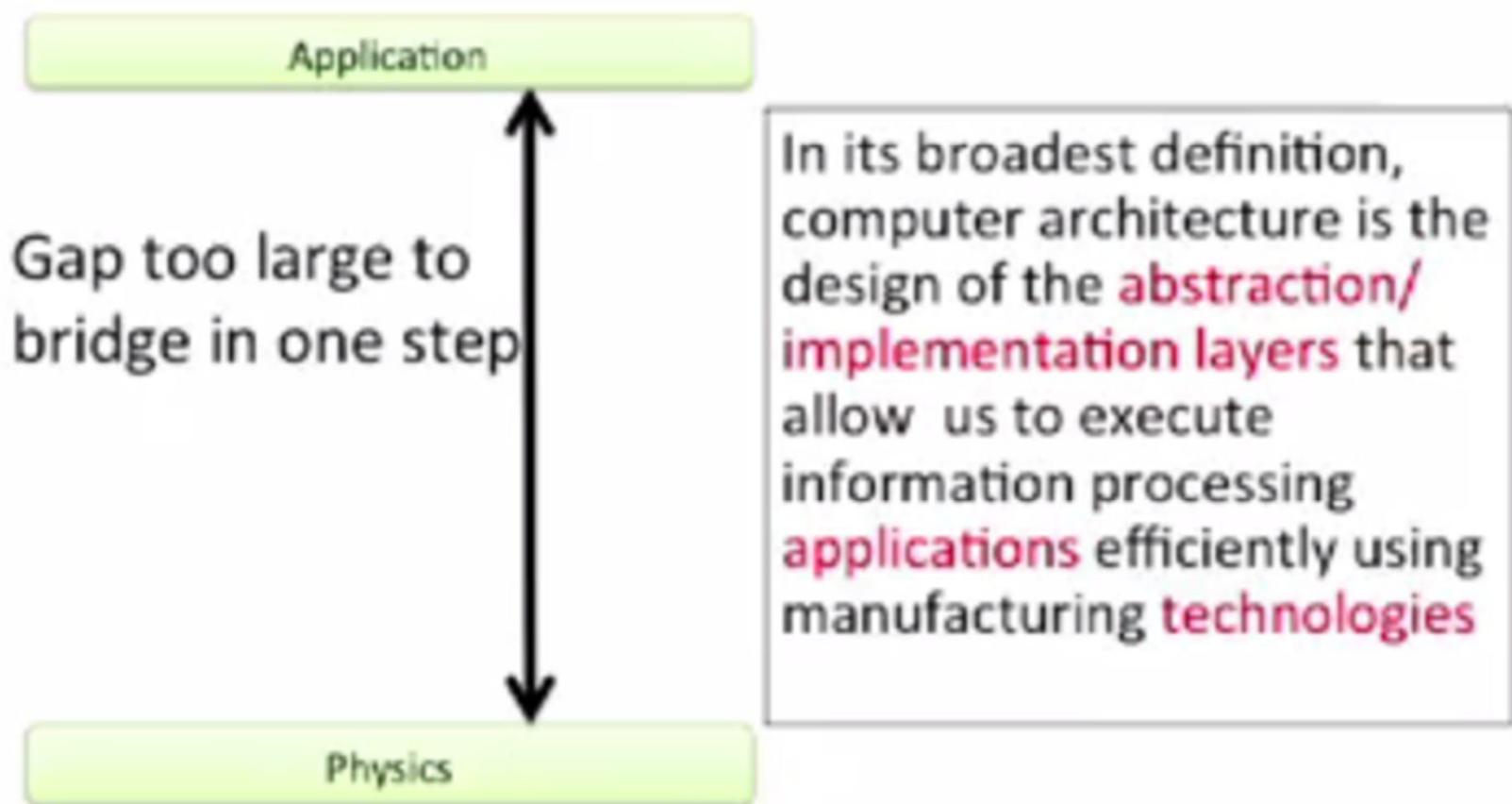


What is Computer Architecture?



Abstractions in Modern Computing Systems

Application

Algorithm

Programming Language

Operating System/Virtual Machines

Instruction Set Architecture

Microarchitecture

Register-Transfer Level

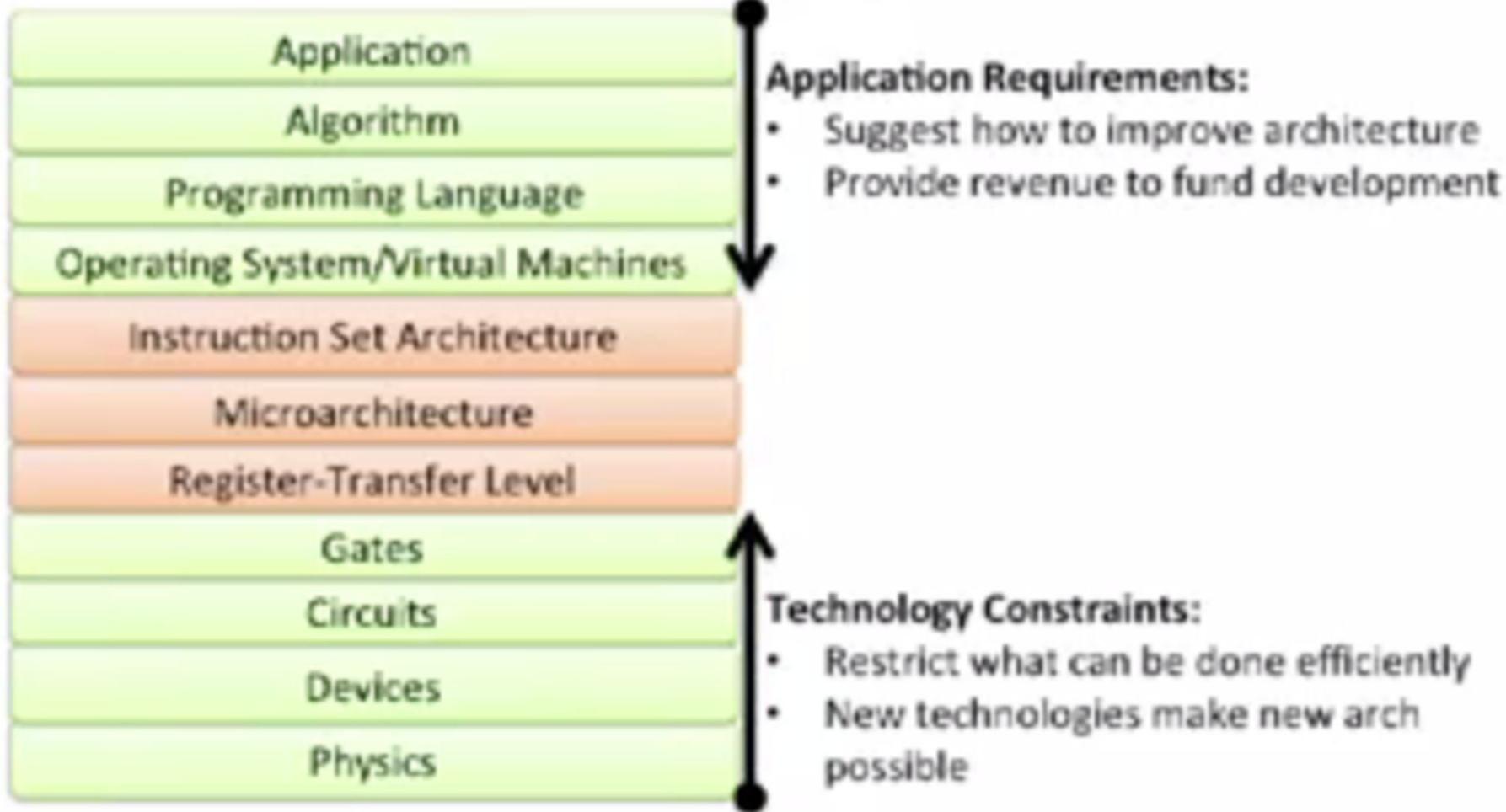
Gates

Circuits

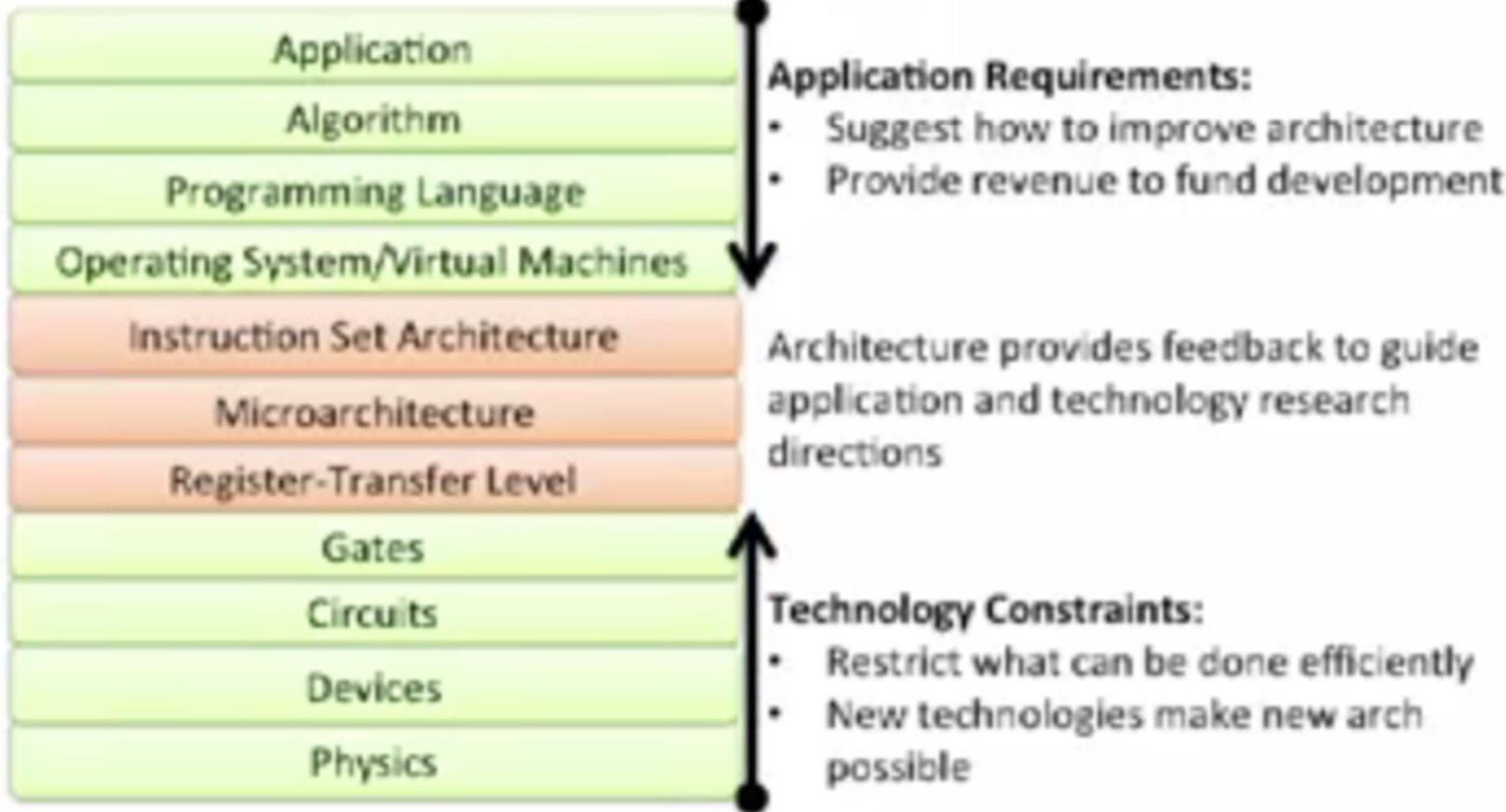
Devices

Physics

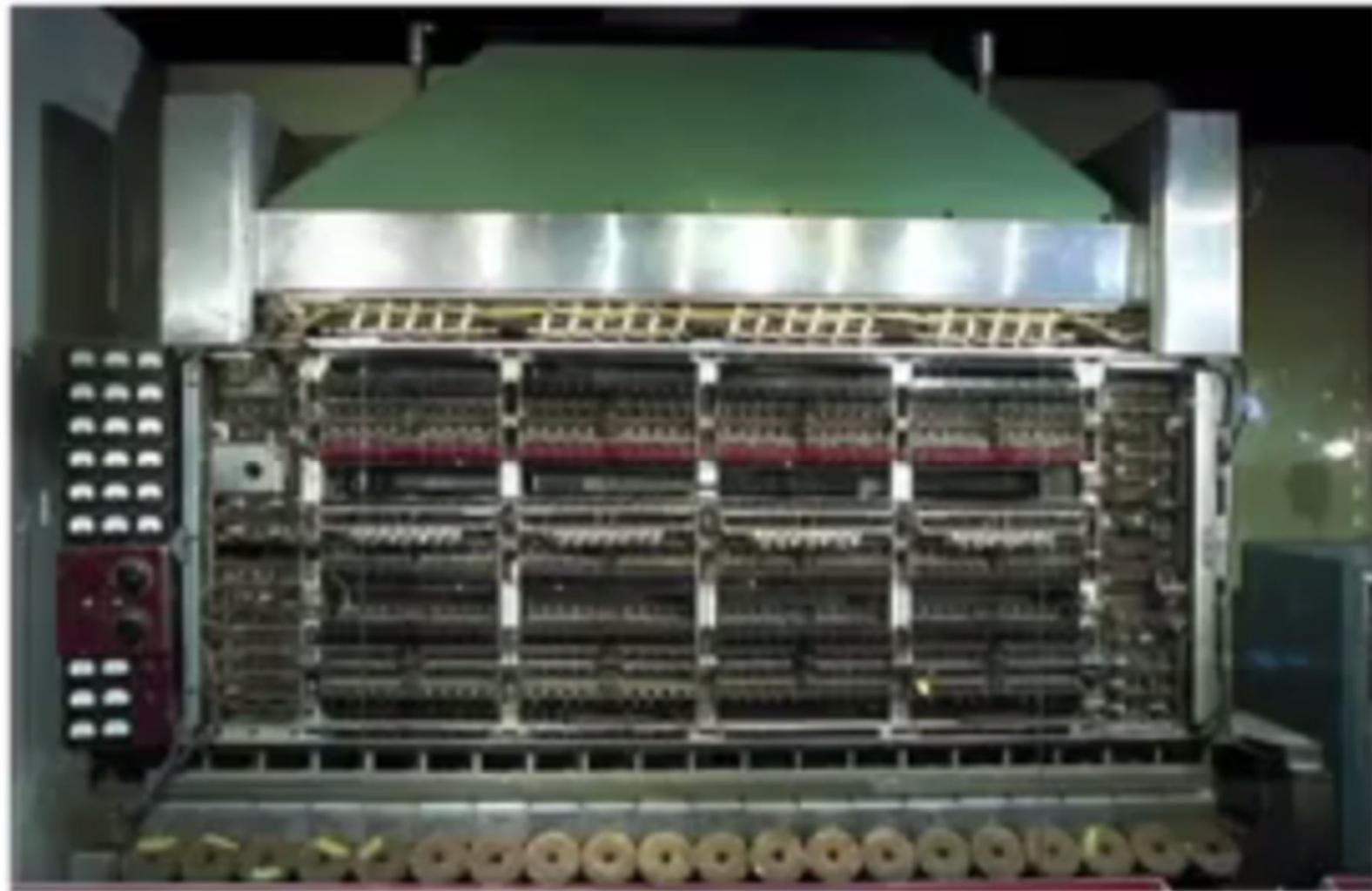
Computer Architecture is Constantly Changing



Computer Architecture is Constantly Changing



Computers Then...



IAS Machine. Design directed by John von Neumann.

