

REVIEW OF COMPUTER ARCHITECTURE

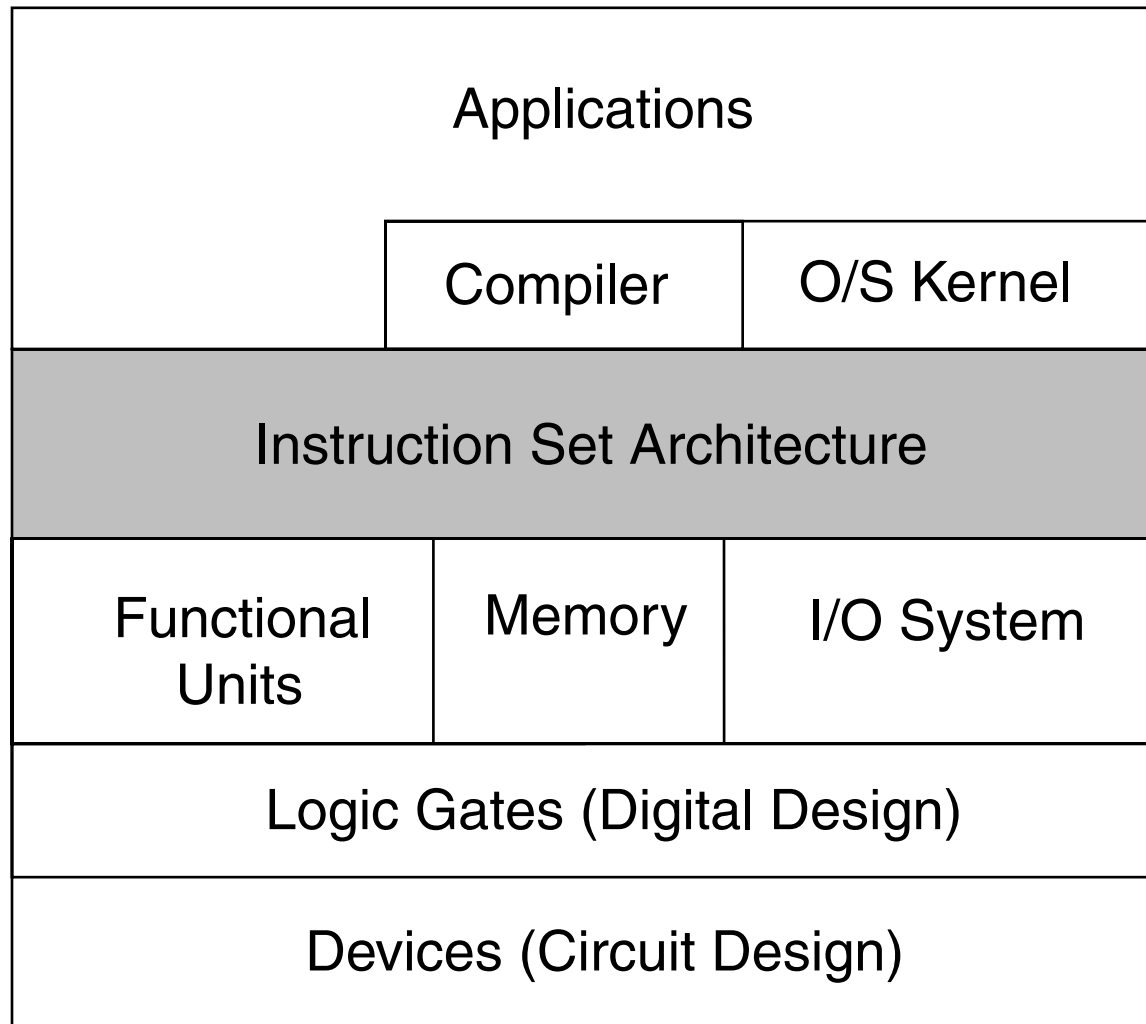
Notes prepared for EE 6481

by

Professor Cyrus D. Cantrell

May–August 2005

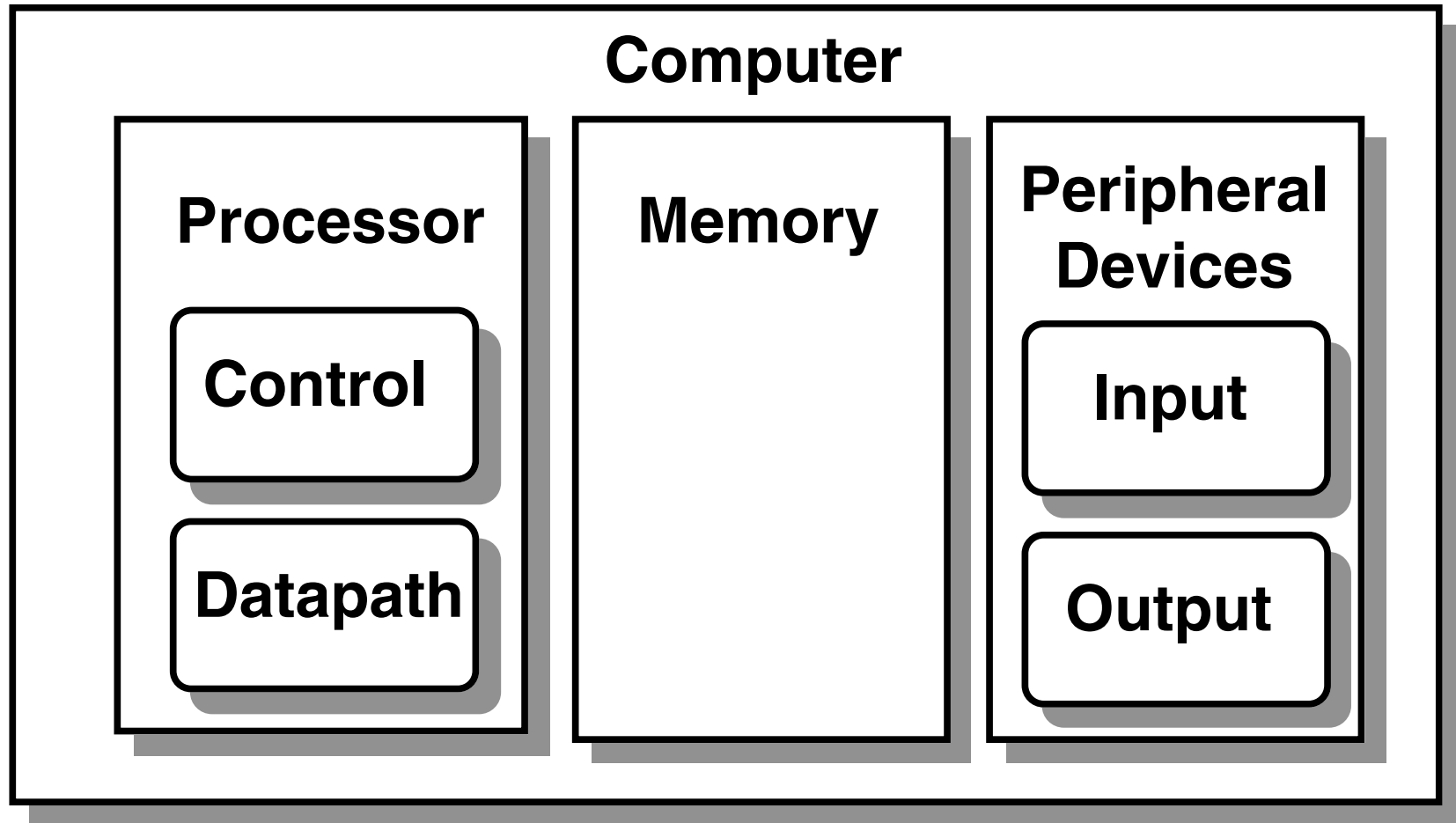
WHAT IS “COMPUTER ARCHITECTURE”?



SHRINK-WRAPPED SOFTWARE

- **Shrink-wrapped software** is a compiled program (or suite of programs) packaged in a box that is sealed inside a plastic envelope that shrinks to fit the box when it is heated
- Three things make shrink-wrapped software usable on computers that are manufactured by different vendors:
 - ▷ Different vendors compile their software for the same instruction set architecture
 - ▷ Different vendors compile their software for the same operating system
 - ▷ Input/output hardware that conforms to a common set of standards is available on different systems

THE MAIN COMPONENTS OF A COMPUTER



HIERARCHICAL APPROACH TO WRITING SOFTWARE

- Large program (100's or 1000's of lines)
 - ▷ Use top-down approach:
 - Divide problem into tasks
 - Define a function (procedure, subroutine) to accomplish each task
 - Code each function in its own module
 - Specify interfaces (parameters to be passed, etc.) for each function module
 - ▷ **Essential concept for systems integration:** The implementation of a function is independent of its interface specification
 - Implementation belongs to a different level in the hierarchy
 - Each module is a “black box” to higher-level modules
- Design is the most important step in creating a large program
 - ▷ Most hard-to-find bugs originate in a faulty design

STORED-PROGRAM COMPUTERS

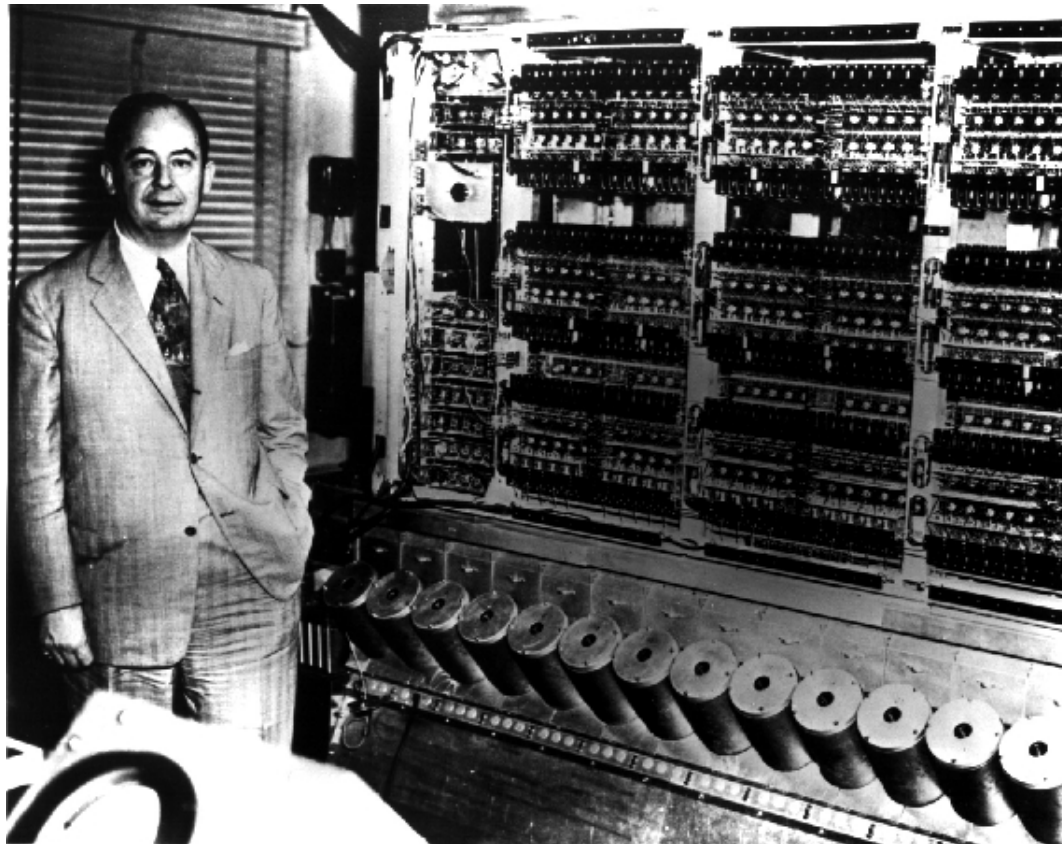
- Instructions and data stored as numbers in a common memory
 - ▷ Same hardware can be used to load/store instructions and data
 - ▷ An instruction must be referenced by its address in memory
 - ▷ Programs are *software*, because they are not *hardwired*
 - ▷ Same hardware can execute many different programs
 - ⇒ general-purpose computers
 - ▷ Same memory can hold several different programs, and the processor can switch execution from one to another
 - ▷ In principle, permits self-organizing structures
 - This feature rarely used intentionally
 - Self-modification by a program usually ⇒ trouble!
 - ▷ Formulation of concept, design of EDVAC: John von Neumann

THE VON NEUMANN ARCHITECTURE (1)

- **General-purpose, stored-program computer**
 - ▷ Instructions and data are stored in the same format in memory:
Instructions are data
 - ▷ In principle, an executing program can modify itself
 - In practice, this usually means that an error has occurred!
- **Strictly sequential execution**
 - ▷ Instruction fetched from memory, then
 - ▷ Instruction decoded, then
 - ▷ Data fetched from memory, then
 - ▷ Instruction executed, then
 - ▷ Result stored to memory

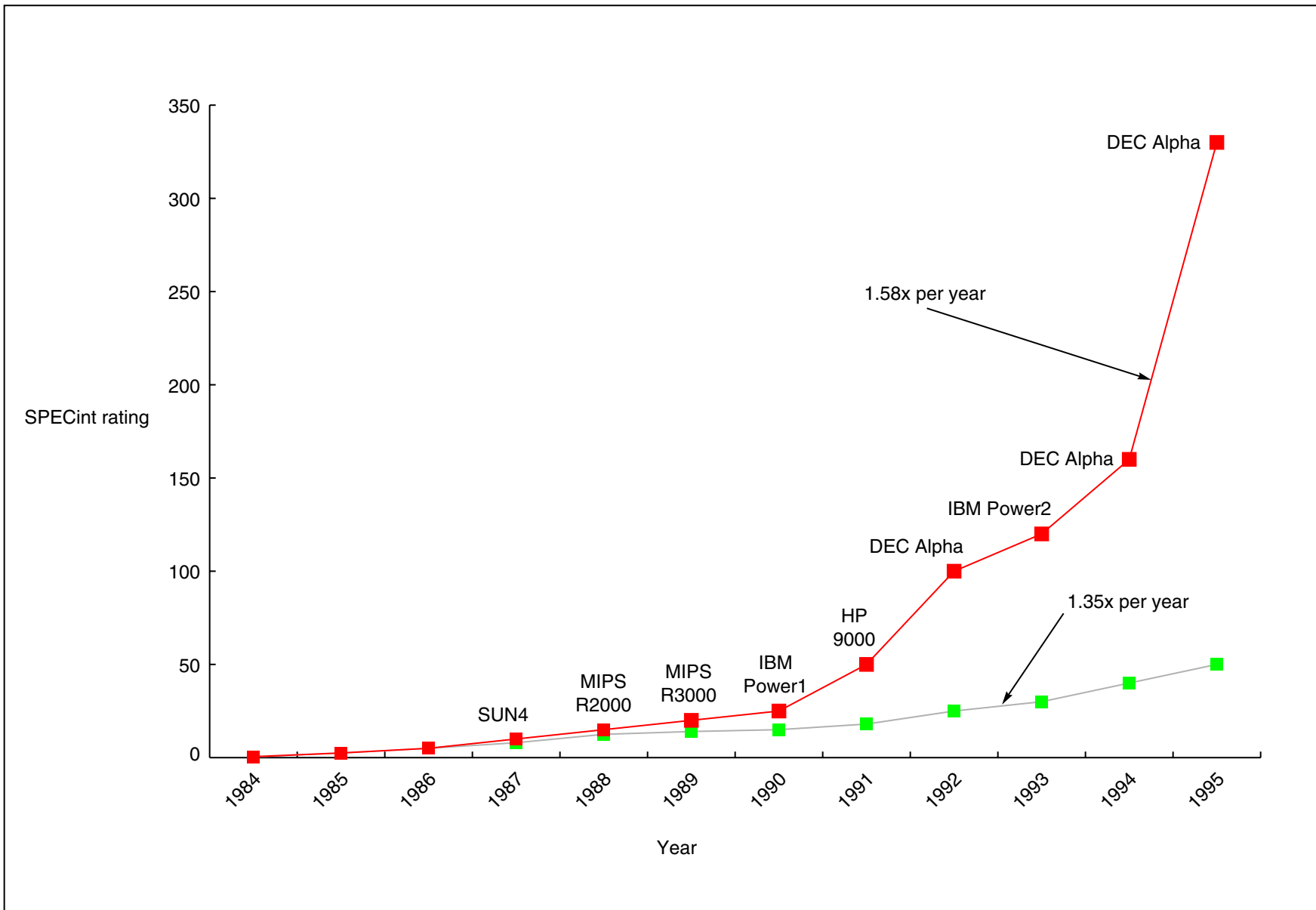
THE VON NEUMANN ARCHITECTURE (2)

- John von Neumann standing in front of the MANIAC computer at the Institute for Advanced Study, Princeton



CONCURRENT COMPUTATION

- Substantial performance improvements have occurred through concurrency in computation
 - ▷ Data and instruction pipelining in hardware ($\sim 5\times$ — $8\times$)
 - ▷ Multiple processors (up to a factor of $P = \text{no. of processors}$)
 - Coarse-grained parallelism: Clusters
 - Medium-grained parallelism: Symmetric multiprocessors
 - ▷ Fine-grained parallelism: VLIW architectures ($\sim 5\times$)
 - ▷ Limitation: Amdahl's law
- Compare with gains due to hardware improvements since 1946:
 - ▷ Factor of $10^{4.5}$ in CPU clock frequency
 - ▷ Factor of 10^3 in bus clock frequency
 - ▷ Factor of $\sim 10^{5.5}$ in latency of random-access memory

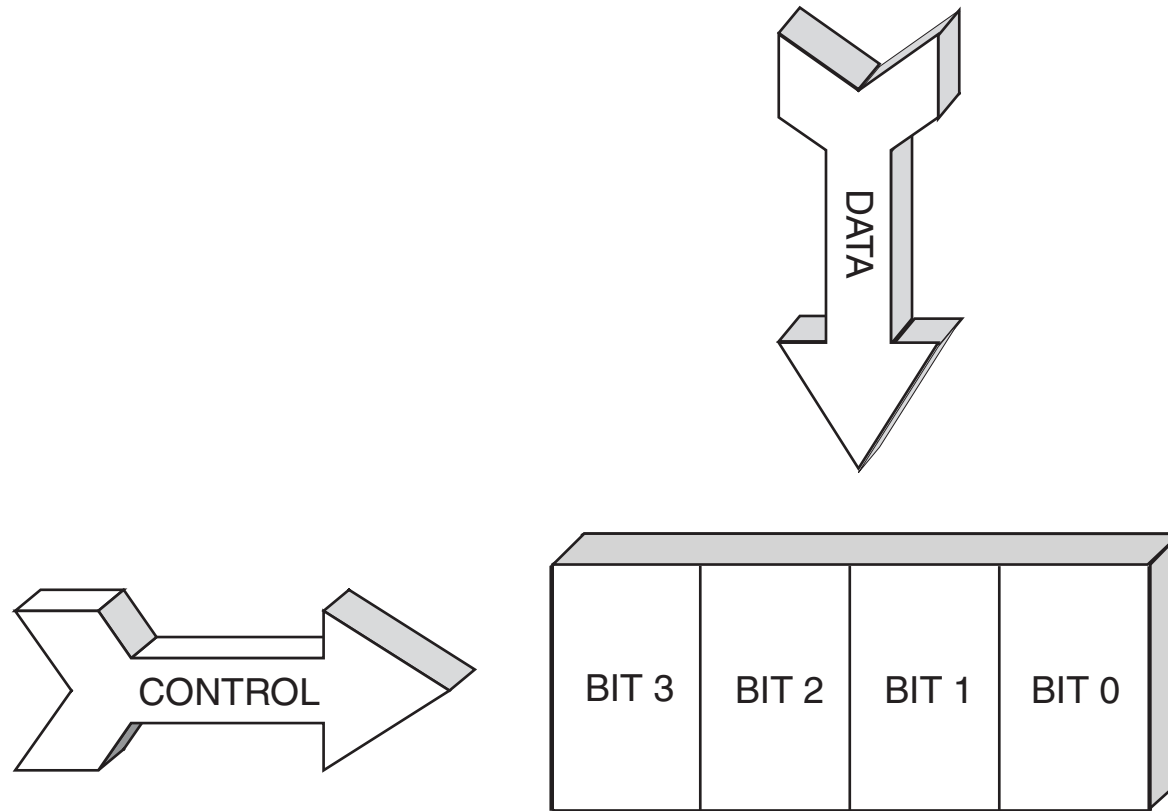


Growth in microprocessor performance since the mid 1980s has been substantially higher than in earlier years

