 = 0+r4
```

WaW and WaR do not occur in simple pipelines, but they limit scheduling freedom!

Problems for your compiler and Pentium!

⇒ use **register renaming** to solve this!

Week 11

215

Instruction Set Design

Lecture overview

216

- ISA and Evolution
- Architecture classes
- Addressing
- Operands
- Operations
- Encoding
- RISC
- SIMD extensions

Instruction Set Architecture

- The instruction set architecture serves as the interface between software and hardware
- It provides the mechanism by which the software tells the hardware what should be done
- Architecture definition:
*“the architecture of a system/processor is (a minimal description of) its behavior **as observed by its immediate users**”*



Instruction Set Design Issues

- **Where** are operands stored?
 - registers, memory, stack, accumulator
- How many **explicit** operands are there?
 - 0, 1, 2, or 3
- How is the operand location **specified**?
 - register, immediate, indirect, . . .
- What **type & size** of operands are supported?
 - byte, int, float, double, string, vector. . .
- **What operations** are supported?
 - basic operations: add, sub, mul, move, compare . . .
 - or also very complex operations?

Operands

219

□ How are operands **designated**?

- fixed – always in the same place
- by opcode – always the same for groups of instructions
- by a field in the instruction – requires decode first

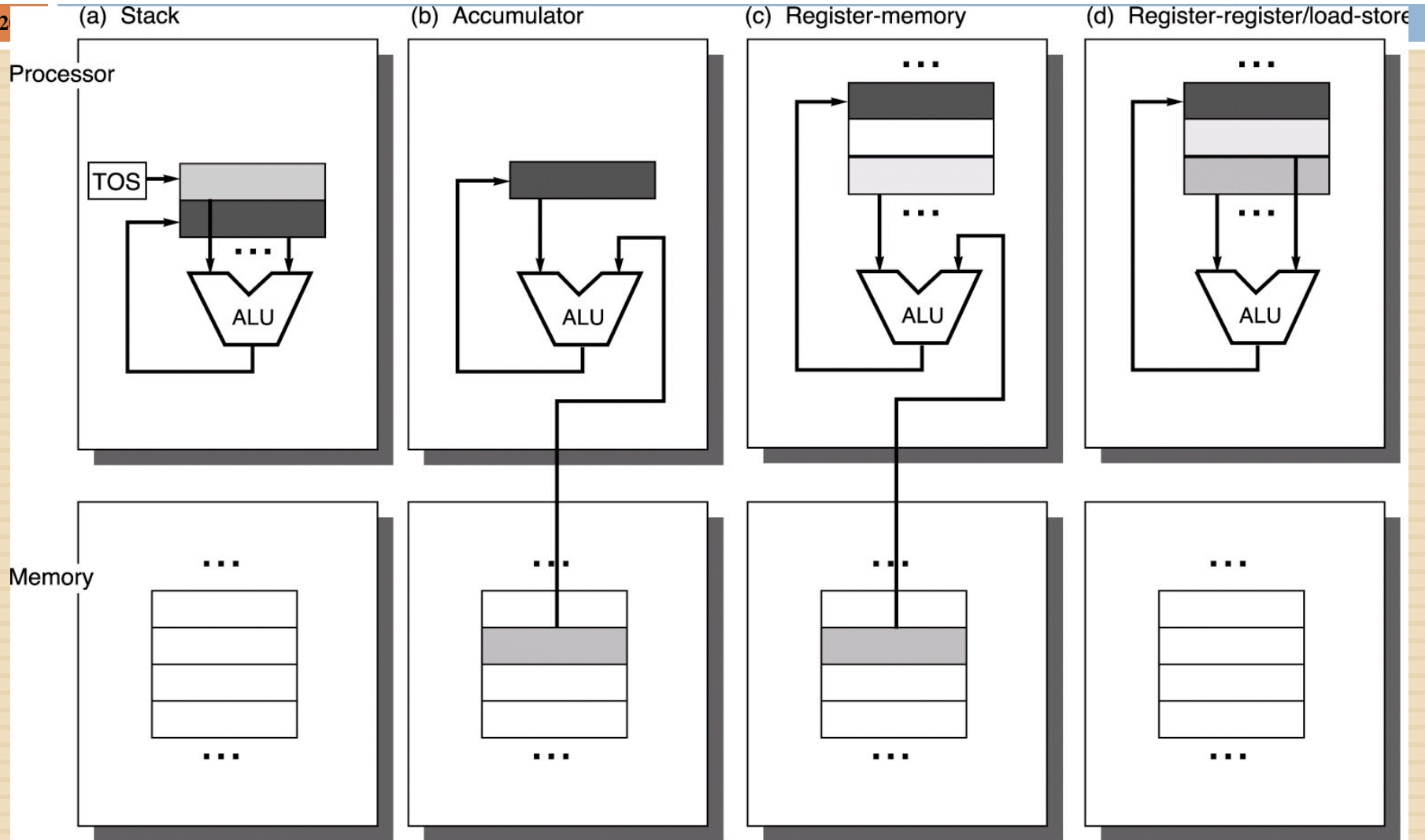
□ What is the **format** of the data?

- binary
- character
- decimal (packed and unpacked)
- floating-point – IEEE 754 (others used less and less)
- size – 8-, 16-, 32-, 64-, 128-bit,
- or **vectors** of above types and sizes

□ What is the influence on the ISA (= Instruction-Set Architecture)?

Operand Locations

22



Classifying ISAs

221

Accumulator (before 1960):

1 address	add A	$\text{acc} \leftarrow \text{acc} + \text{mem}[A]$
-----------	-------	--

Stack (1960s to 1970s):

0 address	add	$\text{tos} \leftarrow \text{tos} + \text{next}$
-----------	-----	--

Memory-Memory (1970s to 1980s):

2 address	add A, B	$\text{mem}[A] \leftarrow \text{mem}[A] + \text{mem}[B]$
3 address	add A, B, C	$\text{mem}[A] \leftarrow \text{mem}[B] + \text{mem}[C]$

Register-Memory (1970s to present):

2 address	add R1, A	$R1 \leftarrow R1 + \text{mem}[A]$
	load R1, A	$R1 \leftarrow \text{mem}[A]$

Register-Register (Load/Store) (1960s to present):

3 address	add R1, R2, R3	$R1 \leftarrow R2 + R3$
	load R1, R2	$R1 \leftarrow \text{mem}[R2]$
	store R1, R2	$\text{mem}[R1] \leftarrow R2$

Evolution of Architectures

Single Accumulator (EDSAC 1950)

Accumulator + Index Registers

(Manchester Mark I, IBM 700 series 1953)

**Separation of Programming Model
from Implementation**

**High-level Language Based
(B5000 1963)**

**Concept of a Processor Family
(IBM 360 1964)**

General Purpose Register Machines

**Complex Instruction Sets
(Vax, Intel 8086 1977-80)**

**Load/Store Architecture
(CDC 6600, Cray 1 1963-76)**

RISC

(Mips, Sparc, 88000, IBM RS6000, . . . 1987+)

Addressing Modes

223

□ Types

- Register – data in a register
- Immediate – data in the instruction
- Memory – data in memory

□ Calculation of *Effective Address*

- Direct – address in instruction
- Indirect – address in register
- Displacement – address = register or PC + offset
- Indexed – address = register + register
- Memory Indirect – address at address in register

□ **Question:** What is the influence on ISA?

Types of Addressing Mode (VAX)

Addressing Mode	Example	Action
1. Register direct	Add R4, R3	$R4 \leftarrow R4 + R3$
2. Immediate	Add R4, #3	$R4 \leftarrow R4 + 3$
3. Displacement	Add R4, 100(R1)	$R4 \leftarrow R4 + M[100 + R1]$
4. Register indirect	Add R4, (R1)	$R4 \leftarrow R4 + M[R1]$
5. Indexed	Add R4, (R1 + R2)	$R4 \leftarrow R4 + M[R1 + R2]$
6. Direct	Add R4, (1000)	$R4 \leftarrow R4 + M[1000]$
7. Memory Indirect	Add R4, @(R3)	$R4 \leftarrow R4 + M[M[R3]]$
8. Autoincrement	Add R4, (R2)+	$R4 \leftarrow R4 + M[R2]$ $R2 \leftarrow R2 + d$
9. Autodecrement	Add R4, (R2)-	$R4 \leftarrow R4 + M[R2]$ $R2 \leftarrow R2 - d$
10. Scaled	Add R4, 100(R2)[R3]	$R4 \leftarrow R4 +$ $M[100 + R2 + R3*d]$

- Studies by [Clark and Emer] indicate that *modes 1-4 account for 93% of all operands on the VAX*

Operations

225

□ Types

- ALU – Integer arithmetic and logical functions
- Data transfer – Loads/stores
- Control – Branch, jump, call, return, traps, interrupts
- System – O/S calls, virtual memory management
- Floating point – Floating point arithmetic
- Decimal – Decimal arithmetic (BCD: binary coded decimal)
- String – moves, compares, search, etc.
- Graphics – Pixel/vertex operations
- Vector – Vector (SIMD) functions
- more complex ones

□ Addressing

- Which addressing modes for which operands are supported?

80x86 Instruction Frequency

<i>Rank</i>	<i>Instruction</i>	<i>Frequency</i>
1	load	22%
2	branch	20%
3	compare	16%
4	store	12%
5	add	8%
6	and	6%
7	sub	5%
8	register move	4%
9	call	1%
10	return	1%
Total		96%

Relative Frequency of Control Instructions

Operation	SPECint92	SPECfp92
Call/Return	13%	11%
Jumps	6%	4%
Branches	81%	87%

- Design hardware to handle branches quickly, since these occur most frequently

Frequency of Operand Sizes on 32-bit Load-Store Machines

Size	SPECint92	SPECfp92
64 bits	0%	69%
32 bits	74%	31%
16 bits	19%	0%
8 bits	19%	0%

- For floating-point want good performance for 64 bit operands.
- For integer operations want good performance for 32 bit operands