First booted in Princeton NJ in 1952

Smithsonian Institution Archives (Smithsonian Image 95-06151)

Computers Now

- Sensor Networks
- Cameras
- Smartphones
- Mobile Audio Players
- Laptops
- Autonomous Cars
- Servers
- Game Players
- Routers
- Flying UAVs
- GPS
- eBooks
- Tablets
- Set-top Boxes

Architecture vs. Microarchitecture

“Architecture”/Instruction Set Architecture:

- Programmer visible state (Memory & Register)
- Operations (Instructions and how they work)
- Execution Semantics (interrupts)
- Input/Output
- Data Types/Sizes

Microarchitecture/Organization:

- Tradeoffs on how to implement ISA for some metric (Speed, Energy, Cost)
- Examples: Pipeline depth, number of pipelines, cache size, silicon area, peak power, execution ordering, bus widths, ALU widths

Computer Architecture

Logical aspects of system implementation as seen by the programmer; such as, instruction sets (ISA) and formats, opcode, data types, addressing modes and I/O.

Instruction set architecture (ISA) is different from “microarchitecture”, which consist of various processor design techniques used to implement the instruction set.

Computers with different microarchitectures can share a common instruction set.

For example, the Intel Pentium and the AMD Athlon implement nearly identical versions of the x86 instruction set, but have radically different internal designs.

Computer architecture is the conceptual design and fundamental operational structure of a computer system. It is a **functional description** of requirements and design implementations for the various parts of a computer.

It is the science and art of selecting and interconnecting hardware components to create computers that meet functional, performance and cost goals.

It deals with the architectural attributes like physical address memory, CPU and how they should be designed and **made to coordinate with each other** keeping the goals in mind.

Analogy: “building the design and architecture of house” – architecture may take more time due to planning and then organization is building house by bricks or by latest technology keeping the basic layout and architecture of house in mind.

Computer architecture comes before computer organization.

Computer organization (CO) is how operational attributes are linked together and contribute to realise the architectural specifications.

CO encompasses all physical aspects of computer systems

e.g. Circuit design, control signals, memory types.

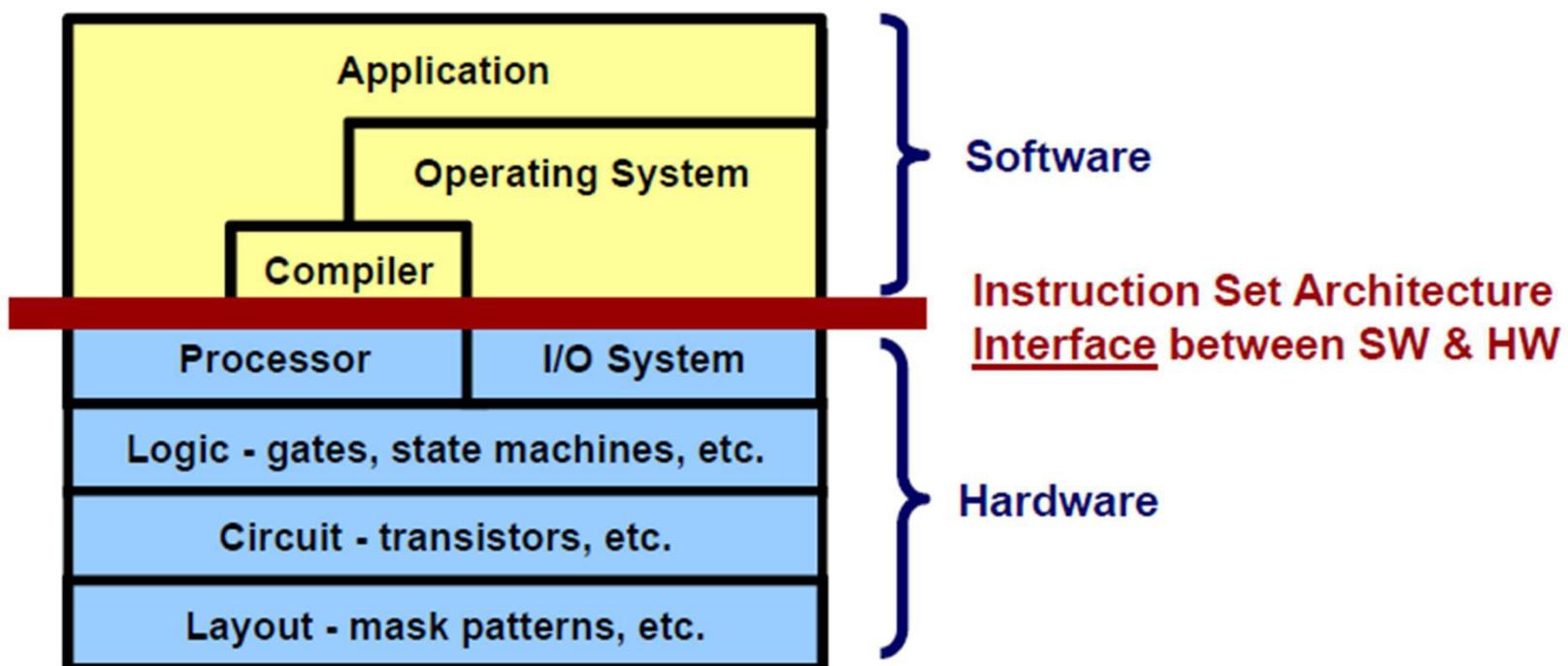
Microarchitecture, also known as **Computer organization** is a lower level, more concrete and detailed, description of the system that involves how the **constituent parts of the system** are **interconnected** and **how they interoperate** in order to implement the ISA.

The size of a computer's cache, for example, is an organizational issue that generally has nothing to do with the ISA.

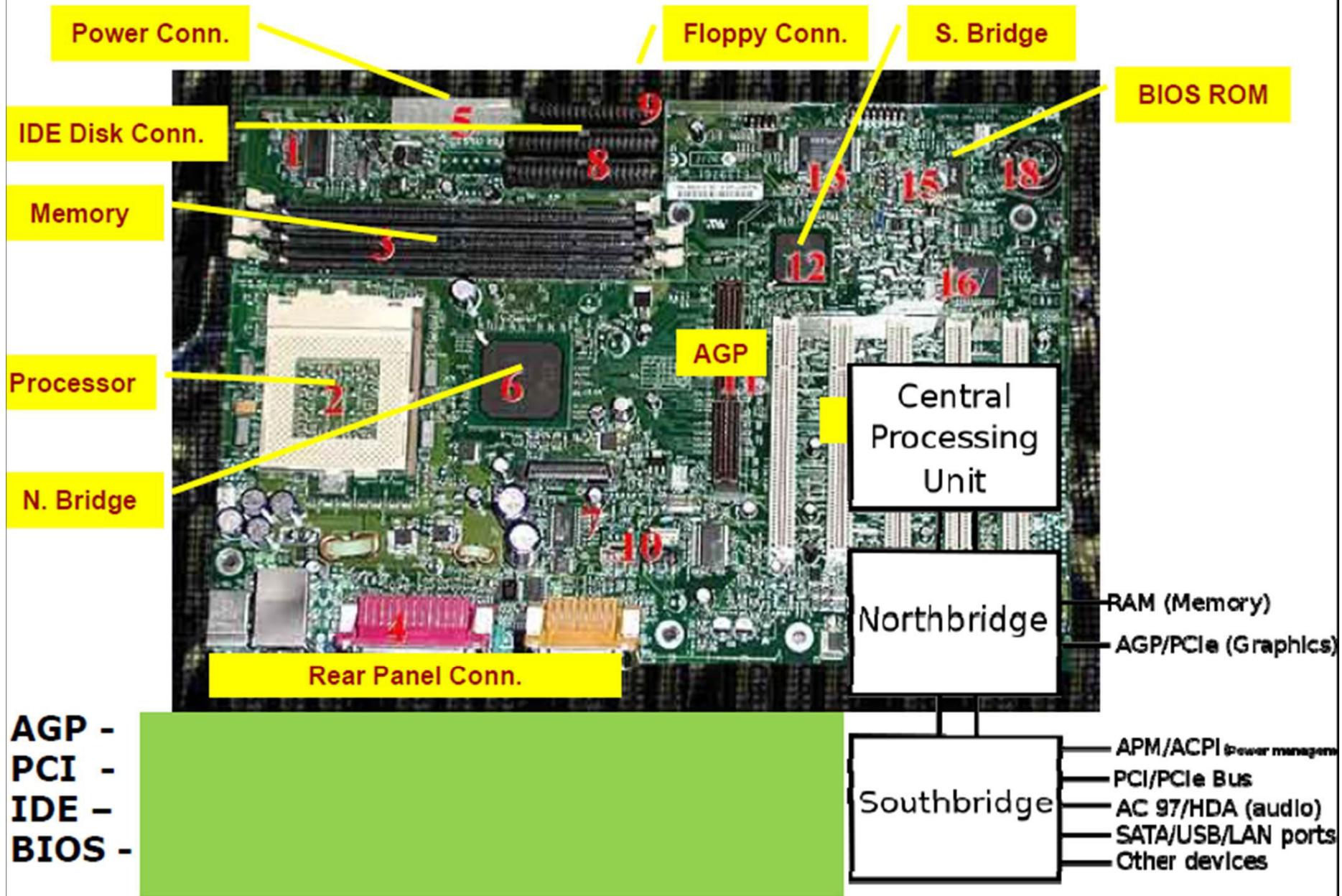
Another example: it is an architectural design issue whether a computer will have a **multiply instruction**. It is an organizational issue whether that instruction will be implemented by a special **multiply unit** or by a mechanism that makes repeated use of the **add unit** of the system.

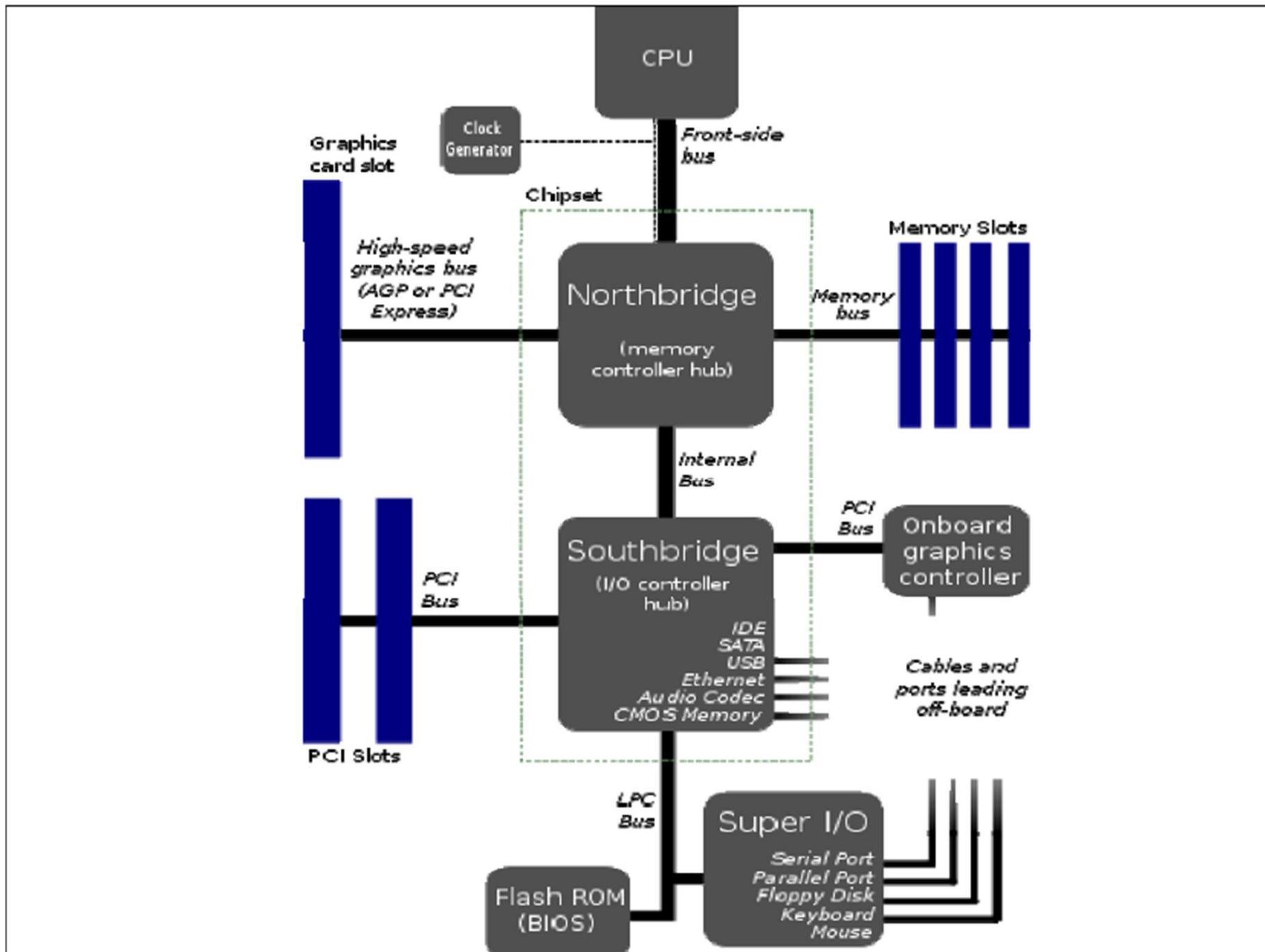
Instruction Set Architecture (ISA) - The Hardware-Software Interface

- ▶ The **most important** abstraction of computer design



Typical Motherboard (Pentium III)





What is Computer Architecture?



Computer Architecture

- The **CPU** is the brain of a computer system.
- It works both consciously and subconsciously.
- Consciously : Executes a program
- Sub-consciously : Runs the operating system, co-ordinates with I/O devices

Figure 1: Courtesy: www.psychologytoday.com

Computer Architecture : Study of the CPU and the peripherals

Where does it fit in?



Figure 2: courtesy: www.coolnerds.com

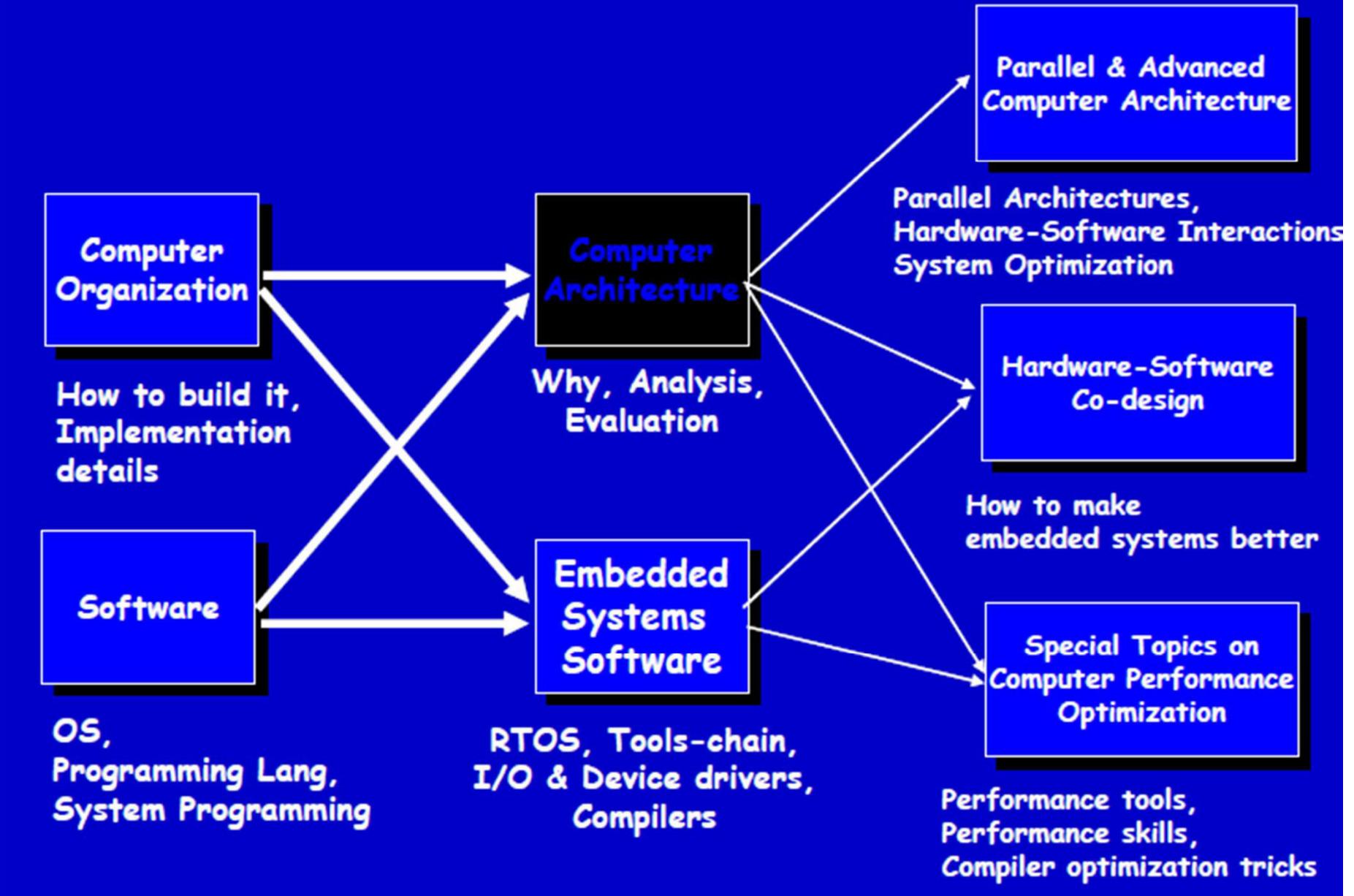
Example

- Computer Architecture → Brain
- Networking → Nervous and Circulatory System
- Computer Vision → Eyes
- Operating System → Endocrine and Immune System
- Databases → Memory
- Algorithms → Intelligence
- Prog. Languages → Linguistic Center
- ...