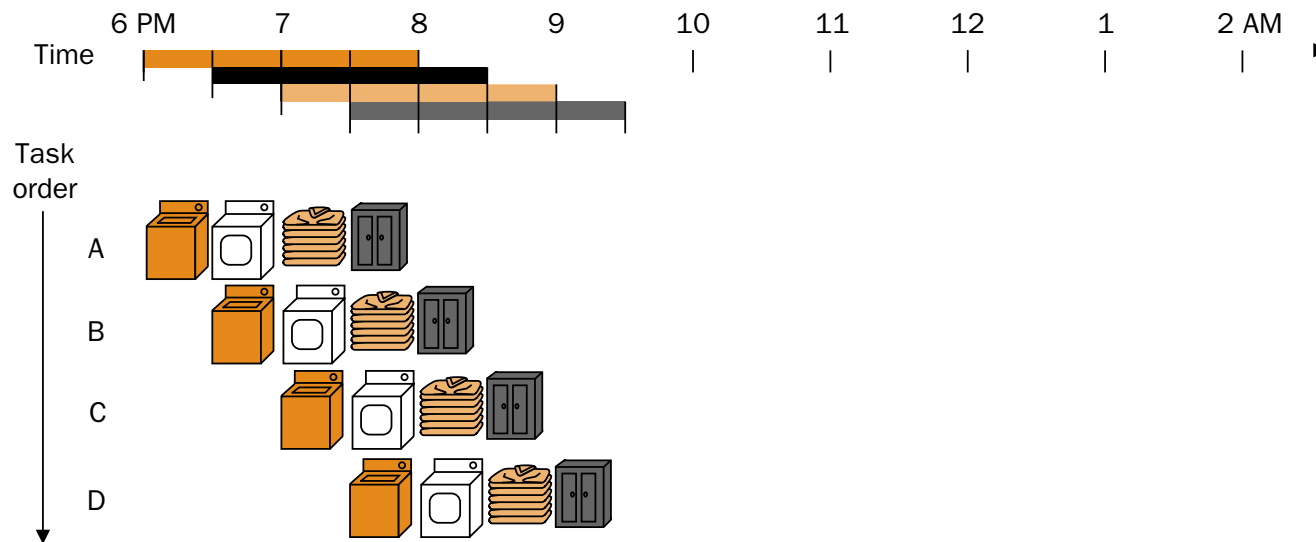
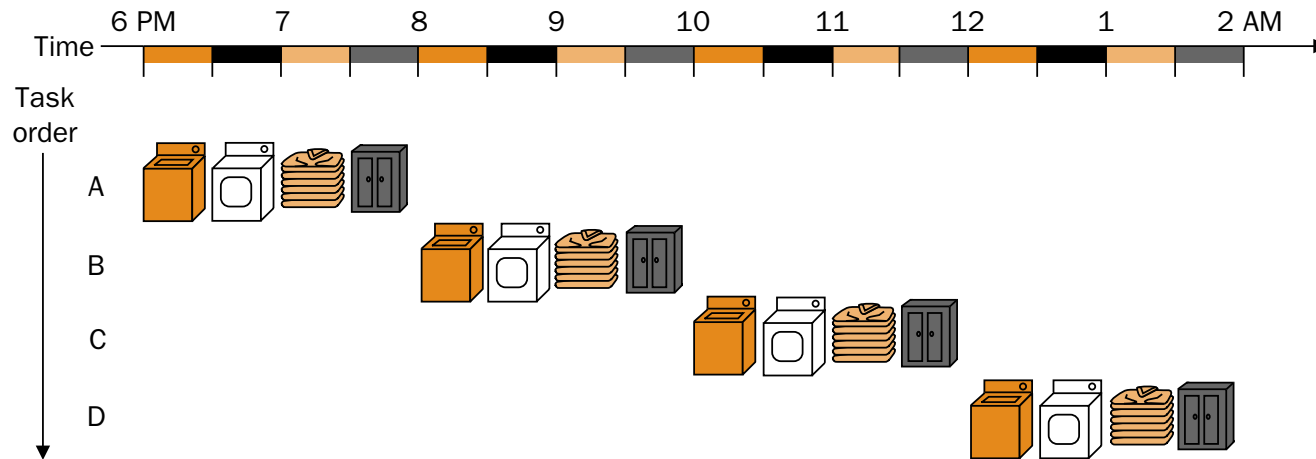
THE DATAPATH

- One of the five basic components of a computer
- All datapath operations are based on one simple idea: **Operate repetitively on chunks of data using the same algorithm**
 - ▷ A “chunk” can be a bit, a byte (8 bits), a short word (16 bits), a long word (32 bits), or a quad word (64 bits)]
 - ▷ The MIPS instruction set architecture uses words of 32 or 64 bits
- RISC machines load data chunks into registers before operating on them
 - ▷ The cycle of datapath operations is
 - Load data from main memory into registers
 - Operate on the data using the ALU, FPU, etc.
 - Store the result back to main memory

BIT SLICING: THE SAME ALGORITHM IS APPLIED TO ALL BITS IN PARALLEL



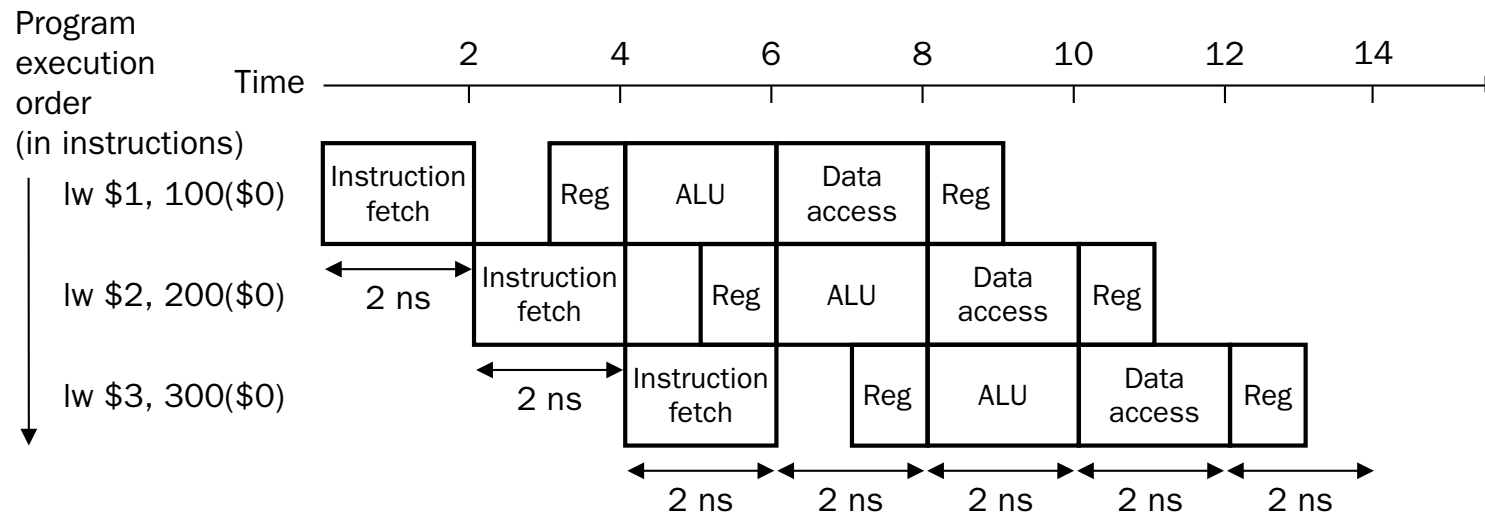
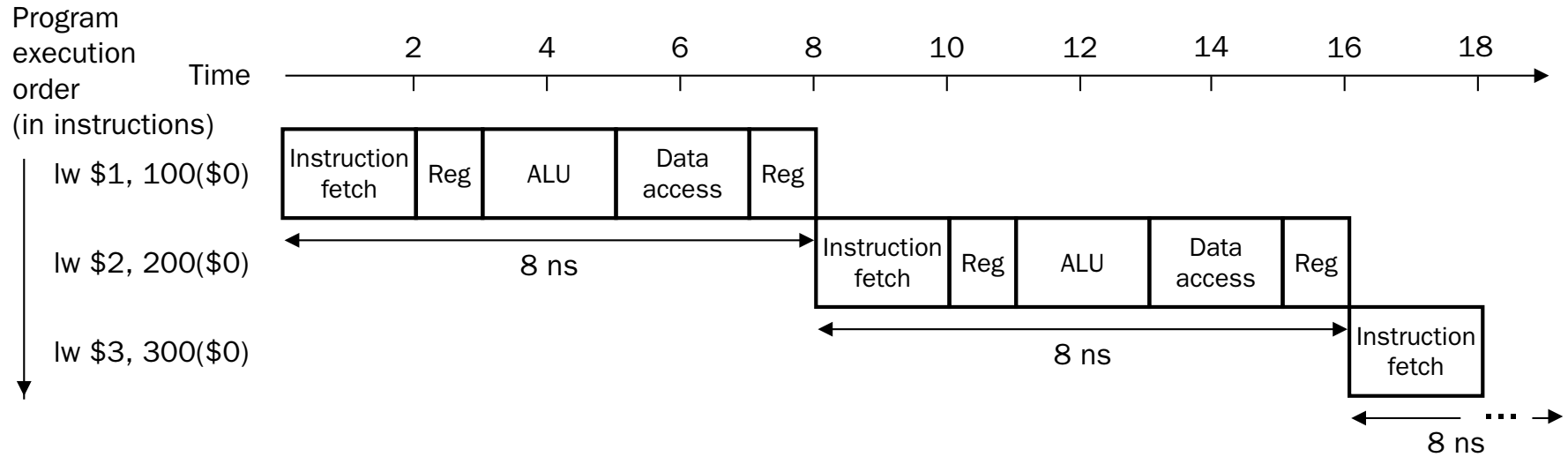
LAUNDRY: SEQUENTIAL vs. PIPELINED EXECUTION



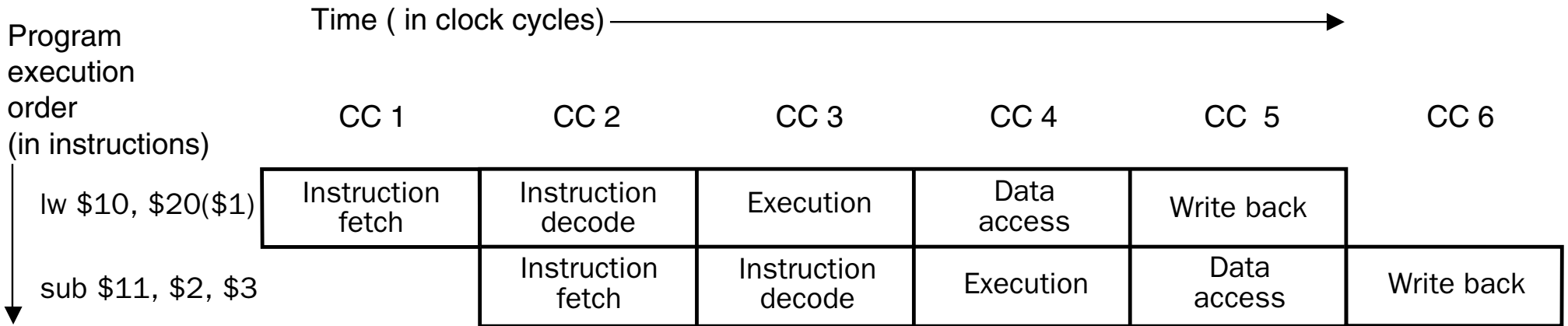
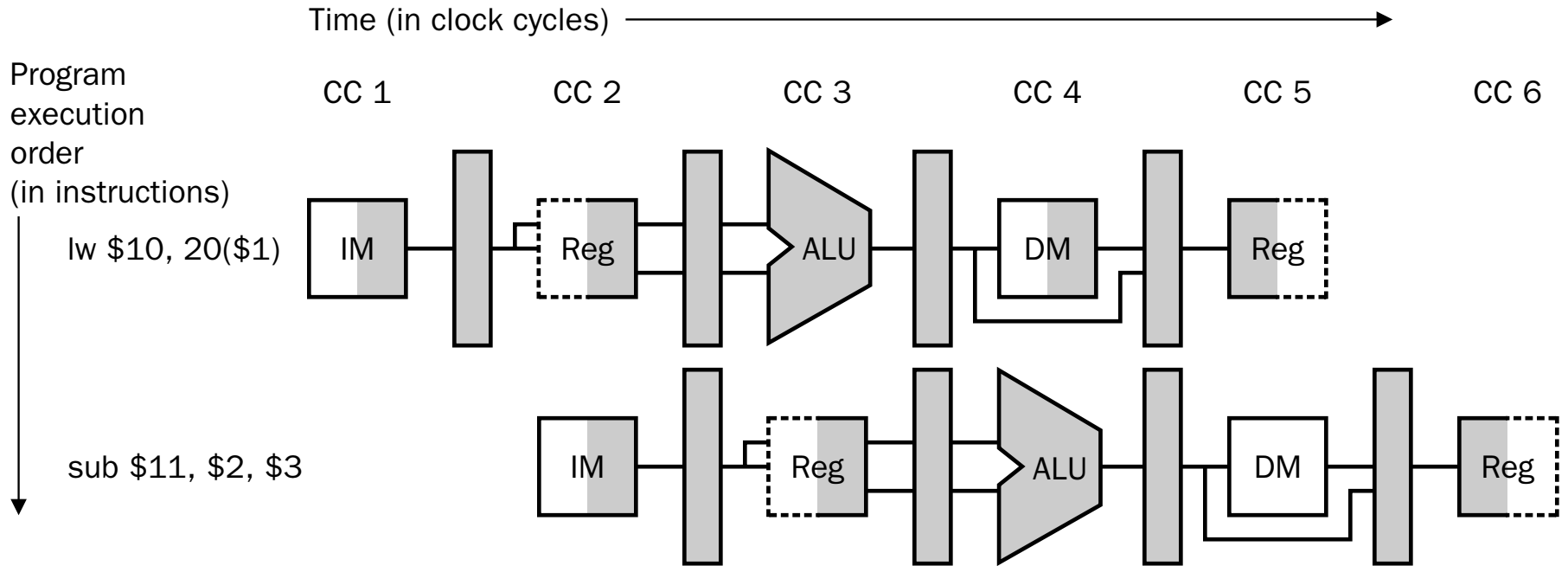
PIPELINING (1)

- In a pipelined computer architecture, a single processor can execute several instructions concurrently
 - ▷ Execution of one instruction uses several hardware functional units (instruction memory, register file, ALU, data memory, etc.)
 - ▷ Most functional units are used once per instruction
 - The register file may be read and written by the same instruction
 - ▷ The hardware functional units are organized into **stages**
 - Execution at each stage takes 1 clock period
 - Stages are separated by clock-controlled **pipeline registers** that preserve the state of execution for the duration of a clock period
 - ▷ The pipeline is subject to **hazards**
 - Data hazards: Write/read conflicts or timing problems
 - Control hazards: Exceptions and branches
- The MIPS R2000 pipeline design strongly influenced the design of all subsequent processors

COMPUTER: SEQUENTIAL vs. PIPELINED EXECUTION



TWO REPRESENTATIONS OF PIPELINED EXECUTION



PIPELINING (2)

● Pipeline speedup:

▷ A pipelined processor with s stages can execute n instructions in

$$ET_P = s + (n - 1) \text{ clock periods}$$

(assuming no hazards)

▷ A serial processor executes the same n instructions in

$$ET_S = ns \text{ clock periods}$$

▷ The ideal pipeline speedup equals the number of stages:

$$S_P = \frac{ET_S}{ET_P} = \frac{ns}{s + (n - 1)} \xrightarrow{n \gg s} s$$

● Amdahl's law applies to pipelining

PROGRAMMING IMPLICATIONS OF PIPELINING

- Avoid function or subprogram calls in an inner loop
 - ▷ Jumps force the pipeline to be flushed
- Avoid recursion in an inner loop
 - ▷ Recursion on the elements of an array generally causes data hazards because the value of $v[n]$ has not been written before it is needed for the computation of $v[n+1]$
- Avoid scalar temporary variables in an inner loop
 - ▷ Reading a memory-resident scalar variable may cause a data hazard
- Avoid **case** and **switch** statements in an inner loop
 - ▷ Conditional branches cause control hazards, and the use of a jump table may cause data hazards

DATA, INSTRUCTIONS AND MEMORY

- Harvard architecture pioneered by Howard Aiken:
 - ▷ Separate memories for instructions and data
 - ▷ Logical extension: Separate memory for each kind of data
 - Advantage: Data types can be recognized unambiguously
 - Disadvantage: Inefficient utilization of memory
- von Neumann architecture:
 - ▷ A single memory is used for both instructions and data
 - ▷ All types of data are stored in the same memory
 - Advantage: Efficient utilization of memory
 - Disadvantage: Data types cannot be recognized unambiguously
- The correct design decision is generally to use a single kind of memory for all data
 - ▷ Software has to decide how a particular data item should be interpreted

TYPES OF MEMORY

- Non-random access
 - ▷ Latency dependent on addresses of last block and current block
 - ▷ Examples:
 - Magnetic disk (latency = seek time + rotation time)
 - Magnetic tape (completely sequential access)
 - CD-ROM
 - ▷ Several contiguous blocks take almost the same time as one block
- Random access memory (RAM)
 - ▷ Latency independent of address
 - ▷ Example: Semiconductor memory
 - Static RAM (SRAM) — Retains data as long as power is on
 - Dynamic RAM (DRAM) — Data must be refreshed periodically

RANDOM-ACCESS MEMORY (1)