- Recent architectures also support 64-bit integers

Instruction Encoding

Variable

- Instruction length varies based on opcode and address specifiers
- For example, VAX instructions vary between 1 and 53 bytes, while x86 instruction vary between 1 and 17 bytes.
- Good code density, but difficult to decode and pipeline

Fixed

- Only a single size for all instructions
- For example MIPS, Power PC, Sparc all have 32 bit instructions
- Not as good code density, but easier to decode and pipeline

Hybrid

- Have multiple format lengths specified by the opcode
- For example, IBM 360/370
- Compromise between code density and ease of decode

Instruction Encoding

230

Operation and no. of operands	Address specifier 1	Address field 1	...	Address specifier	Address field
----------------------------------	------------------------	--------------------	-----	----------------------	------------------

(a) Variable (e.g., VAX, Intel 80x86)

Operation	Address field 1	Address field 2	Address field 3
-----------	--------------------	--------------------	--------------------

(b) Fixed (e.g., Alpha, ARM, MIPS, PowerPC, SPARC, SuperH)

Operation	Address specifier	Address field
-----------	----------------------	------------------

Operation	Address specifier 1	Address specifier 2	Address field
-----------	------------------------	------------------------	------------------

Operation	Address specifier	Address field 1	Address field 2
-----------	----------------------	--------------------	--------------------

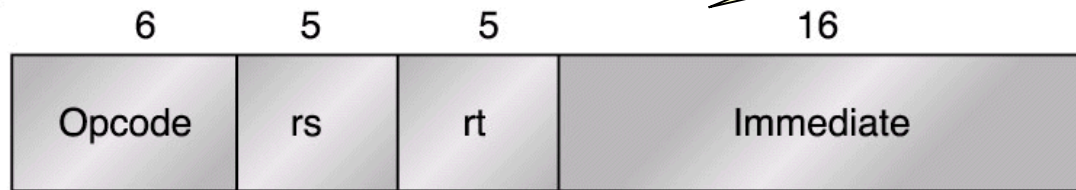
(c) Hybrid (e.g., IBM 360/70, MIPS16, Thumb, TI TMS320C54x)

Example: MIPS

231

Operands mostly at fixed positions

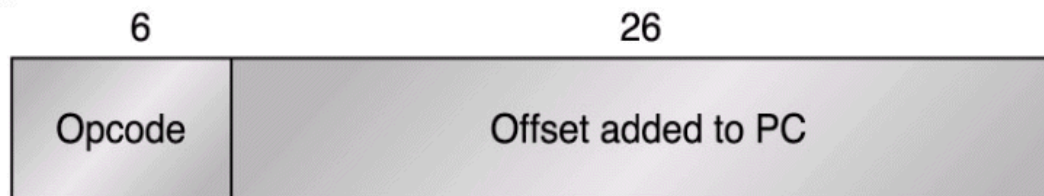
I-type instruction



R-type instruction



J-type instruction



Fixed instruction size; few formats

Compilers and ISA

□ Compiler Goals

- All correct programs compile correctly
- Most compiled programs execute quickly
- Most programs compile quickly
- Achieve small code size
- Provide debugging support

□ Multiple Source Compilers

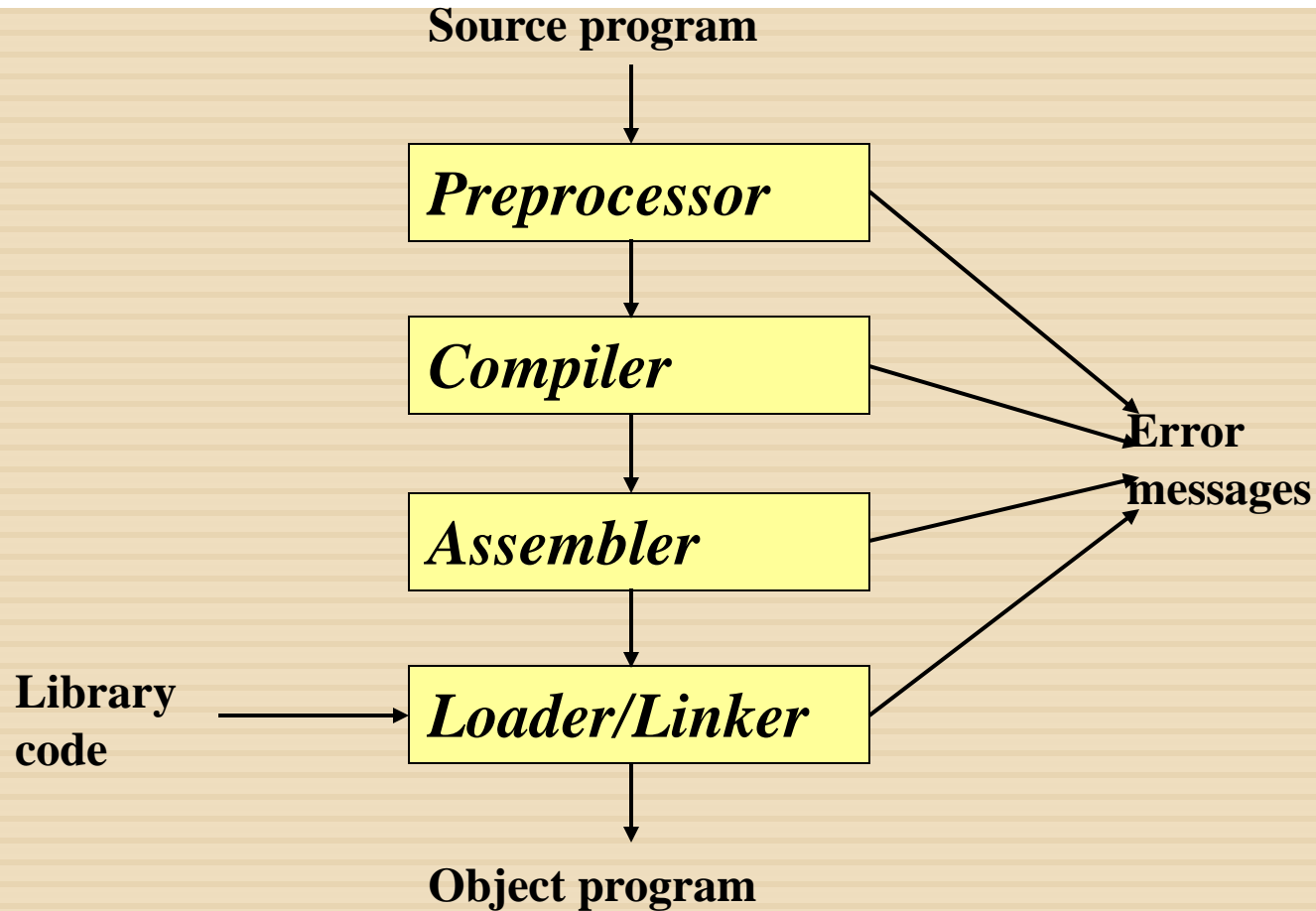
- Same compiler can compile different languages

□ Multiple Target Compilers

- Same compiler can generate code for different machines
- 'cross-compiler'

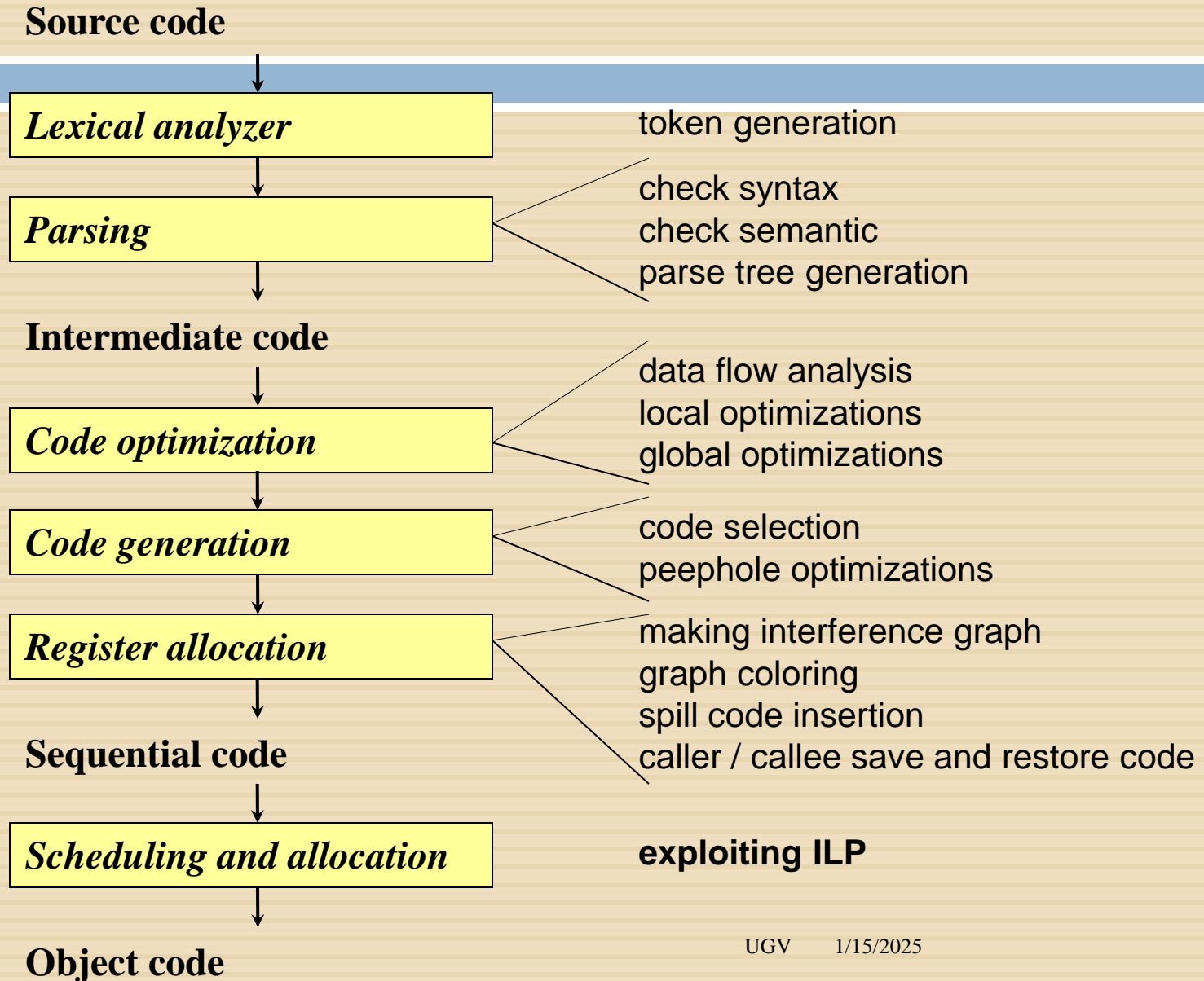
Compiler basics: trajectory

233



Compiler basics: **structure** / **passes**

234

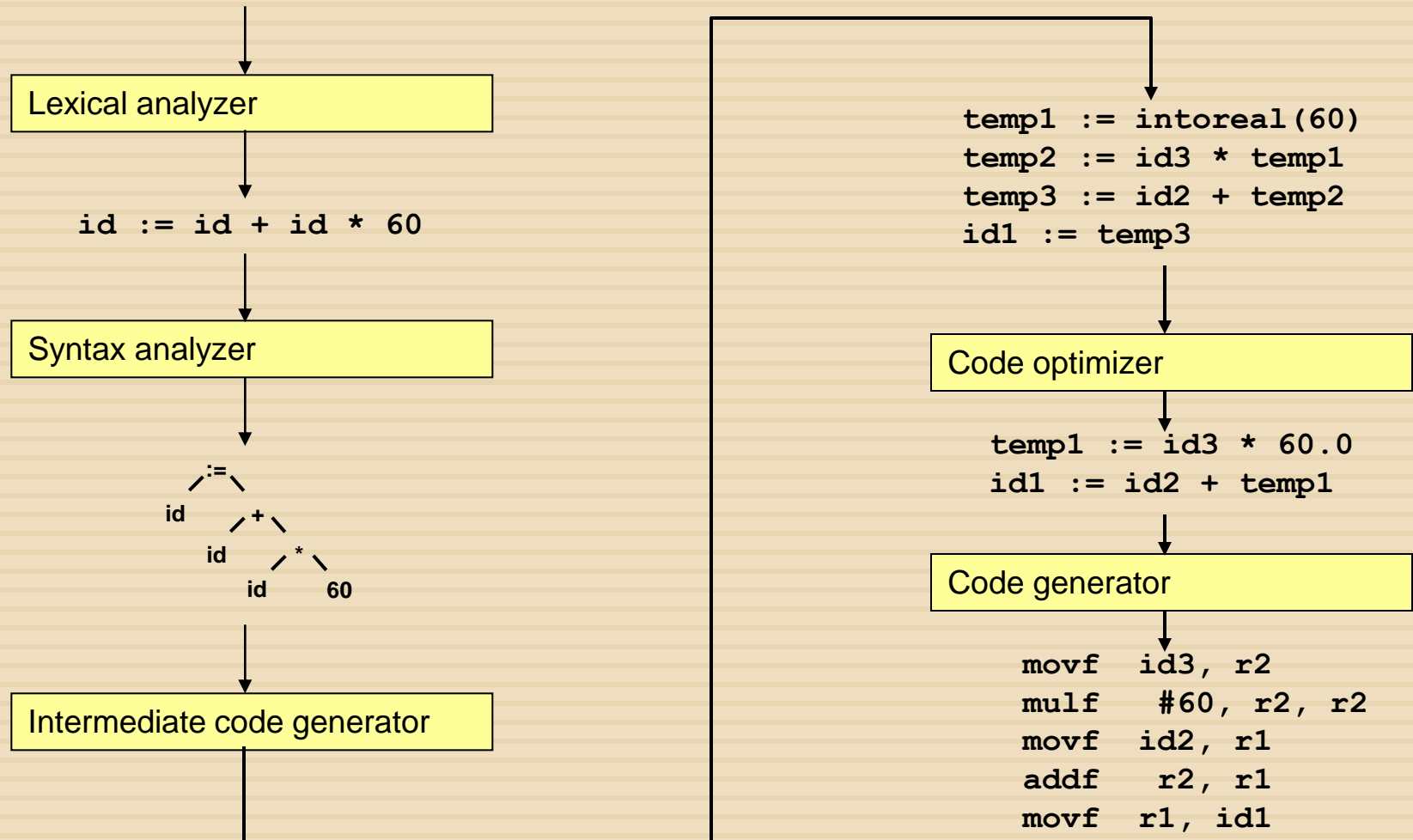


Compiler basics: structure

Simple compilation example

235

```
position := initial + rate * 60
```



Week 12

236

ILP architectures
with emphasis on Superscalar

Topics

237

- Introduction
- Hazards
- Dependences limit ILP: scheduling
- Out-Of-Order execution: Hardware speculation
- Branch prediction
- Multiple issue
- How much ILP is there?

Introduction

238

ILP = Instruction level parallelism

- multiple operations (or instructions) can be executed in parallel

Needed:

- Sufficient resources
- Parallel scheduling
 - ▣ Hardware solution
 - ▣ Software solution
- Application should contain ILP

Hazards

239

- Three types of hazards (see previous lecture)
 - ▣ Structural
 - multiple instructions need access to the same hardware at the same time
 - ▣ Data dependence
 - there is a dependence between operands (in register or memory) of successive instructions
 - ▣ Control dependence
 - determines the order of the execution of basic blocks
- Hazards cause scheduling problems

Data dependences

240

- **RaW** read after write
 - real or flow dependence
 - can only be avoided by value prediction (i.e. speculating on the outcome of a previous operation)
- **WaR** write after read
- **WaW** write after write
 - WaR and WaW are false dependencies
 - Could be avoided by **renaming** (if sufficient registers are available)

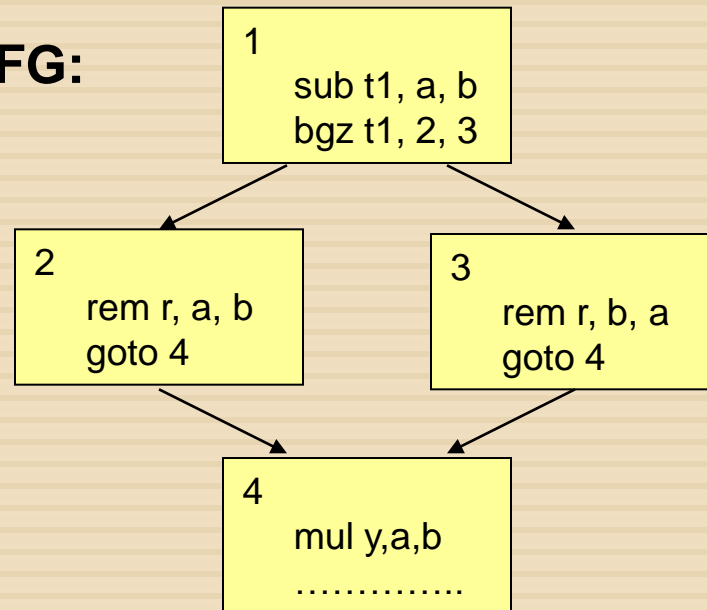
Note: data dependences can be both between register data and memory data operations

Control Dependences

241

C input code: `if (a > b) { r = a % b; }
 else { r = b % a; }
 y = a*b;`

CFG:



Question: How real are control dependences?

Dynamic Scheduling Principle

242

What we examined so far is *static scheduling*

- Compiler reorders instructions so as to avoid hazards and reduce stalls
- *Dynamic scheduling*: hardware rearranges instruction execution to reduce stalls
- **Example:**

DIV.D F0,F2,F4 ; takes 24 cycles and
; is not pipelined

ADD.D F10,F0,F8

SUB.D F12,F8,F14

- Key idea: Allow instructions behind stall to proceed
- Book describes Tomasulo algorithm, but we describe general idea

***This instruction cannot continue
even though it does not depend
on anything***

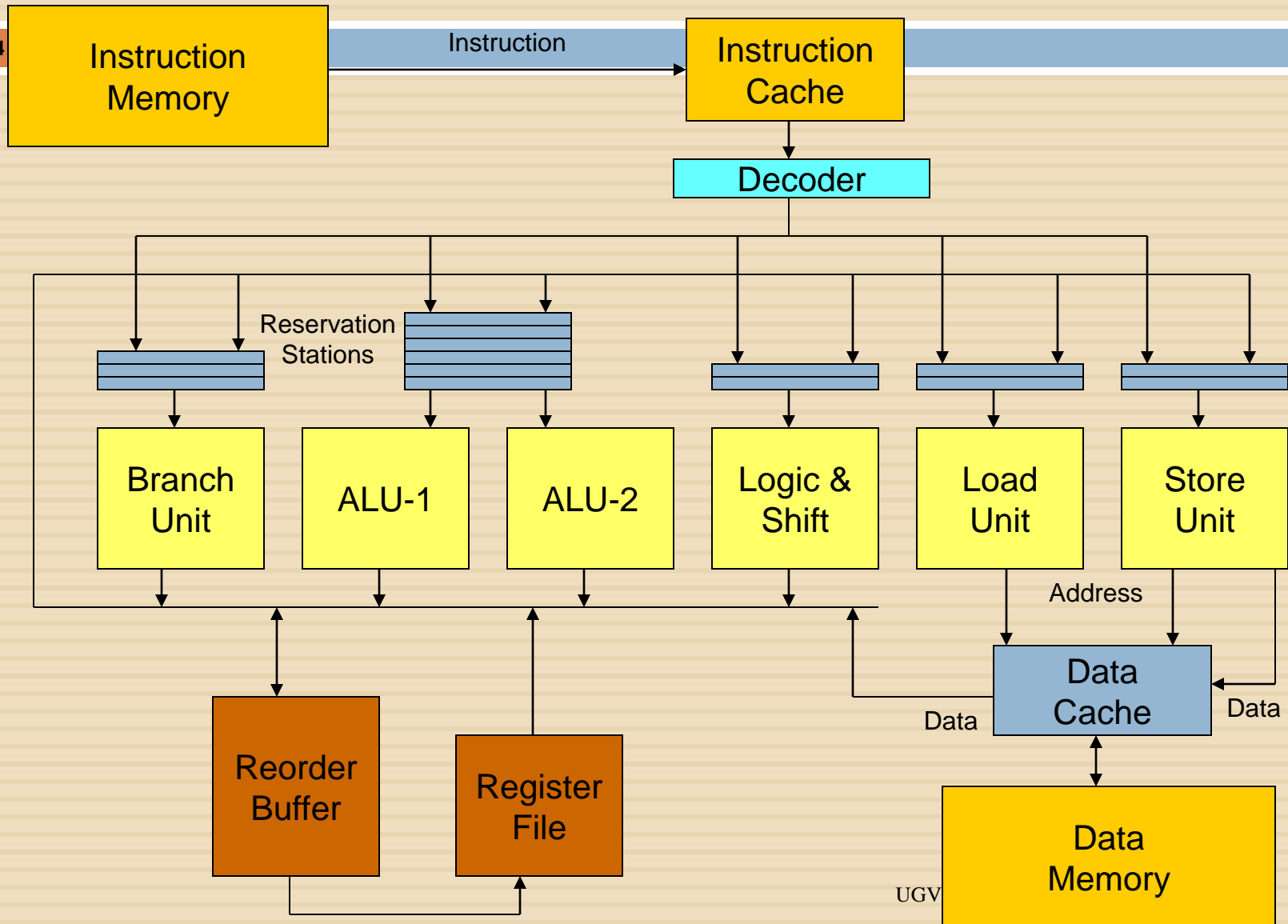
Advantages of Dynamic Scheduling

243

- Handles cases when dependences unknown at compile time
 - e.g., because they may involve a memory reference
- It simplifies the compiler
- Allows code compiled for one machine to run efficiently on a different machine, with different number of function units (FUs), and different pipelining
- Hardware **speculation**, a technique with significant performance advantages, that builds on dynamic scheduling

Superscalar Concept

244



Superscalar Issues

245

- How to fetch multiple instructions in time (across basic block boundaries) ?
- Predicting branches
- Non-blocking memory system
- Tune #resources(FUs, ports, entries, etc.)
- Handling dependencies
- How to support precise interrupts?
- How to recover from a mis-predicted branch path?