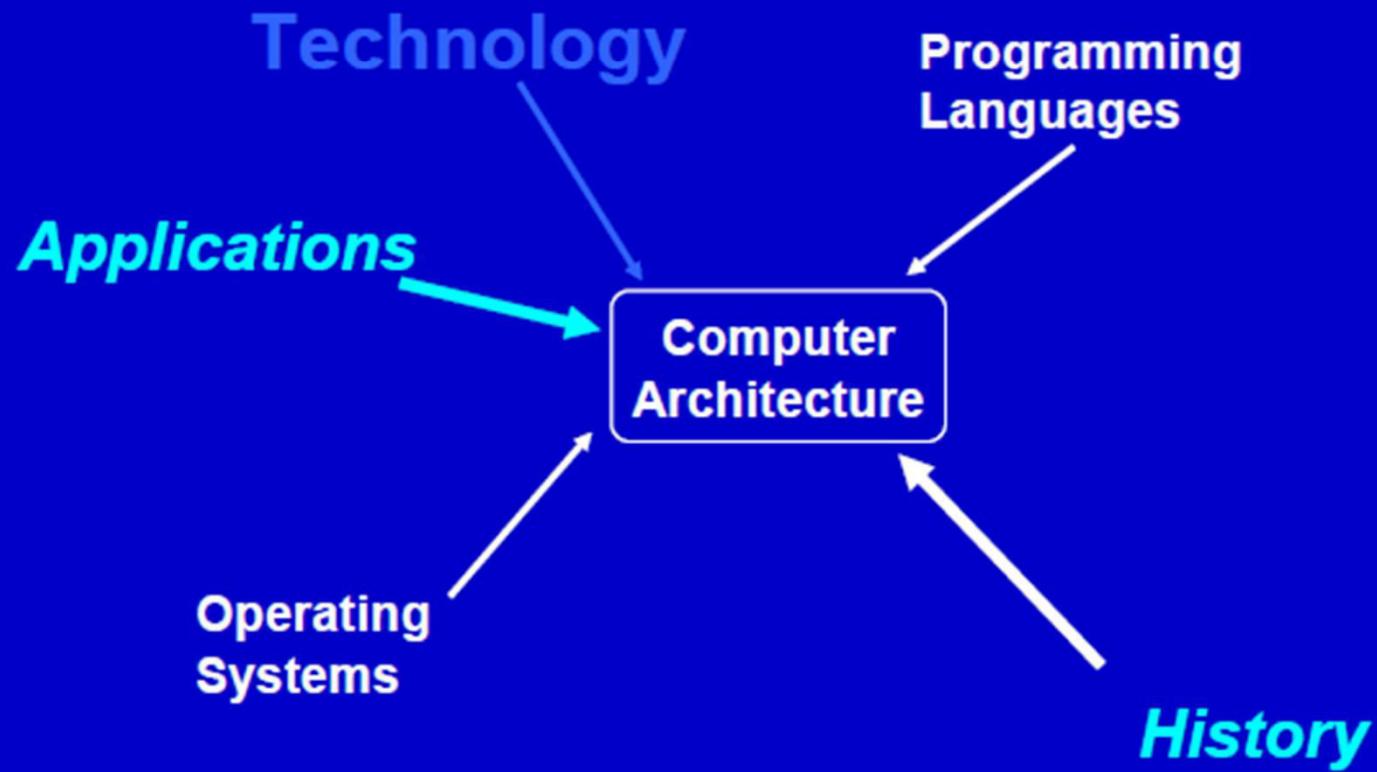
Why Study Computer Architecture?

- Understanding
 - Learn the inner workings of processors
 - Understand hardware/software interaction
 - Design better operating systems and compilers
- Career Prospects
 - Companies directly working in architecture
 - Intel, AMD, Sun/Oracle, Arm, IBM
 - Systems Software
 - Google, Samsung, VMWare, Wind River, McAfee
- Higher Studies ...

Related Courses

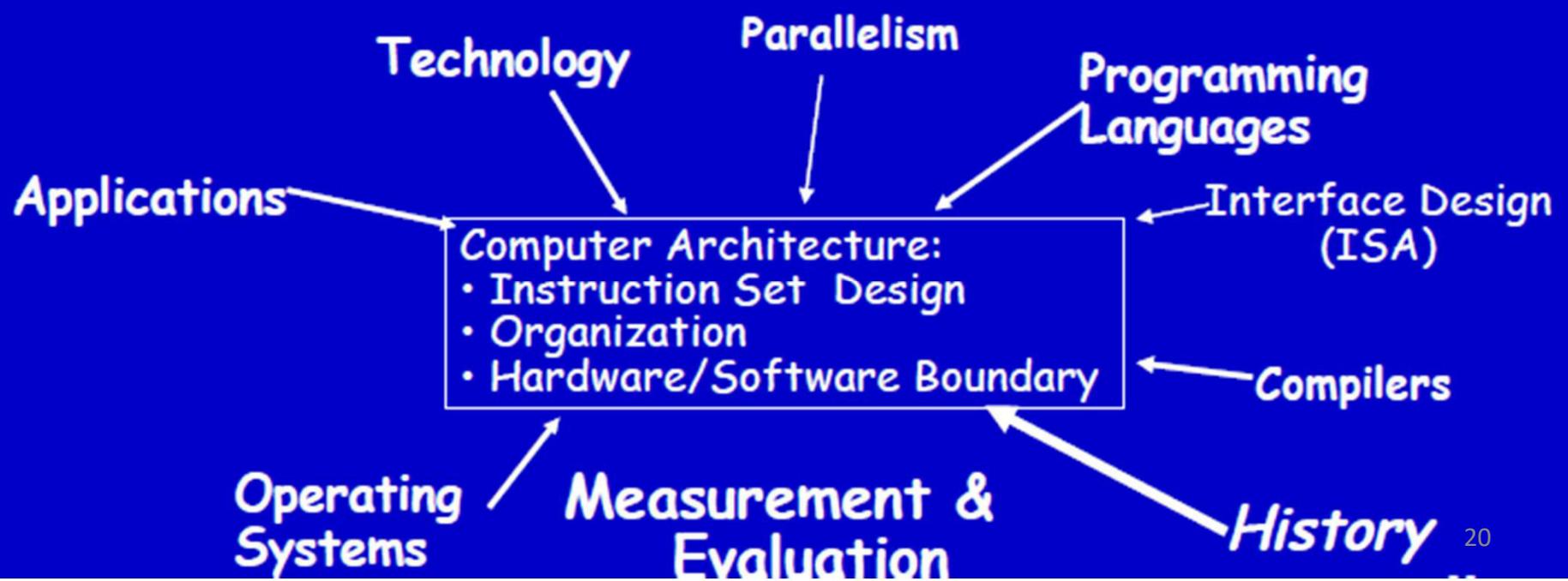


Forces on Computer Architecture



Course Focus

Understanding the design techniques, machine structures, technology factors, evaluation methods that will determine the form of computers in 21st Century



Input/ Output Unit Overview

- Input units
 - Keyboard, mouse, microphone, CDROM, etc.
- Output units
 - Graphical display, printer, etc.
- The collective term input/ output (I/O) units
 - Input units, output units, disk drives, etc.

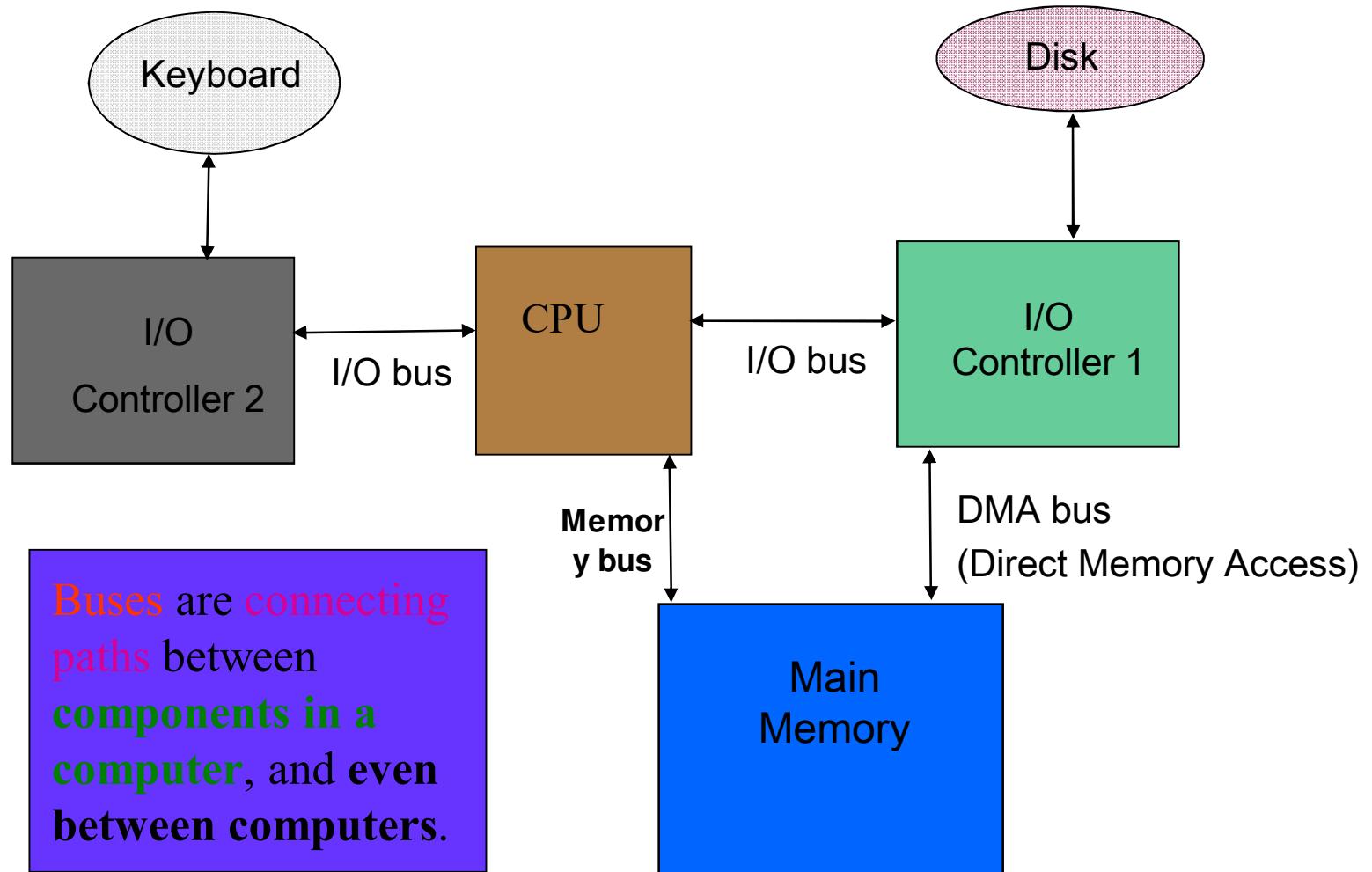
Memory Unit Overview

- Memory is used to **store programs** and **data**
- [Low-level] Unit of access is an **n-bit word**
 - Unique location is its address
 - Retrieval is in units of words
 - Commonly 32-bit today, moving to 64-bits
 - Typically 16-bit – 64-bit machines nowadays
- Primary storage: random-access memory (RAM)
- Secondary storage: hard disk, CDROM, etc.

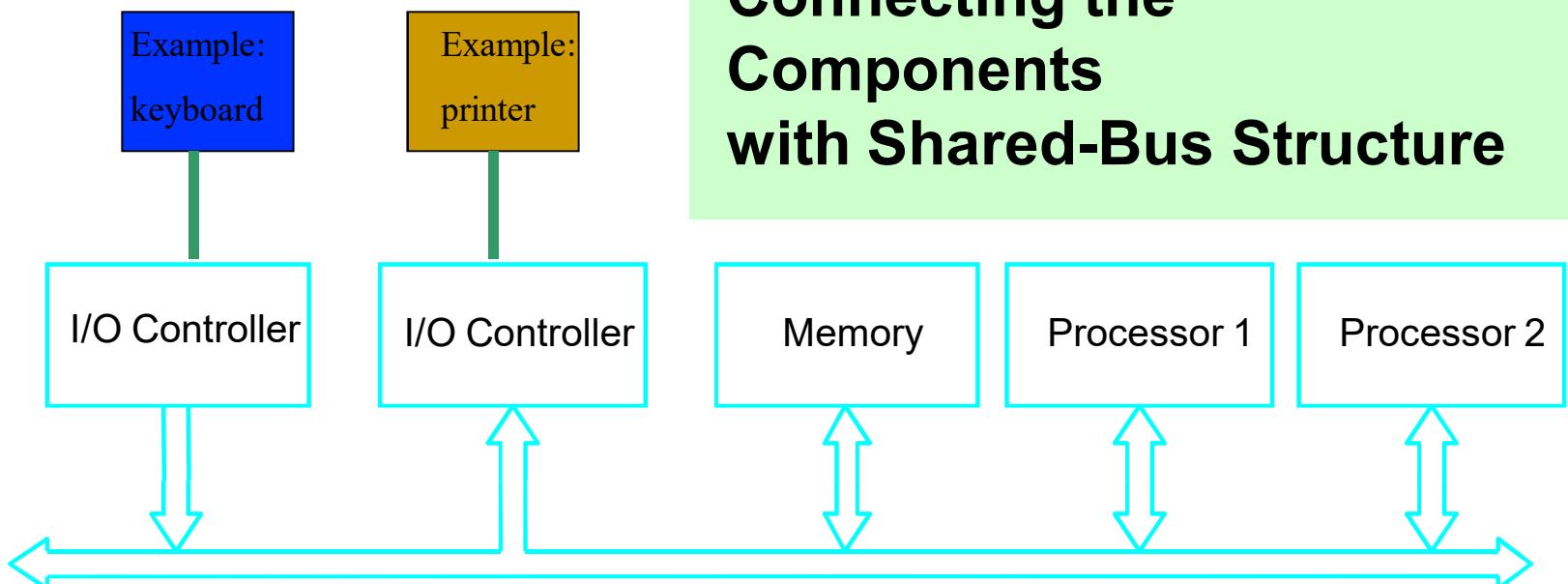
Processor Overview

- Registers
 - Small but fast storage of intermediate values in a computation
- Arithmetic logic unit (ALU): performs computations
 - e.g. arithmetic operations: add, subtract, multiply, divide, etc.
 - e.g. logical operations: and, or, not, xor, etc.
 - c.f. calculator
 - Operands taken from registers
- Control
 - Orchestrates the transfer of data and sequencing of operations between memory, registers, ALU, I/O devices