- Read-only
 - ▷ Mask-programmed ROM
 - ▷ User-programmable (PROM)
- Read mostly
 - ▷ EPROM, EEPROM, NVRAM
- Read/write
 - ▷ Static RAM (SRAM): Semiconductor logic
 - Fast; no need for refresh cycle
 - Costly; hot
 - ▷ Dynamic RAM (DRAM): Arrays of CMOS capacitors
 - Low cost & heat dissipation; a commodity product
 - High capacity
 - Slow because of complex addressing and refresh cycle

RANDOM-ACCESS MEMORY (2)

- SRAM
 - ▷ Access time: 5–15 ns
 - ▷ Price (8/96): \$80–200/MB
 - ▷ Hierarchical design of SRAM memory:
 - Cell design
 - IC design
 - SRAM memory system design
- SRAM cell design:
 - ▷ Clocked D-latch (single or master-slave configuration)
 - ▷ Tristate output
 - Enable asserted: Normal latch operation
 - Enable deasserted: Latch acts as an unconnected wire (high-impedance state)

WAYS TO MATCH CPU AND MEMORY SPEEDS

- Solutions to the problem that

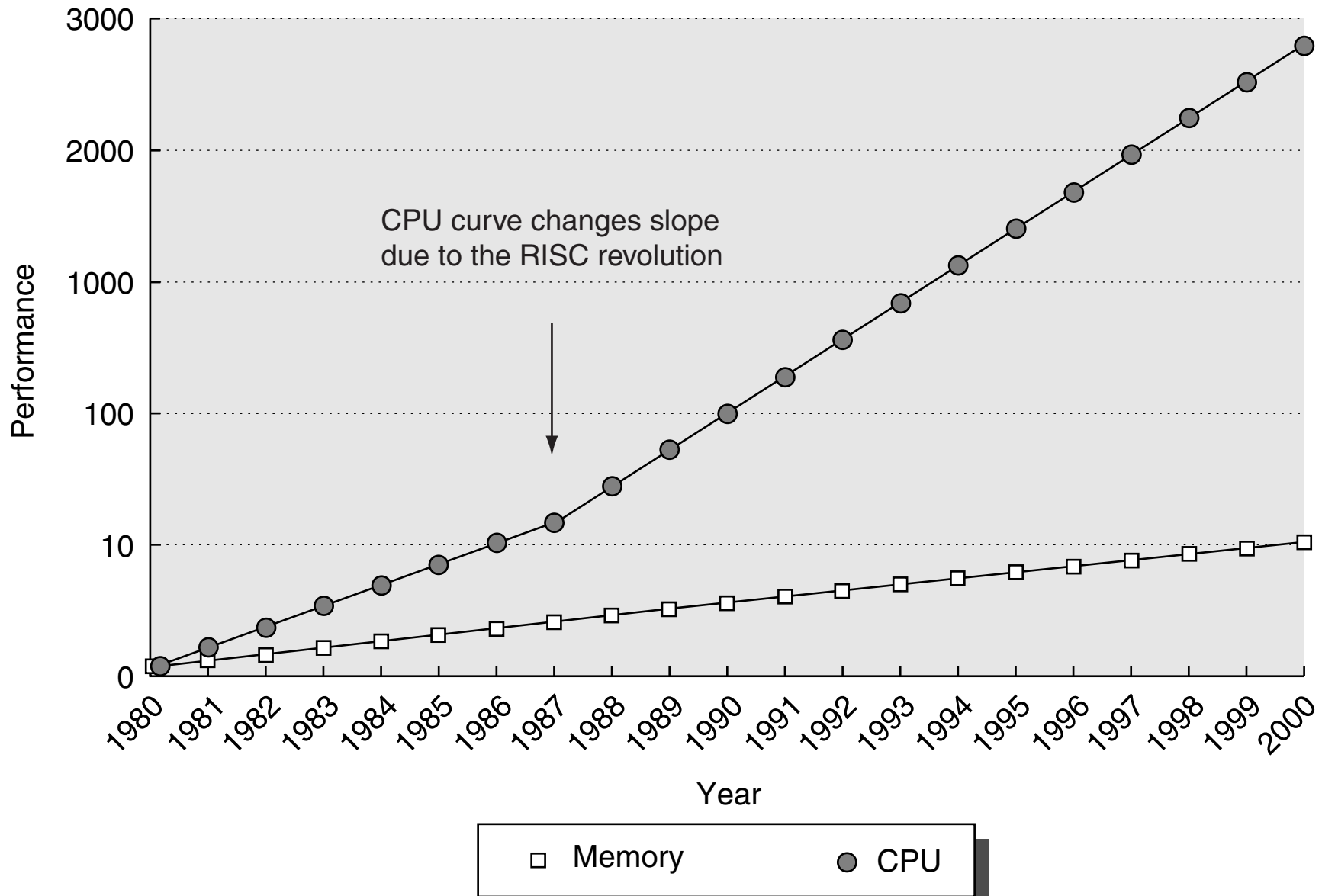
CPU clock period \approx 0.05–0.1 DRAM access time

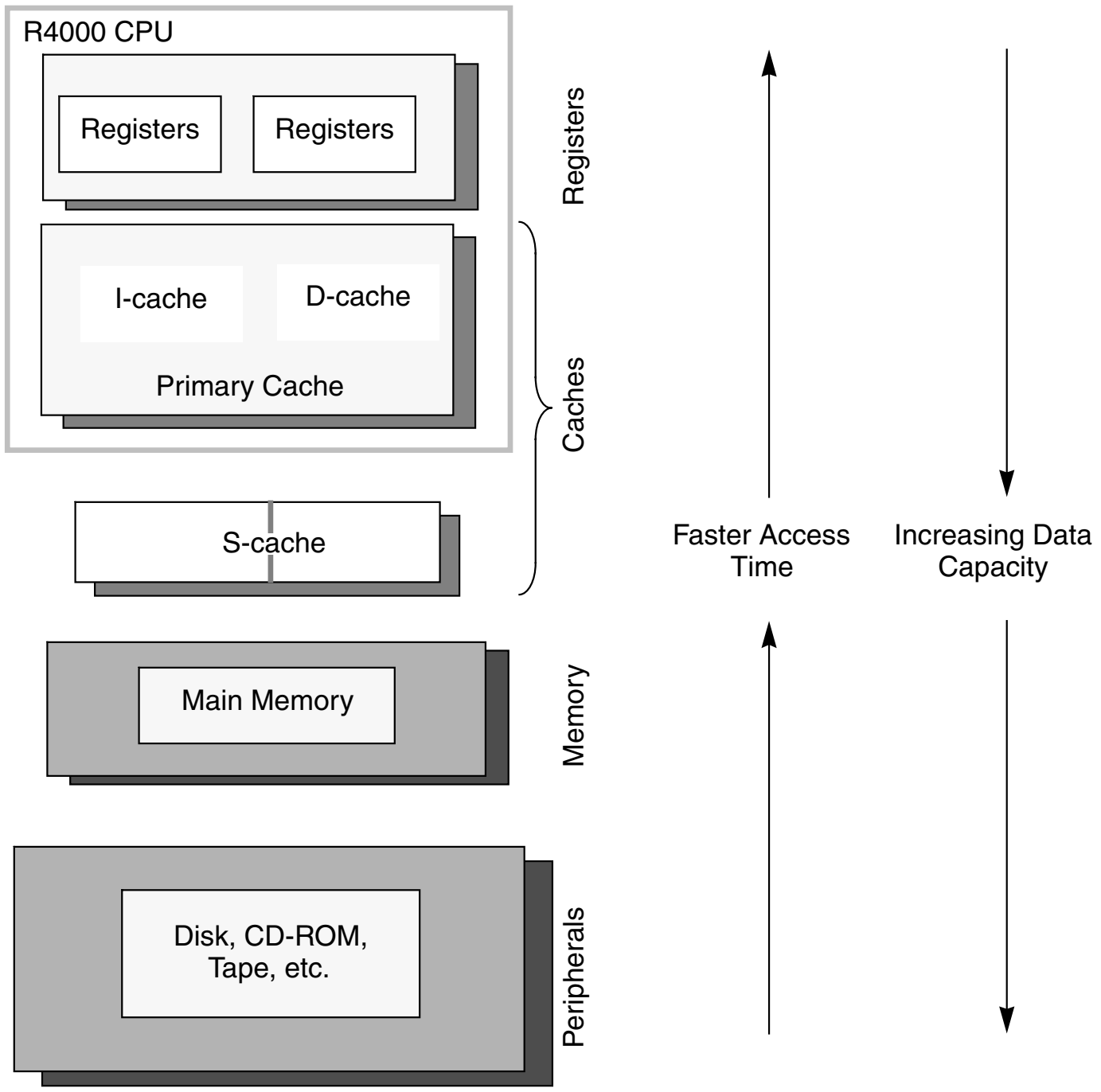
- ▷ Interleave the memory system

- Permits sequential reads or writes to main memory on successive CPU clock periods
- Cray's solution: interleaved SRAM memory (\$\$\$\$\$!)

- ▷ Hierarchical memory

- Keep recently-used data and instructions (and items with nearby addresses) in fast memory (**cache**)
- Fetch data and instructions from main memory if not in cache
- Other examples of the same approach:
 - ◇ Main memory (SRAM) and Solid State Disk (DRAM) (CRAY)
 - ◇ Main memory (faster) and magnetic disk (slower)
 - ◇ Magnetic disk (faster) and magnetic tape (slower)





Logical Hierarchy of R4000 Memory

HIERARCHICAL MEMORY (1)

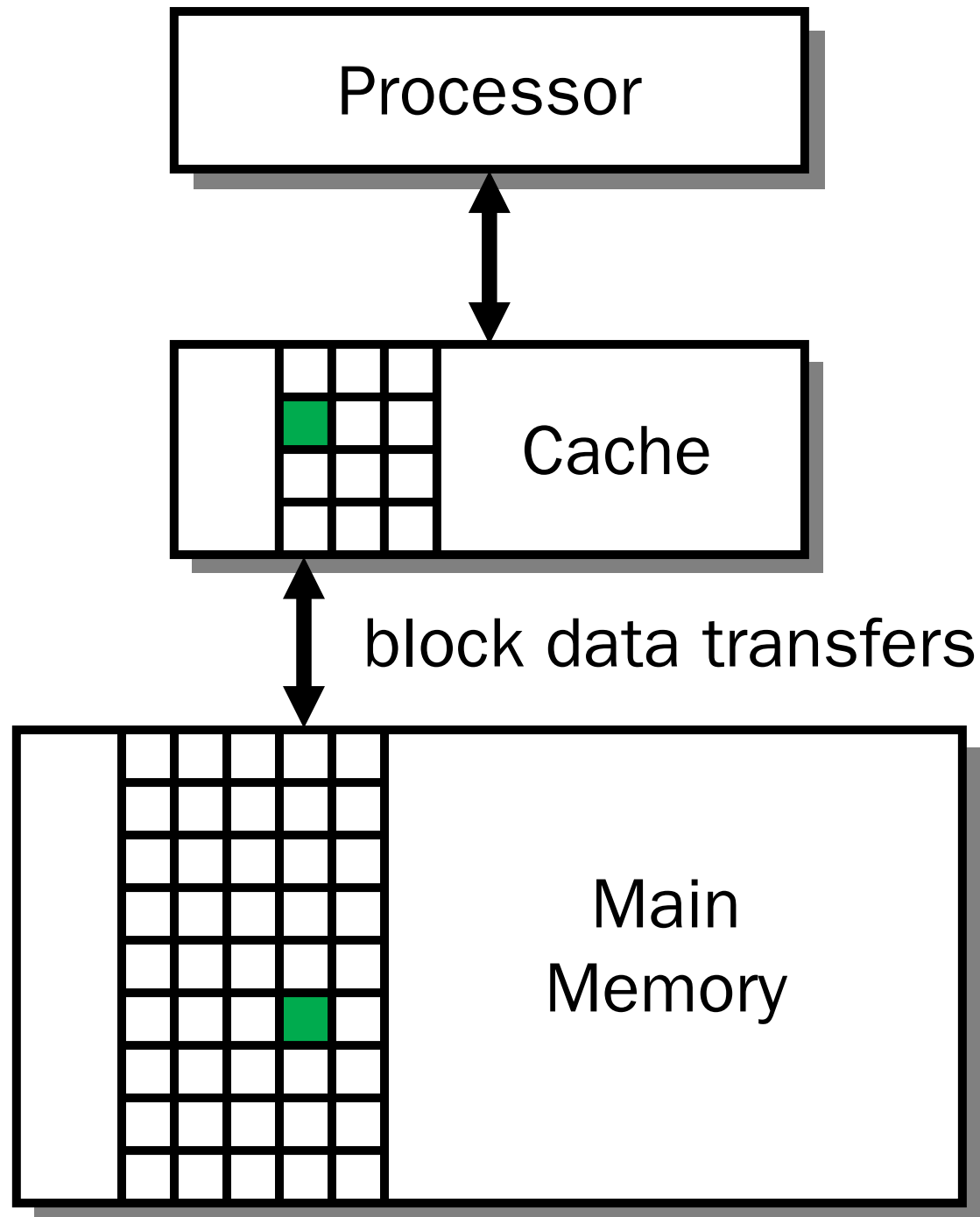
- Goal: Create the illusion that the processor has access to a huge amount of very fast memory
 - ▷ Must be an illusion because price increases with speed:

Technology	Latency (ns)	\$ per MByte
SRAM	1–10	80–200
DRAM	40–60	5–7
Magnetic Disk	3×10^6	10^{-3}

- ▷ Strategy:
 - Keep most frequently used data & instructions in fastest memory
 - Put less frequently used items in next lower level of memory
 - Use locality in space and time as initial criteria for deciding what to put in fast memory

HIERARCHICAL MEMORY (2)

- Memory references in running processes or threads tend to obey the **Principles of Locality**:
 - ▷ If a data item or an instruction is accessed, it is likely to be used again soon (**temporal locality**)
 - ▷ If a data item or an instruction is referenced, a data item or an instruction that is stored at a nearby address in main memory is likely to be accessed soon (**spatial locality**)
 - Keep most frequently used data & instructions in fastest memory
 - Put less frequently used items in next lower level of memory
 - Use locality in space and time as initial criteria for deciding what to put in fast memory
 - ▷ Locality generally is not the same for data and instructions
 - ▷ Example: matrix multiplication with a very large stride in memory
 - Instructions are highly local (inner loop), but data is not local



HIERARCHICAL MEMORY (3)

- Terminology

- ▷ A **hit** occurs in an upper level of hierarchical memory when the data requested is in a block in the upper level

- Hit rate = fraction of accesses found in upper level
- Hit time = time to access upper level + hit/miss decision time

- ▷ A **miss** occurs in an upper level of hierarchical memory when the data requested is *not* in a block in the upper level

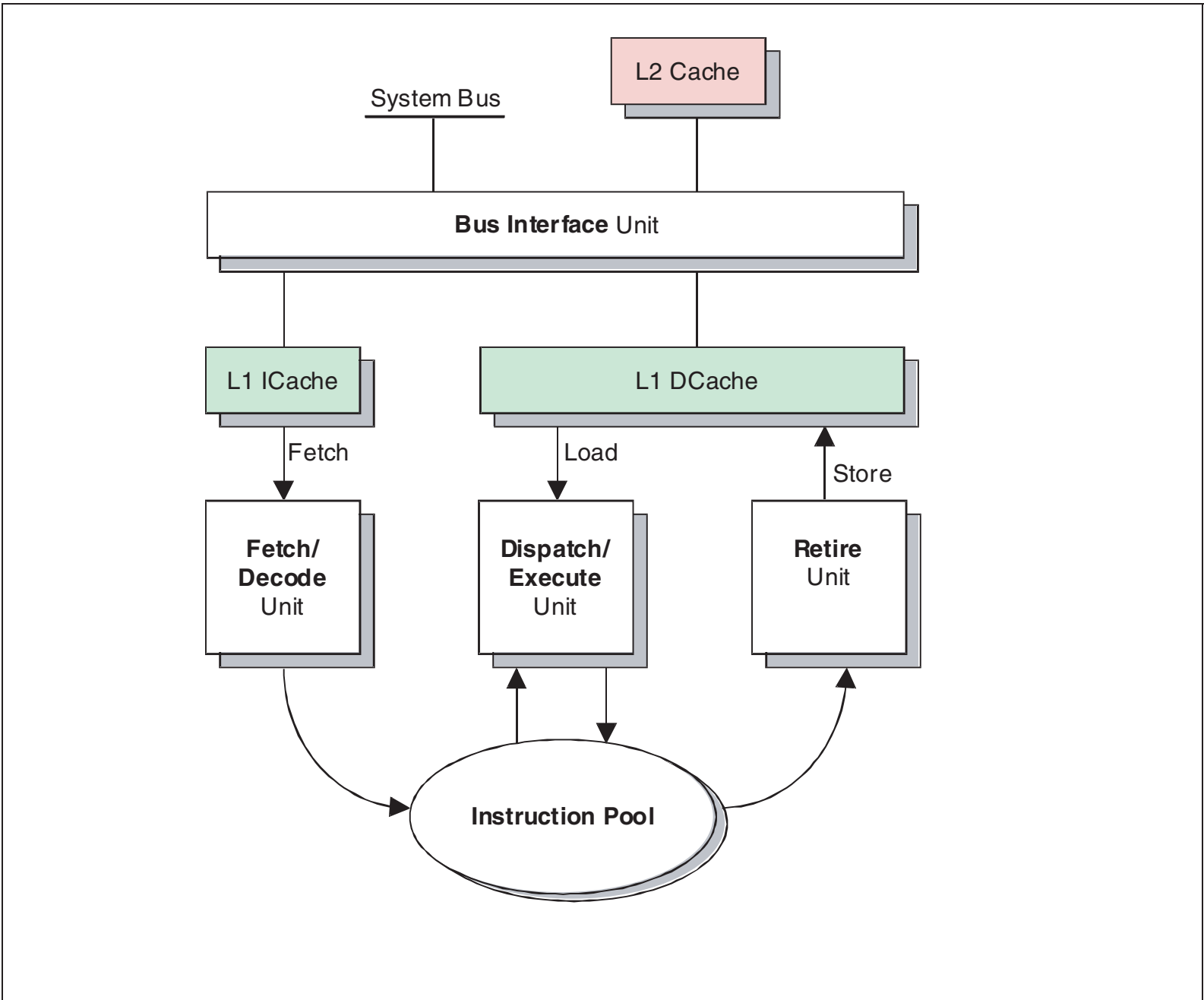
- Data must be obtained from a block in the lower level
- Instruction processing stops until the data is fetched
 - ◊ Then the processor restarts the access
- Miss rate = $1 - \text{hit rate}$

Miss penalty = Time to get block from lower level
+ Time to place block in upper level
+ Time to deliver data to the processor

- Hit time \ll miss penalty

HIERARCHICAL MEMORY (4)

- The levels of hierarchical memory between the processor and main memory are called **cache**
 - ▷ **Level 1 cache** is on the same die as the processor
 - Size is limited by available real estate (8–64 kB)
 - May contain:
 - ◇ Instructions only (common in processors intended for use in embedded systems)
 - ◇ Data only (not common, because instructions have better locality than data)
 - ◇ Instructions and data in same cache
 - ◇ Instructions and data in separate caches
 - ▷ **Level 2 cache** is implemented in SRAM on a separate card or chip, or on the processor die
 - Used with all RISC processors
 - Size dictated by heat dissipation and maximum number of transistors per processor