- For the latter two issues you may have look at sequential, look-ahead, and architectural state
 - ▣ Ref: Johnson 91 (PhD thesis)

Example of Superscalar Processor Execution

246

Superscalar processor organization:

- simple pipeline: IF, EX, WB
- fetches 2 instructions each cycle
- 2 ld/st units, dual-ported memory; 2 FP adders; 1 FP multiplier
- Instruction window (buffer between IF and EX stage) is of size 2
- FP ld/st takes 1 cc; FP +/- takes 2 cc; FP * takes 4 cc; FP / takes 8 cc

Cycle	1	2	3	4	5	6	7
L.D	F6, 32 (R2)						
L.D	F2, 48 (R3)						
MUL.D	F0, F2, F4						
SUB.D	F8, F2, F6						
DIV.D	F10, F0, F6						
ADD.D	F6, F8, F2						
MUL.D	F12, F2, F4						

Example of Superscalar Processor Execution

247

Superscalar processor organization:

- simple pipeline: IF, EX, WB
- fetches 2 instructions each cycle
- 2 ld/st units, dual-ported memory; 2 FP adders; 1 FP multiplier
- Instruction window (buffer between IF and EX stage) is of size 2
- FP ld/st takes 1 cc; FP +/- takes 2 cc; FP * takes 4 cc; FP / takes 8 cc

Cycle		1	2	3	4	5	6	7
L.D	F6, 32 (R2)	IF						
L.D	F2, 48 (R3)	IF						
MUL.D	F0, F2, F4							
SUB.D	F8, F2, F6							
DIV.D	F10, F0, F6							
ADD.D	F6, F8, F2							
MUL.D	F12, F2, F4							

Example of Superscalar Processor Execution

248

Superscalar processor organization:

- simple pipeline: IF, EX, WB
- fetches 2 instructions each cycle
- 2 ld/st units, dual-ported memory; 2 FP adders; 1 FP multiplier
- Instruction window (buffer between IF and EX stage) is of size 2
- FP ld/st takes 1 cc; FP +/- takes 2 cc; FP * takes 4 cc; FP / takes 8 cc

Cycle		1	2	3	4	5	6	7
L.D	F6, 32 (R2)	IF	EX					
L.D	F2, 48 (R3)	IF	EX					
MUL.D	F0, F2, F4		IF					
SUB.D	F8, F2, F6		IF					
DIV.D	F10, F0, F6							
ADD.D	F6, F8, F2							
MUL.D	F12, F2, F4							

Example of Superscalar Processor Execution

249 Superscalar processor organization:

- simple pipeline: IF, EX, WB
- fetches 2 instructions each cycle
- 2 ld/st units, dual-ported memory; 2 FP adders; 1 FP multiplier
- Instruction window (buffer between IF and EX stage) is of size 2
- FP ld/st takes 1 cc; FP +/- takes 2 cc; FP * takes 4 cc; FP / takes 8 cc

Cycle		1	2	3	4	5	6	7
L.D	F6, 32 (R2)	IF	EX	WB				
L.D	F2, 48 (R3)	IF	EX	WB				
MUL.D	F0, F2, F4		IF	EX				
SUB.D	F8, F2, F6		IF	EX				
DIV.D	F10, F0, F6			IF				
ADD.D	F6, F8, F2			IF				
MUL.D	F12, F2, F4							

Example of Superscalar Processor Execution

250

Superscalar processor organization:

- simple pipeline: IF, EX, WB
- fetches 2 instructions each cycle
- 2 ld/st units, dual-ported memory; 2 FP adders; 1 FP multiplier
- Instruction window (buffer between IF and EX stage) is of size 2
- FP ld/st takes 1 cc; FP +/- takes 2 cc; FP * takes 4 cc; FP / takes 8 cc

Cycle		1	2	3	4	5	6	7
L.D	F6, 32 (R2)	IF	EX	WB				
L.D	F2, 48 (R3)	IF	EX	WB				
MUL.D	F0, F2, F4		IF	EX	EX			
SUB.D	F8, F2, F6		IF	EX	EX			
DIV.D	F10, F0, F6			IF				
ADD.D	F6, F8, F2			IF				
MUL.D	F12, F2, F4							

stall because
of data dep.

cannot be fetched because window full

Example of Superscalar Processor Execution

251 Superscalar processor organization:

- simple pipeline: IF, EX, WB
- fetches 2 instructions each cycle
- 2 ld/st units, dual-ported memory; 2 FP adders; 1 FP multiplier
- Instruction window (buffer between IF and EX stage) is of size 2
- FP ld/st takes 1 cc; FP +/- takes 2 cc; FP * takes 4 cc; FP / takes 8 cc

Cycle		1	2	3	4	5	6	7
L.D	F6, 32 (R2)	IF	EX	WB				
L.D	F2, 48 (R3)	IF	EX	WB				
MUL.D	F0, F2, F4		IF	EX	EX	EX		
SUB.D	F8, F2, F6		IF	EX	EX	WB		
DIV.D	F10, F0, F6			IF				
ADD.D	F6, F8, F2			IF		EX		
MUL.D	F12, F2, F4					IF		

Example of Superscalar Processor Execution

252

Superscalar processor organization:

- simple pipeline: IF, EX, WB
- fetches 2 instructions each cycle
- 2 ld/st units, dual-ported memory; 2 FP adders; 1 FP multiplier
- Instruction window (buffer between IF and EX stage) is of size 2
- FP ld/st takes 1 cc; FP +/- takes 2 cc; FP * takes 4 cc; FP / takes 8 cc

Cycle		1	2	3	4	5	6	7
L.D	F6, 32 (R2)	IF	EX	WB				
L.D	F2, 48 (R3)	IF	EX	WB				
MUL.D	F0, F2, F4		IF	EX	EX	EX	EX	
SUB.D	F8, F2, F6		IF	EX	EX	WB		
DIV.D	F10, F0, F6			IF				
ADD.D	F6, F8, F2			IF		EX	EX	
MUL.D	F12, F2, F4					IF		

Example of Superscalar Processor Execution

253

Superscalar processor organization:

- simple pipeline: IF, EX, WB
- fetches 2 instructions each cycle
- 2 ld/st units, dual-ported memory; 2 FP adders; 1 FP multiplier
- Instruction window (buffer between IF and EX stage) is of size 2
- FP ld/st takes 1 cc; FP +/- takes 2 cc; FP * takes 4 cc; FP / takes 8 cc

Cycle		1	2	3	4	5	6	7
L.D	F6, 32 (R2)	IF	EX	WB				
L.D	F2, 48 (R3)	IF	EX	WB				
MUL.D	F0, F2, F4		IF	EX	EX	EX	EX	WB
SUB.D	F8, F2, F6		IF	EX	EX	WB		
DIV.D	F10, F0, F6			IF				EX
ADD.D	F6, F8, F2			IF		EX	EX	WB
MUL.D	F12, F2, F4					IF		?

Register Renaming

254

- A technique to eliminate anti- and output dependencies
- Can be implemented
 - ▣ by the compiler
 - advantage: low cost
 - disadvantage: “old” codes perform poorly
 - ▣ in hardware
 - advantage: binary compatibility
 - disadvantage: extra hardware needed
- We describe the general idea

Register Renaming

- there's a *physical register file* larger than *logical register file*
- *mapping table* associates logical registers with physical register
- when an instruction is decoded
 - its physical source registers are obtained from mapping table
 - its physical destination register is obtained from a *free list*
 - mapping table is updated

before: add r3, r3, 4

current mapping table:

r0	R8
r1	R7
r2	R5
r3	R1
r4	R9

current free list: R2 R6

after: add R2, R1, 4

new mapping table:

r0	R8
r1	R7
r2	R5
r3	R2
r4	R9

new free list: R6

Week 13

256

Exploiting ILP with SW approaches

Topics

257

- Static branch prediction and speculation
- Basic compiler techniques
- Multiple issue architectures
- Advanced compiler support techniques
 - ▣ Loop-level parallelism
 - ▣ Software pipelining
- Hardware support for compile-time scheduling

We discussed previously dynamic branch
prediction

This does not help the compiler !!!

We need Static Branch Prediction

Static Branch Prediction and Speculation

259

- Static branch prediction useful for code scheduling
- Example:

```
        ld      r1, 0(r2)
        sub     r1, r1, r3          # hazard
        beqz    r1, L
        or      r4, r5, r6
        addu    r10, r4, r3
L:      addu    r7, r8, r9
```

- If the branch is **taken** most of the times and since $r7$ is not needed on the fall-through path, we could move
addu $r7, r8, r9$ directly after the ld
- If the branch is **not taken** most of the times and assuming that $r4$ is not needed on the taken path, we could move
or $r4, r5, r6$ after the ld

Static Branch Prediction Methods

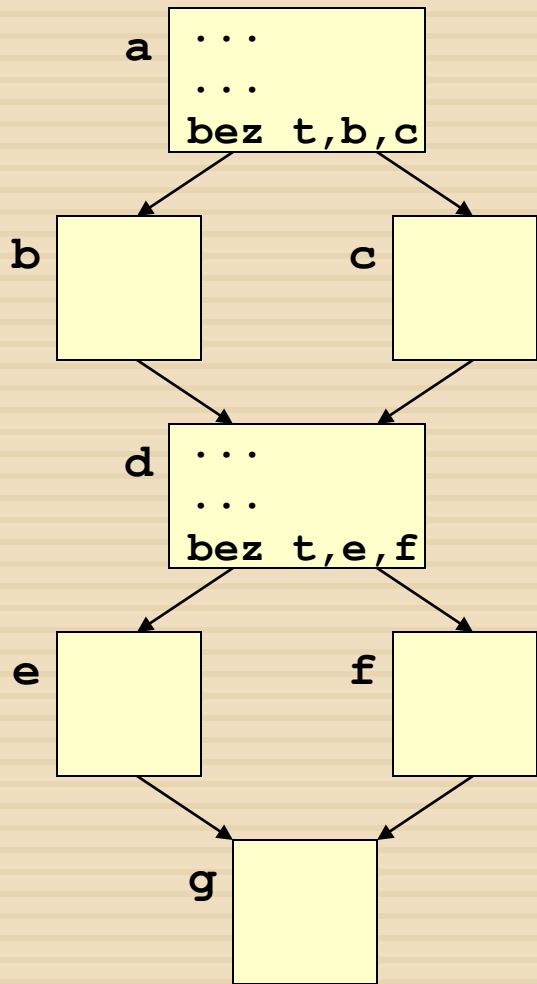
260

- Always predict taken
 - ▣ Average misprediction rate for SPEC: 34% (9%-59%)
- Backward branches predicted taken, forward branches not taken
 - ▣ In SPEC, most forward branches are taken, so always predict taken is better
- Profiling
 - ▣ Run the program and profile all branches. If a branch is taken (not taken) most of the times, it is predicted taken (not taken)
 - ▣ Behavior of a branch is often biased to taken or not taken
 - ▣ Average misprediction rate for SPECint: 15% (11%-22%), SPECfp: 9% (5%-15%)
- Can we do better? YES, use control flow restructuring to **exploit correlation**

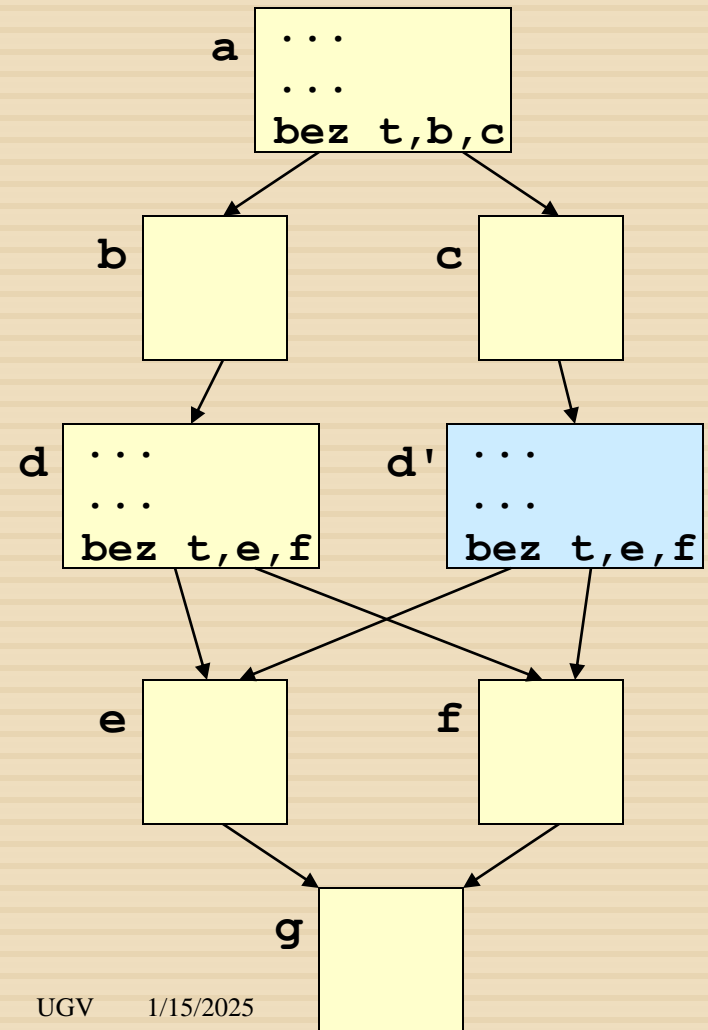
Static exploitation of correlation

261

If correlation,
branch direction
in block d depends
on branch in block a



control flow
restructuring



Basic compiler techniques

262

- Dependencies limit ILP (Instruction-Level Parallelism)
 - ▣ We can not always find sufficient independent operations to fill all the delay slots
 - ▣ May result in pipeline stalls
- **Scheduling** to avoid stalls
- **Loop unrolling**: create more exploitable parallelism

Dependencies Limit ILP: Example

2 C loop:

```
for (i=1; i<=1000; i++)  
    x[i] = x[i] + s;
```

MIPS assembly code:

```
; R1 = &x[1]  
; R2 = &x[1000]+8  
; F2 = s
```

Loop: L.D	F0, 0(R1)	; F0 = x[i]
ADD.D	F4, F0, F2	; F4 = x[i]+s
S.D	0(R1), F4	; x[i] = F4
ADDI	R1, R1, 8	; R1 = &x[i+1]
BNE	R1, R2, Loop	; branch if R1!=&x[1000]+8

Schedule this on a MIPS Pipeline

264

- FP operations are mostly **multicycle**
- The pipeline must be **stalled** if an instruction uses the result of a not yet finished multicycle operation
- We'll assume the following latencies

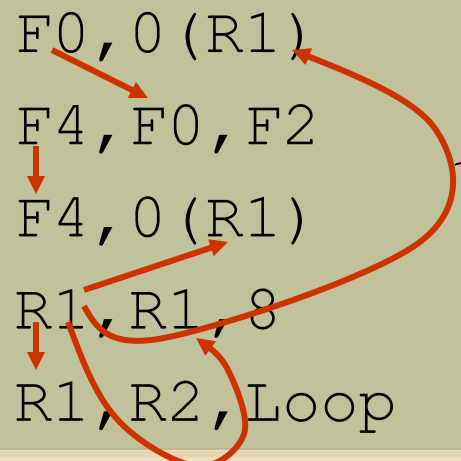
<i>Producing instruction</i>	<i>Consuming instruction</i>	<i>Latency (clock cycles)</i>
FP ALU op	FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0

Where to Insert Stalls?

265

- How would this loop be executed on the MIPS FP pipeline?

```
Loop:      L.D      F0, 0(R1)
           ADD.D    F4, F0, F2
           S.D      F4, 0(R1)
           ADDI     R1, R1, 8
           BNE      R1, R2, Loop
```



Inter-iteration
dependence

Which true (flow) dependences?

Where to Insert Stalls

266

- How would this loop be executed on the MIPS FP pipeline?
- 10 cycles per iteration

```
Loop: L.D      F0, 0(R1)
      stall
      ADD.D    F4, F0, F2
      stall
      stall
      S.D      0(R1), F4
      ADDI     R1, R1, 8
      stall
      BNE      R1, R2, Loop
      stall
```

Code Scheduling to Avoid Stalls

267

- Can we reorder the order of instruction to avoid stalls?
- Execution time reduced from 10 to 6 cycles per iteration

```
Loop:  L.D    F0, 0(R1)
        ADDI   R1, R1, 8
        ADD.D  F4, F0, F2
        stall
        BNE    R1, R2, Loop
        S.D    -8(R1), F4
```

watch out!

- But only 3 instructions perform useful work, rest is loop overhead.

How to avoid this ???

Loop Unrolling: increasing ILP

268

At source level:

```
for (i=1; i<=1000; i++)  
    x[i] = x[i] + s;
```



```
for (i=1; i<=1000; i=i+4)  
{  
    x[i]    = x[i] + s;  
    x[i+1]  = x[i+1]+s;  
    x[i+2]  = x[i+2]+s;  
    x[i+3]  = x[i+3]+s;  
}
```

MIPS code after scheduling:

```
Loop: L.D      F0, 0(R1)  
      L.D      F6, 8(R1)  
      L.D      F10, 16(R1)  
      L.D      F14, 24(R1)  
      ADD.D    F4, F0, F2  
      ADD.D    F8, F6, F2  
      ADD.D    F12, F10, F2  
      ADD.D    F16, F14, F2  
      S.D      0(R1), F4  
      S.D      8(R1), F8  
      ADDI     R1, R1, 32  
      SD       -16(R1), F12  
      BNE     R1, R2, Loop  
      SD       -8(R1), F16
```

□ Any drawbacks?

- loop unrolling increases code size
- more registers needed

Multiple issue architectures

269 How to get $CPI < 1$?

- Superscalar: multiple instructions issued per cycle
 - ▣ Statically scheduled
 - ▣ Dynamically scheduled (see previous lecture)
- VLIW ?
 - ▣ single instruction issue, but multiple operations per instruction
- SIMD / Vector ?
 - ▣ single instruction issue, single operation, but multiple data sets per operation
- Multi-processor ?