Connecting the Components with Dedicated/ Multiple Buses



Connecting the Components with Shared-Bus Structure



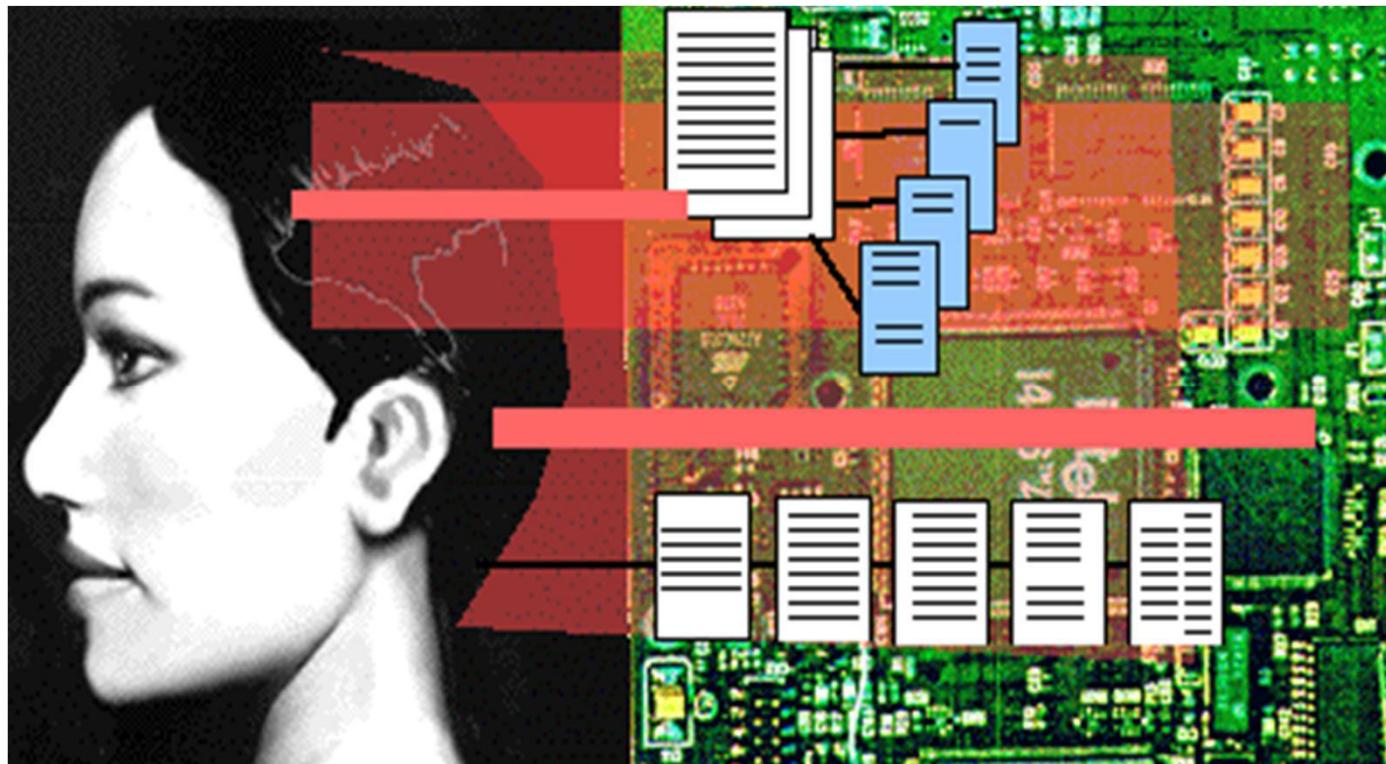
Information carried along a bus: **address, data , control**

- All devices have same **address** structure
- All devices can be **controlled** by common machine instructions
- Only two devices can do **data** communication **simultaneously**

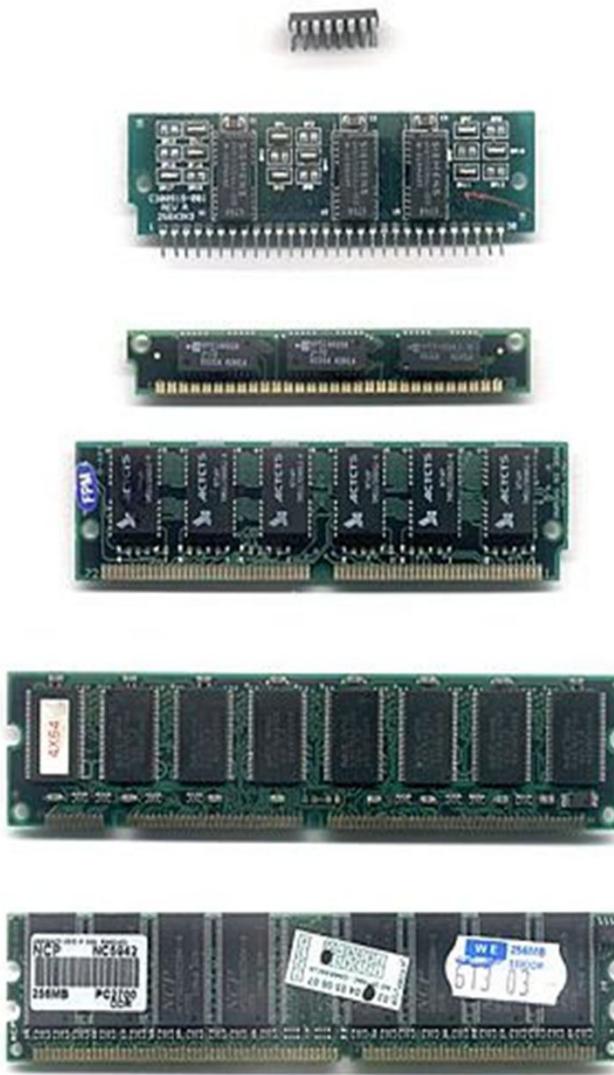
Bus

- Allows computer to be customized for different applications e.g. different peripherals
- Design criteria – speed, cost, etc.
- Word length can be different depending on application
 - E.g. USB is serial (1-bit), PCI bus is 32-bit

Computer Memory



Memory Hardware: Chips and Modules

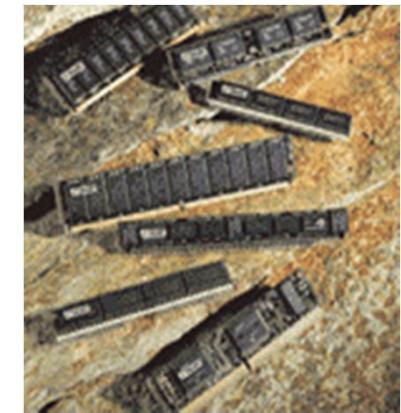


Digital (Binary) Memory

- Primary storage, main storage, main memory
- Fast Access (when compared to I/O)

Units

bit (1 binary digit, a value of 0 or 1)



Byte (1 byte = 8 bits)

Word

Manipulation of data by CPU is in words.

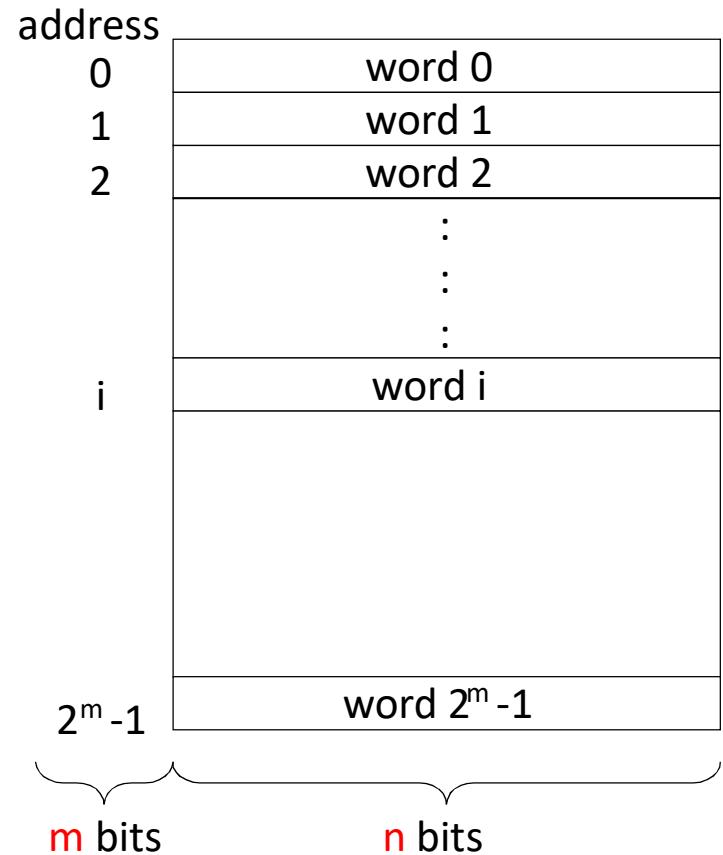
A single access results in one word of data being transferred.

Word length is specified in number of bits/ bytes:

for example, 8-bit, 16-bit, 32-bit, 64-bit, 4-byte, etc.

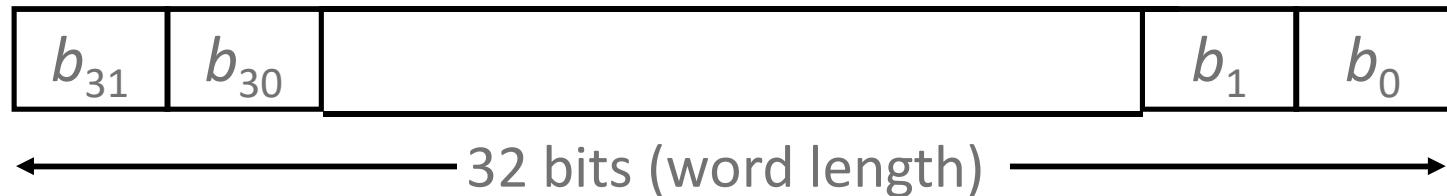
Main Memory (MM) Organization

- In a main memory with m -bit addresses,
0 to $2^m - 1$ words are available.
- Each word stores n bits
- m, n are independent
- m specifies the number of units;
 n specifies the unit size
- What is the total number of bits?



Memory: Contents of a Word (I)

Here is an example of a 32-bit word.



A word can store **information**. For example,

1. Four English characters, each encoded in a common 8-bit code

ASCII: American Standard Code for Information Interchange
or

EBCDIC: Extended Binary Coded Decimal Interchange Code

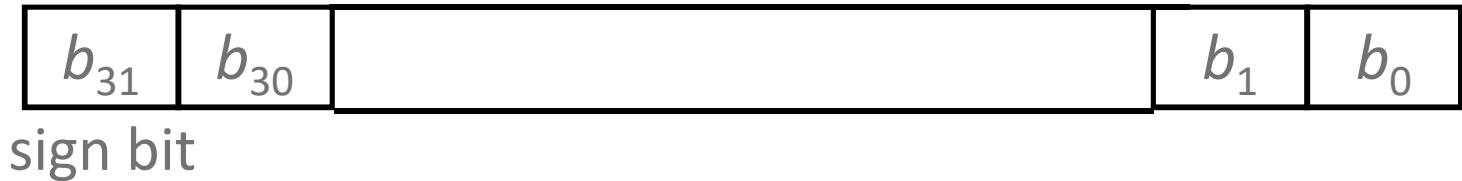


Memory: Contents of a Word (II)

2. Two Chinese characters

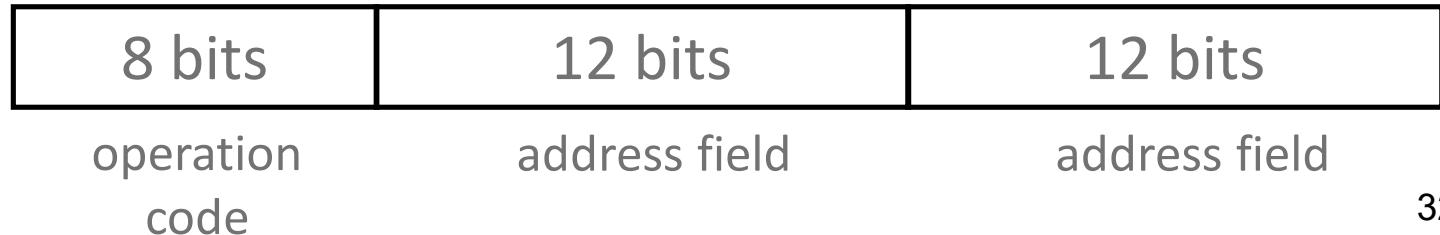


3. A 32-bit Signed integer



$b_{31}=0$ means positive integer, $b_{31}=1$ means negative integer
magnitude = $b_{30} \cdot 2^{30} + \dots + b_0 \cdot 2^0$

4. A 32-bit machine instruction



Addresses

- Use **addresses** to store or retrieve a single item of information
- For some k , memory consists of 2^k unique addresses which range from $0 - (2^k - 1)$
 - The **possible** addresses are the **address space** of the computer
 - e.g. 28-bit address $\rightarrow 2^{28} = 268435456$ locations
 - Talk about word address: we use words
 - Talk about byte address: we use bytes
 - Talk about memory size: we use bytes/ sometimes bits

Byte addresses

- Information quantities: bit, byte, word
- 1 Byte = 8 bits
- Word typically varies 16-64 bits (the IA-32 architecture has 32-bit words which we will assume from now on)
Intel Architecture, 32-bit
- Most machines address memory in units of bytes (byte-addressable)
 - Implies for a 32-bit machine, successive words are at addresses 0, 4, 8, 12 ...

More/ Less Significant Bytes

- Consider the hexadecimal (base 16) 32-bit number

12342A3F₁₆

$$\begin{aligned} &= 1 \times 16^7 + 2 \times 16^6 + 3 \times 16^5 + 4 \times 16^4 + 2 \times 16^3 + \textcolor{red}{10} \times 16^2 + 3 \times 16^1 + \textcolor{blue}{15} \times 16^0 \\ &= 305408575_{10} \end{aligned}$$

- This 32-bit number has four bytes 12, 34, 2A, 4F
(4 bytes x 8 bits/byte = 32 bits)
- Bytes/bits with higher weighting are “ more significant”
 - e.g. the byte 34 is more significant than 2A
- Bytes/bits with lower weighting are “ less significant”
- We also use terms “ most significant byte/ bit” and “ least significant byte/ bit”

Big/ Little Endian

- Two ways byte addresses can be assigned/ arranged across words

○ more significant bytes first (big endian)

- e.g. Motorola

○ less significant bytes first (little endian)

- e.g. Intel

Big/ Little Endian

What would $12342A3F_{16}$ look like in these two endianship assignments?

Word address	Byte address			
0	0	1	2	3
4	4	5	6	7
	•	•	•	
$2^k - 4$	$2^k - 4$	$2^k - 3$	$2^k - 2$	$2^k - 1$

(a) Big-endian assignment
e.g. Motorola

Word address	Byte address			
0	3	2	1	0
4	7	6	5	4
	•	•	•	
$2^k - 4$	$2^k - 1$	$2^k - 2$	$2^k - 3$	$2^k - 4$

(b) Little-endian assignment
e.g. Intel

Assume k-bit address, 4-byte (32-bit) word

Word alignment

- 32-bit words *align* naturally at addresses 0, 4, 8 ...
 - These are aligned addresses
- Unaligned accesses are either not allowed or slower, why?
 - e.g. read a 32-bit word from address 00000001
- What about for 16- and 64-bit words?