P6 processor interface to memory

HIERARCHICAL MEMORY (6)

- Systems approach to data transfers between two levels of hierarchical memory (Amdahl's law again!):
 - ▷ Logically organize all data stored in the lower level into blocks
 - ▷ Store the most-frequently-used blocks in the upper level
 - **Hit rate**
 - H = fraction of memory references to data in the upper level
 - **Miss rate** = $1 - H$; upper-level latency = T_u ;
miss penalty = T_m ; lower-level latency = T_l
 - Average memory latency:

$$T_{\text{eff}} = HT_u + (1 - H)T_m$$

- Speedup:

$$S = \frac{T_l}{T_{\text{eff}}} = \frac{T_l/T_m}{1 - H(1 - T_u/T_m)}$$

REGISTERS (1)

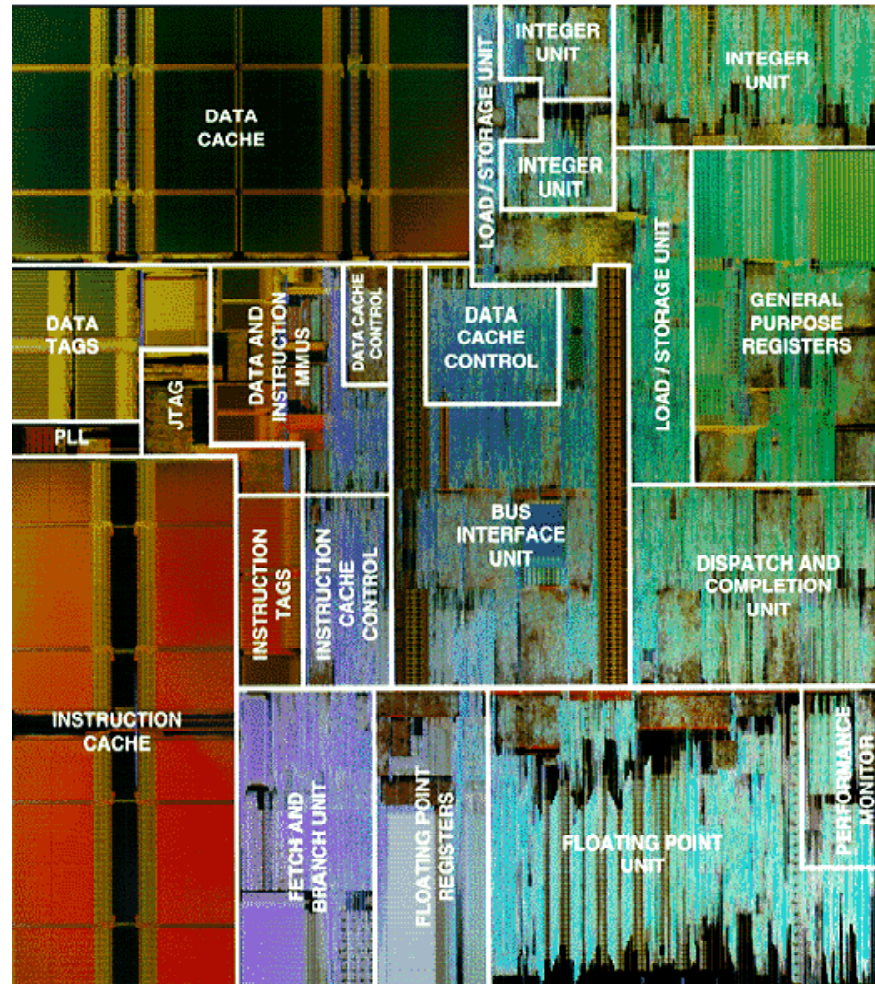
- A **register** is an array of flip-flops with special properties:
 - ▷ Registers are the fastest kind of addressable memory
 - ▷ Registers are located on the same die as the processor
 - Registers are **NOT** located in main memory
 - ▷ The set of registers in a processor is called the **register file**
- Registers are addressed separately & differently from main memory
 - ▷ MIPS R2000 architecture:
 - Main memory addresses are 32 bits
 - ⇒ 4,294,967,296 addressable 8-bit bytes
 - Register addresses are 5 bits
 - ⇒ 32 addressable 32-bit general-purpose registers
 - Register addresses run from $00000_2 = 0_{10}$ to $11111_2 = 31_{10}$
 - MIPS assembler: Register addresses are written \$0, ..., \$31

REGISTERS (2)

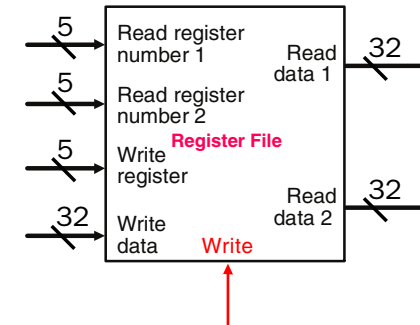
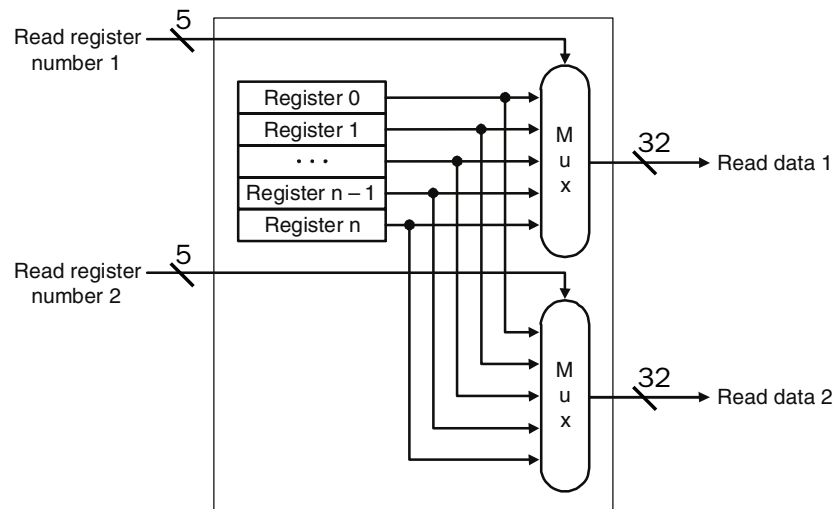
- Why are registers necessary?
 - ▷ The processor needs a “scratch pad” for operands and intermediate results
 - ▷ Some data items must be accessed in 1 clock period to avoid wait states
- Why isn't the register file a part of main memory?
 - ▷ Really fast memory is really expensive
 - ⇒ number of words in registers \ll number of words in main memory
 - ▷ Really fast memory is really hot
 - ⇒ major heat-dissipation problem, even if you're rich!
 - ▷ Propagation velocity of a signal \approx 6 inches per nanosecond
 - ⇒ fast memory must be physically close to the ALU & control unit

A POWERPC 604e MICROPROCESSOR

Motorola's PowerPC 604e™ RISC Microprocessor

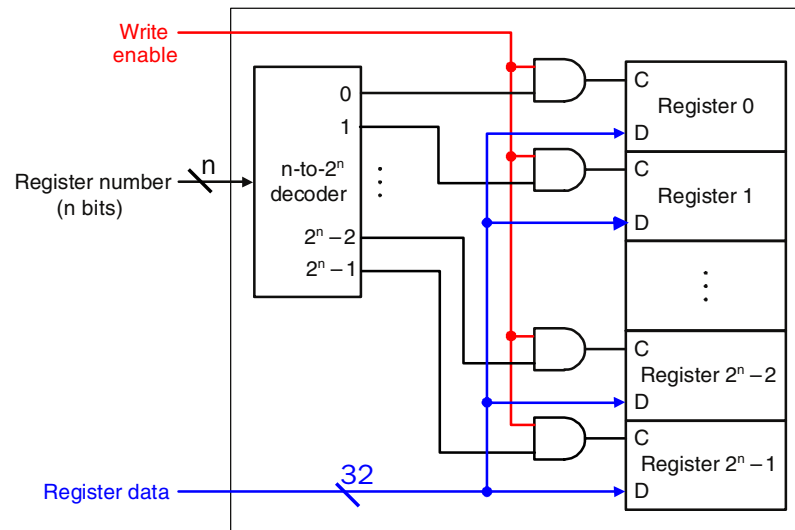


REGISTER FILE



The register file is an array of arrays of flip-flops, addressed using a decoder, read using multiplexors. Data to be written to a particular register is **broadcast** to all registers, but only the specific register that is **selected** by an asserted “write enable” signal is modified.

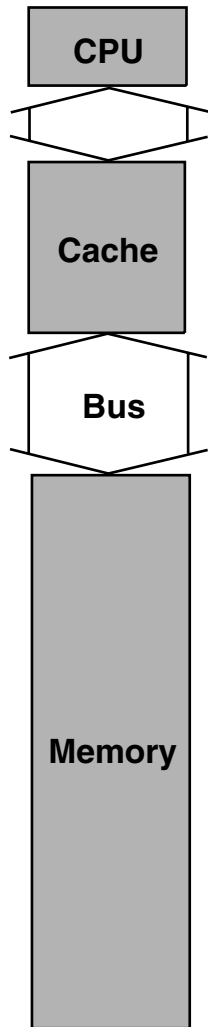
APPLICATION OF DECODER TO REGISTER ADDRESSING



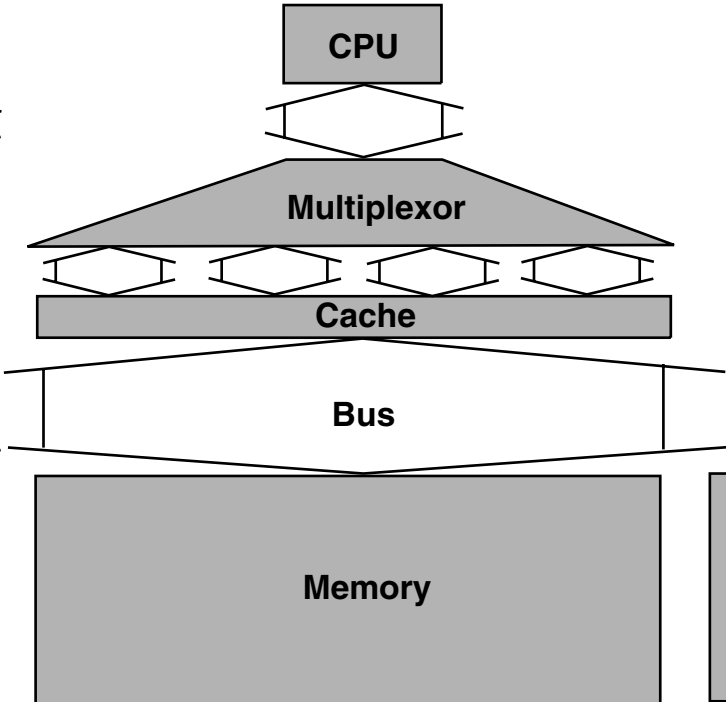
Data is **broadcast** to all registers, but only the register **selected** by the decoder is modified

- Each register has an “enable” input (labeled C in the figure)
 - ▷ A register’s enable input must be asserted in order for data to be written to the register through the “data” input (labeled D)
 - ▷ The enable input is controlled by an AND gate
 - ▷ Both the signal from the decoder and the “write enable” signal must be asserted in order for the register’s enable input to be asserted

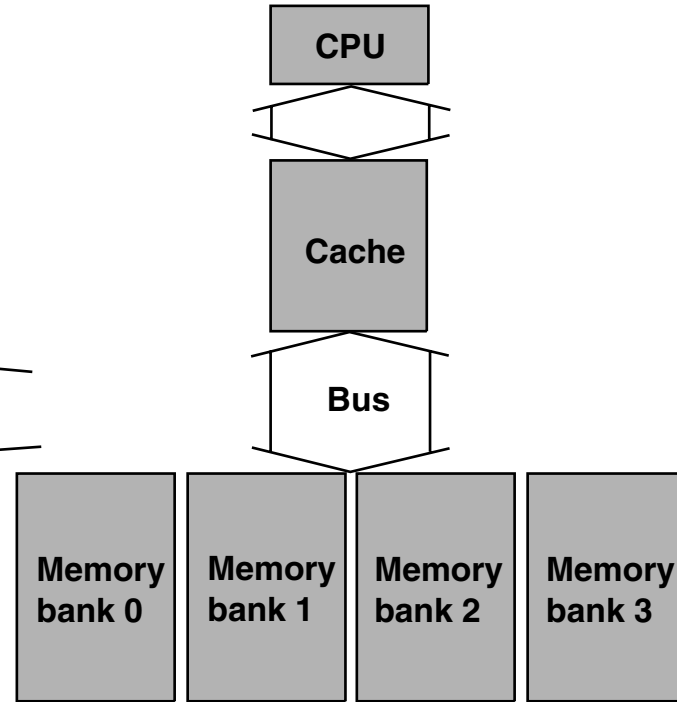
(a) One-word-wide memory organization



(b) Wide memory organization



(c) Interleaved memory organization



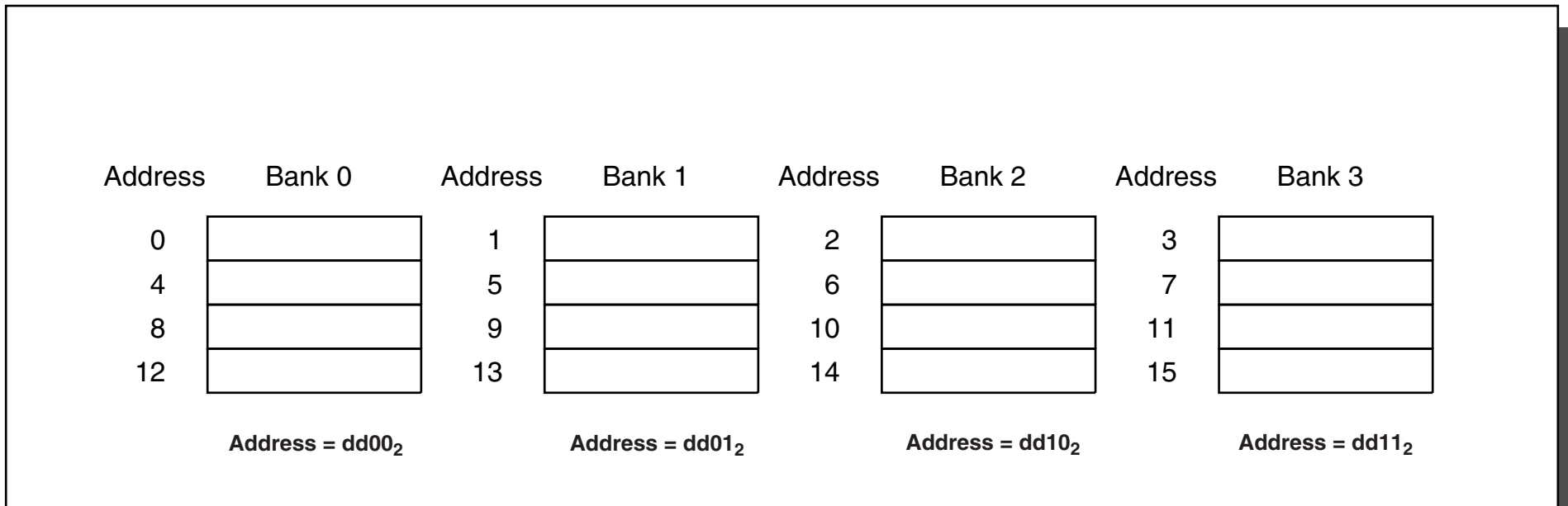
Three techniques for achieving higher memory bandwidth

INTERLEAVED MEMORY (1)

- Goal: Eliminate (or greatly reduce) the effects of DRAM cycle time, permitting memory accesses in successive clock periods
 - ▷ Physical memory is partitioned into 2^n banks of equal size
 - ▷ The address of the bank in which the byte or word at address M is stored is the least significant n bits of M :

$$\text{bank address} \equiv \text{word address} \pmod{2^n}$$

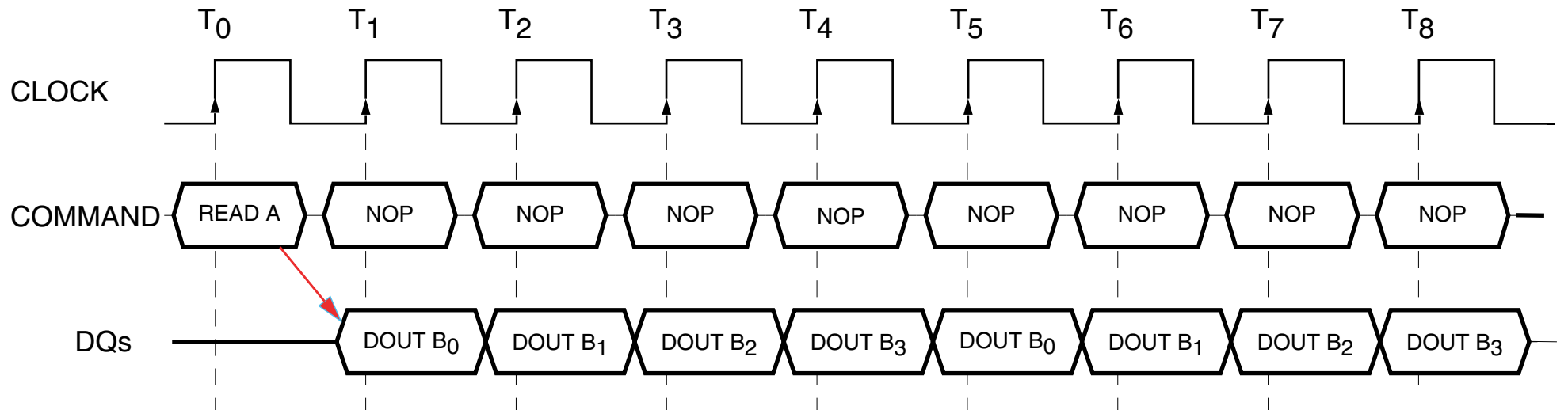
- ▷ The number of banks should be no smaller than the number of clock periods in one memory cycle time
 - Example: If the bank cycle time is 6 clock periods, then at least $2^n = 8$ banks should be used (8-way interleaving)
- ▷ Stride = 1 + number of words between successively accessed words
- ▷ Bank conflicts occur whenever the stride in memory has a common factor with the number of banks \Rightarrow bank accessed before it's ready
 - Example: With 8 banks, a bank cycle time of 6 clock periods, and a stride of 10, a bank conflict occurs on every 4th access



Four-way interleaved memory (assuming 4-bit addresses)

Hennessy and Patterson, *Computer Architecture: A Quantitative Approach*, 2nd Edition

Pipelined read operation with 4 memory banks (B_0, B_1, B_2, B_3)