Instruction Parallel (ILP) Processors

270

The name ILP is used for:

□ **Multiple-Issue Processors**

- **Superscalar**: varying no. instructions/cycle (0 to 8), scheduled by HW (dynamic issue capability)

 - IBM PowerPC, Sun UltraSparc, DEC Alpha, Pentium III/4, etc.



- **VLIW** (very long instr. word): fixed number of instructions (4-16) scheduled by the compiler (static issue capability)

 - Intel Architecture-64 (IA-64, Itanium), TriMedia, TI C6x

□ **(Super-) pipelined processors**

- Anticipated success of multiple instructions led to Instructions Per Cycle (IPC) metric instead of CPI

Week 14

271

SMT

Simultaneously Multi-Threading

Lecture overview

272

- How to achieve speedup
- Simultaneous Multithreading
- Examples
 - ▣ Power 4 vs. Power 5
- Head to Head: VLIW vs. Superscalar vs. SMT
- Conclusion

- Book: sections 3.4 – 3.6

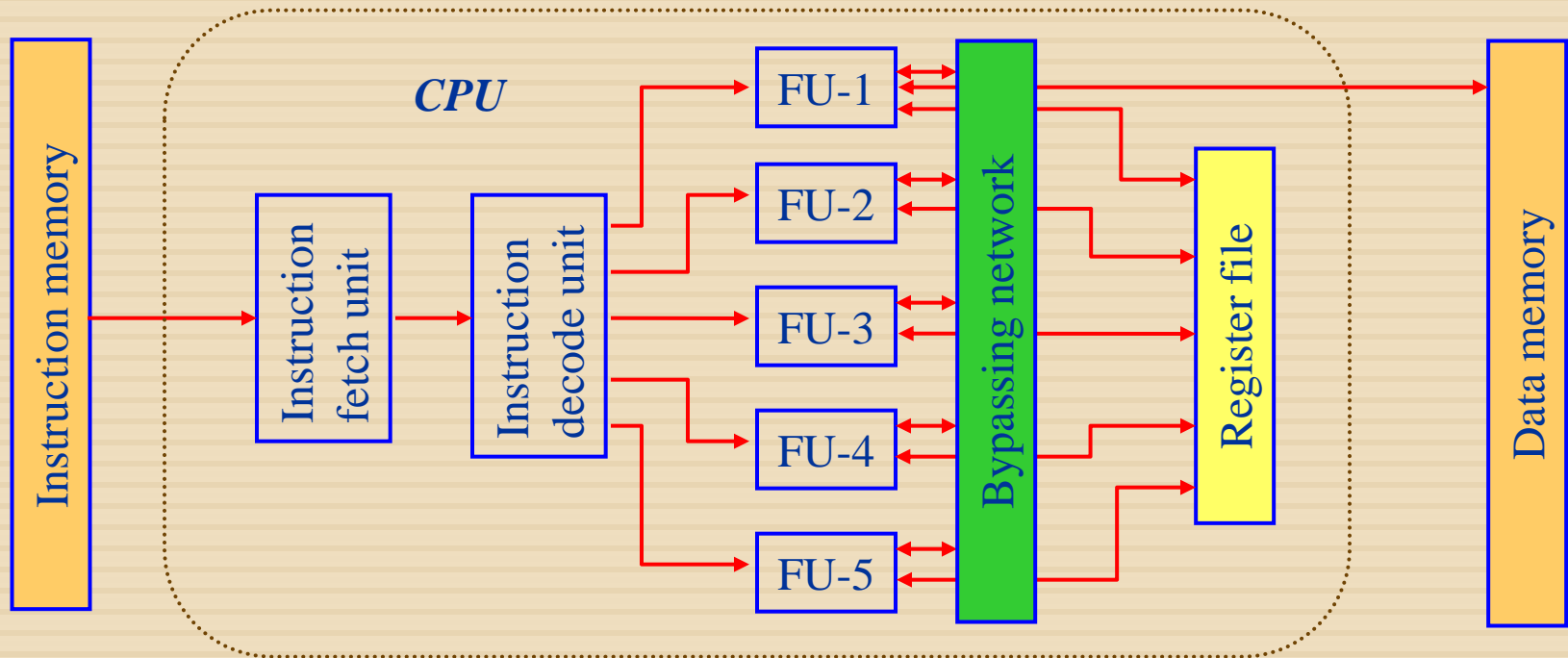
5 ways to speed up: parallelism

273

- TLP: task level parallelism
 - ▣ multiple threads of control
- ILP: instruction level parallelism
 - ▣ issue (and execute) multiple instructions per cycle
 - ▣ Superscalar approach
- OLP: operation level parallelism (usually also called ILP)
 - ▣ multiple operations per instruction
 - ▣ VLIW approach
- DLP: data level parallelism
 - ▣ multiple operands per operations
 - ▣ SIMD / sub-word parallel / vector computing approach
- Pipelining: overlapped execution
 - ▣ every architecture following RISC principles

General organization of an ILP / OLP architecture

274



ILP / OLP limits

275

- ILP and OLP everywhere, but limited, due to:
 - ▣ true dependences
 - ▣ branch miss predictions
 - ▣ cache misses
 - ▣ architecture complexity
 - bypass network complexity quadratic in number of FUs
 - register file: too many ports needed
 - issue, renaming and select logic (not for VLIW)

Should we go Multi-Processing?

276

In the past MP hindered by:

- Increase in single thread performance 50% per year
 - ▣ 30 % by faster transistors (silicon improvements)
 - ▣ deeper pipelining
 - ▣ multi-issue: ILP
 - ▣ better compilers
- Few highly task-level parallel applications
- Programmers are not educated in 'parallelism'

Should we go Multi-Processing?

277

- Today:
 - ▣ Diminishing returns for exploiting ILP
 - ▣ Power issues
 - ▣ Wiring issues (faster transistors do not help that much)
 - ▣ More parallel applications
 - ▣ Multi-core architectures hit the market

- In chapter 4 we go multi-processor, first we look at an alternative

New Approach: Multi-Threaded

278

- Multithreading: multiple threads share the functional units of **1** processor
 - ▣ duplicate independent state of each thread e.g., a separate copy of register file, a separate PC
 - ▣ HW for fast thread switch; much faster than full process switch $\approx 100\text{s}$ to 1000s of clocks
- When to switch?
 - ▣ Next instruction next thread (**fine grain**), or
 - ▣ When a thread is stalled, perhaps for a cache miss, another thread can be executed (**coarse grain**)

Fine-Grained Multithreading

279

- Switches between threads on **each instruction**, causing the execution of multiples threads to be interleaved
- Usually done in a **round-robin** fashion, skipping any stalled threads
- CPU must be able to switch threads every clock
- Advantage: it can hide both short and long stalls, since instructions from other threads executed when one thread stalls
- Disadvantage: may slow down execution of individual threads
- Used in e.g. Sun's Niagara

Course-Grained Multithreading

280

- Switches threads only on costly stalls, such as L2 cache misses
- Advantages
 - ▣ Relieves need to have very fast thread-switching
 - ▣ Doesn't slow down thread, since instructions from other threads issued only when the thread encounters a costly stall
- Disadvantage: hard to overcome throughput losses from shorter stalls, due to pipeline start-up costs
 - ▣ Since CPU issues instructions from 1 thread, when a stall occurs, the pipeline must be emptied or frozen
 - ▣ New thread must fill pipeline before instructions can complete
- Because of this start-up overhead, coarse-grained multithreading is better for reducing penalty of high cost stalls, where pipeline refill \ll stall time
- Used in e.g. IBM AS/400

Simultaneous Multi-threading ...

One thread, 8 units

Cycle M M FX FX FP FP BR CC

1	█							█
2	█	█					█	
3				█	█			
4								
5								
6								
7	█			█		█		
8		█			█			
9				█				

Two threads, 8 units

Cycle M M FX FX FP FP BR CC

1	█	█	█					█
2	█	█	█			█	█	
3	█			█	█			
4	█	█				█		
5		█						█
6								
7	█		█	█	█	█		
8		█		█	█	█		
9	█	█		█		█		

M = Load/Store, FX = Fixed Point, FP = Floating Point, BR = Branch, CC = Completion Codes

Simultaneous Multithreading (SMT)

282

- SMT: dynamically scheduled processors already has many HW mechanisms to support multithreading:
 - ▣ Large set of virtual registers that can be used to hold the register sets of independent threads
 - ▣ Register renaming provides unique register identifiers, so instructions from multiple threads can be mixed in datapath without confusing sources and destinations across threads
 - ▣ Out-of-order completion allows the threads to execute out of order, and get better utilization of the HW