Central Processing Unit (CPU)



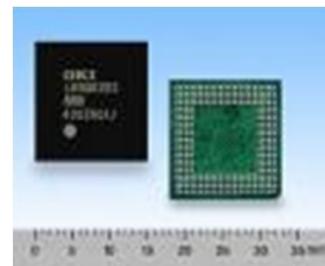
IBM Power PC



Intel Pentium



Digital signal processor IC



ARM 9



Some Intel CPUs

4004

8008

8080

8088/ 8086 [x86]

80186

80286 [286]

80386 [386]

80486 [486]

Pentium [586]

Pentium MMX

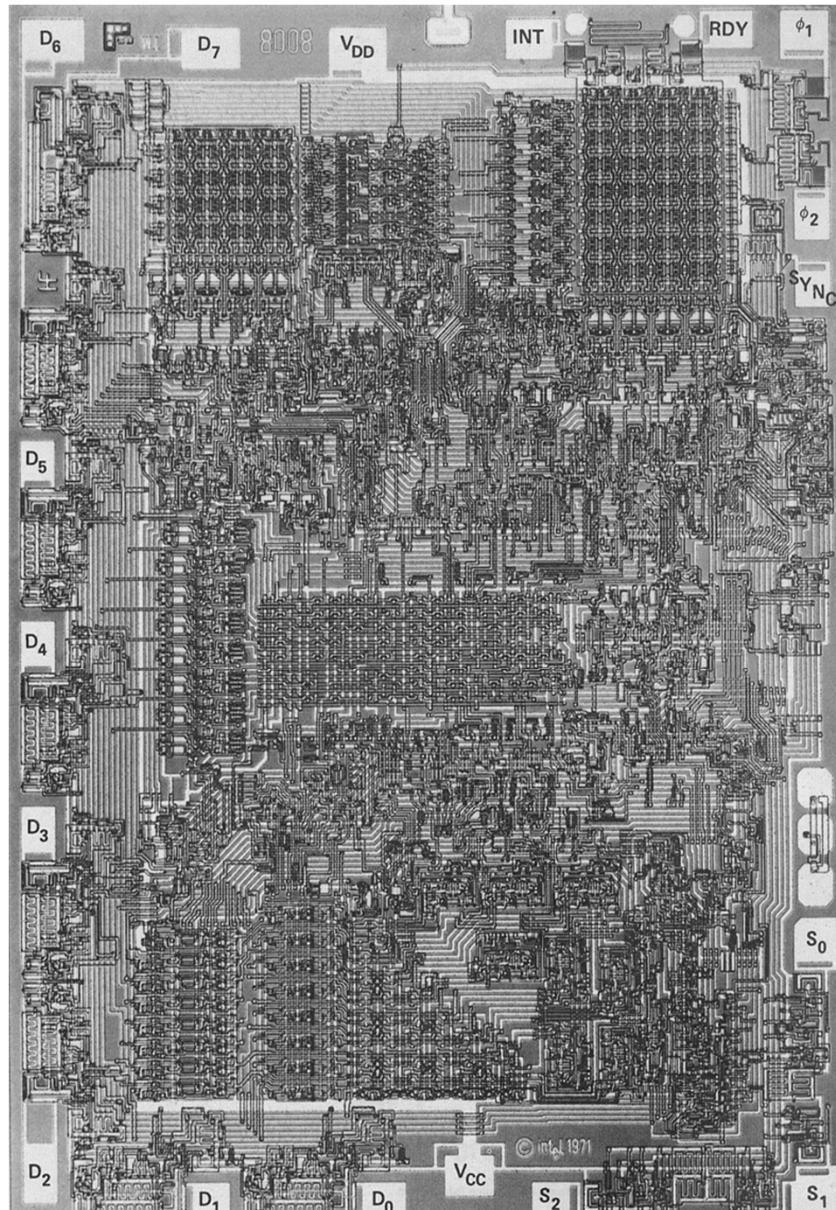
Pentium PRO [686]

Pentium II

Pentium III

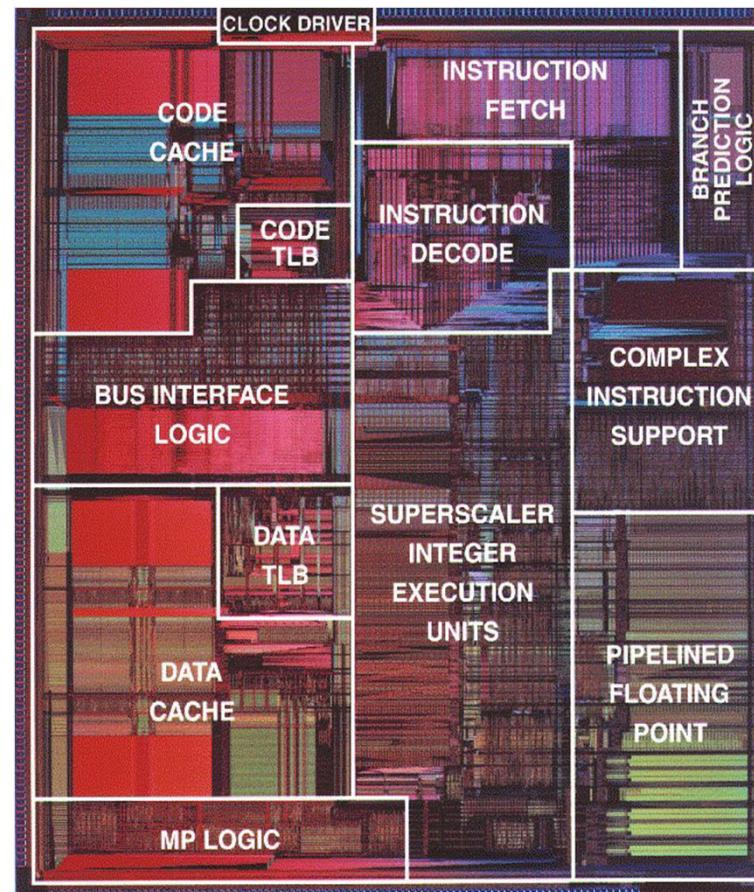
Pentium 4

Core2 Duo



8008 Photomicrograph With Pin Designations

CPU on a Chip → Microprocessor



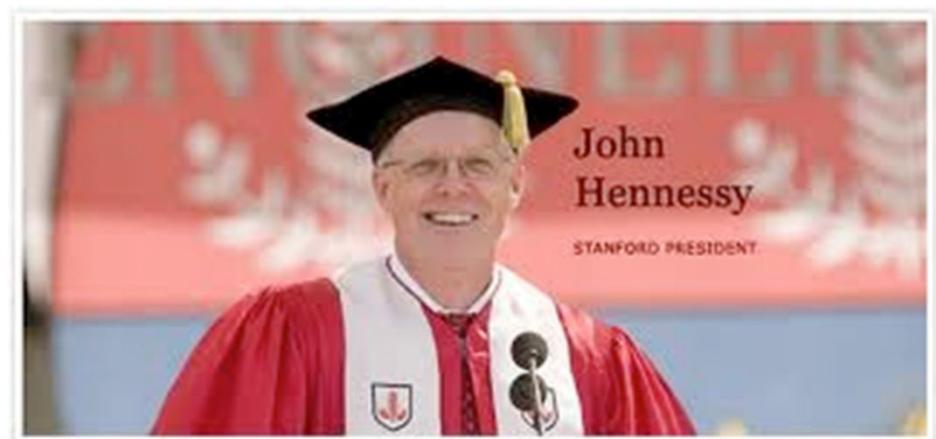
<http://micro.magnet.fsu.edu/chipshots/index.html>

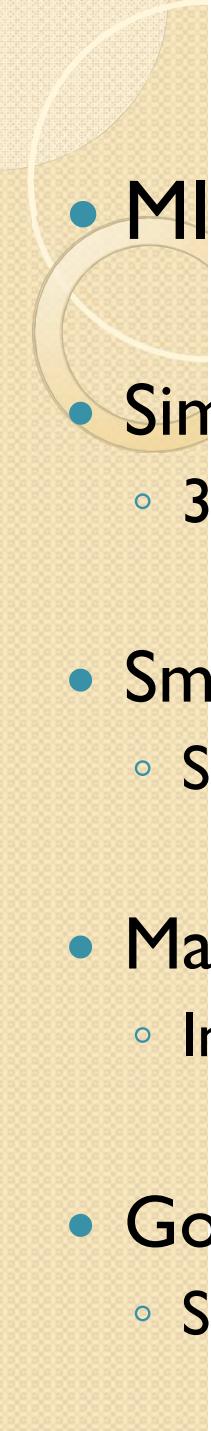
CPU

- What does a CPU do?
It executes programs.
- What is a program?
Program = Instruction + Data
- Where are the instructions and data?
In Memory – when they are not being processed
In CPU – when they are being processed

Reduced Instruction Set Computer (RISC)

- Dave Patterson
- RISC Project, 1982
- UC Berkeley
- RISC-I: $\frac{1}{2}$ transistors & 3x faster
- Influences: Sun SPARC, namesake of industry
- John L. Hennessy
- MIPS, 1981
- Stanford
- Simple pipelining, keep full
- Influences: MIPS computer system, PlayStation, Nintendo





Reduced Instruction Set Computer (RISC)

- MIPS Design Principles
- Simplicity favors regularity
 - 32 bit instructions
- Smaller is faster
 - Small register file
- Make the common case fast
 - Include support for constants
- Good design demands good compromises
 - Support for different type of interpretations/classes

Reduced Instruction Set Computer

- MIPS = Reduced Instruction Set Computer (RISC)
 - ≈ 200 instructions, 32 bits each, 3 formats
 - all operands in registers
 - almost all are 32 bits each
 - ≈ 1 addressing mode: Mem[reg + imm]
- x86 = Complex Instruction Set Computer (CISC)
 - > 1000 instructions, 1 to 15 bytes each
 - operands in dedicated registers, general purpose registers, memory, on stack, ...
 - can be 1, 2, 4, 8 bytes, signed or unsigned
 - 10s of addressing modes
 - e.g. Mem[segment + reg + reg*scale + offset]

RISC vs CISC

• RISC Philosophy

- Regularity & simplicity

- Leaner means faster

- Optimize the common case

- Energy efficiency

- Embedded Systems

- Phones/Tablets

CISC Rebuttal

- Compilers can be smart

- Transistors are plentiful

- Legacy is important

- Code size counts

- Micro-code!

Desktops/Servers

Role of Performance

Introduction

- Performance dictates the effectiveness of an entire system, including hardware and software
- Performance measurement is one of the most important and difficult problems in computers
 - Consider the code to initialize a million integers using a loop vs using a system call
- Different aspects of performance may require different performance metrics
- Our goal for understanding performance
 - Effect of software on performance (see the above example)
 - Effect of instruction set architecture
 - Hardware features
- Defining performance
 - Needs and desires, buying a car
 - Response time
 - Execution time
 - Clock time is dependent on computer load, I/O wait, and OS overhead
 - Throughput
 - For our purpose,

$$\text{Performance} = \frac{c}{\text{Execution time}}$$

where c is a constant

- For two machines, performance (p_i) and execution time (e_i) obey the relation

$$\frac{p_i}{p_j} = \frac{e_j}{e_i} = n$$

and we say that machine i is n times faster than machine j

Measuring performance

- Amount of work and amount of time
- Simplest time definition is the real clock time
 - System time, user time, I/O time, overhead
- *System performance* – Elapsed time on unloaded system
- *CPU performance* – CPU time
- *Clock cycles*
 - Constant time interval for the clock within the system
 - Dictates how fast a CPU can execute each instruction
- Clock rate
 - Inverse of clock cycle

- 500 MHz
- Clock cycle for 500 MHz is 2ns

Performance metrics

- CPU execution time is given by the product of CPU clock cycles for program and clock cycle time
- It can also be measured by

$$\frac{\text{CPU clock cycles for program}}{\text{Clock rate}}$$

- Improving performance

- Current system
 - * Execution time – 10 sec
 - * Clock speed – 400 MHz
- New system
 - * Execution time – 6 sec
 - * Clock speed – ?
 - * Number of clock cycles – 1.2 times current system

- Compute the number of clock cycles for current system

*

$$\begin{aligned} \text{CPU time} &= \frac{\text{CPU clock cycles for program}}{\text{Clock rate}} \\ 10\text{sec} &= \frac{\text{CPU clock cycles for program}}{400 \times 10^6 \text{cps}} \end{aligned}$$

* CPU clock cycles for program = 4000×10^6

- Compute the clock speed for new system

*

$$\begin{aligned} \text{CPU time} &= \frac{\text{CPU clock cycles for program}}{\text{Clock rate}} \\ 6\text{sec} &= \frac{1.2 \times 4000 \times 10^6}{\text{Clock rate}} \end{aligned}$$

*

$$\begin{aligned} \text{Clock rate} &= \frac{1.2 \times 4000 \times 10^6}{6} \\ &= 800 \times 10^6 \\ &= 800\text{MHz} \end{aligned}$$

- *Clock cycles per instruction*, or CPI

- Average number of cycles for all instructions for the program being executed
- CPU clock cycles is given by the product of number of instructions and CPI

- Using performance equation

- Two implementations of the same ISA – machines M_a and M_b
- M_a clock cycle time 1ns and CPI 2.0 for some code p
- M_b clock cycle time 2ns and CPI 1.2 for p

- Identify faster machine
 - * Let total clock cycles for the program on respective machines be c_a and c_b , and number of instructions be I
 - *
- $$\begin{aligned} c_a &= T \times 2.0 \\ c_b &= T \times 1.2 \end{aligned}$$
- * CPU time $t = \text{CPU clock cycles} \times \text{Clock cycle time}$
 - * $t_a = I \times 2.0 \times 1 = 2I \text{ ns}$
 - * $t_b = I \times 1.2 \times 2 = 2.4I \text{ ns}$
 - * Machine m_a is faster; since performance is inversely proportional to time, the performance gain is given by

$$\frac{t_b}{t_a} = \frac{2.4}{2} = 1.2$$

- Basic performance equation

$$\text{CPU time} = \text{Instruction count} \times \text{CPI} \times \text{Clock cycle time}$$

or

$$\text{CPU time} = \frac{\text{Instruction count} \times \text{CPI}}{\text{Clock rate}}$$

- Measuring the performance factors

- Measure CPU time by actually running the program
- Clock cycle time is usually available as part of documentation
- Instruction count and CPI are more difficult to obtain
- Instruction count can be measured by using profiling tools, for example, `gprof(1)` in Unix

```
$ gcc -pg -o foobar foobar.c
$ foobar
$ gprof > foobar.profile
```

- CPI can be obtained by detailed simulation of an implementation or by combining hardware counters and simulation
- You may be able to compute CPU clock cycles by looking at different types of instructions and using their individual clock cycle counts

$$\text{CPU clock cycles} = \sum_{i=1}^n (\text{CPI}_i \times C_i)$$

- * C_i is the number of instructions of class i
- * CPI_i is the average number of cycles per instruction for class i
- * n is the number of instruction classes

- Comparing code segments – deciding how to write efficient code for a given machine by selecting a set of instructions

- Instruction classes

Instruction class	CPI
A	1
B	2
C	3

- Instruction count for different code sequences

Code sequence	Number of instructions		
	A	B	C
c_1	2	1	2
c_2	4	1	1

- Find out the number of instructions for each code sequence, the faster code sequence, and CPI for each code sequence
 - * Number of instructions in sequence $c_1 = 2 + 1 + 2 = 5$
 - * Number of instructions in sequence $c_2 = 4 + 1 + 1 = 6$
 - * Obviously, sequence c_1 executes fewer instructions
 - * CPU clock cycles₁ = $(2 \times 1) + (1 \times 2) + (2 \times 3) = 2 + 2 + 6 = 10$
 - * CPU clock cycles₂ = $(4 \times 1) + (1 \times 2) + (1 \times 3) = 4 + 2 + 3 = 9$
 - * Code sequence c_2 is faster
 - * $\text{CPI} = \frac{\text{CPU clock cycles}}{\text{Instruction count}}$
 - * $\text{CPI}_1 = \frac{10}{5} = 2$
 - * $\text{CPI}_2 = \frac{9}{6} = 1.5$