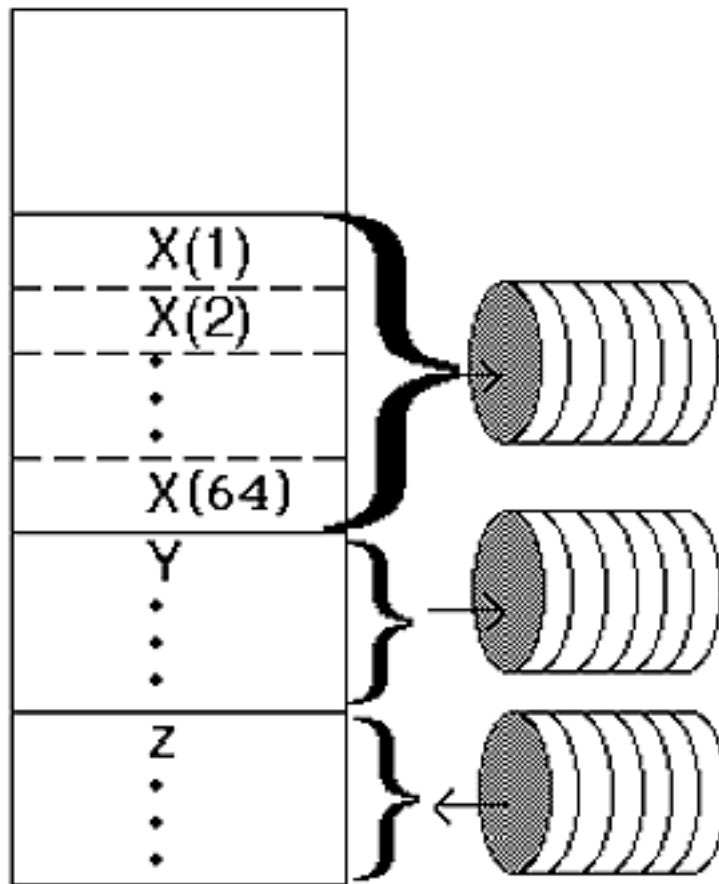


INTERLEAVED MEMORY (2)

- Memory organization of a CRAY J90 or C90 processor
 - ▷ 64 memory banks
 - Consecutive words in address space stored in consecutive banks
 - Bank cycle time: 6 clock periods
 - ▷ 3 vector ports to memory in J90, 4 in C90
 - ▷ In every clock period, two operands can be loaded from memory and a result can be stored to memory

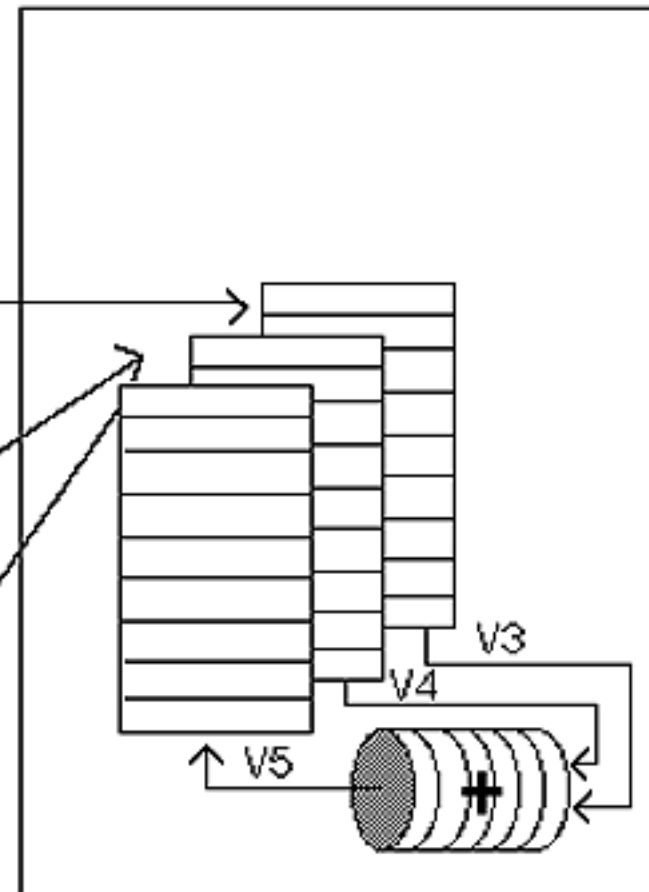
Vector pseudocode

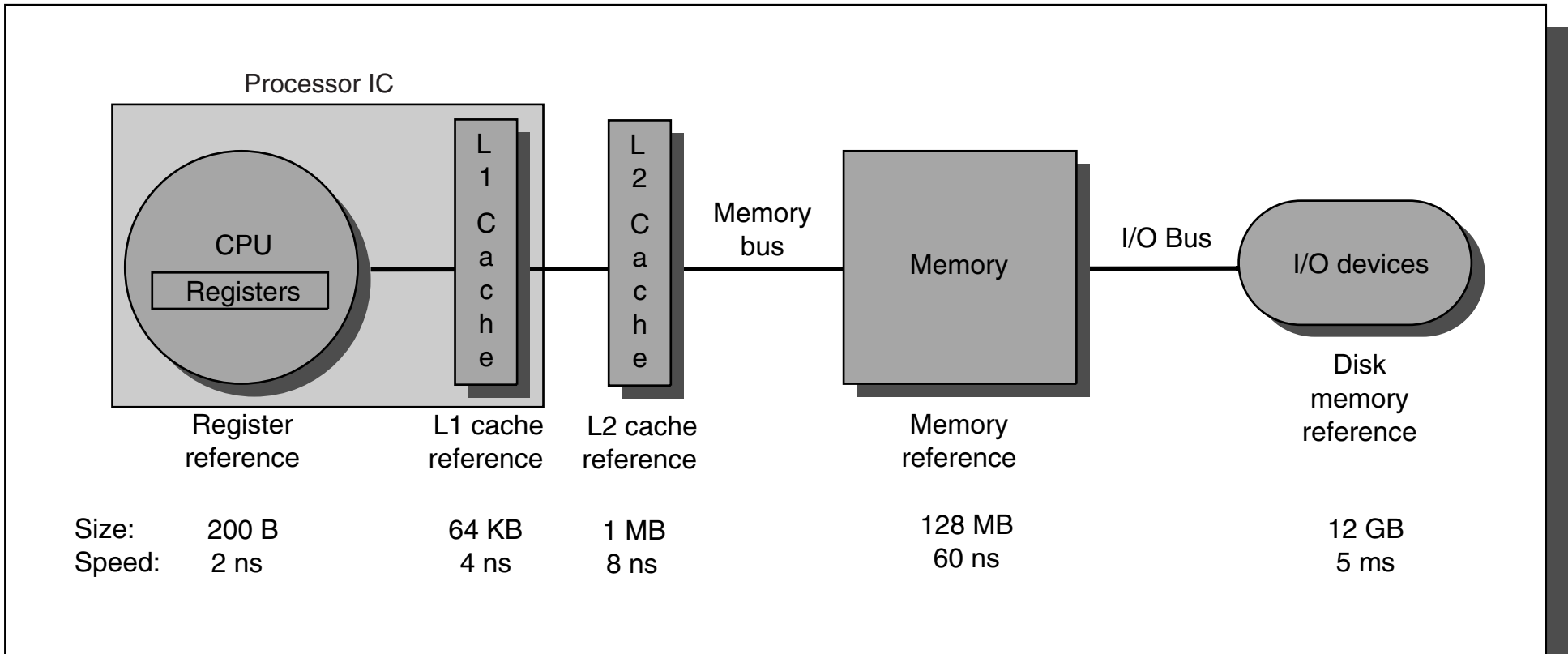
```
LOAD  $V_3$ , X(1:64)  
LOAD  $V_4$ , Y(1:64)  
 $V_5 = V_3 + V_4$   
Store Z(1:64),  $V_5$ 
```



Scalar pseudocode

```
LOAD R1, 1  
LOAD R3, X(R1)  
LOAD R4, Y(R1)  
R5=R3+R4  
STORE Z(R1), R5  
R1=R1+1  
JUMP 10 IF R1 <= 64
```

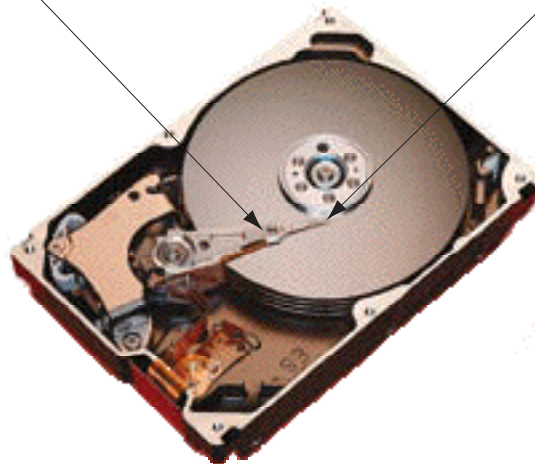


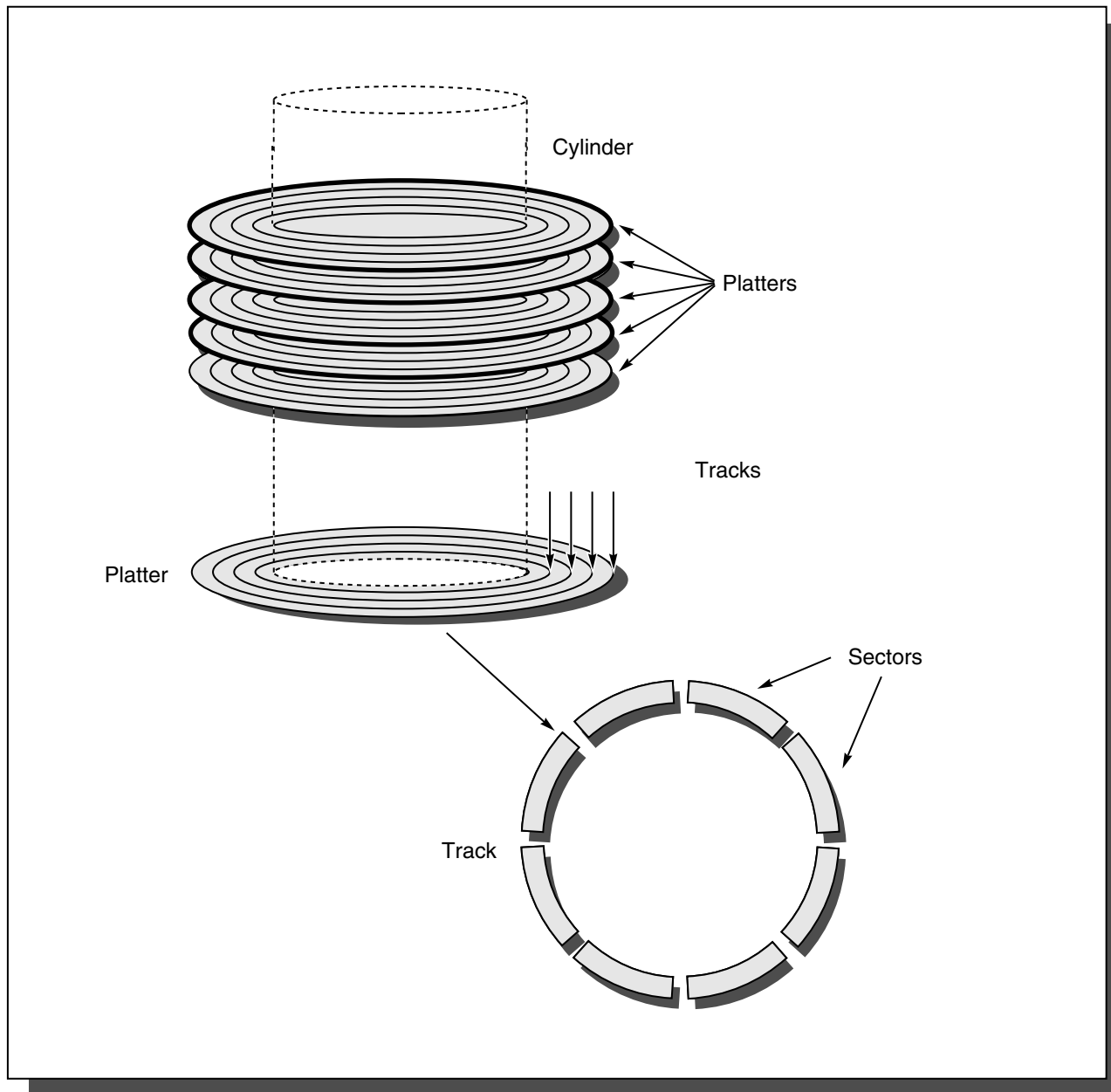


Levels in a typical memory hierarchy

DISK MEMORY (1)

- A **hard disk** records bits as patterns of magnetization in a thin coating on the surface of a rigid platter
 - ▷ Information is read or written through a set of **heads** attached to the ends of an **arm** with multiple segments





Disks are organized into platters, cylinders, tracks, and sectors

DISK MEMORY (2)

- The latency of a magnetic disk access is given by the formula

latency = seek time + rotational delay + transfer time + controller delay (not counting delays due to the fact that other processes may be using the disk)

- Terminology:

- ▷ Seek time = time for the arm to move from its present track to the track where the requested data is located
- ▷ Rotational time = time for the requested sector to rotate to the position of the arm

$$\text{rotational time} \leq \frac{60}{\text{disk rpm}} \quad \left(\text{average} = \frac{30}{\text{disk rpm}} \right) \quad (\text{seconds})$$

- ▷ Transfer time = time for data transfer from disk to main memory

$$\text{transfer time} \geq \frac{\text{amount of data in MB}}{\text{transfer rate (in MB/s)}}$$

DISK MEMORY (3)

- Example: latency of writing one 512-byte sector on a magnetic disk rotating at 5400 rpm, with the following parameters:

Average seek time = 12 ms

Transfer rate = 5 MB/s

Controller delay = 2 ms

$$\begin{aligned}\text{average latency} &= 12 \text{ ms} + \frac{30}{5400} \text{ s} + \frac{0.5 \times 10^{-3} \text{ MB}}{5 \text{ MB/s}} + 2 \text{ ms} \\ &= 19.7 \text{ ms}\end{aligned}$$

WHO MANAGES HIERARCHICAL MEMORY?

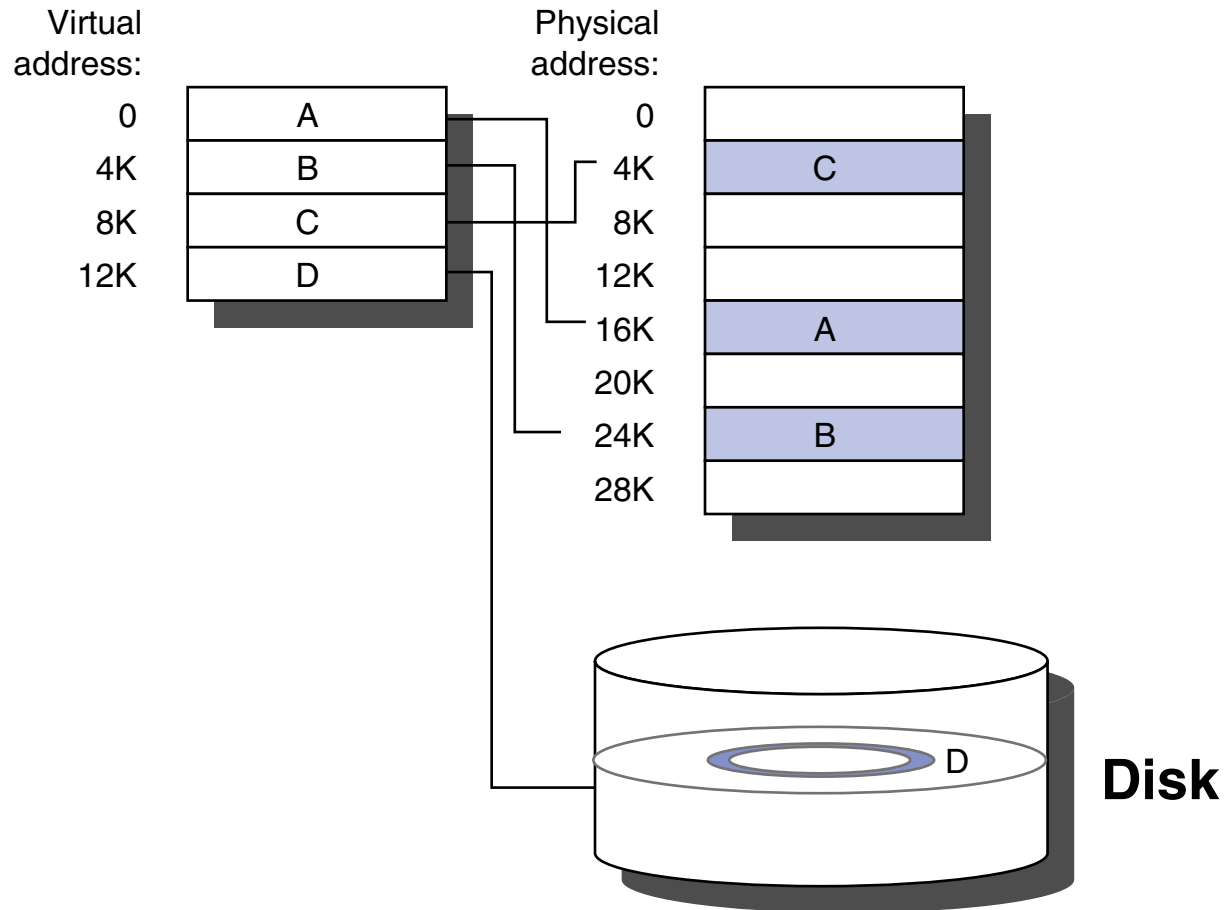
- The user
 - ▷ No virtual memory
 - ▷ Program overlays may be needed
 - ▷ The user must make explicit transfers of data blocks among levels
 - ▷ Context switching is accomplished by swapping entire processes
 - ▷ Requires very fast backup of memory (e.g., Cray's SSD)
 - ▷ Optimized for single-user mode
- The operating system
 - ▷ Virtual memory
 - In most OSs, VM is optimized for rotating magnetic disks
 - Solid state disks may work best with other algorithms
 - ▷ Context switching is performed by the kernel
 - ▷ Optimized for multi-user mode

VIRTUAL MEMORY

- Useful in a multiprogramming environment
 - ▷ Each process has its own logical address space
 - ▷ Logical to physical address translation on every memory reference
 - ▷ The operating system protects each process's physical address space
- Strategy: Main memory functions as a “cache” for magnetic disk
 - ▷ Keep most recently used data & instructions in main memory
- Size of “cached” blocks:
 - ▷ All microprocessors permit fixed-size blocks (**pages**)
 - Fixed size & relocation eliminate memory fragmentation
 - A miss is a page fault (penalty $\approx 10^5 \times$ cache miss penalty)
 - ▷ 80x86 architecture also permits variable-size blocks (**segments**)
 - Address translation and memory fragmentation are problems

Virtual memory

Physical main memory



A logical program in its contiguous virtual address space is shown on the left; it consists of four pages A, B, C, and D

Hennessy and Patterson, *Computer Architecture: A Quantitative Approach*, 2nd Edition

PROGRAMMING IMPLICATIONS OF HIERARCHICAL MEMORY

- Maintain instruction and data locality in the innermost loop
 - ▷ Program a short innermost loop
 - No conditional statements
 - Avoid function and subprogram calls
 - ◇ Inline code instead of calling a function
 - ◇ Pre-compute function values and store them in a table, if possible
(I once increased the speed of a program by a factor of 10^3 this way)
 - Use no statements that throw interrupts
 - ◇ No I/O in the innermost loop
 - ▷ To maximize data locality:
 - Arrange unit stride (or the shortest possible stride) for all arrays that are referenced in the innermost loop