- Just adding a per thread renaming table and keeping separate PCs

Week 15

283

Multi Processing -1

Flynn's Taxonomy

□ SISD (Single Instruction, Single Data)

284

□ Uniprocessors

□ SIMD (Single Instruction, Multiple Data)

□ Vector architectures also belong to this class

- Multimedia extensions (MMX, SSE, VIS, AltiVec, ...)

□ Examples: Illiac-IV, CM-2, MasPar MP-1 /2, Xetal, IMAP, Imagine, GPUs,

□ MISD (Multiple Instruction, Single Data)

□ Systolic arrays / stream based processing

□ MIMD (Multiple Instruction, Multiple Data)

□ Examples: Sun Enterprise 5000, Cray T3D/T3E, SGI Origin

- Flexible

□ Most widely used

Compare the earlier presented classification!!

TCV / 3/2025

Why parallel processing

285

- Performance drive
- Diminishing returns for exploiting ILP and OLP
- Multiple processors fit easily on a chip
- Cost effective (just connect existing processors or processor cores)
- Low power: parallelism may allow lowering V_{dd}

However:

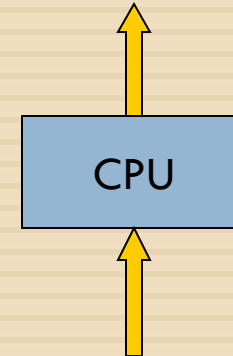
- Parallel programming is hard

Low power through parallelism

286

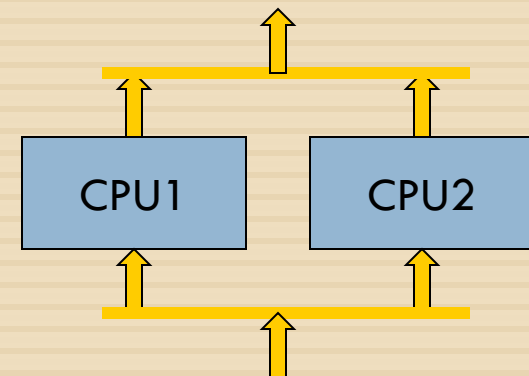
□ Sequential Processor

- Switching capacitance C
- Frequency f
- Voltage V
- $P_1 = \alpha f C V^2$



□ Parallel Processor (two times the number of units)

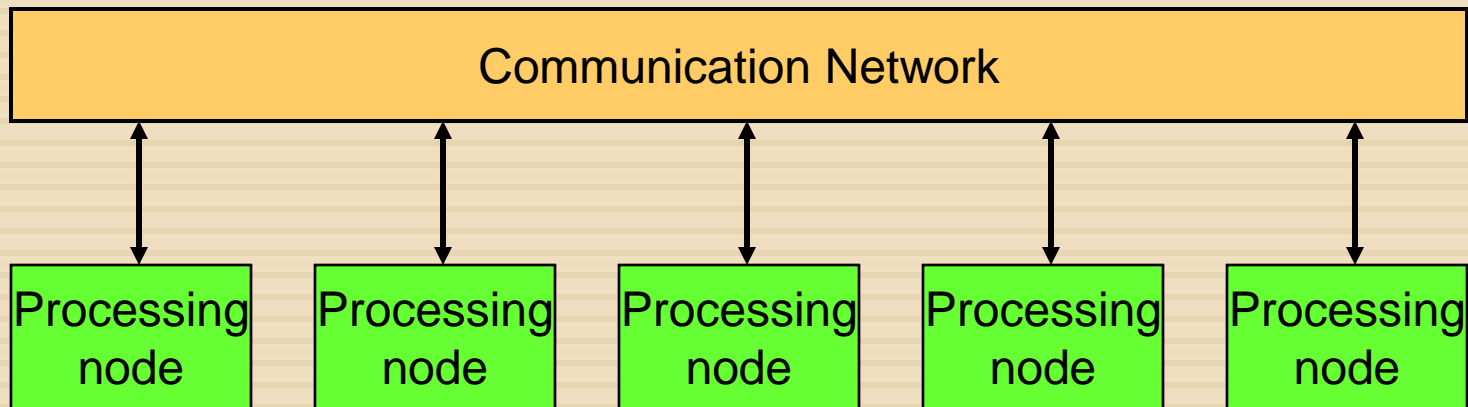
- Switching capacitance $2C$
- Frequency $f/2$
- Voltage $V' < V$
- $P_2 = \alpha f/2 \cdot 2C V'^2 = \alpha f C V'^2 < P_1$



Parallel Architecture

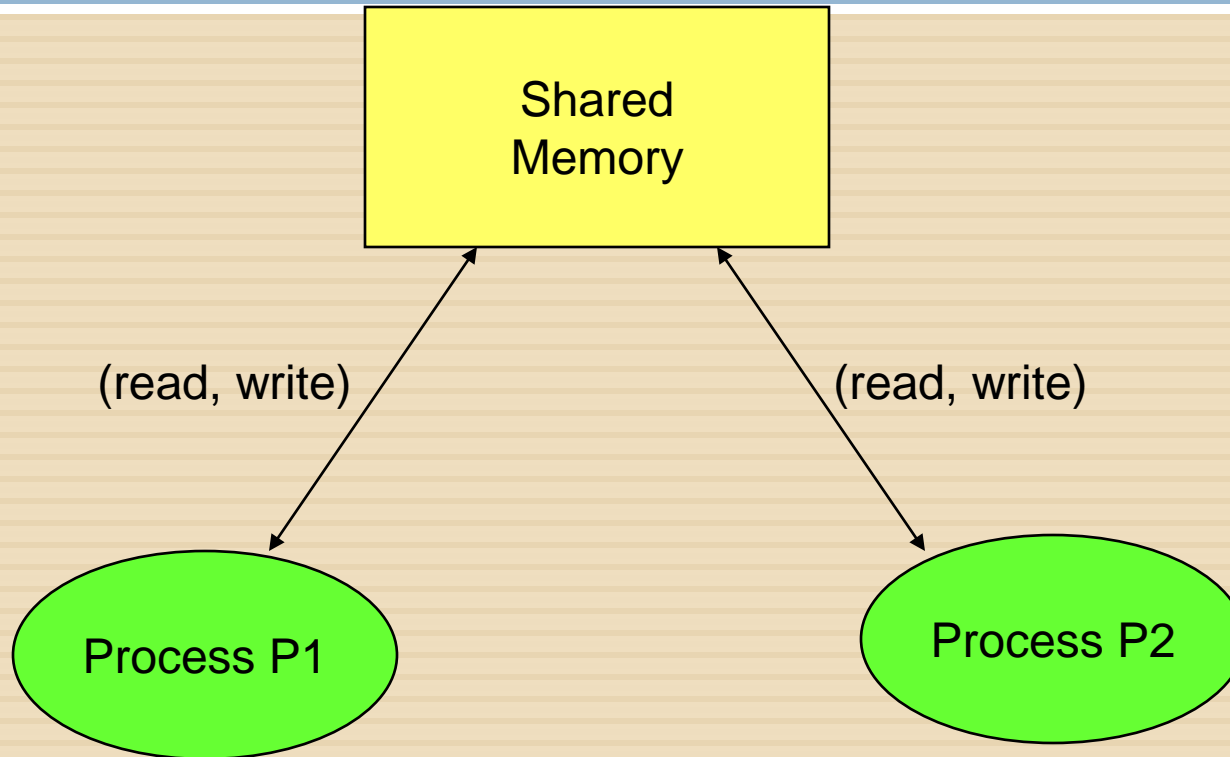
287

- Parallel Architecture extends traditional computer architecture with a **communication network**
 - ▣ abstractions (HW/SW interface)
 - ▣ organizational structure to realize abstraction efficiently



Communication models: Shared Memory

288



- Coherence problem
- Memory consistency issue
- Synchronization problem

Communication models: **Shared memory**

289

- Shared address space
- Communication primitives:
 - load, store, atomic swap

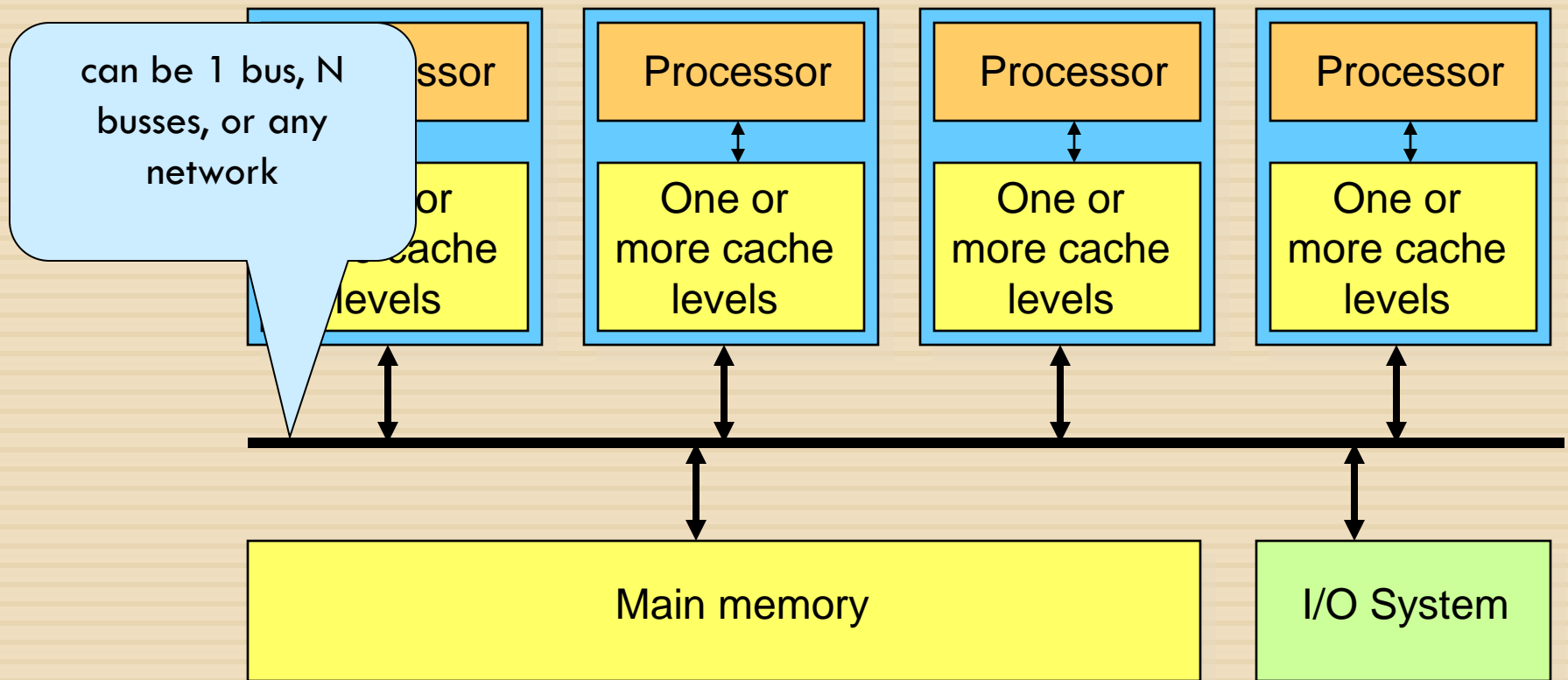
Two varieties:

- Physically shared => **Symmetric Multi-Processors (SMP)**
 - usually combined with local caching
- Physically distributed => **Distributed Shared Memory (DSM)**

SMP: Symmetric Multi-Processor

290

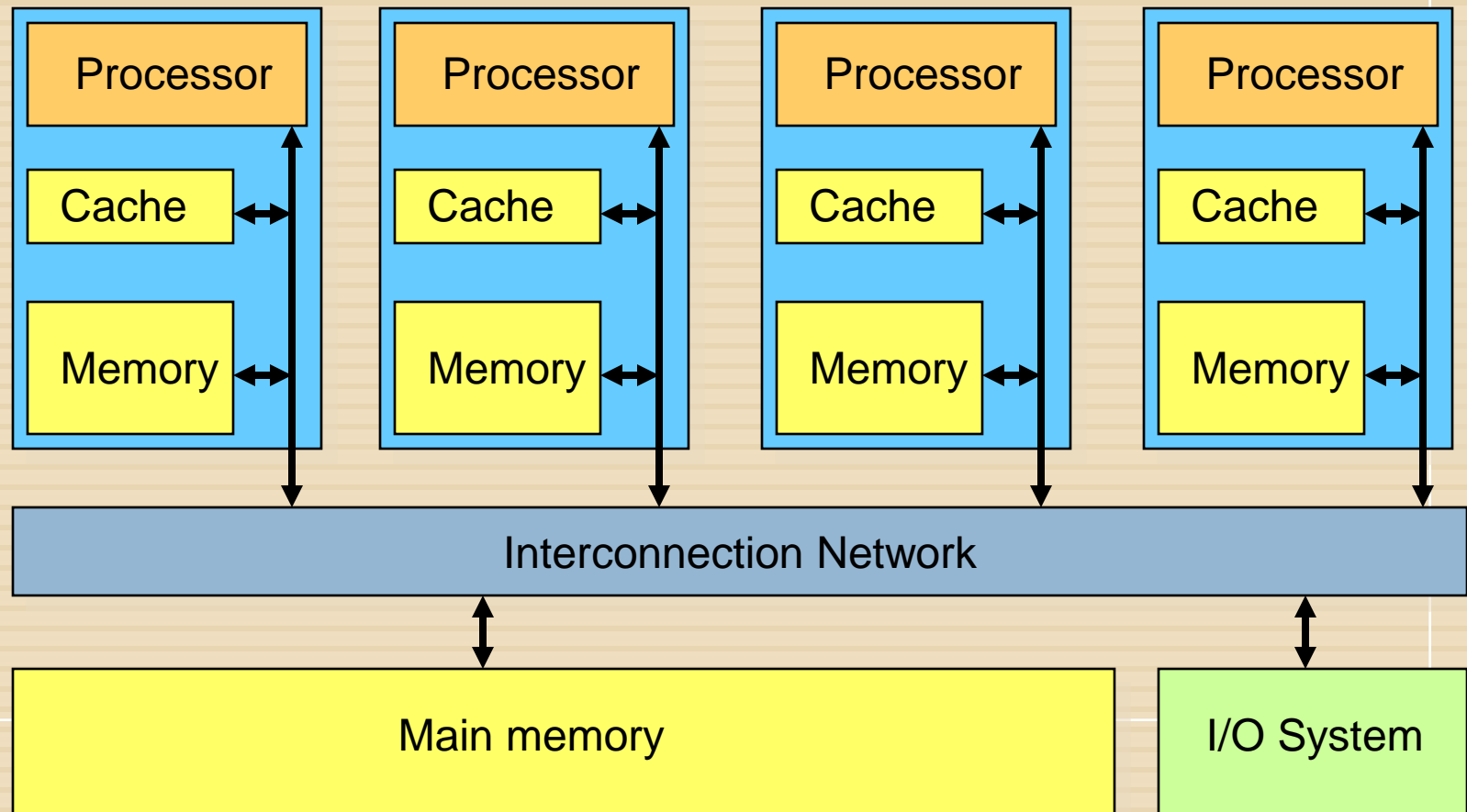
- Memory: centralized with uniform access time (**UMA**) and bus interconnect, I/O
- Examples: Sun Enterprise 6000, SGI Challenge, Intel



DSM: Distributed Shared Memory

291

- Nonuniform access time (**NUMA**) and scalable interconnect (distributed memory)



Shared Address Model Summary

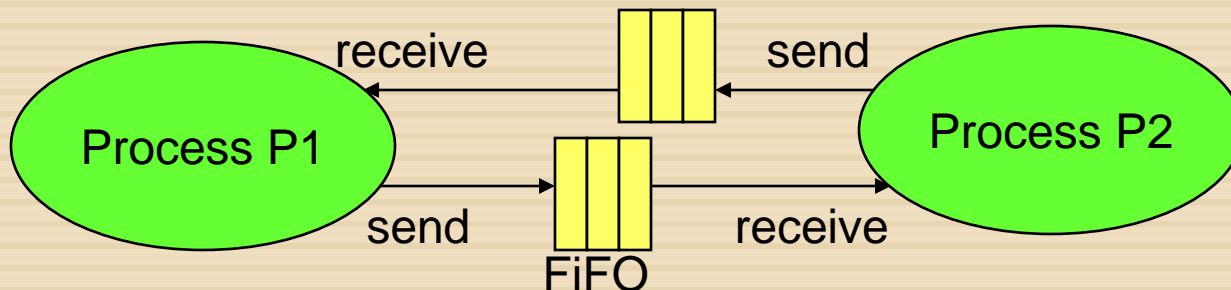
292

- Each processor can name every physical location in the machine
- Each process can name all data it shares with other processes
- Data transfer via load and store
- Data size: byte, word, ... or cache blocks
- Memory hierarchy model applies:
 - ▣ communication moves data to local proc. cache

Communication models: **Message Passing**

293

- Communication primitives
 - e.g., send, receive library calls
 - standard MPI: Message Passing Interface
 - www.mpi-forum.org
- *Note that MP can be build on top of SM and vice versa !*

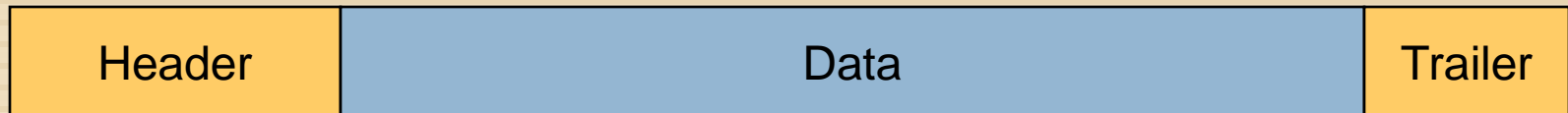


Message Passing Model

294

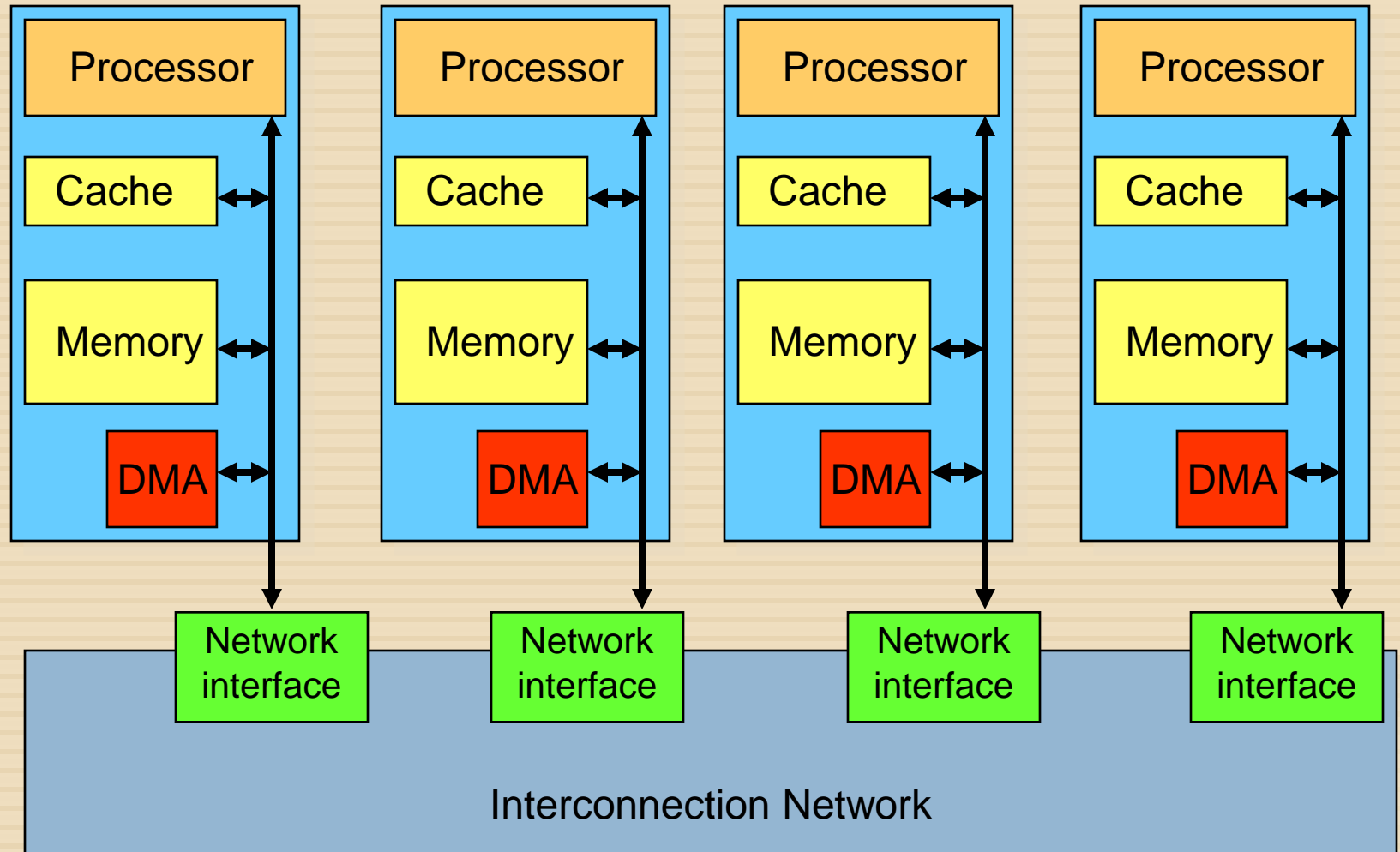
- Explicit message send and receive operations
- Send specifies local buffer + receiving process on remote computer
- Receive specifies sending process on remote computer + local buffer to place data
- Typically blocking communication, but may use DMA

Message structure



Message passing communication

295



Communication Models: Comparison

296

□ Shared-Memory

- Compatibility with well-understood (language) mechanisms
- Ease of programming for complex or dynamic communications patterns
- Shared-memory applications; sharing of large data structures
- Efficient for small items
- Supports hardware caching

□ Messaging Passing

- Simpler hardware
- Explicit communication
- Implicit synchronization (with any communication)

Network: Performance metrics

297

□ Network Bandwidth

- Need high bandwidth in communication
- How does it scale with number of nodes?

□ Communication Latency

- Affects performance, since processor may have to wait
- Affects ease of programming, since it requires more thought to overlap communication and computation

How can a mechanism help hide latency?

- overlap message send with computation,
- prefetch data,
- switch to other task or thread

Week 16

298

Multi Processing -2

Challenges of parallel processing

299

Q1: can we get linear speedup

Suppose we want speedup 80 with 100 processors. What fraction of the original computation can be sequential (i.e. non-parallel)?

Answer: $f_{seq} = 0.25\%$

Q2: how important is communication latency

Suppose 0.2 % of all accesses are remote, and require 100 cycles on a processor with base CPI = 0.5

What's the communication impact?

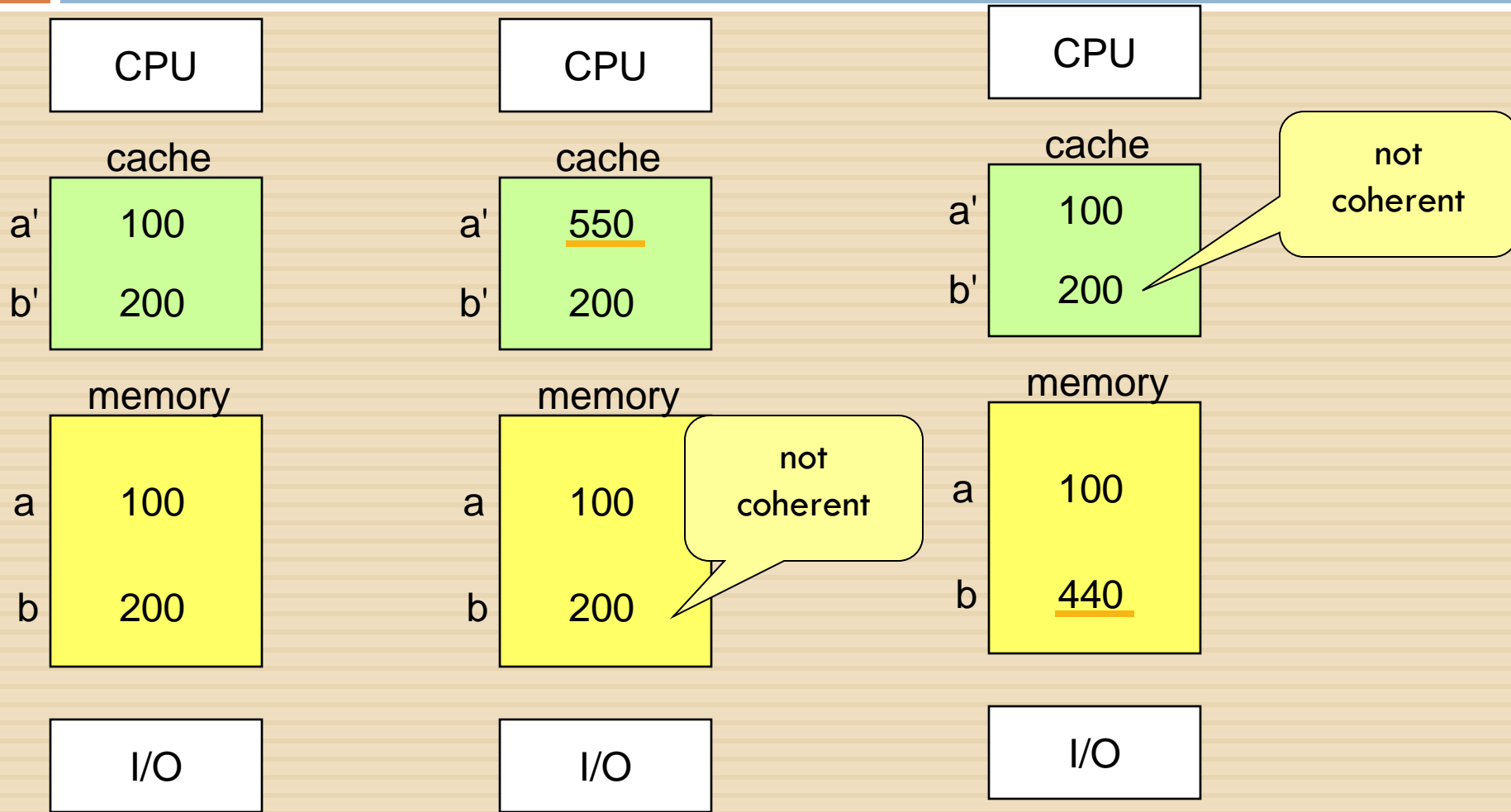
Three fundamental issues for shared memory multiprocessors

300

- **Coherence,**
about: *Do I see the most recent data?*
- **Consistency,**
about: *When do I see a written value?*
 - e.g. do different processors see writes at the same time (w.r.t. other memory accesses)?