Benchmarks for performance evaluation

- *Workload*
 - Typical set of programs run in day-to-day work
 - Compare the execution time of workload on two computers to evaluate their relative performance
 - Not always feasible for real world
 - * Too expensive (taking machines to prospective buyers' sites)
 - * Proprietary issues (sending code and data to vendor sites)
- Benchmarks
 - Programs specifically chosen to simulate the actual workload performance
 - Selection of programs based on expected usage environment
 - Compiler optimization to beat benchmarks
 - * Compiler may beat the benchmark but not guaranteed to produce correct working code at similar performance level
 - * Code optimization to beat benchmark, especially if the benchmark is skewed towards some code
 - Benchmarks are used for
 - * Easy coding and simulation
 - * Simplicity
 - * More easily standardized than large code
- Reproducibility
 - Most important component of a benchmark
 - Contains everything required to simulate a benchmark

Comparing and summarizing performance

- Summarizing implies loss of information but ease of understanding

- Should not cause confusion with contradictory but true statements
 - * *Machine A is 10 times faster than machine B for program 1*
 - * *Machine B is 10 times faster than machine A for program 2*
- Total execution time
 - Compare total execution time of a set of programs taken together
 - If P_i is performance of machine i and E_i is execution time of machine i , then,
$$\frac{P_a}{P_b} = \frac{E_b}{E_a} = \frac{1001}{110} = 9.1$$
- Average execution time
 - Computed over a number of small benchmarks
 - Arithmetic mean $AM = \frac{1}{n} \sum_{i=1}^n E_i$
 - Smaller mean implies smaller execution time
- Weighted average execution time
 - Applies a weight to each task such that sum of all weights w_i is 1
 - Weighted arithmetic mean $WAM = \frac{1}{n} \sum_{i=1}^n w_i \times E_i$, with the condition that $\sum_{i=1}^n w_i = 1$

SPEC95 Benchmark

- SPEC – *System Performance Evaluation Cooperative*
- Most comprehensive and popular set of CPU benchmarks
- 8 integer programs written in C and 10 floating point programs written in Fortran 77
- Separate time measurement for each set
 - Measurement normalized by dividing the execution time of a Sun SPARCstation 10/40 by the execution time on measured machine, yielding SPEC ratio
 - SPECint95 or SPECfp95 – Summary measurement by taking the geometric mean of the SPEC ratios
- For a given ISA, performance improvement comes from
 1. Increase in clock rate
 2. Improvements in processor organization to lower the CPI
 3. Compiler enhancements to lower the instruction count, or generate instructions with a lower average CPI
- In Figure 2.7, we see that Pentium Pro is 1.4 to 1.5 times faster on SPECint95 and 1.6 to 1.7 times faster on SPECfp95, *at the same clock rate*
- Increasing clock speed (Figure 2.8) does not increase the SPEC performance by the same level because of memory speed bottleneck

Fallacies and pitfalls

Pitfall 1 *Expecting the improvement of one aspect of a machine to increase performance by an amount proportional to the size of the improvement.*

- A program runs in 100 sec on a machine, with multiply operations taking up 80 seconds of this time. How much does the speed of multiplication need to improve to get a five-fold increase in code execution?

$$\begin{aligned} \text{Execution time after improvement} &= \frac{\text{Execution time affected by improvement}}{\text{Amount of improvement}} + \text{Execution time unaffected} \\ \frac{100}{5} &= \frac{80}{n} + (100 - 80) \\ 20 &= \frac{80}{n} + 20 \\ 0 &= \frac{80}{n} \end{aligned}$$

There is no amount by which we can improve the performance of multiply to realize a five-fold increase in overall performance

- This is *Amdahl's Law* in computing, or the law of diminishing returns in everyday life
- Opportunity of improvement is affected by how many time the event occurs
- Common theme (Corollary of Amdahl's law) – make the common case fast

Fallacy 1 *Hardware-independent metrics predict performance.*

- Code size as a measure of speed
- ISA with smallest instruction set is the fastest

Pitfall 2 *Using MIPS as a performance metric.*

- MIPS = $\frac{\text{Instruction count}}{\text{Execution time} \times 10^6}$
- Intuitive, as more MIPS implies faster execution
- Problems
 1. MIPS does not account for capabilities of instructions
 2. A machine cannot have same MIPS rating for all programs
 3. MIPS can vary inversely with performance
- Consider the machine with three instruction classes and CPI measurements as follows:

- Instruction classes

Instruction class	CPI
A	1
B	2
C	3

- Instruction count (in billions of instructions for each class) for same program from two different compilers

Code from	Instruction count		
	A	B	C
Compiler 1	5	1	1
Compiler 2	10	1	1

- Machine clock rate – 500 MHz
- Which code sequence executes faster according to MIPS? According to execution time?

- Solution

- Find the execution time on each compiler using the equation

$$\text{Execution time} = \frac{\text{CPU clock cycles}}{\text{Clock rate}}$$

- If C_i is the number of instructions of class i executed

$$\text{CPU clock cycles} = \sum_{i=1}^n (\text{CPI}_i \times C_i)$$

$$\begin{aligned}\text{CPU clock cycles}_1 &= (5 \times 1 + 1 \times 2 + 1 \times 3) \times 10^9 = 10 \times 10^9 \\ \text{CPU clock cycles}_2 &= (10 \times 1 + 1 \times 2 + 1 \times 3) \times 10^9 = 15 \times 10^9\end{aligned}$$

- Execution time for two compilers

$$\begin{aligned}\text{Execution time}_1 &= \frac{10 \times 10^9}{500 \times 10^6} = 20s \\ \text{Execution time}_2 &= \frac{15 \times 10^9}{500 \times 10^6} = 30s\end{aligned}$$

- MIPS rate

$$\begin{aligned}\text{MIPS} &= \frac{\text{Instruction count}}{\text{Execution time} \times 10^6} \\ \text{MIPS}_1 &= \frac{(5 + 1 + 1) \times 10^9}{20 \times 10^6} = 350 \\ \text{MIPS}_2 &= \frac{(10 + 1 + 1) \times 10^9}{30 \times 10^6} = 400\end{aligned}$$

- Conclusion – Code from compiler 1 runs faster but code from compiler 2 has higher MIPS

Fallacy 2 Synthetic benchmarks predict performance.

- Goal to create a benchmark where execution frequency of a synthetic benchmark matches the characteristics of a large set of programs
- Most popular synthetic benchmarks – Whetstone and Dhrystone
- Whetstone – Measurement of Algol programs in a scientific/engineering environment (converted to Fortran)
- Dhrystone – Systems programming environments, originally in Ada and later converted to C

Pitfall 3 Using arithmetic mean of normalized execution times to predict performance.

- Normalized arithmetic mean is dependent on the machine used for normalization
- Better way is to use geometric mean given by

$$\sqrt[n]{\prod_{i=1}^n \text{Execution time ratio}_i}$$

where $\text{Execution time ratio}_i$ is the execution time, normalized to the reference machine, for the i th program of a total of n in the total workload

- Geometric mean is independent of the data series used for normalization because of the property

$$\frac{\text{Geometric mean}(X_i)}{\text{Geometric mean}(Y_i)} = \text{Geometric mean} \left(\frac{X_i}{Y_i} \right)$$

implying that mean of ratios, or ratio of means, is equal

Fallacy 3 *The geometric mean of execution time ratios is proportional to total execution time.*

- Geometric mean does not predict execution time

Performance

- Introduction
- Defining Performance
- The Iron Law of Processor Performance
- Processor performance enhancement
- Performance Evaluation Approaches
- Performance Reporting
- Amdahl's Law

Introduction

- ❑ Performance measurement is important:
 - ❑ Helps us to determine if one processor or computer works faster than other
 - ❑ Helps us to know how much performance improvement has taken place after incorporating some performance enhancement feature
 - ❑ Help to see through the marketing hype!
- ❑ Provides answer to the following questions:
 - ❑ Why is some hardware better than others for different programs?
 - ❑ What factors affect system performance?
 - ❑ Hardware, OS or compiler?
 - ❑ How does the machine's instruction set affect performance?

Defining Performance in terms of time

- ❑ Time is the final measure of computer performance
- ❑ A computer exhibits higher performance if it executes program faster
- ❑ Response Time (elapsed time, Latency):
 - ❑ How long does it take for my job to run?
 - ❑ How long does it take to execute (start to finish) my job?
 - ❑ How long must/wait for the database query?
- ❑ Throughput:
 - ❑ How many jobs can the machine run at once?
 - ❑ What is the average execution rate?
 - ❑ How much work is getting done?

Individual user concern

System Manager concern

Execution Time

Elapsed time

- Count everything (disk and memory access, waiting for IO, running other programs, etc) from start to finish
- A useful number, but often not good for comparison purpose

Elapsed time = **CPU time + wait time (IO, other program, etc.)**



CPU time

- Doesn't count waiting for IO or time spent running other programs
- Can be divided into user CPU time and system CPU time(OS calls)

CPU time = user CPU time + System CPU Time

Elapsed time = **user CPU time + System CPU Time + wait time**

Our focus: User CPU time

- CPU execution time or simply execution time: time spent executing the lines of code that are in our program

Measuring performance

- ❑ for some program running on machine X:

$$\text{Performance} = \frac{1}{\text{execution time}_x}$$

- ❑ X is n times faster than Y means:

$$\frac{\text{Performance}_x}{\text{Performance}_y} = n$$

The IRON law of processor performance

$$\text{Processor performance} = \frac{\text{Time}}{\text{Program}}$$

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

(code Size)

(CPI)

(Cycle time)

Architecture →

Implementation →

Realization

Compiler Designer

Processor designer

Chip designer

- ❑ Instructions/Program (Instruction count)

- Instructions executed, not static code size
- Determined by algorithm, compiler, ISA

- ❑ Cycles/Instruction (CPI)

- Determined by ISA and CPU organization
- Overlap among instructions reduces this term

- ❑ Time/cycle (Cycle time)

- Determined by technology, organization, clever circuit design

MIPS and MFLOPS

- ❑ Used extensively 30 years back.
- ❑ **MIPS:** millions of instructions processed per second.
- ❑ **MFLOPS:** Millions of Floating-point Operations completed per Second

$$\text{MIPS} = \frac{\text{Instruction Count}}{\text{Exec. Time} \times 10^6} = \frac{\text{Clock Rate}}{\text{CPI} \times 10^6}$$



Problems with MIPS

- Three significant problems with using MIPS:
- So severe, made some one term:
 - “Meaningless Information about Processing Speed”
- Problem 1:
 - MIPS is instruction set dependent.
- Problem 2:
 - MIPS varies between programs on the same computer.
- Problem 3:
 - MIPS can vary inversely to performance!

□ Consider the following computer:

The machine runs at 100MHz.

Instruction counts (in millions)
for each instruction class

Code type-	A (1 cycle)	B (2 cycle)	C (3 cycle)
Compiler 1	5	1	1
Compiler 2	10	1	1

Instruction A requires 1 clock cycle, Instruction B requires 2 clock cycles, Instruction C requires 3 clock cycles.

$$CPI = \frac{\text{CPU Clock Cycles}}{\text{Instruction Count}} = \frac{\sum_{i=1}^n CPI_i \times N_i}{\text{Instruction Count}}$$

$$CPI_1 = \frac{\text{count} \times \text{cycles}}{(5 + 1 + 1) \times 10^6} = 10/7 = 1.43$$

$$MIPS_1 = \frac{100 \text{ MHz}}{1.43} = 69.9$$

$$CPI_2 = \frac{[(10 \times 1) + (1 \times 2) + (1 \times 3)] \times 10^6}{(10 + 1 + 1) \times 10^6} = 15/12 = 1.25$$

$$MIPS_2 = \frac{100 \text{ MHz}}{1.25} = 80.0$$

So, compiler 2 has a higher MIPS rating and should be faster?

□ Now let's compare CPU time:

$$\text{CPU Time} = \frac{\text{Instruction Count} \times \text{CPI}}{\text{Clock Rate}}$$

$$\text{CPU Time}_1 = \frac{7 \times 10^6 \times 1.43}{100 \times 10^6} = 0.10 \text{ seconds}$$

$$\text{CPU Time}_2 = \frac{12 \times 10^6 \times 1.25}{100 \times 10^6} = 0.15 \text{ seconds}$$

Therefore program 1 is faster despite a lower MIPS!

Computer Performance

"X is N% faster than Y."

$$\frac{\text{Execution Time of Y}}{\text{Execution Time of X}} = 1 + \frac{N}{100}$$

Amdahl's law for overall speedup

$$\text{Overall Speedup} = \frac{1}{(1 - F) + \frac{F}{S}}$$

F = The fraction enhanced

S = The speedup of the enhanced fraction

Using Amdahl's law

Overall speedup if we make 90% of a program run 10 times faster.

$$F = 0.9 \quad S = 10$$

$$\text{Overall Speedup} = \frac{1}{(1 - 0.9) + \frac{0.9}{10}} = \frac{1}{0.1 + 0.09} = 5.26$$

Overall speedup if we make 80% of a program run 20% faster.

$$F = 0.8 \quad S = 1.2$$

$$\text{Overall Speedup} = \frac{1}{(1 - 0.8) + \frac{0.8}{1.2}} = \frac{1}{0.2 + 0.66} = 1.153$$

Amdahl's Law

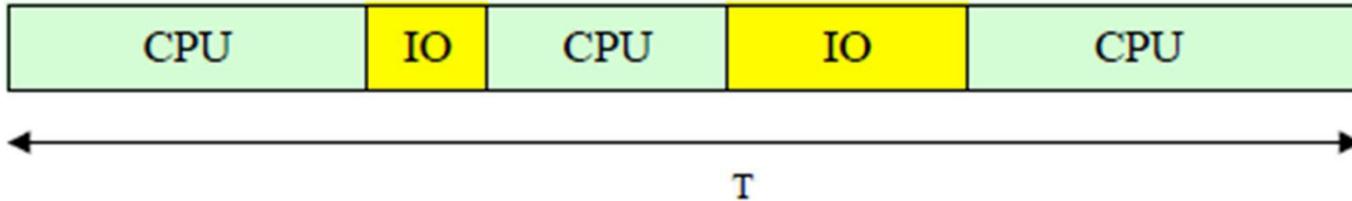
How is system performance altered when some component is changed?

Example 1:

Program execution time is made up of 75% CPU time and 25% I/O time. Which is the better enhancement:

- (a) Increasing the CPU speed by 50% or (b) reducing I/O time by half?