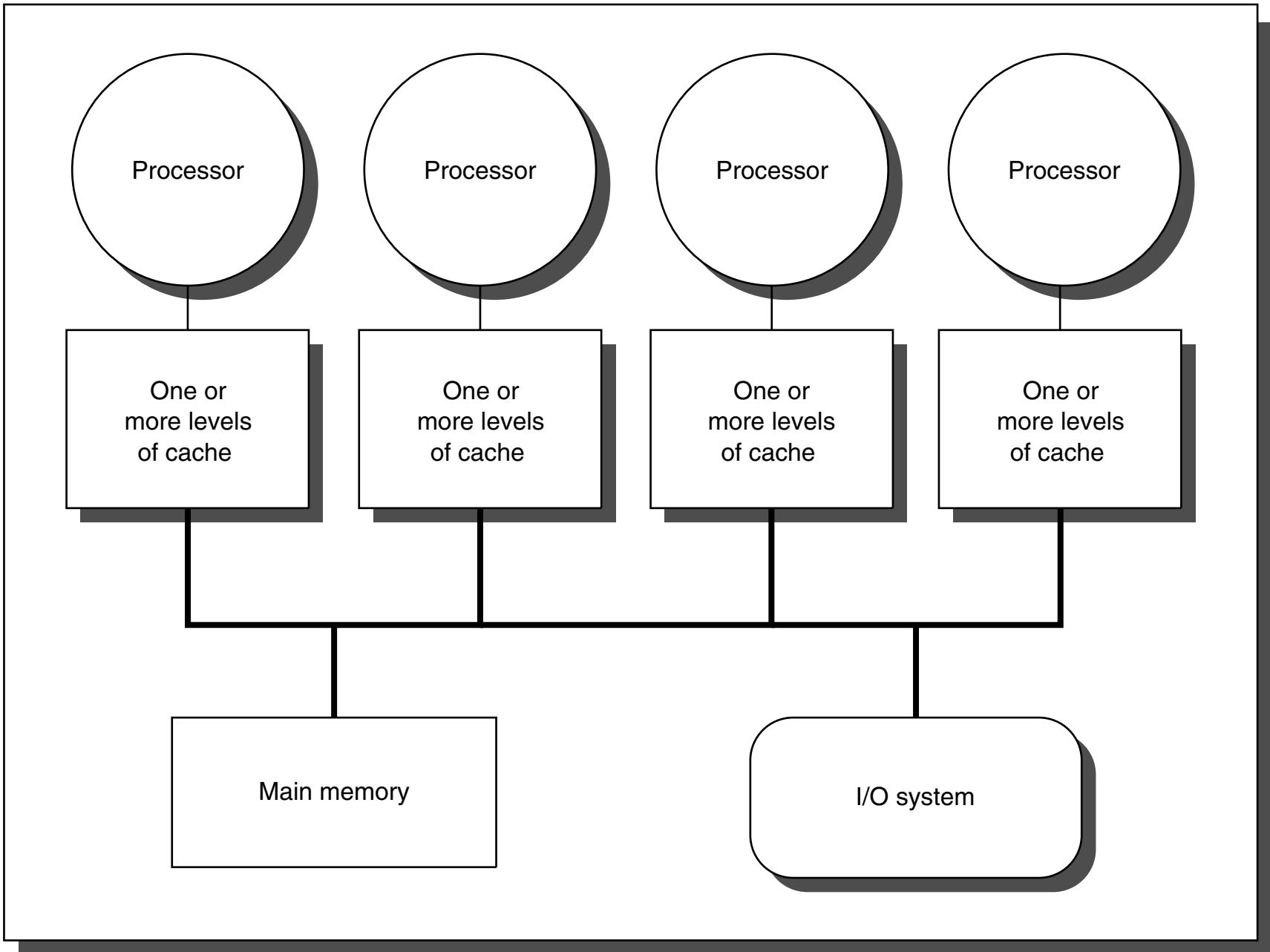
PARALLEL PROCESSING

- Motivated by need for speeds greater than can be obtained with single processors at reasonable cost (see next slide)
- Parallel computer architectures
 - ▷ Shared-memory multiprogramming
 - Multiple threads, running on one or more processors
 - ▷ Multiprocessors
 - A single OS kernel controls all processors
 - ◇ The processor that runs the kernel can't be used for computation
 - Memory: Shared, distributed or both
 - Communication: Shared memory or message passing
 - ▷ Multicomputers
 - Each processor is an autonomous computer with its own OS kernel, memory and I/O
 - Communication: Message passing

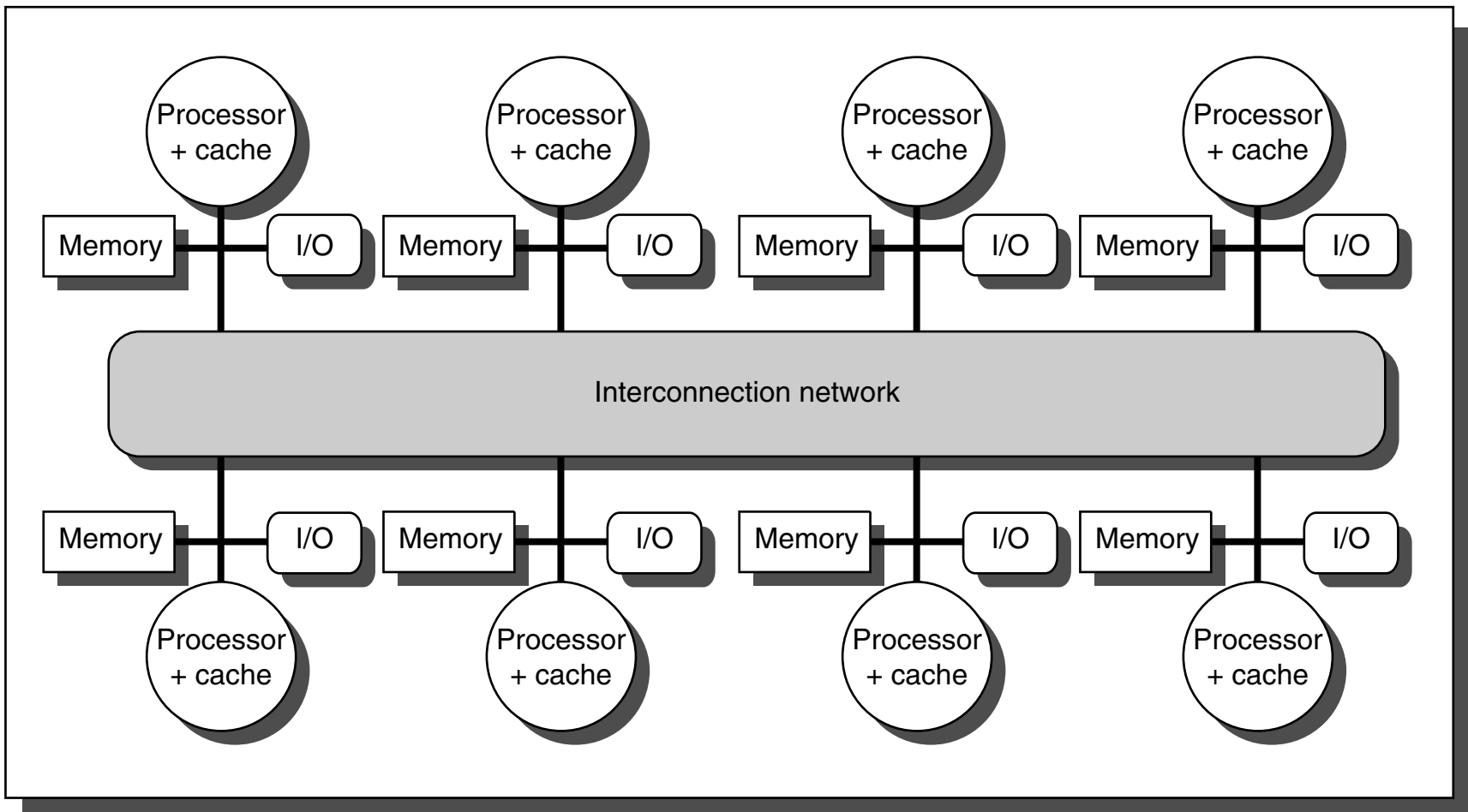
COMPUTATIONAL PROBLEMS

- Numerical simulation has become as important as experimentation
 - ▷ Permits exploration of a much larger region in parameter space
 - ▷ All-numerical designs
- The most CPU-intensive application areas include:
 - ▷ Computational fluid dynamics
 - ▷ Computational electromagnetics
 - ▷ Molecular modeling
- Typical problem sizes (in double-precision floating-point operations) and execution times (in CPU-days @ 10^9 FLOPS):

Problem size	Execution time (CPU-days)
10^{14}	1.16
10^{15}	11.57
10^{16}	115.7
10^{17}	1157