- **Synchronization**
How to synchronize processes?
 - how to protect access to shared data?

Coherence problem, in single CPU system

301



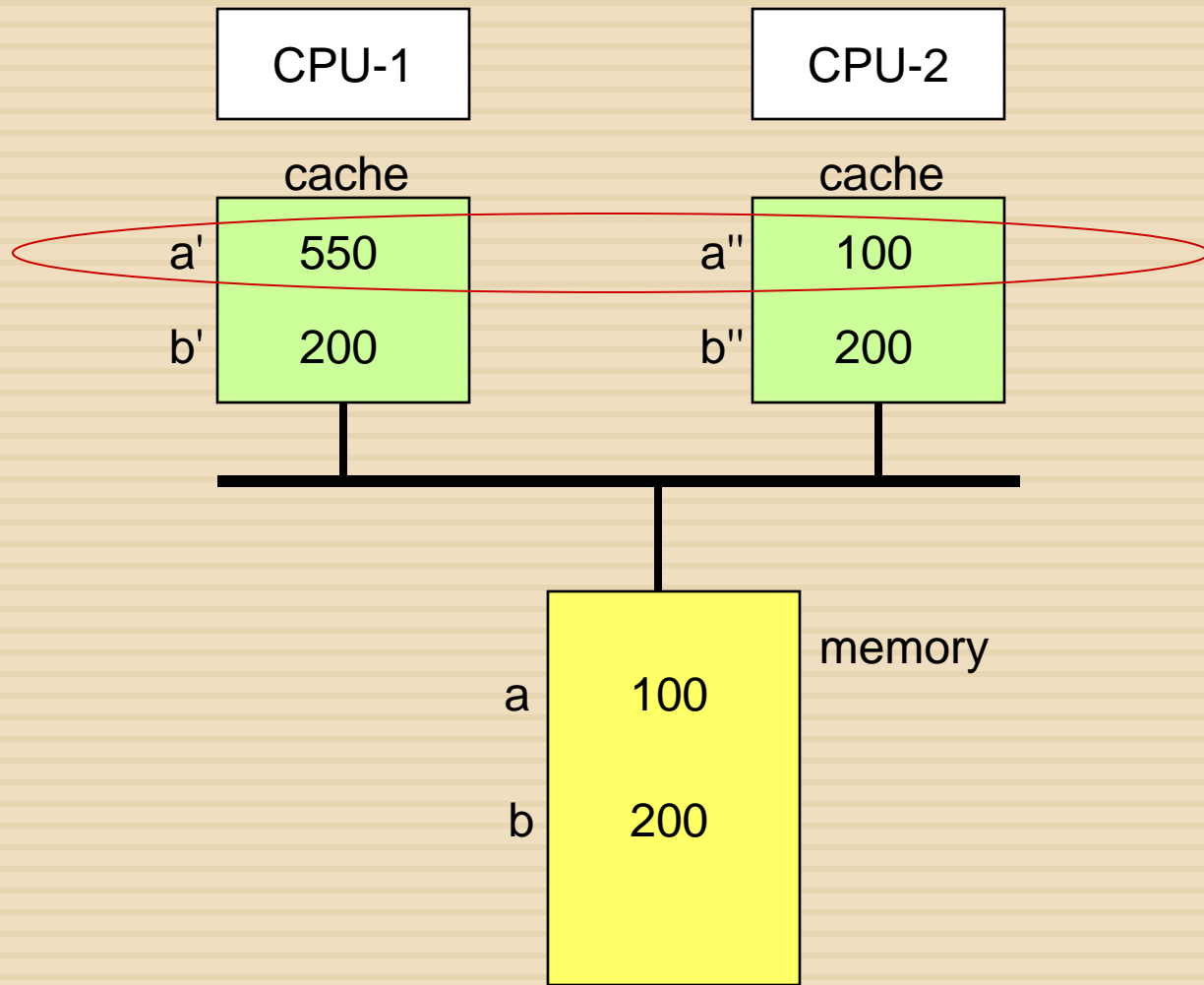
1) CPU writes to a

2) I/O writes b

UGV II 6/2025

Coherence problem, in Multi-Proc system

302



What Does Coherency Mean?

303

Informally:

- “Any read must return the most recent write (to the same address)”
- Too strict and too difficult to implement

Better:

- A write followed by a read by the same processor P with no writes in between returns the value written
- “Any write must eventually be seen by a read”
 - If P writes to X and P' reads X then P' will see the value written by P if the read and write are *sufficiently separated in time*
- Writes to the same location by different processors are seen *in the same order* by all processors ("**serialization**")
 - Suppose P1 writes location X, followed by P2 writing also to X. If no serialization, then some processors would read value of P1 and others of P2. Serialization guarantees that all processors read the same sequence.

Two rules to ensure coherency

304

- “If P1 writes x and P2 reads it, P1’s write will be seen by P2 if the read and write are sufficiently far apart”
- Writes to a single location are serialized:
 - seen in one order
 - ▣ Latest write will be seen
 - ▣ Otherwise could see writes in illogical order (could see older value after a newer value)

Potential HW Coherency Solutions

305

□ **Snooping Solution (Snoopy Bus):**

- Send all requests for data to all processors (or local caches)
- Processors snoop to see if they have a copy and respond accordingly
- Requires broadcast, since caching information is at processors
- Works well with bus (natural broadcast medium)
- Dominates for small scale machines (most of the market)

□ **Directory-Based Schemes**

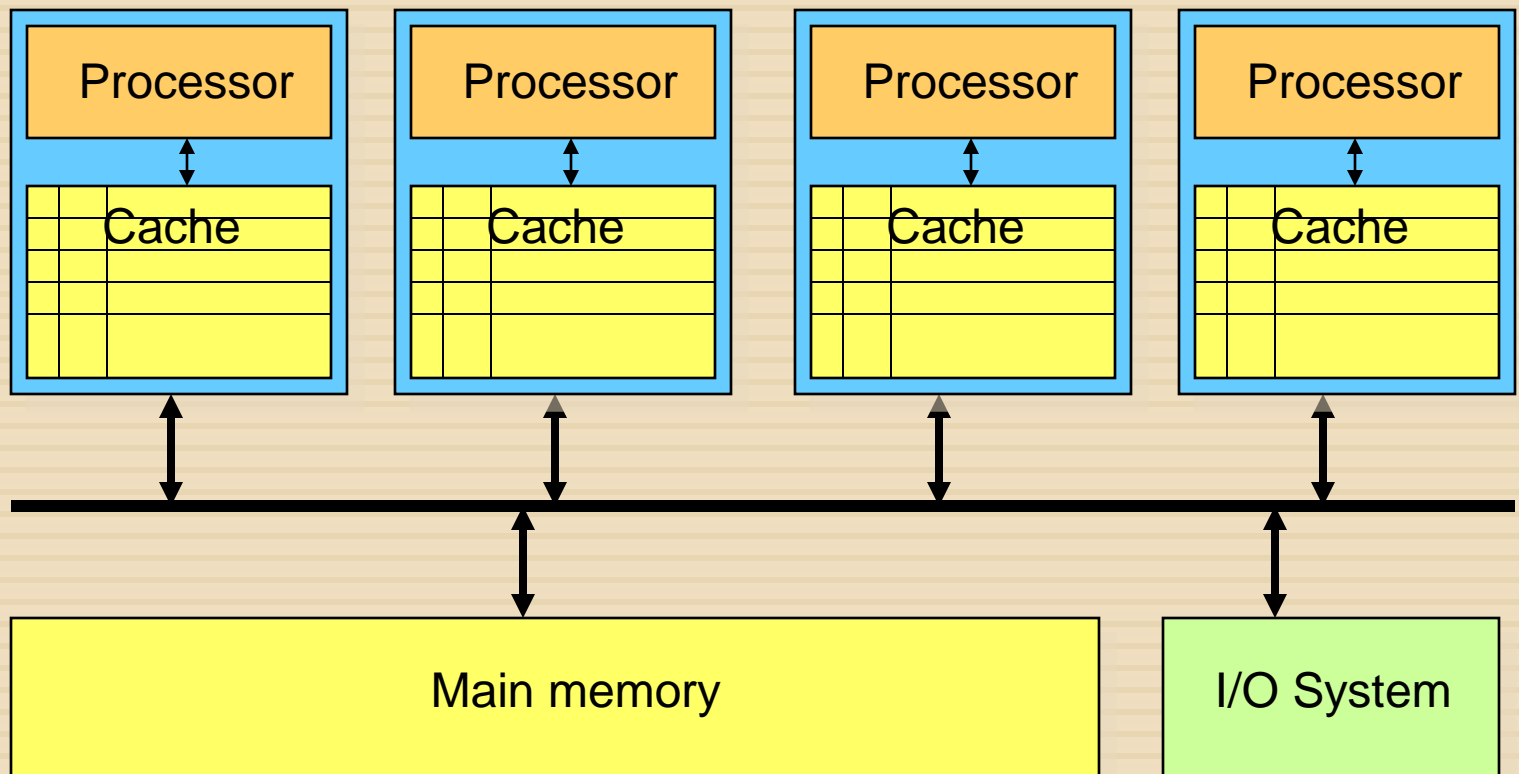
- Keep track of what is being shared in one centralized place
- Distributed memory => distributed directory for scalability (avoids bottlenecks, **hot spots**)
- Scales better than Snooping
- Actually existed BEFORE Snooping-based schemes

Example Snooping protocol

306

3 states for each cache line:

- invalid, shared (read only), modified (also called exclusive, you may write it)
- FSM per cache, gets requests from processor and bus



Snooping Protocol 1: Write Invalidate

307

- Get exclusive access to a cache block (invalidate all other copies) before writing it
- When processor reads an invalid cache block it is forced to fetch a new copy
- If two processors attempt to write simultaneously, one of them is first (having a bus helps). The other one must obtain a new copy, thereby enforcing serialization

Processor activity	Bus activity	Cache CPU A	Cache CPU B	Memory addr. X
				0
CPU A reads X	Cache miss for X	0		0
CPU B reads X	Cache miss for X	0	0	0
CPU A writes 1 to X	Invalidation for X	1	<i>invalidated</i>	0
CPU B reads X	Cache miss for X	1	1	1

Example: address X in memory initially contains value '0'

Basics of Write Invalidate

□ Use the bus to perform invalidates

308

To perform an invalidate, acquire bus access and broadcast the address to be invalidated

- all processors snoop the bus, listening to addresses
- if the address is in my cache, invalidate my copy

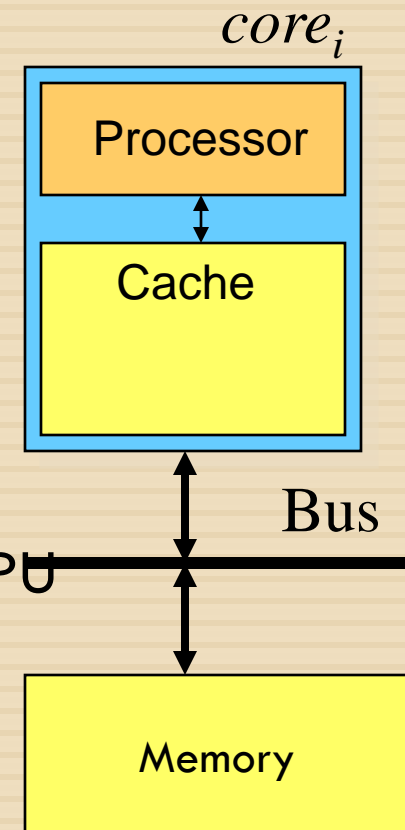
□ Serialization of bus access enforces write serialization

□ Where is the most recent value?