Execution model: No overlap between CPU and I/O operations



Program execution time $T = T_{cpu} + T_{io}$

$$T_{cpu} / T = 0.75 \text{ and } T_{io} / T = 0.25$$

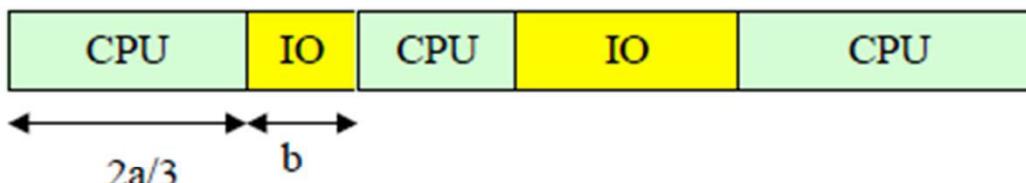
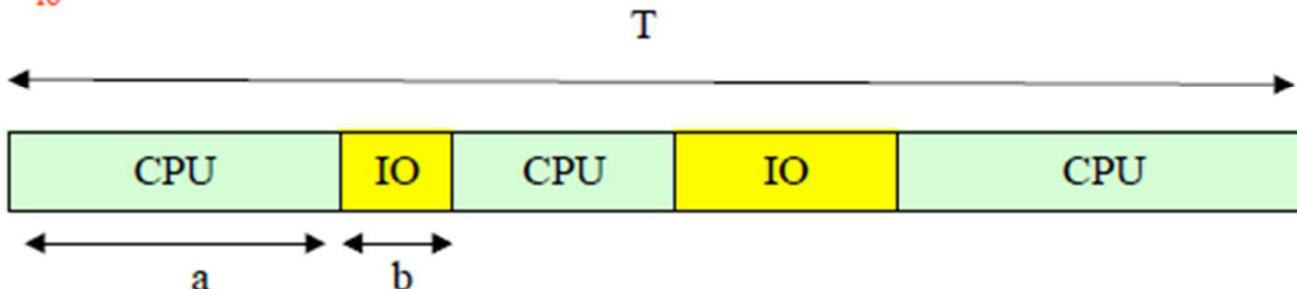
Amdahl's Law

- (a) Increasing the CPU speed by 50%

Program execution time $T = T_{cpu} + T_{io}$ $T_{old} = T$

$$T_{cpu} / T = 0.75$$

$$T_{io} / T = 0.25$$



Program execution time $T_{new} = T_{cpu} / 1.5 + T_{io}$

$$T_{new} = T_{cpu} / 1.5 + T_{io} = 0.75 T / 1.5 + 0.25 T = 0.75 T$$

For a 50% improvement in CPU speed: Execution time decreases by 25%

$$\text{Speedup} = T_{old} / T_{new} = T / 0.75 T = 1.33$$

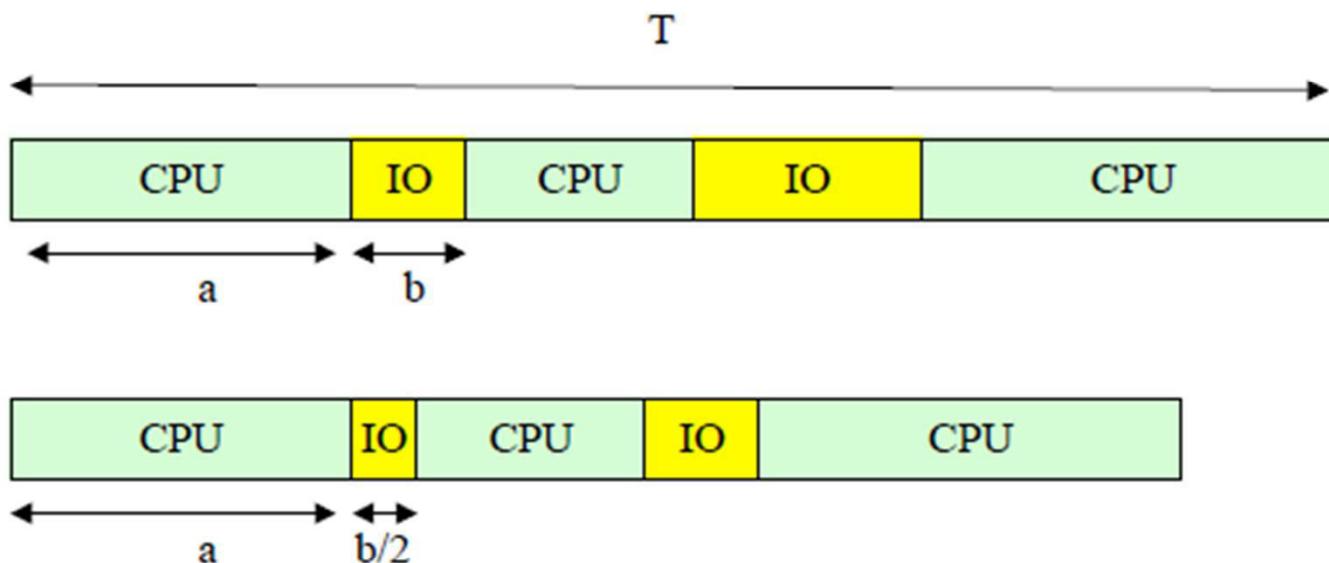
Amdahl's Law

(b) Halve the IO Time

$$\text{Program execution time } T = T_{\text{cpu}} + T_{\text{io}} \quad T_{\text{old}} = T$$

$$T_{\text{cpu}} / T = 0.75$$

$$T_{\text{io}} / T = 0.25$$



$$\text{Program execution time } T_{\text{new}} = T_{\text{cpu}} + T_{\text{io}} / 2$$

$$T_{\text{new}} = 0.75 T + 0.25 T / 2 = 0.875 T$$

For a 100% improvement in IO speed: Execution time decreases by 12.5%

$$\text{Speedup} = T_{\text{old}} / T_{\text{new}} = T / 0.875 T = 1.14$$

Amdahl's Law

Limiting Cases

- CPU speed improved infinitely so T_{CPU} tends to zero
 $T_{new} = T_{IO} = 0.25T$ Speedup limited to 4
- IO speed improved infinitely so T_{IO} tends to zero
 $T_{new} = T_{CPU} = 0.75T$ Speedup limited to 1.33

- Improving performance

- Current system

- * Execution time – 10 sec
 - * Clock speed – 400 MHz

- New system

- * Execution time – 6 sec
 - * Clock speed – ?
 - * Number of clock cycles – 1.2 times current system

- Compute the number of clock cycles for current system

- *

$$\text{CPU time} = \frac{\text{CPU clock cycles for program}}{\text{Clock rate}}$$
$$10\text{sec} = \frac{\text{CPU clock cycles for program}}{400 \times 10^6 \text{cps}}$$

- * CPU clock cycles for program = 4000×10^6

Problem

- Compute the clock speed for new system

*

$$\text{CPU time} = \frac{\text{CPU clock cycles for program}}{\text{Clock rate}}$$
$$6\text{sec} = \frac{1.2 \times 4000 \times 10^6}{\text{Clock rate}}$$

*

$$\text{Clock rate} = \frac{1.2 \times 4000 \times 10^6}{6}$$
$$= 800 \times 10^6$$
$$= 800\text{MHz}$$

- Basic performance equation

$$\text{CPU time} = \text{Instruction count} \times \text{CPI} \times \text{Clock cycle time}$$

or

$$\text{CPU time} = \frac{\text{Instruction count} \times \text{CPI}}{\text{Clock rate}}$$

$$\text{CPU clock cycles} = \sum_{i=1}^n (\text{CPI}_i \times C_i)$$

- * C_i is the number of instructions of class i
- * CPI_i is the average number of cycles per instruction for class i
- * n is the number of instruction classes

Code sequence	Number of instructions		
	A	B	C
c_1	2	1	2
c_2	4	1	1

Find out the number of instructions for each code sequence, the faster code sequence, and CPI for each code sequence

Number of instructions in sequence $c_1 = 2 + 1 + 2 = 5$

Number of instructions in sequence $c_2 = 4 + 1 + 1 = 6$

Obviously, sequence c_1 executes fewer instructions

CPU clock cycles₁ = $(2 \times 1) + (1 \times 2) + (2 \times 3) = 2 + 2 + 6 = 10$

CPU clock cycles₂ = $(4 \times 1) + (1 \times 2) + (1 \times 3) = 4 + 2 + 3 = 9$

Code sequence c_2 is faster

$$CPI = \frac{\text{CPU clock cycles}}{\text{Instruction count}}$$

$$CPI_1 = \frac{10}{5} = 2$$

$$CPI_2 = \frac{9}{6} = 1.5$$

Bench mark sample with problem

- A program runs in 100 sec on a machine, with multiply operations taking up 80 seconds of this time. How much does the speed of multiplication need to improve to get a five-fold increase in code execution?

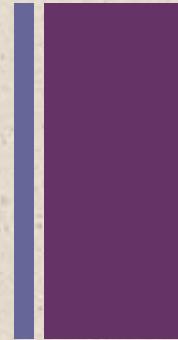
$$\text{Execution time after improvement} = \frac{\text{Execution time affected by improvement}}{\text{Amount of improvement}} + \text{Execution time unaffected}$$
$$\frac{100}{5} = \frac{80}{n} + (100 - 80)$$
$$20 = \frac{80}{n} + 20$$
$$0 = \frac{80}{n}$$

There is no amount by which we can improve the performance of multiply to realize a five-fold increase in overall performance

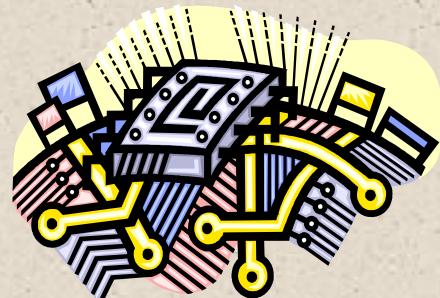
- This is *Amdahl's Law* in computing, or the law of diminishing returns in everyday life
- Opportunity of improvement is affected by how many time the event occurs



Arithmetic & Logic Unit (ALU)



- Part of the computer that actually performs arithmetic and logical operations on data
- All of the other elements of the computer system are there mainly to bring data into the ALU for it to process and then to take the results back out
- Based on the use of simple digital logic devices that can store binary digits and perform simple Boolean logic operations



ALU Inputs and Outputs

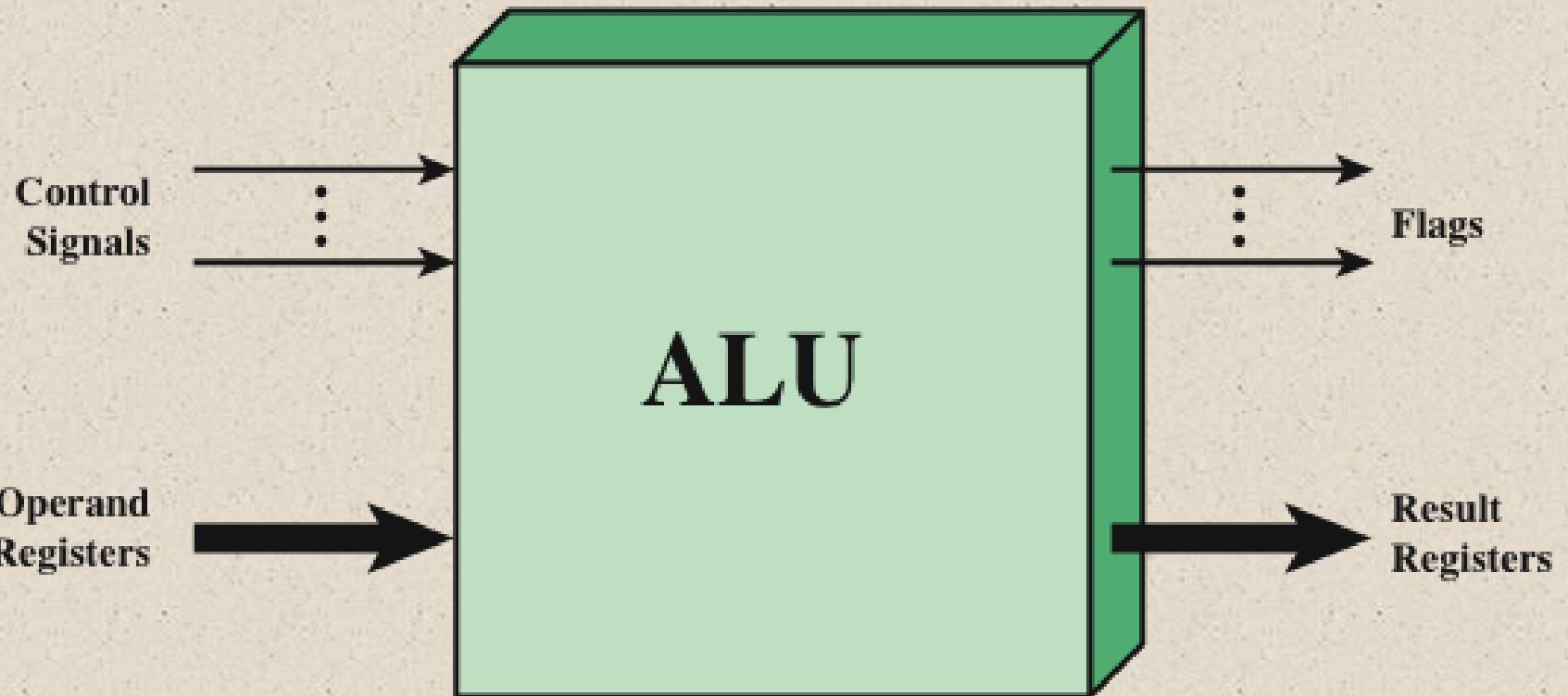


Figure 10.1 ALU Inputs and Outputs

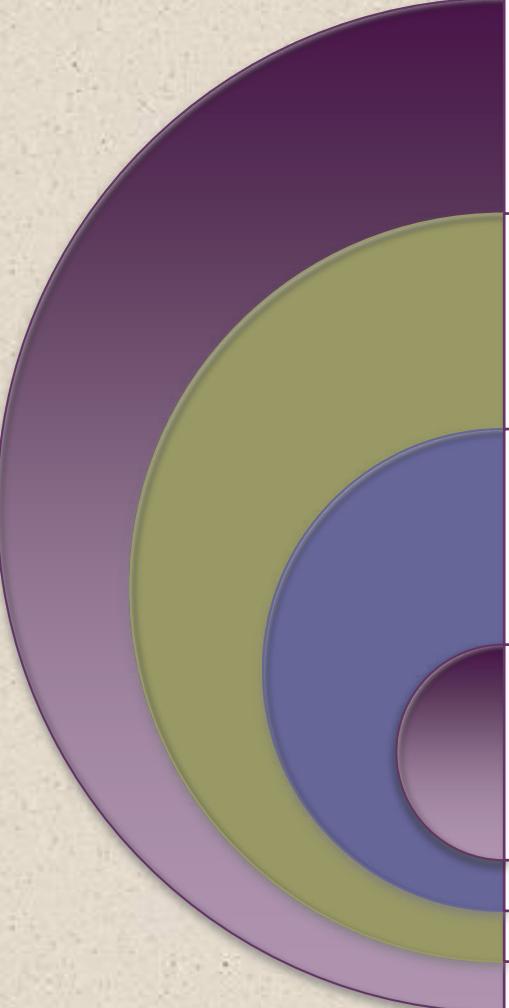


Integer Representation



- In the binary number system arbitrary numbers can be represented with:
 - The digits zero and one
 - The minus sign (for negative numbers)
 - The period, or *radix point* (for numbers with a fractional component)
- For purposes of computer storage and processing we do not have the benefit of special symbols for the minus sign and radix point
- Only binary digits (0,1) may be used to represent numbers

Sign-Magnitude Representation



There are several alternative conventions used to represent negative as well as positive integers

- All of these alternatives involve treating the most significant (leftmost) bit in the word as a sign bit
- If the sign bit is 0 the number is positive
- If the sign bit is 1 the number is negative

Sign-magnitude representation is the simplest form that employs a sign bit

Drawbacks:

- Addition and subtraction require a consideration of both the signs of the numbers and their relative magnitudes to carry out the required operation
- There are two representations of 0

Because of these drawbacks, sign-magnitude representation is rarely used in implementing the integer portion of the ALU



Twos Complement Representation

- Uses the most significant bit as a sign bit
- Differs from sign-magnitude representation in the way that the other bits are interpreted

Range	-2_{n-1} through $2_{n-1} - 1$
Number of Representations of Zero	One
Negation	Take the Boolean complement of each bit of the corresponding positive number, then add 1 to the resulting bit pattern viewed as an unsigned integer.
Expansion of Bit Length	Add additional bit positions to the left and fill in with the value of the original sign bit.
Overflow Rule	If two numbers with the same sign (both positive or both negative) are added, then overflow occurs if and only if the result has the opposite sign.
Subtraction Rule	To subtract B from A , take the twos complement of B and add it to A .