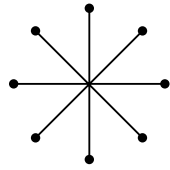


A centralized, shared-memory multiprocessor

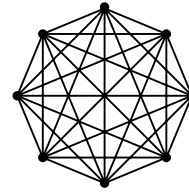


A distributed-memory multicomputer

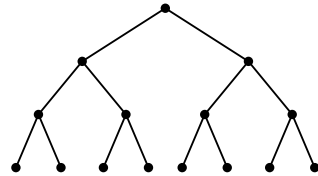
Interconnection topologies



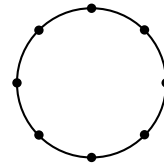
star



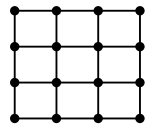
full interconnection



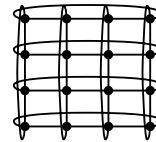
tree



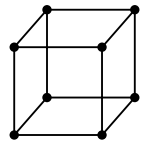
ring



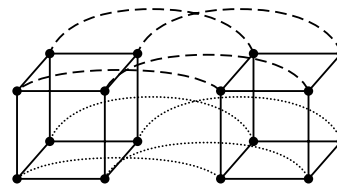
grid



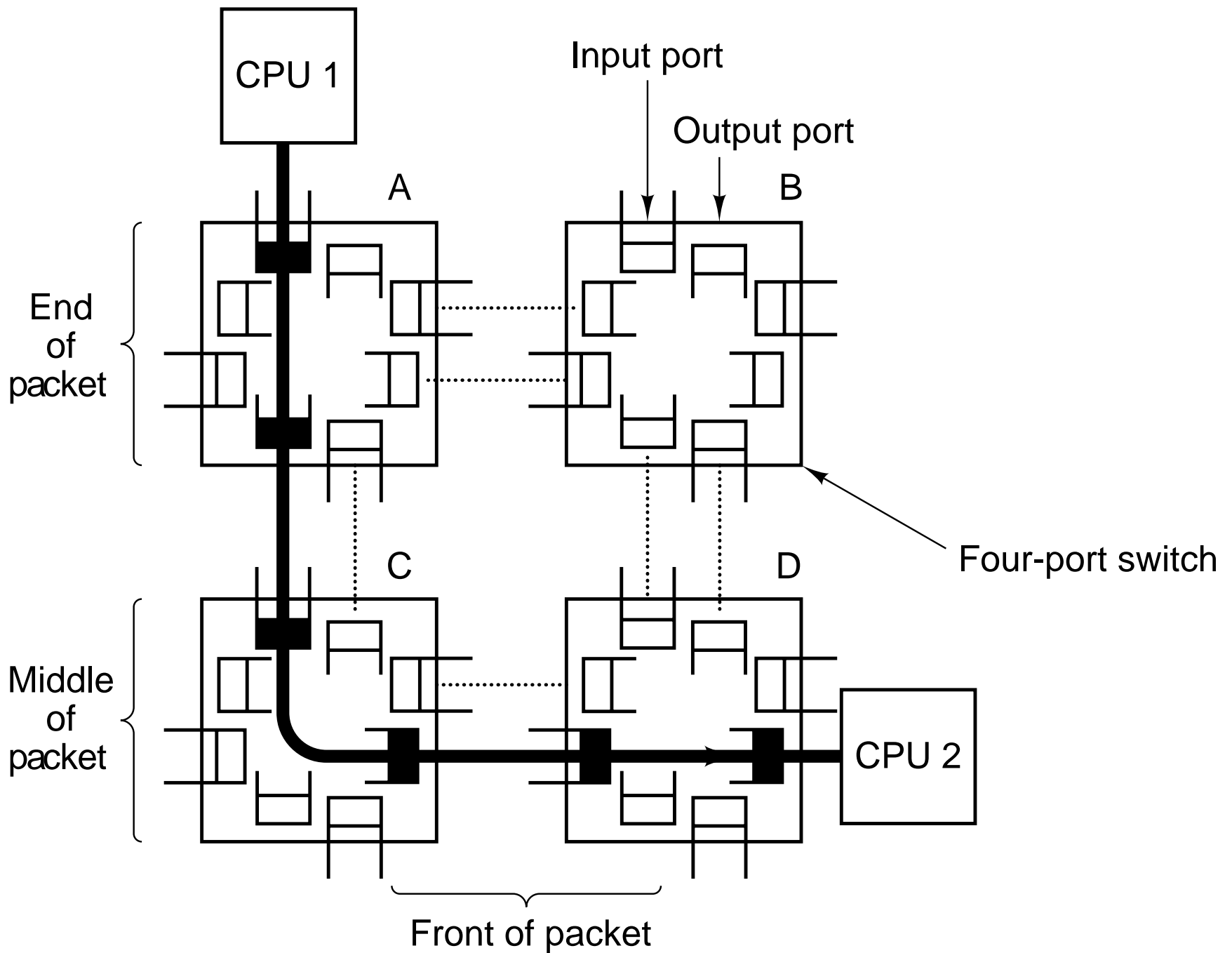
double torus



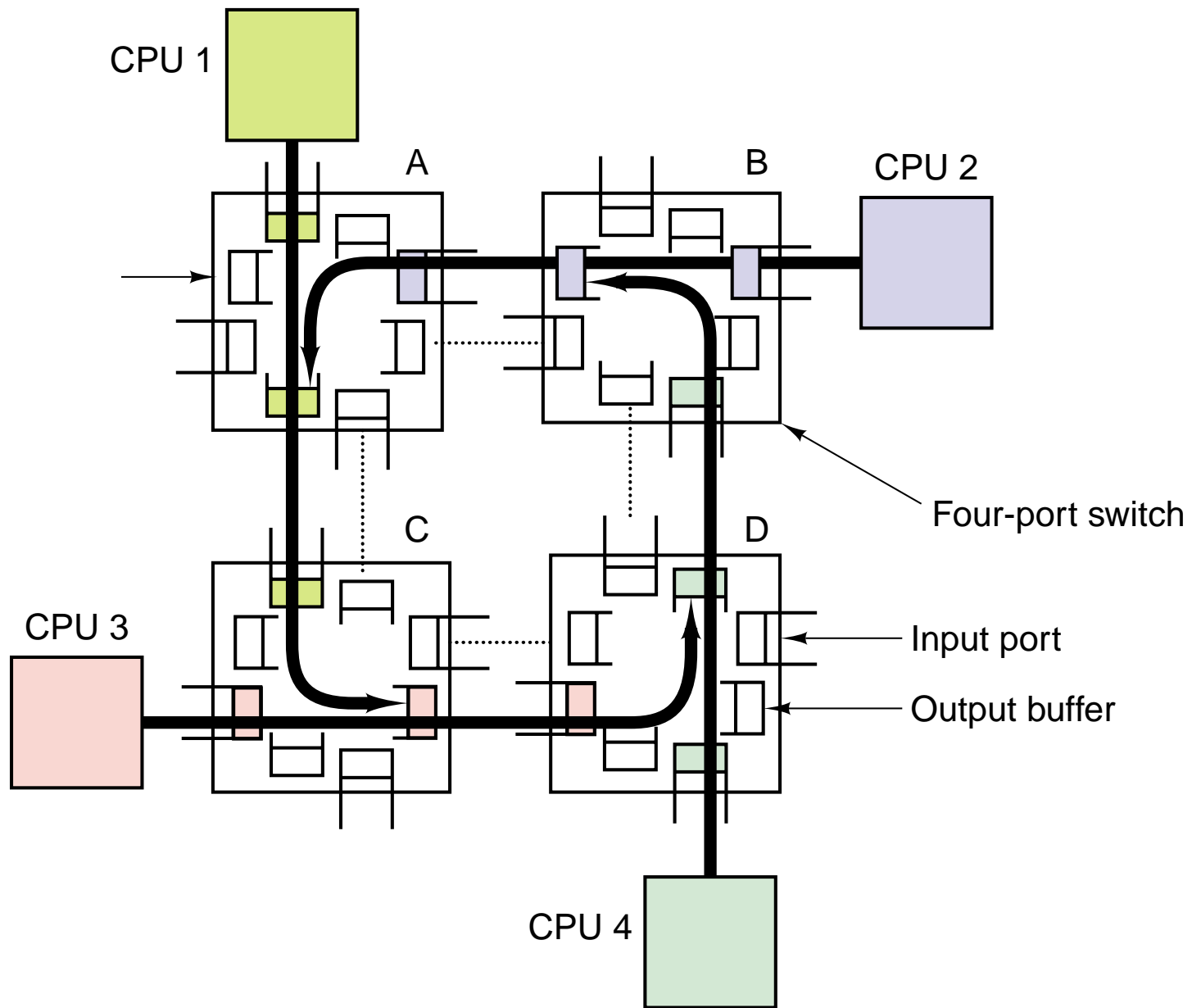
cube



4-D hypercube



Interconnection network consisting of a 4-switch square grid

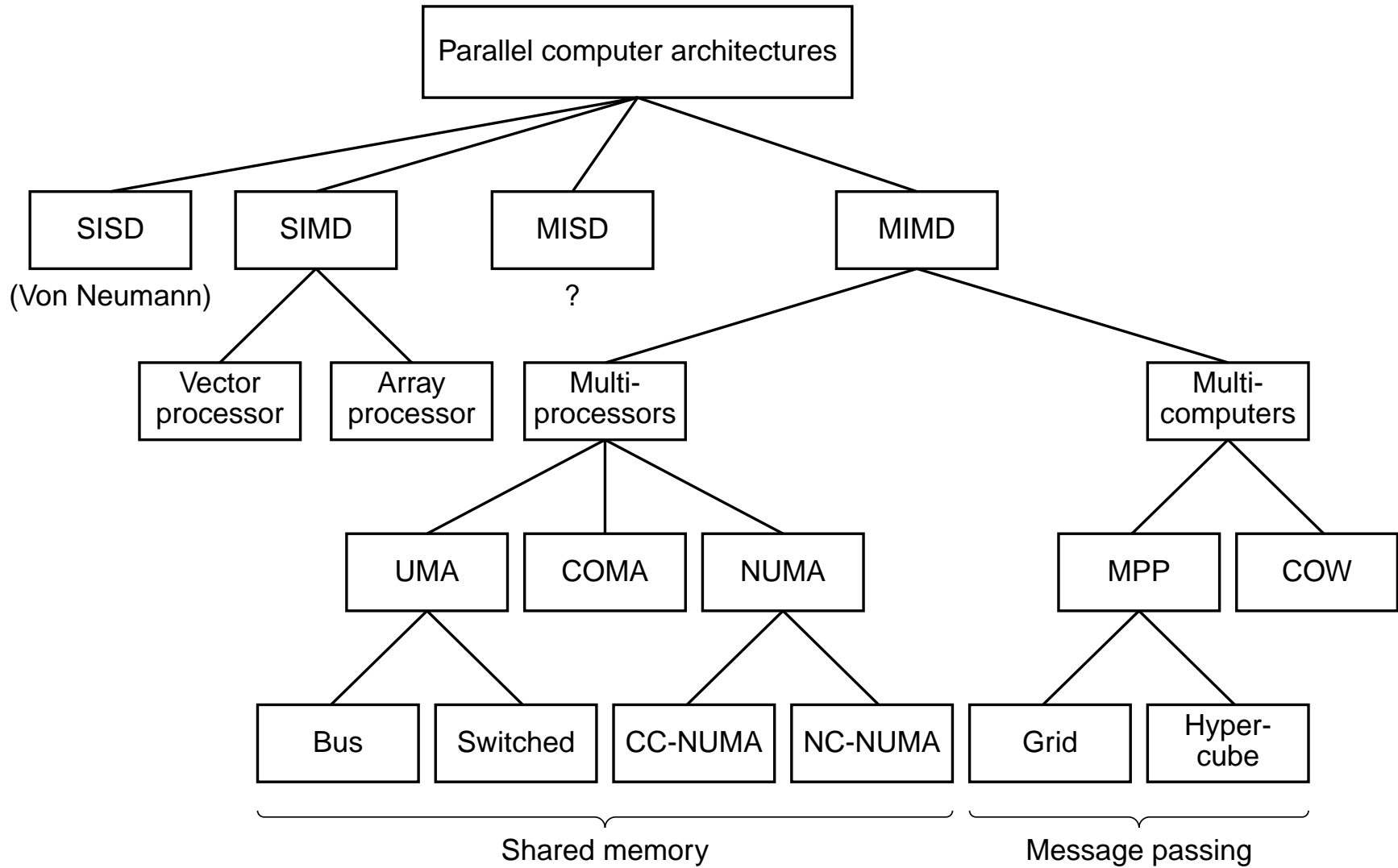


Deadlock in a circuit-switched interconnection network

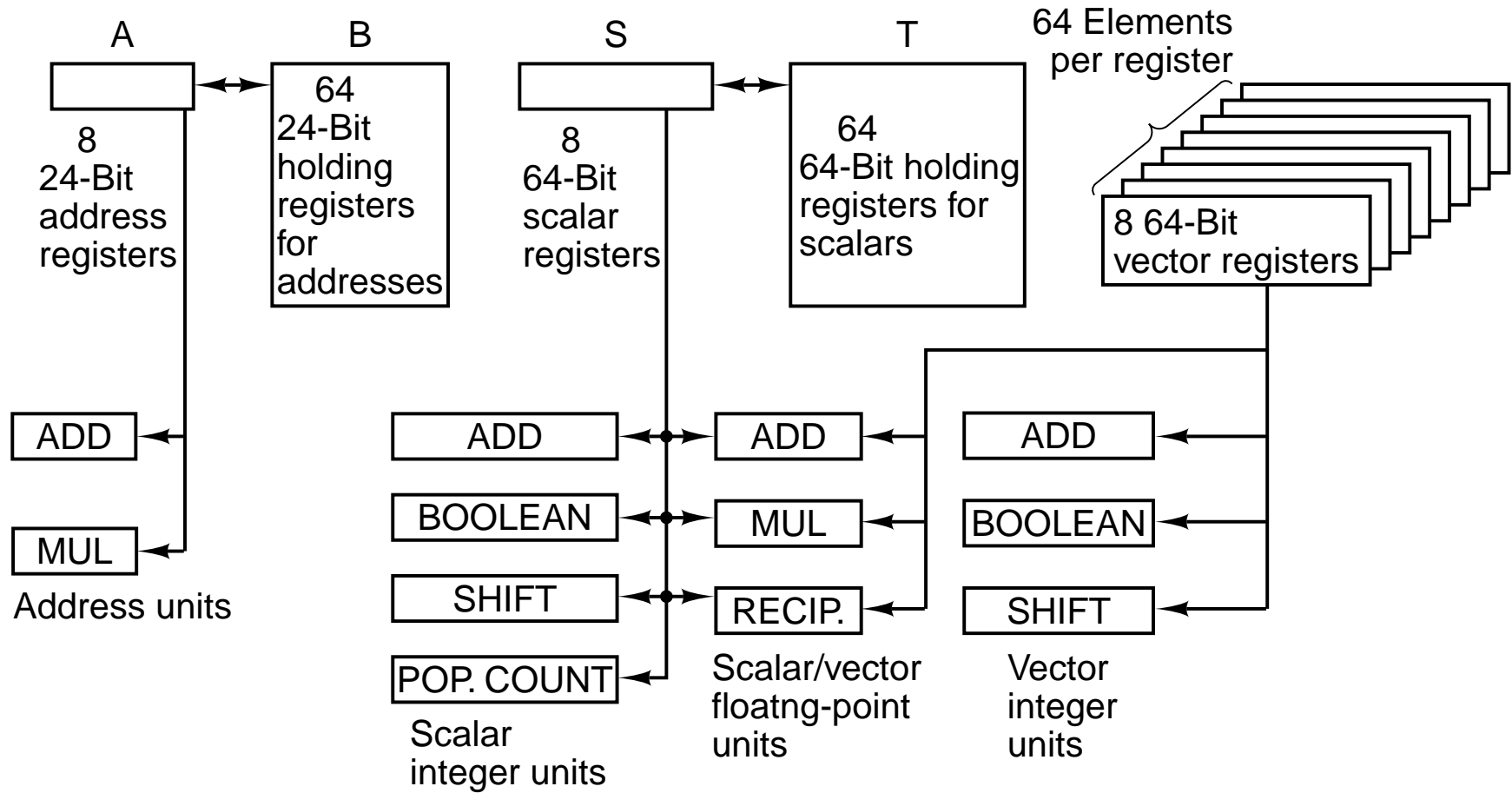
Flynn's taxonomy of parallel computers

Instruction streams	Data streams	Name	Examples
1	1	SISD	Classical Von Neumann machine
1	Multiple	SIMD	Vector supercomputer, array processor
Multiple	1	MISD	Arguably none
Multiple	Multiple	MIMD	Multiprocessor, multicomputer

A taxonomy of parallel computers



CRAY-1 registers and functional units



INSTRUCTION SET DESIGN

- A key step in both hardware and software design
 - ▷ Software:
 - The instruction set architecture (ISA), and its implementation, determine what kinds of programs perform well
 - ▷ Hardware:
 - Up to now, we have worked at the gate level
 - In order to design or understand an ISA, we have to work at a higher level of organization, using modules or functional units
 - ◇ Main memory
 - ◇ Cache and cache controller
 - ◇ Register file
 - ◇ Integer unit (ALU)
 - ◇ Floating-point unit (FPU)
 - ◇ Branch unit
 - ◇ Exception/interrupt processing unit
 - ◇ Vector unit

INSTRUCTION SET DESIGN ISSUES

- What operations should be provided in hardware?
 - ▷ An optimization problem
 - ▷ Any computation can be done using only load/store/increment/branch
 - Resulting code: large, with many memory references \Rightarrow slow
- How many / what kinds of operands?
 - ▷ Must provide for one operand; should provide for two
 - ▷ Examples: $a \mapsto \bar{a}$, $\langle a, b \rangle \mapsto c$
- What addressing modes should be implemented?
 - ▷ Choices made here affect complexity of both datapath and control
 - ▷ Fixed-length vs. variable-length instructions
- How to encode operations in a few bits?
 - ▷ Hardware design issue: how to translate operation encoding and addressing into control signals?
- How should interrupts and exceptions be handled?

KINDS OF INSTRUCTION SET ARCHITECTURES

- Accumulator
 - ▷ Requires only one address
 - ▷ Example: `add x` means `acc` \mapsto `acc + M[x]`
- Stack
 - ▷ Requires no explicit memory addresses
 - ▷ Example: `add` means `tos` \mapsto `tos + next`
- General-purpose register set
 - ▷ Some operands may be in memory
 - ▷ Example of adding two numbers in memory locations `a` and `b`:
`load r1,a; add r1,b; store r1,c`
 - ▷ Load/store architectures
 - All arithmetic/logical operations use register operands
 - Only `load` and `store` instructions reference memory

LOAD-STORE ARCHITECTURES

- All instructions other than **load** and **store** operate only on:
 - ▷ Data in registers
 - ▷ Immediate data (data encoded in the assembled instruction)
- Advantages of RISC load-store architectures:
 - ▷ Speed
 - ALU operations execute in 1 clock period
 - ▷ Small number of addressing modes
 - MIPS processors (RISC): 1 form of add instruction
 - VAX processors (CISC): $\approx 20,000$ forms of add instruction
 - ▷ Simple control unit
 - Short clock period
 - Easily & quickly scalable IC design

ADVANTAGES AND DISADVANTAGES OF VARIOUS ISAs

- Accumulator
 - + Minimizes internal state of CPU
 - Maximizes memory-to-CPU traffic
- Stack
 - + Reverse-polish expression evaluation (like HP calculator)
 - Stack can't be randomly accessed \Rightarrow bottleneck
- General-purpose register set
 - + Most general model for code generation
 - Each operand must be named \Rightarrow longer instructions
- Load/store
 - + Fixed-length instruction encoding, with one addressing mode
 - ▷ Simplifies hardware datapath and control \Rightarrow maximum speed
 - Increases total instruction count

WHY REGISTER ARCHITECTURES ARE DOMINANT

- Since 1980, logic circuits have become much faster than dynamic RAM
 - ▷ Registers are faster than main memory
 - ▷ Registers are faster than a stack
 - A stack is built in main memory
 - Operands that are in registers can be used in any order
 - ▷ Registers can hold variables as they are updated in a multi-step computation
 - Reduction of main memory traffic makes computations faster
 - Register addresses (typically 5 bits) are shorter than main memory addresses (typically 32 to 64 bits)
 - ◇ Code density (information per bit in an instruction) is higher in load-store architectures than in architectures that allow memory-resident operands
 - ◇ Short, fixed-length addressing \Rightarrow fixed-length instructions
 - For a register machine, one can optimize the most frequently used instructions (which are also simple, by experiment)

REDUCED INSTRUCTION SETS

- Several different groups “discovered” more or less independently that ~ 10 instructions account for more than 90% of the executions
 - ▷ Suppose that one can design hardware that executes almost all of the top 10 instructions in one clock period, instead of 10
 - ▷ Amdahl’s Law predicts that the overall speedup is

$$\text{Overall speedup} = \frac{1}{(1 - 0.9) + \frac{0.9}{10}} = 5.26$$

- ▷ Result: **REDUCED INSTRUCTION SET COMPUTER (RISC)** architectures such as the MIPS R2000
- ▷ General principle: **“Make the common case fast”**
- ▷ When Sun Microsystems changed from Motorola 68020 (CISC) processors to SPARC (RISC) processors *at the same clock frequency*, performance improved by a factor of ~ 5

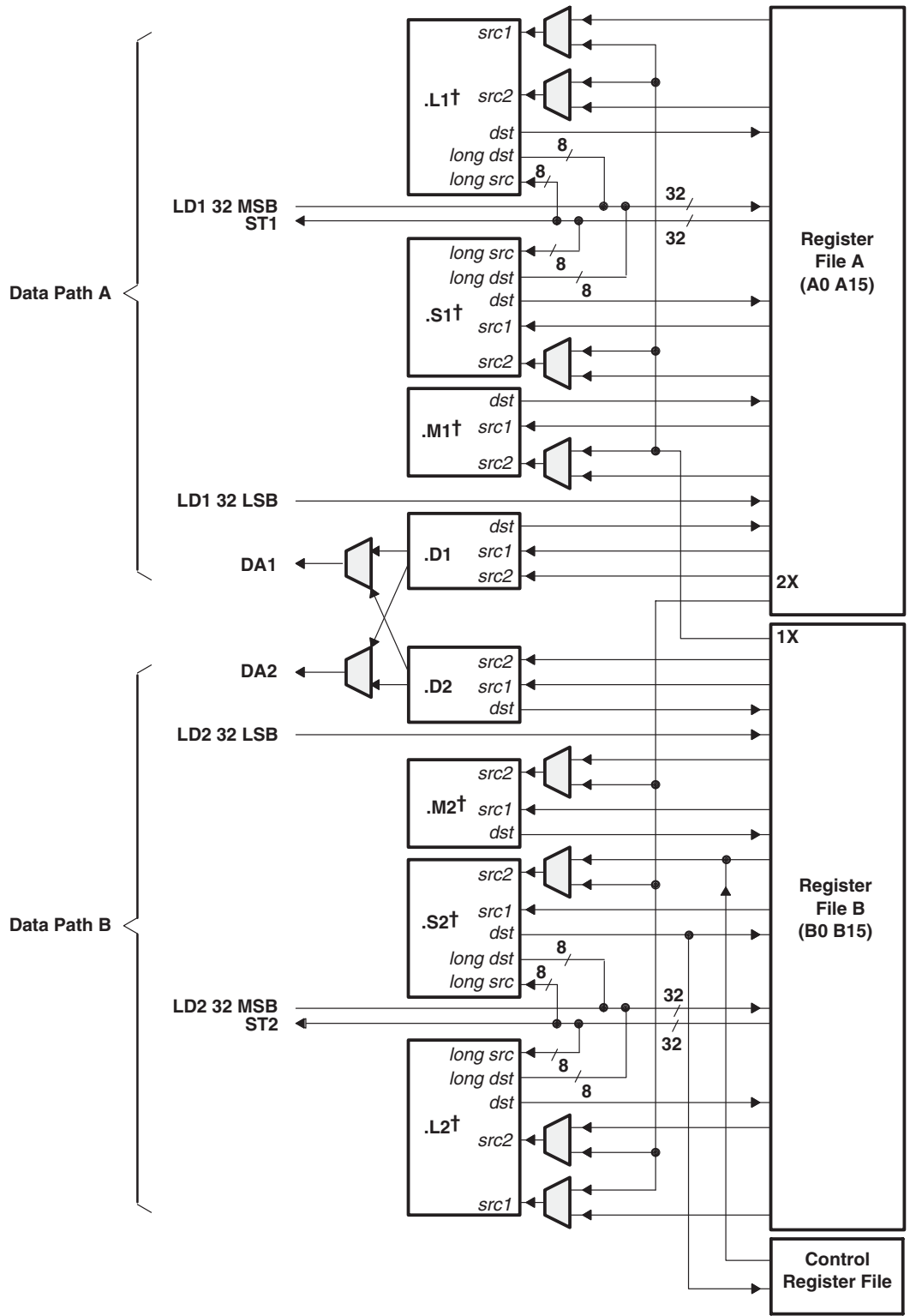
The top 10 instructions for the 80x86

Rank	80x86 instruction	Integer average (% total executed)
1	load	22%
2	conditional branch	20%
3	compare	16%
4	store	12%
5	add	8%
6	and	6%
7	sub	5%
8	move register-register	4%
9	call	1%
10	return	1%
Total		96%

VLIW ARCHITECTURES

- VLIW = **Very Long Instruction Word**
 - ▷ Goal: Reduce CPI (clocks/instruction) below 1.0
 - ▷ Architecture: Multiple streams of execution, supported by multiple sets of functional units
 - ▷ Implementation: Compiler must produce code that takes advantage of multiple functional units
 - ▷ Fine-grained parallelism
 - ▷ Implemented in recent DSP's

CPU (DSP core) description (continued)

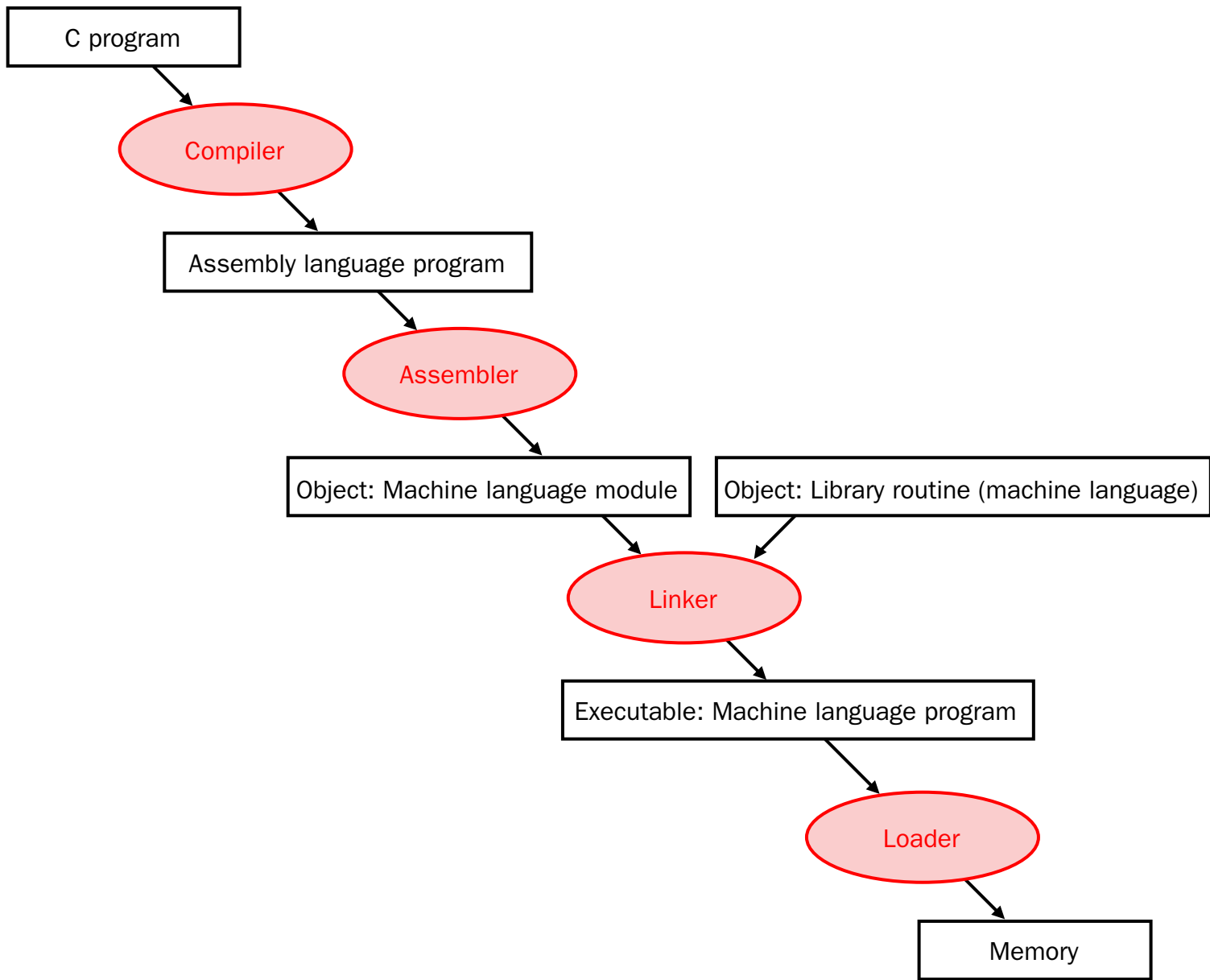


† In addition to fixed-point instructions, these functional units execute floating-point instructions.