- Easy for write-through caches: in the memory
- For write-back caches, again use snooping

□ Can use cache tags to implement snooping

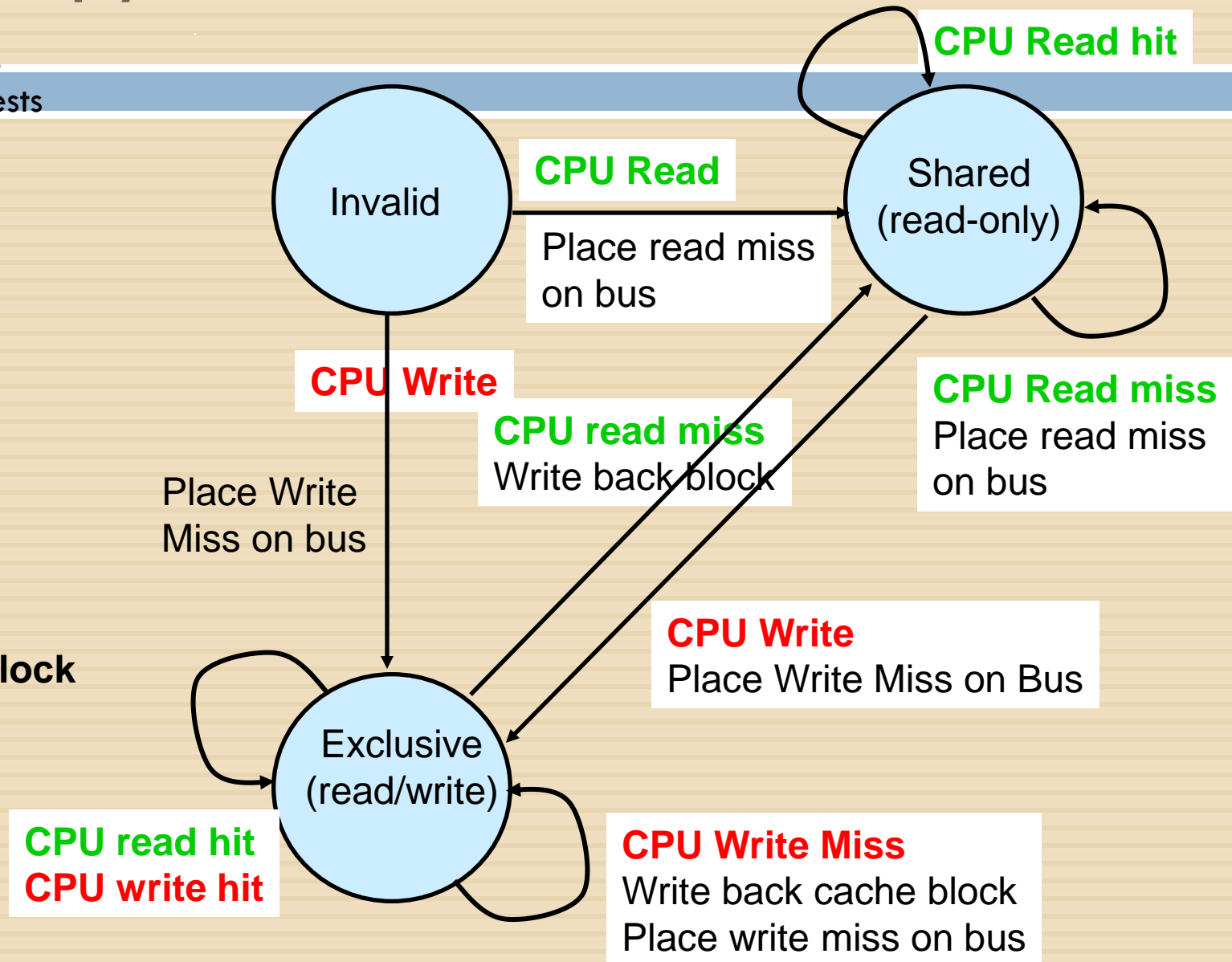
- Might interfere with cache accesses coming from CPU
- Duplicate tags, or employ multilevel cache with inclusion



Snoopy-Cache State Machine-I

State machine
for CPU requests
for each
cache block

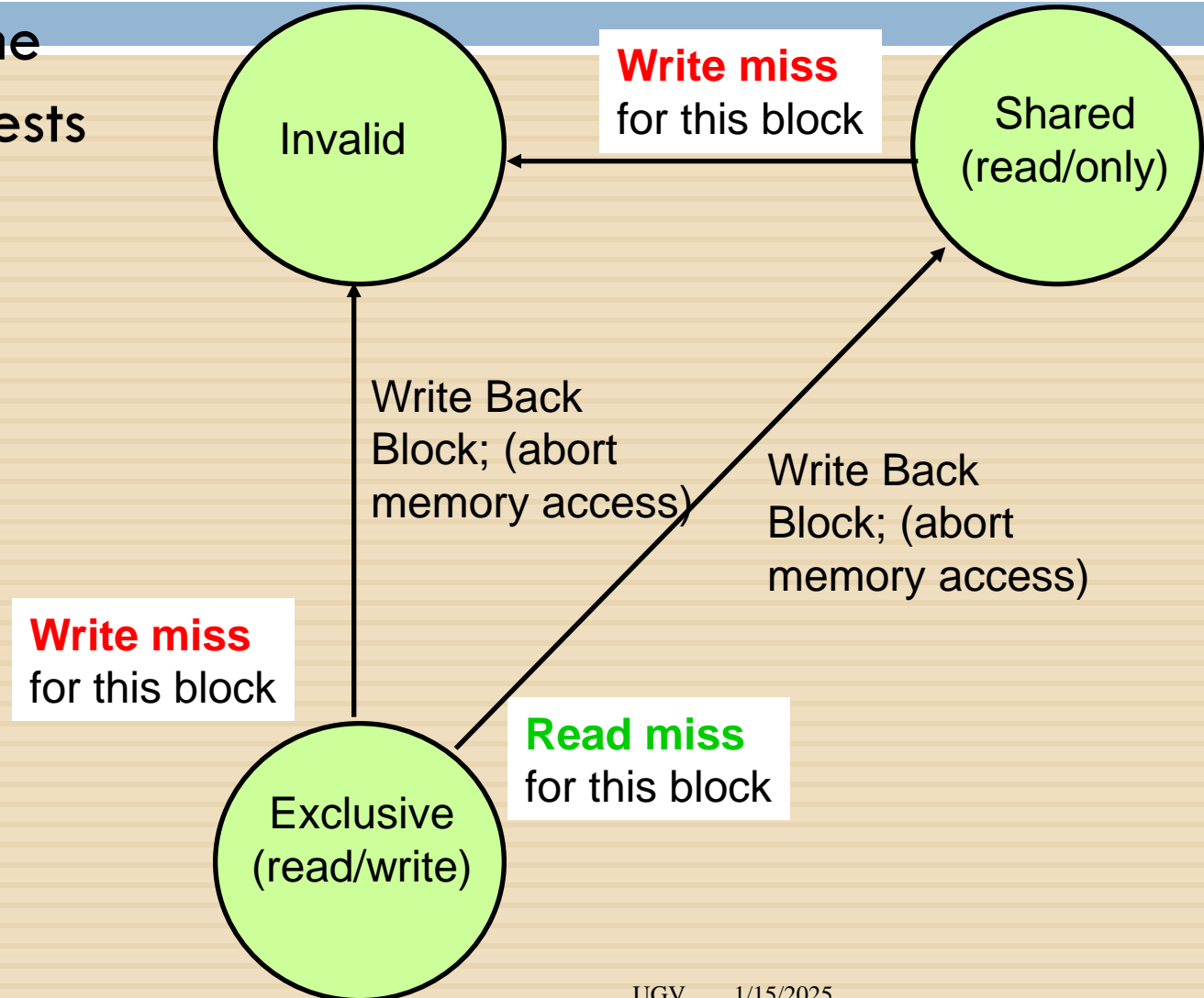
Cache Block
State



Snoopy-Cache State Machine-II

310

State machine
for bus requests
for each
cache block



Week 17

311

Multi Processing -3

Homogeneous or Heterogeneous

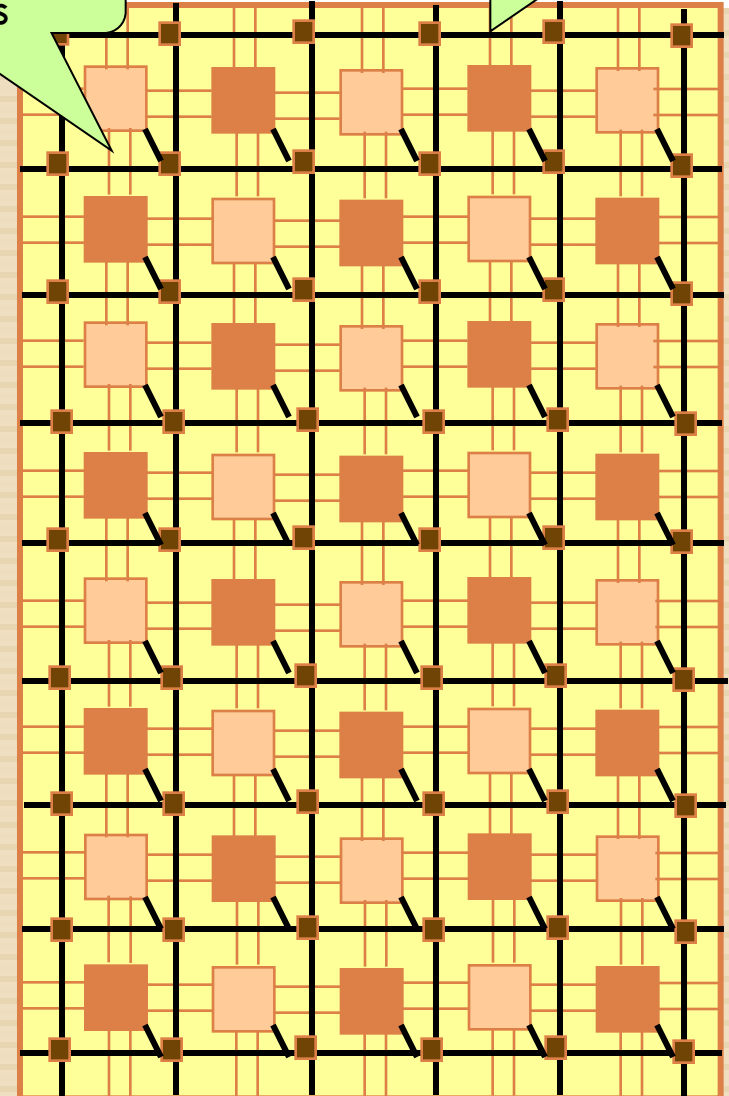
312

Homogenous:

- ▣ replication effect
 - easy to desing
 - fault tolerance can be built-in
 - process migration possible
- ▣ solve realization issues once and for all (i.e. highly tuning of nodes and network)
- ▣ less flexible nodes
- ▣ Reasoning: *future chips are memory dominated any way, so adding some logic to make all nodes equal does not matter*

Local inter-node connections

Global Network-on-Chip

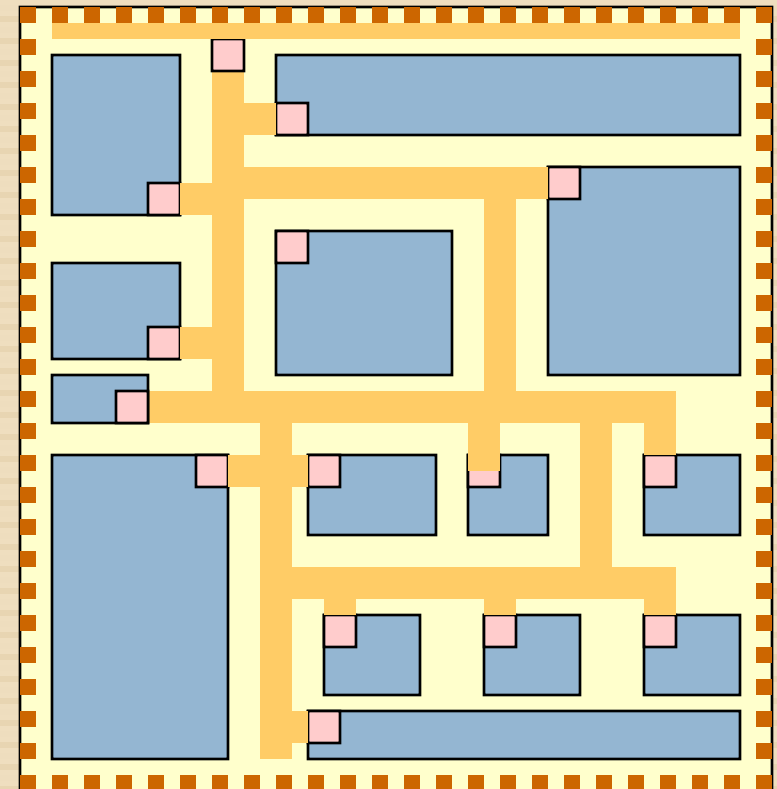


Homogeneous or Heterogeneous

313

□ Heterogeneous

- better fit to application domain
- smaller increments
- more costly design
- no replication advantage

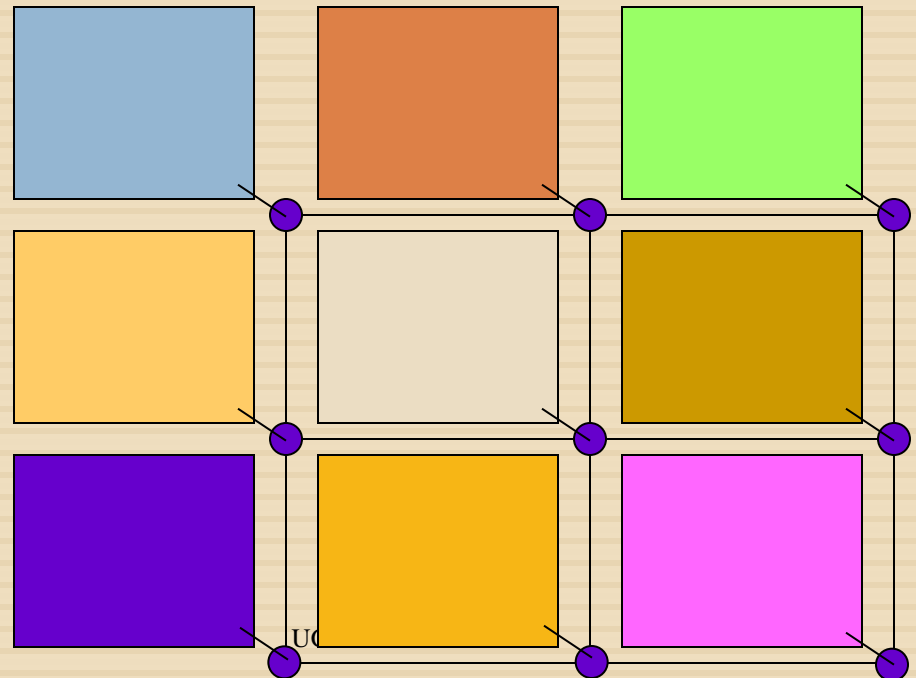


Homogeneous or Heterogeneous

314

□ Middle of the road approach

- Flexible tiles
- Fixed tile structure at top level



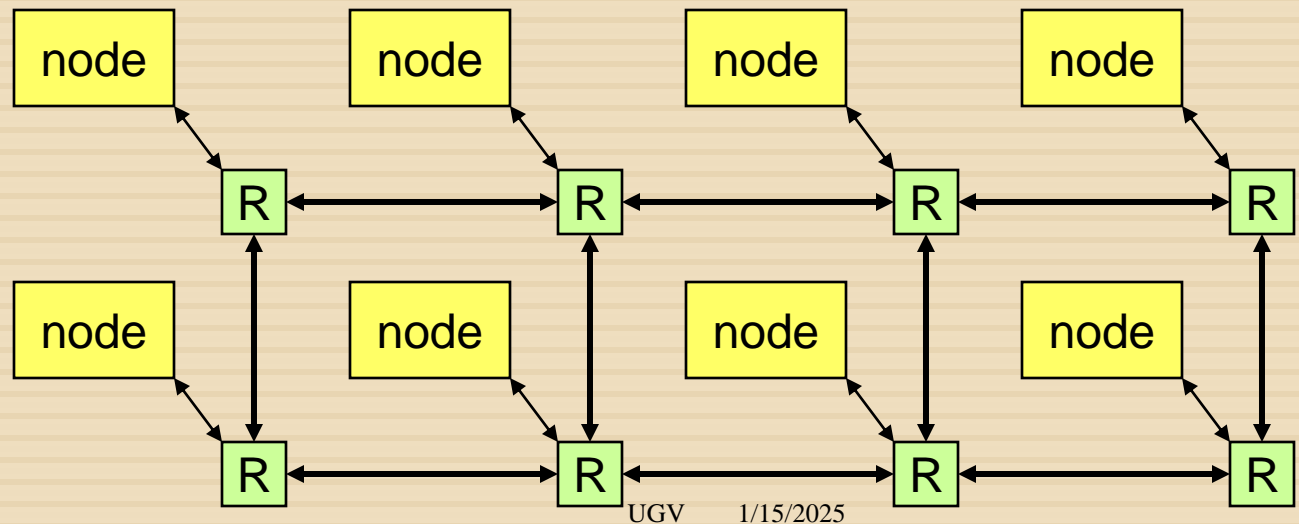
Bus (shared) or Network (switched)

315

Network:

- ▣ claimed to be more scalable
- ▣ no bus arbitration
- ▣ point-to-point connections
- ▣ but router overhead

Example:
NoC with 2x4 mesh
routing network

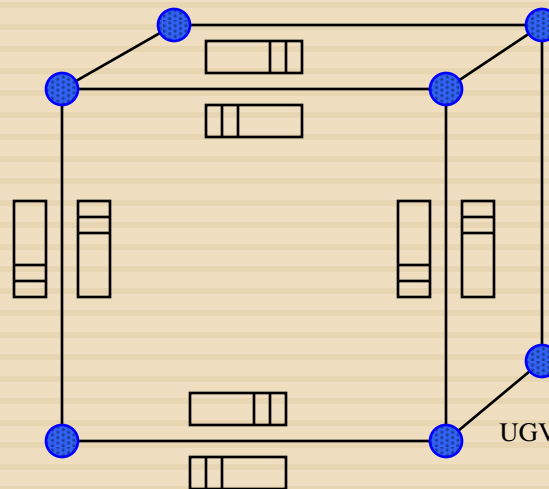


Historical Perspective

316

Early machines were:

- ▣ Collection of microprocessors.
- ▣ Communication was performed using bi-directional queues between nearest neighbors.
- ▣ Messages were forwarded by processors on path
 - ▣ “Store and forward” networking
- ▣ There was a strong emphasis on topology in algorithms, in order to minimize the number of hops => minimize time



UGV

1/15/2025

Design Characteristics of a Network

317

- Topology (how things are connected):
 - Crossbar, ring, 2-D and 3-D meshes or torus, hypercube, tree, butterfly, perfect shuffle,
- Routing algorithm (path used):
 - Example in 2D torus: all east-west then all north-south (avoids deadlock)
- Switching strategy:
 - Circuit switching: full path reserved for entire message, like the telephone.
 - Packet switching: message broken into separately-routed packets, like the post office.
- Flow control and buffering (what if there is congestion):
 - Stall, store data temporarily in buffers
 - re-route data to other nodes
 - tell source node to temporarily halt, discard, etc.
- QoS guarantees
- Error handling
- etc, etc.