Figure 1. TMS320C67x CPU (DSP Core) Data Paths

STEPS IN CREATING AN EXECUTABLE PROGRAM

- Creation of source “code”
 - ▷ Accomplished with a text editor or a programming environment
- Compilation (if source program is in a higher-level language)
 - ▷ Includes one of various levels of optimization
 - ▷ Result of compilation may be an assembly-language program
- Assembly produces an **object module**
 - ▷ Human-readable instructions and macros \mapsto machine language
- Linking (a.k.a. link editing) produces a **load module**
 - ▷ Resolution of calls to user and library functions
 - ▷ Unresolved function calls produce run-time exceptions
- Loading produces a memory-resident executable module
 - ▷ Memory references to code and data are updated to reflect actual locations
 - ▷ Executable program is copied into memory



INPUT/OUTPUT (I/O) SUBSYSTEMS

- Overview of I/O performance measurement and analysis
- Processor interface issues
- Buses
- Types and characteristics of I/O devices
 - ▷ Hard disk storage
 - ▷ Network interfaces
- I/O system design

MOTIVATION FOR STUDYING I/O

- CPU performance improves by 50% to 100% per year
- I/O systems' performance improvements are limited by physics (in some cases)
 - ▷ Mechanical delays (disk drives):
Latency improvement is of order 5% per year
 - ▷ Electrical and optical phenomena (dispersion, attenuation, crosstalk): Improvement is 5% to 25% per year
- Amdahl's law implies that, sooner or later, most of the latency will be due to the part that is hardest to improve
 - ▷ Given: 10% of instructions perform I/O, CPU is 10x faster
 - ▷ Improvement is only 5x \Rightarrow lose 50% of improvement
- **I/O bottleneck lowers the value of CPU improvements**
 - ▷ As technology evolves, a diminishing fraction of total latency is due to the CPU

I/O PERFORMANCE METRICS

- **Bandwidth** (bits or bytes per second):
 - ▷ Peak
 - ▷ Sustained
 - ▷ Useful for buses and networks
- **Throughput** (I/O processes per second)
 - ▷ Useful for file serving and transaction processing
- **Latency** = total time for an I/O process from start to finish
 - ▷ Most important to users
 - Latency too great \Rightarrow user loses train of thought
 - Latency
 - = controller time + wait time + $\frac{\text{no. bytes}}{\text{bandwidth}}$ + CPU time
 - overlap

PROCESSOR INTERFACE ISSUES

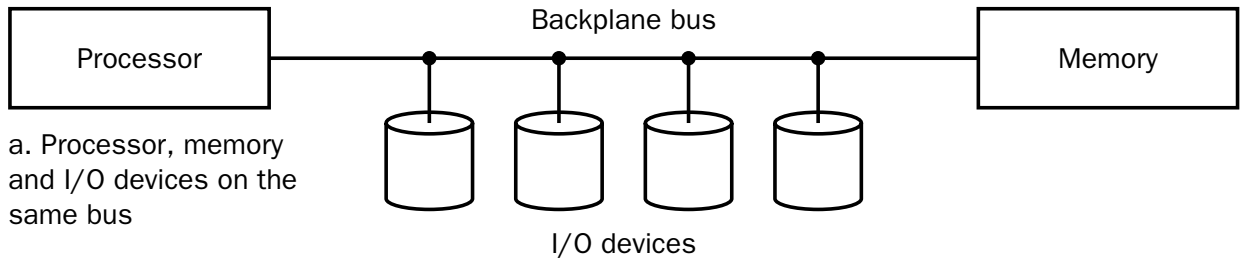
- Interconnections
 - ▷ Buses
- Processor interface
 - ▷ Interrupts
 - ▷ Memory-mapped I/O
- I/O control structures
 - ▷ Polling
 - ▷ Interrupts
 - ▷ DMA
 - ▷ I/O controllers
 - ▷ I/O processors
- Capacity, access time, bandwidth, cost

BUSES

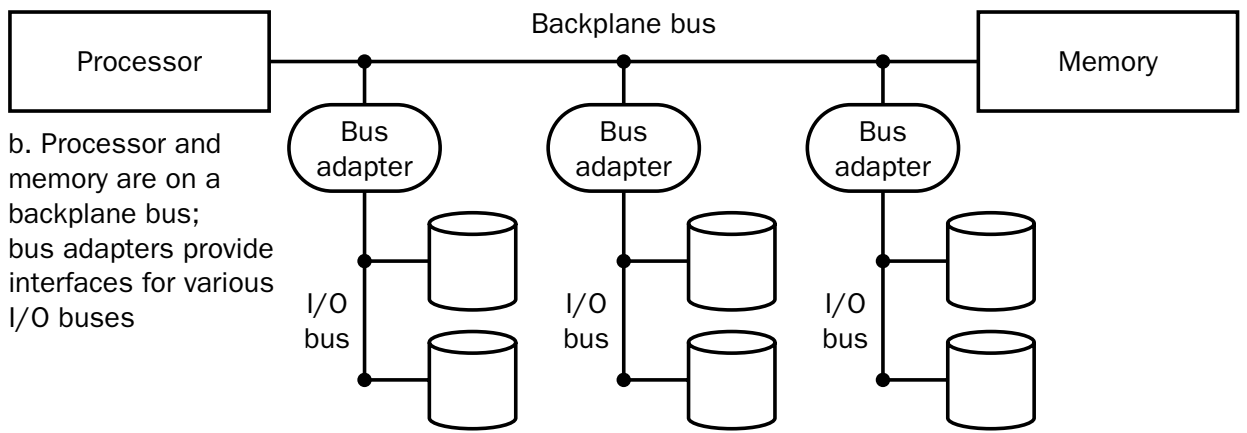
- **Bus:** A communication link shared by multiple subsystems
 - ▷ Physically: Parallel conductors (traces on die or PC board; cable)
 - ▷ Advantages:
 - Low cost (compared to point-to-point wiring)
 - Versatility of interconnections
 - ▷ Disadvantages:
 - Electrical problems \Rightarrow short length
 - ◇ Bus skew
 - ◇ Dispersion
 - ◇ Crosstalk
 - Shared resource \Rightarrow contention
 - ▷ Organization:
 - Control lines to signal & acknowledge requests
 - Data lines to carry addresses, data or commands

PROCESSOR-I/O INTERFACE BUS TYPES

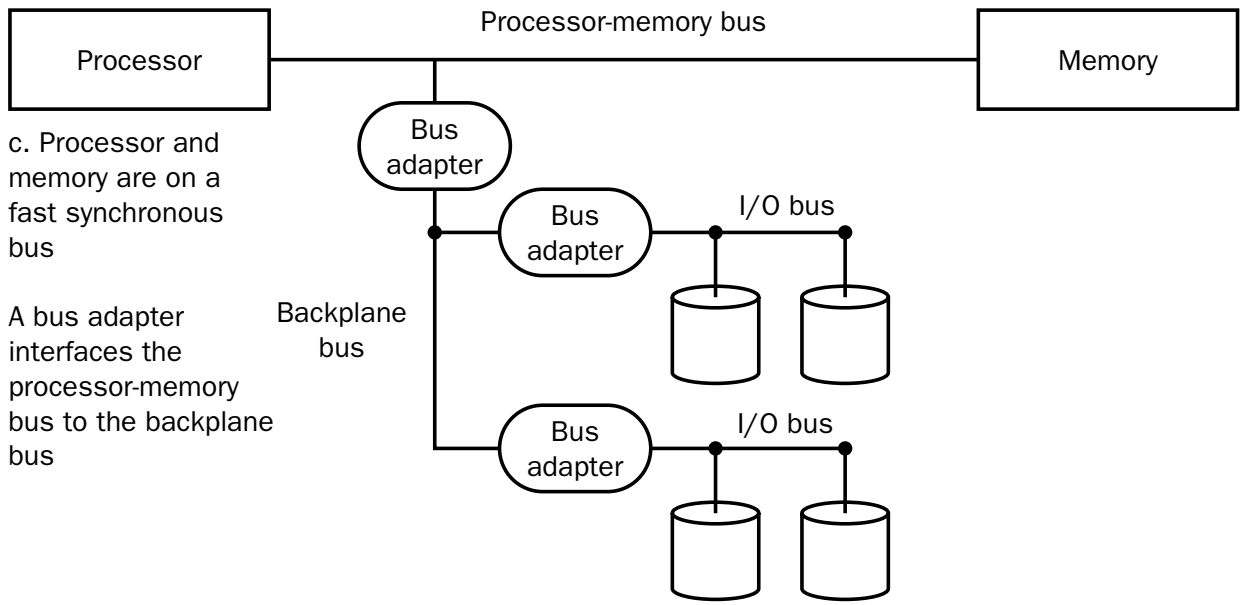
- Backplane bus
 - ▷ Processor, memory and I/O devices coexist on the same bus
 - ▷ In olden times, often built into the backplane of a computer
 - An interconnection structure that was part of the chassis
 - ▷ Processor architecture includes explicit I/O instructions (IN, OUT)
 - ▷ Standard backplane buses: VMEbus, Multibus, NuBus, PCI, ISA (Industry Standard Architecture) bus
- I/O bus
 - ▷ Examples: IDE, SCSI



a. Processor, memory and I/O devices on the same bus

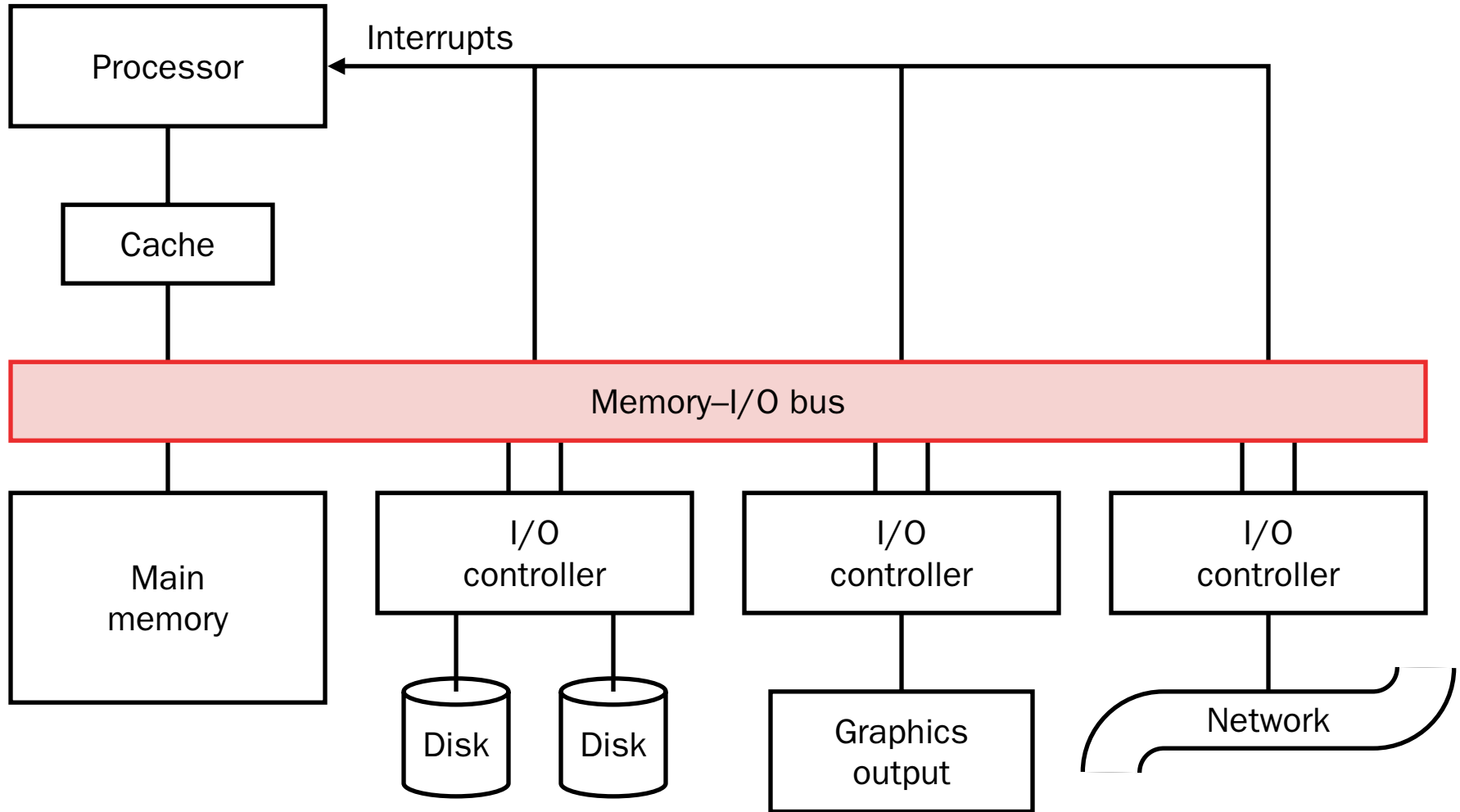


b. Processor and memory are on a backplane bus; bus adapters provide interfaces for various I/O buses

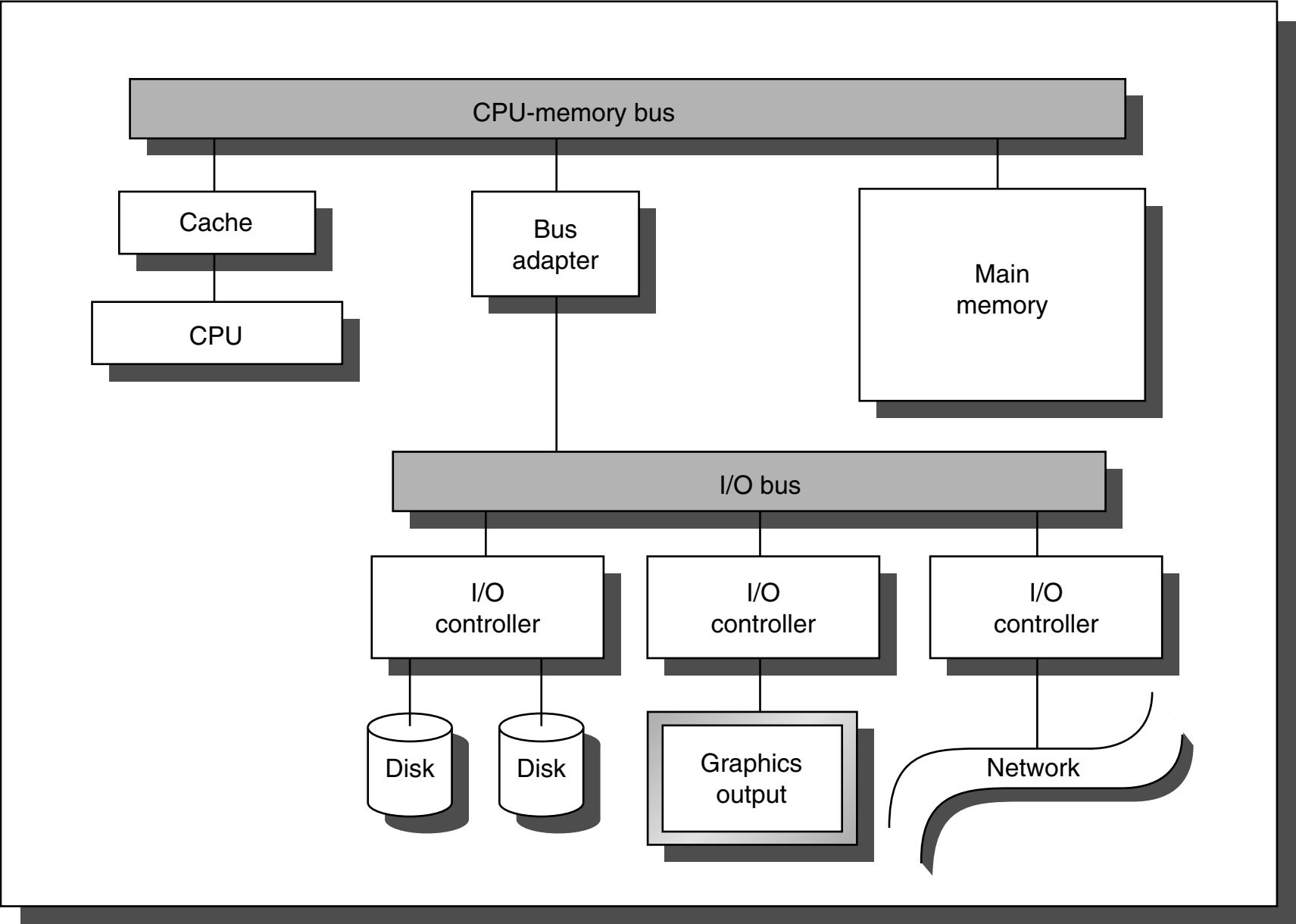


c. Processor and memory are on a fast synchronous bus
A bus adapter interfaces the processor-memory bus to the backplane bus

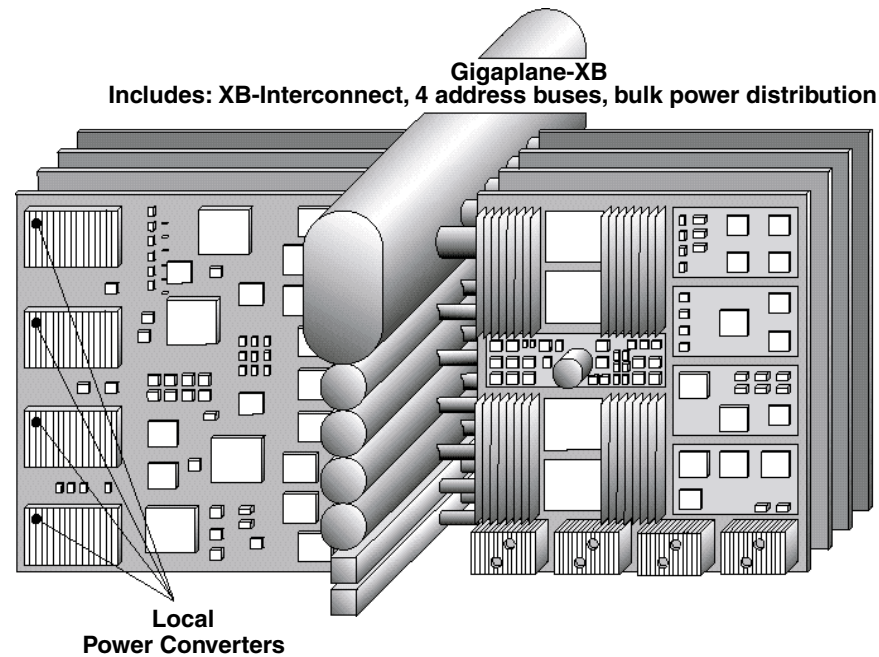
I/O SYSTEM USING ONLY A BACKPLANE BUS



I/O SYSTEM USING AN I/O BUS



Enterprise 10000 hardware architecture



- Data is packet-switched using a crossbar
- Addresses are broadcast