Switch / Network Topology

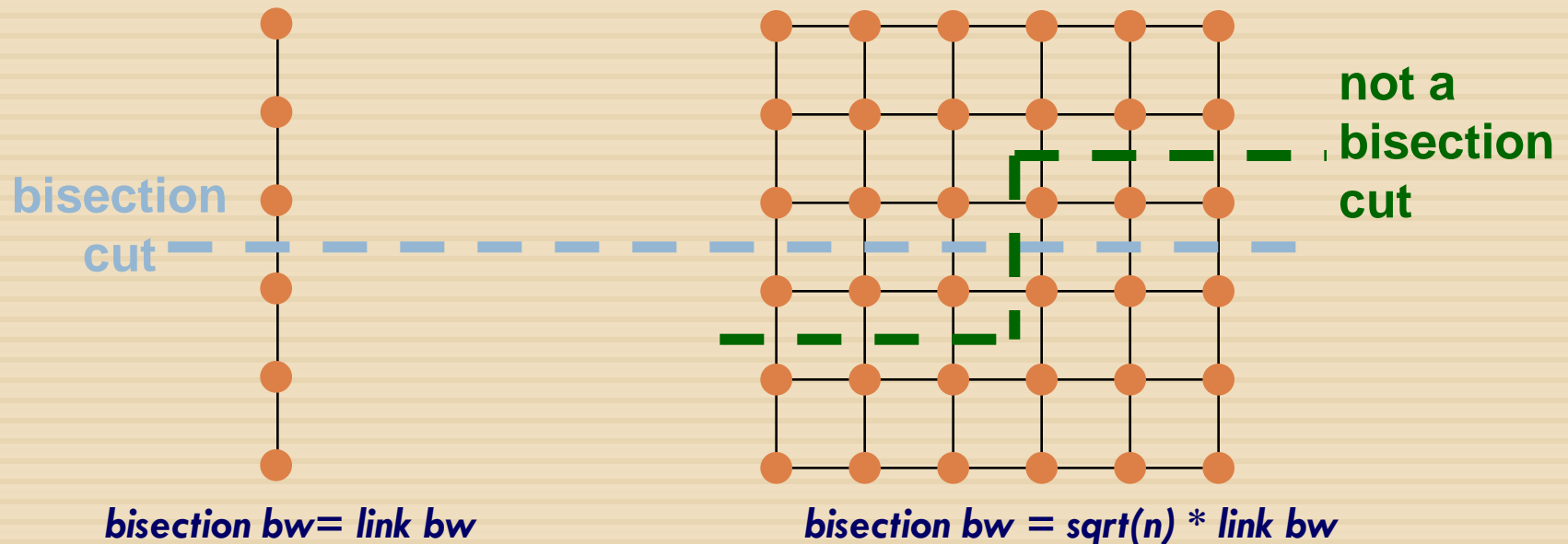
318

Topology determines:

- Degree: number of links from a node
- Diameter: max number of links crossed between nodes
- Average distance: number of links to random destination
- Bisection: minimum number of links that separate the network into two halves
- Bisection bandwidth = link bandwidth * bisection

Bisection Bandwidth

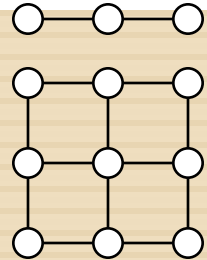
- **Bisection bandwidth:** bandwidth across smallest cut that divides network into two equal halves
- Bandwidth across “narrowest” part of the network



- Bisection bandwidth is important for algorithms in which all processors need to communicate with all others

Common Topologies

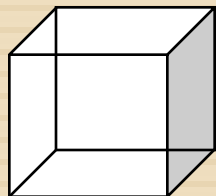
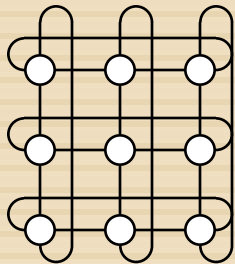
320



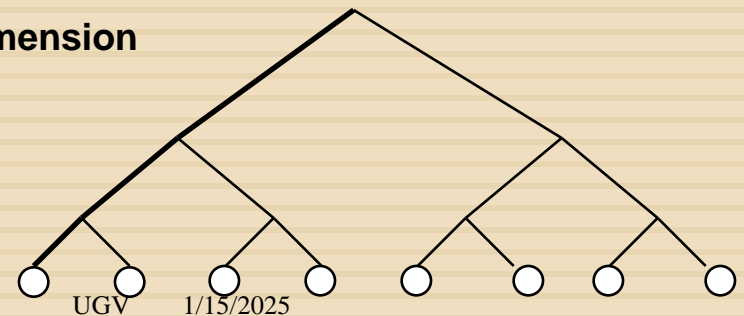
Type	Degree	Diameter	Ave Dist	Bisection
1D mesh	2	$N-1$	$N/3$	1
2D mesh	4	$2(N^{1/2} - 1)$	$2N^{1/2} / 3$	$N^{1/2}$
3D mesh	6	$3(N^{1/3} - 1)$	$3N^{1/3} / 3$	$N^{2/3}$
nD mesh	$2n$	$n(N^{1/n} - 1)$	$nN^{1/n} / 3$	$N^{(n-1)/n}$



Ring	2	$N/2$	$N/4$	2
2D torus	4	$N^{1/2}$	$N^{1/2} / 2$	$2N^{1/2}$
Hypercube	$\log_2 N$	$n = \log_2 N$	$n/2$	$N/2$
2D Tree	3	$2\log_2 N$	$\sim 2\log_2 N$	1
Crossbar	$N-1$	1	1	$N^2/2$



N = number of nodes, n = dimension



Linear and Ring Topologies

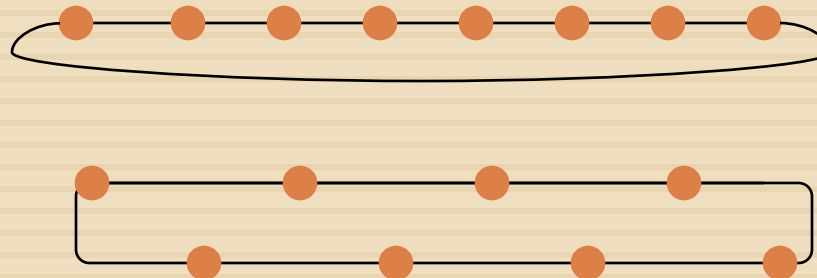
321



□ Linear array

- Diameter = $n-1$; average distance $\sim n/3$
- Bisection bandwidth = 1 (in units of link bandwidth)

□ Torus or Ring



- Diameter = $n/2$; average distance $\sim n/4$
- Bisection bandwidth = 2
- Natural for algorithms that work with 1D arrays

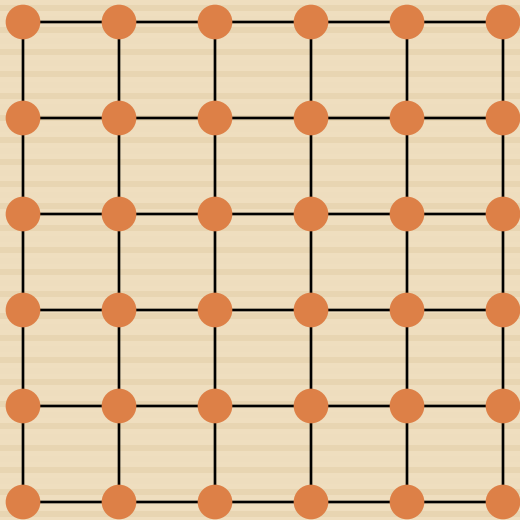
Meshes and Tori

322

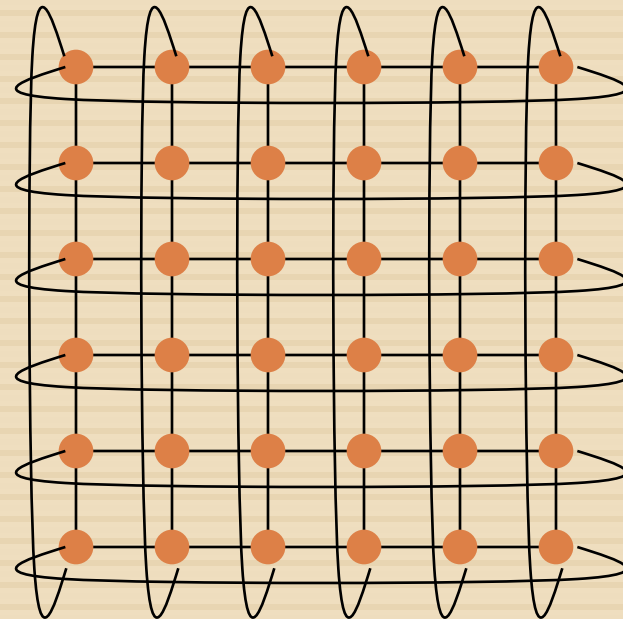
Two dimensional torus

Two dimensional mesh

- Diameter = $2 * (\text{sqrt}(n) - 1)$
- Bisection bandwidth = $\text{sqrt}(n)$



- Diameter = $\text{sqrt}(n)$
- Bisection bandwidth = $2 * \text{sqrt}(n)$



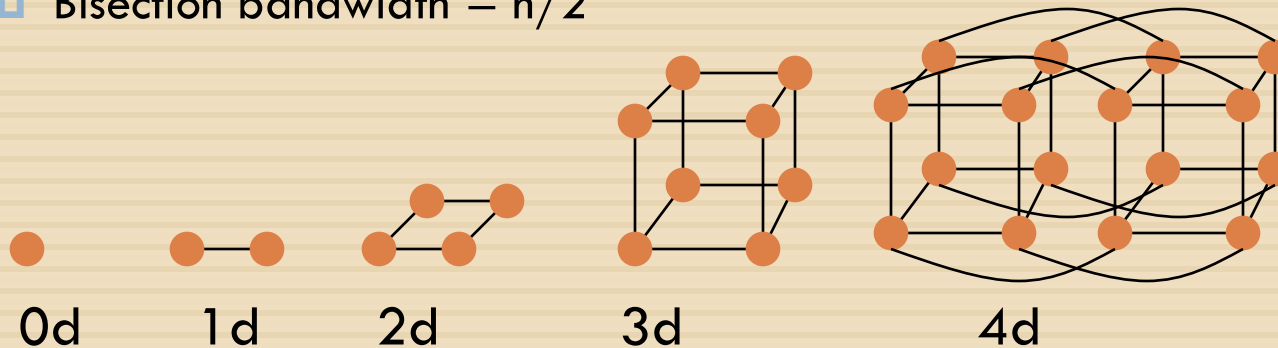
- Generalizes to higher dimensions
- Natural for algorithms that work with 2D and/or 3D arrays

Hypercubes

323

- Number of nodes $n = 2^d$ for dimension d

- Diameter = d
- Bisection bandwidth = $n/2$

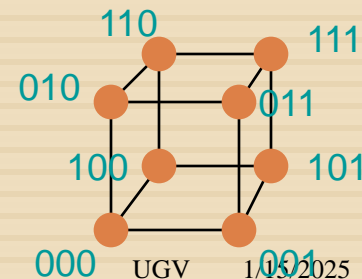


- Popular in early machines (Intel iPSC, NCUBE, CM)

- Lots of clever algorithms
- Extension: k -ary n -cubes

- Greyscale addressing:

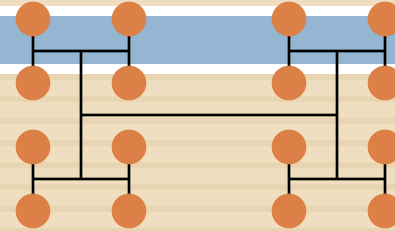
- Each node connected to others with 1 bit different



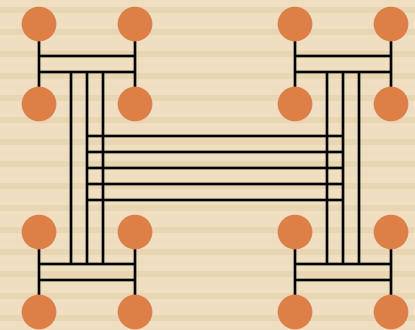
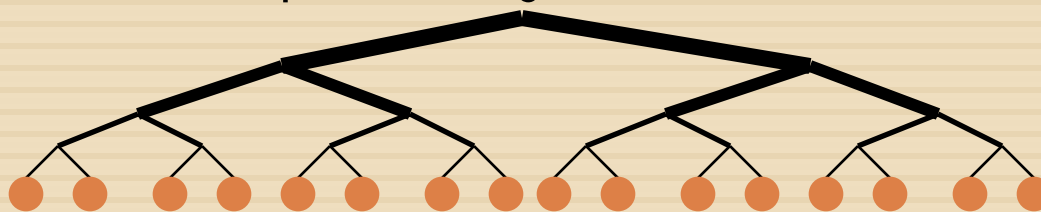
Trees

324

- Diameter = $\log n$.
- Bisection bandwidth = 1
- Easy layout as planar graph
- Many tree algorithms (e.g., summation)

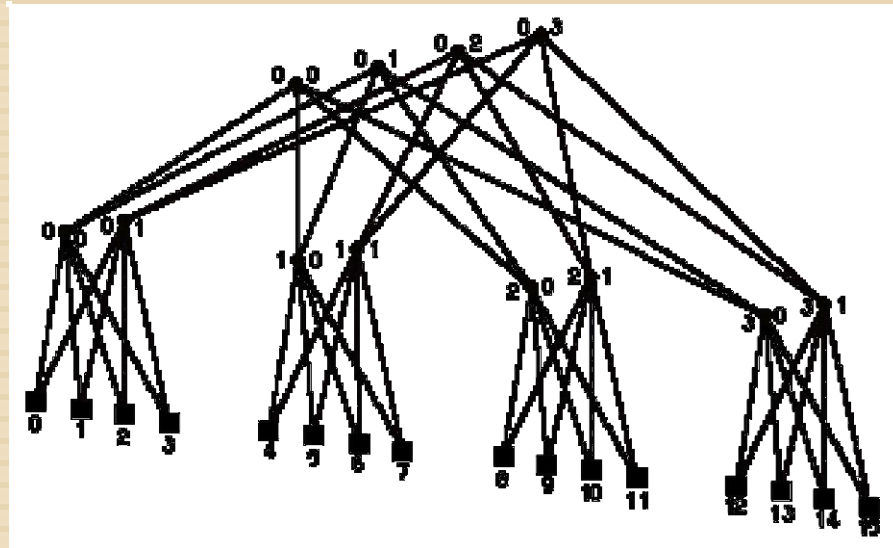


- **Fat trees** avoid bisection bandwidth problem:
 - More (or wider) links near top
 - Example: Thinking Machines CM-5



Fat Tree example

325

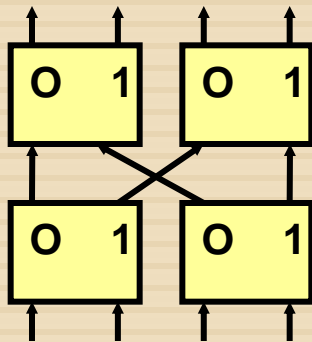


- A multistage fat tree (CM-5) avoids congestion at the root node
- Randomly assign packets to different paths on way up to spread the load
- Increase degree near root, decrease congestion

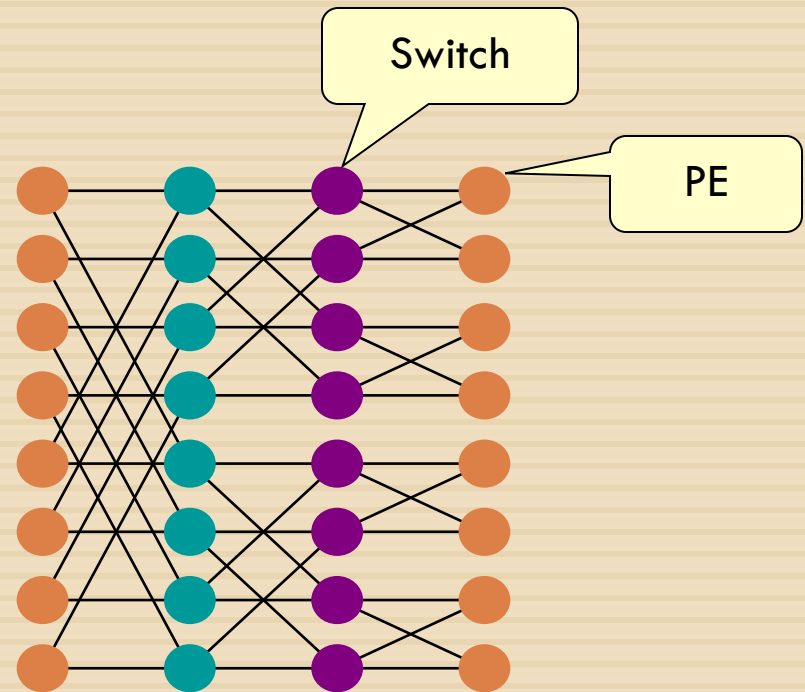
Butterflies with $n = (k-1)2^k$ switches

326

- Connecting 2^k processors, with Bisection bandwidth $= 2 \cdot 2^k$
- Cost: lots of wires
- $2 \log(k)$ hop-distance for all connections, however blocking possible
- Used in BBN Butterfly
- Natural for FFT



Butterfly switch



Multistage butterfly network: $k=3$