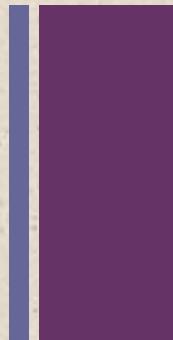
15-July-2018 | Arithmetic | **Table 10.1 Characteristics of Twos Complement Representation and Arithmetic**

Alternative Representations for 4-Bit Integers

Decimal Representation	Sign-Magnitude Representation	Twos Complement Representation	Biased Representation
+8	—	—	1111
+7	0111	0111	1110
+6	0110	0110	1101
+5	0101	0101	1100
+4	0100	0100	1011
+3	0011	0011	1010
+2	0010	0010	1001
+1	0001	0001	1000
+0	0000	0000	0111
-0	1000	—	—
-1	1001	1111	0110
-2	1010	1110	0101
-3	1011	1101	0100
-4	1100	1100	0011
-5	1101	1011	0010
-6	1110	1010	0001
-7	1111	1001	0000
-8	—	1000	—

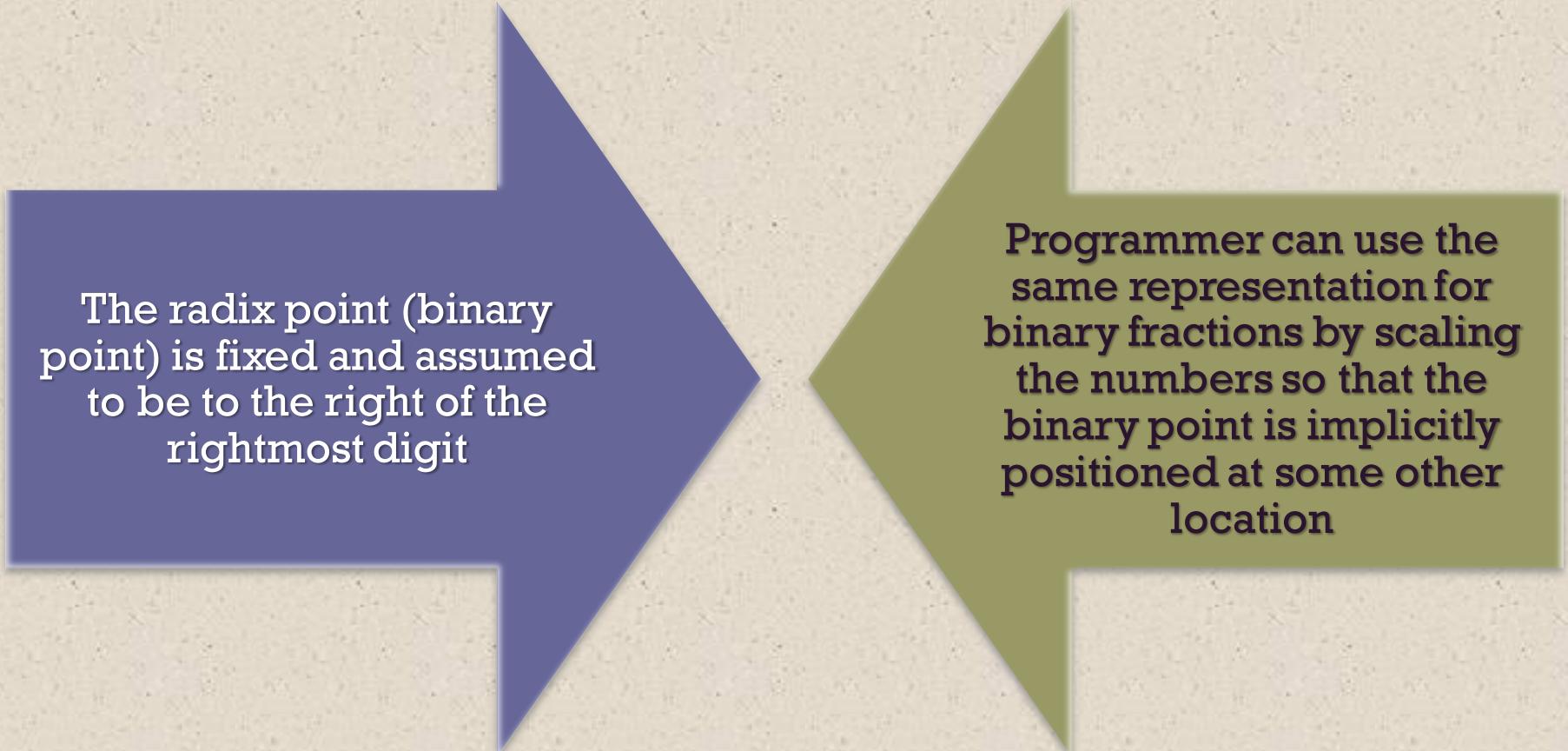


Range Extension



- Range of numbers that can be expressed is extended by increasing the bit length
- In sign-magnitude notation this is accomplished by moving the sign bit to the new leftmost position and fill in with zeros
- This procedure will not work for twos complement negative integers
 - Rule is to move the sign bit to the new leftmost position and fill in with copies of the sign bit
 - For positive numbers, fill in with zeros, and for negative numbers, fill in with ones
 - This is called *sign extension*

Fixed-Point Representation

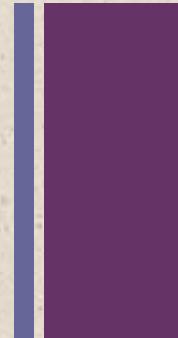


The radix point (binary point) is fixed and assumed to be to the right of the rightmost digit

Programmer can use the same representation for binary fractions by scaling the numbers so that the binary point is implicitly positioned at some other location



Negation



- Twos complement operation
 - Take the Boolean complement of each bit of the integer (including the sign bit)
 - Treating the result as an unsigned binary integer, add 1

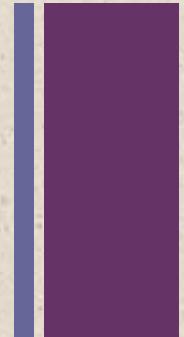
$$\begin{array}{r} +18 = 00010010 \text{ (twos complement)} \\ \text{bitwise complement} = 11101101 \\ + \quad \quad \quad 1 \\ \hline 11101110 = -18 \end{array}$$

- The negative of the negative of that number is itself:

$$\begin{array}{r} -18 = 11101110 \text{ (twos complement)} \\ \text{bitwise complement} = 00010001 \\ + \quad \quad \quad 1 \\ \hline 00010010 = +18 \end{array}$$

+

Negation Special Case 1



0 = 00000000 (twos complement)

Bitwise complement = 11111111

Add 1 to LSB
+ _____ 1

Result 100000000

Overflow is ignored, so:

$$-0 = 0$$



Negation Special Case 2

-128 = 10000000 (twos complement)

Bitwise complement = 0111111

Add 1 to LSB + 1

Result 10000000

So:

$-(-128) = -128 \times$

Monitor MSB (sign bit)

It should change during negation



Addition

$\begin{array}{r} 1001 = -7 \\ +0101 = 5 \\ \hline 1110 = -2 \end{array}$	$\begin{array}{r} 1100 = -4 \\ +0100 = 4 \\ \hline 10000 = 0 \end{array}$
(a) $(-7) + (+5)$	(b) $(-4) + (+4)$
$\begin{array}{r} 0011 = 3 \\ +0100 = 4 \\ \hline 0111 = 7 \end{array}$	$\begin{array}{r} 1100 = -4 \\ +1111 = -1 \\ \hline 11011 = -5 \end{array}$
(c) $(+3) + (+4)$	(d) $(-4) + (-1)$
$\begin{array}{r} 0101 = 5 \\ +0100 = 4 \\ \hline 1001 = \text{Overflow} \end{array}$	$\begin{array}{r} 1001 = -7 \\ +1010 = -6 \\ \hline 10011 = \text{Overflow} \end{array}$
(e) $(+5) + (+4)$	(f) $(-7) + (-6)$



Overflow

OVERFLOW RULE:

If two numbers are added, and they are both positive or both negative, then overflow occurs if and only if the result has the opposite sign.

Rule



Subtraction

SUBTRACTION RULE:

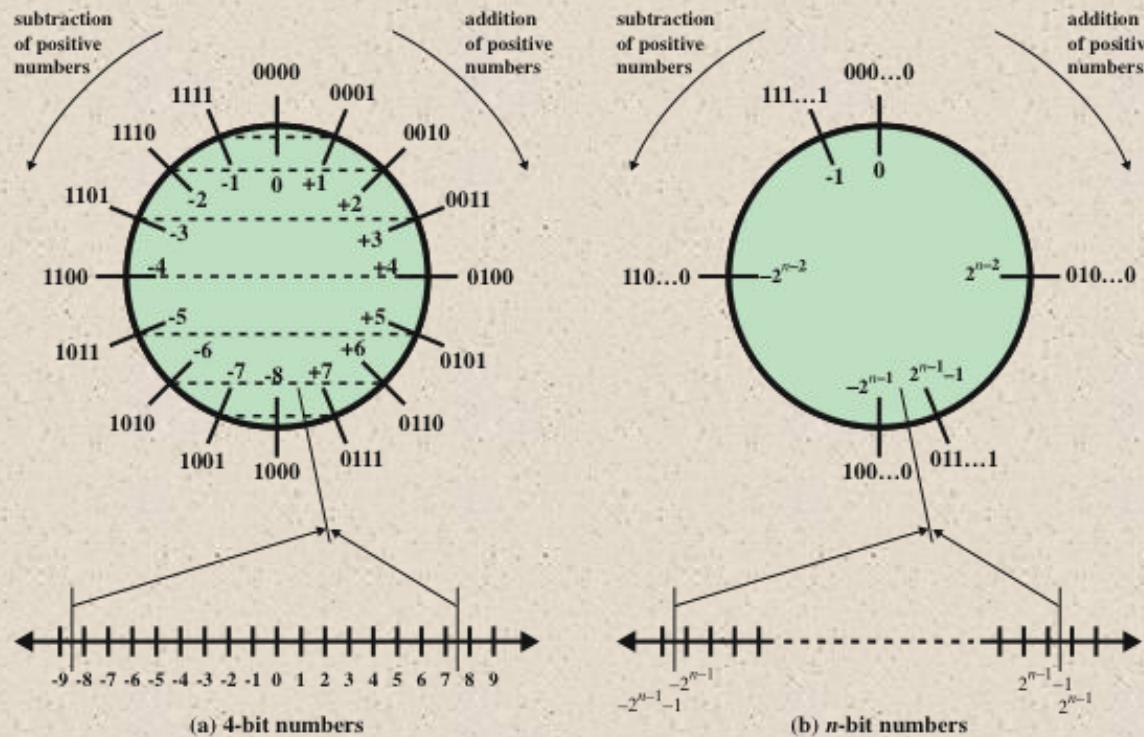
To subtract one number (subtrahend) from another (minuend), take the twos complement (negation) of the subtrahend and add it to the minuend.

Rule

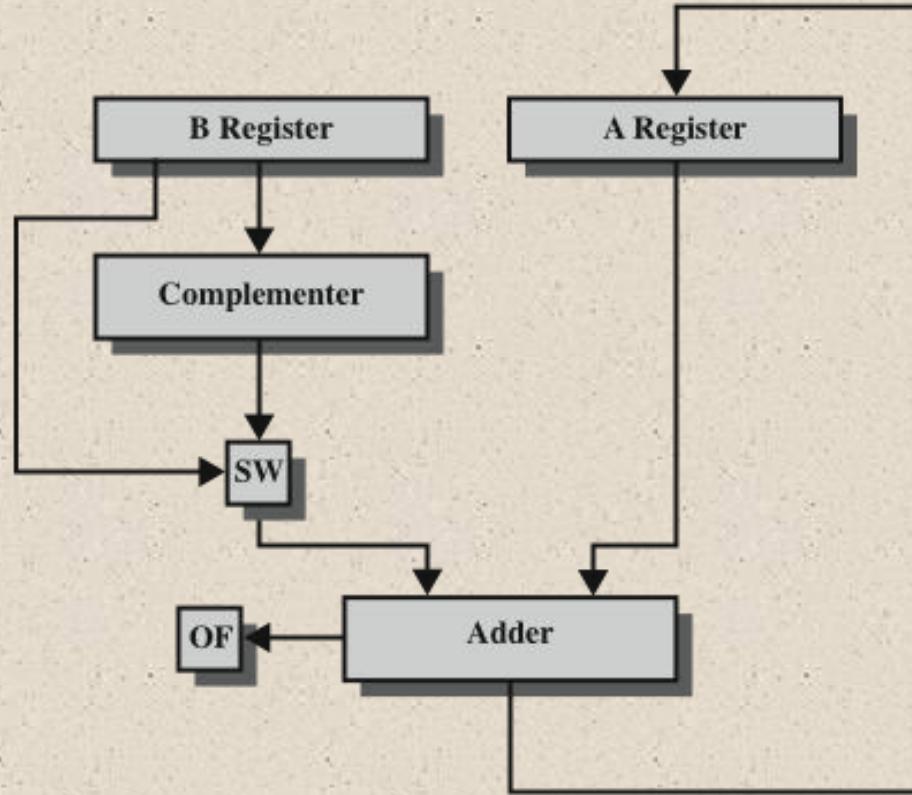
Subtraction

$\begin{array}{r} 0010 = 2 \\ +1001 = -7 \\ \hline 1011 = -5 \end{array}$	$\begin{array}{r} 0101 = 5 \\ +1110 = -2 \\ \hline 10011 = 3 \end{array}$
(a) $M = 2 = 0010$ $S = 7 = 0111$ $-S = 1001$	(b) $M = 5 = 0101$ $S = 2 = 0010$ $-S = 1110$
$\begin{array}{r} 1011 = -5 \\ +1110 = -2 \\ \hline 11001 = -7 \end{array}$	$\begin{array}{r} 0101 = 5 \\ +0010 = 2 \\ \hline 0111 = 7 \end{array}$
(c) $M = -5 = 1011$ $S = 2 = 0010$ $-S = 1110$	(d) $M = 5 = 0101$ $S = -2 = 1110$ $-S = 0010$
$\begin{array}{r} 0111 = 7 \\ +0111 = 7 \\ \hline 1110 = \text{Overflow} \end{array}$	$\begin{array}{r} 1010 = -6 \\ +1100 = -4 \\ \hline 10110 = \text{Overflow} \end{array}$
(e) $M = 7 = 0111$ $S = -7 = 1001$ $-S = 0111$	(f) $M = -6 = 1010$ $S = 4 = 0100$ $-S = 1100$

Geometric Depiction of Twos Complement Integers



Hardware for Addition and Subtraction



OF = overflow bit

SW = Switch (select addition or subtraction)

15-Jun-2021_arithmetic1 **Figure 10.6 Block Diagram of Hardware for Addition and Subtraction**

Subtraction using r's complement:

To find $M-N$ in base r , we add $M + r$'s complement of N

Result is $M + (r^n - N)$

1) If $M > N$ then result is $M - N + r^n$ (r^n is an end carry and can be neglected).

2) If $M < N$ then result is $r^n - (N - M)$ which is the r's complement of the result.



Signed Number Representation

■ *Signed Magnitude Method*

- $N = \pm (a_{n-1} \dots a_0.a_{-1} \dots a_{-m})_r$ is represented as

$$N = (sa_{n-1} \dots a_0.a_{-1} \dots a_{-m})_{rsm}, \quad (1.6)$$

where $s = 0$ if N is positive and $s = r - 1$ otherwise.

- $N = -(15)_{10}$
- In binary: $N = -(15)_{10} = -(1111)_2 = (1, 1111)_{2sm}$
- In decimal: $N = -(15)_{10} = (9, 15)_{10sm}$

■ *Complementary Number Systems*

- *Radix complements* (r 's complements)

$$[N]_r = r^n - (N)_r \quad (1.7)$$

where n is the number of digits in $(N)_r$.

- *Positive full scale*: $r^{n-1} - 1$
- *Negative full scale*: $-r^n - 1$
- *Diminished radix complements* ($r-1$'s complements)

$$[N]_{r-1} = r_n - (N)_r - 1$$

Radix Complement Number Systems

(1)

- Two's complement of $(N)_2 = (101001)_2$

$$[N]_2 = 2^6 - (101001)_2 = (1000000)_2 - (101001)_2 = (010111)_2$$

- $(N)_2 + [N]_2 = (101001)_2 + (010111)_2 = (1000000)_2$

If we discard the carry, $(N)_2 + [N]_2 = 0$.

Hence, $[N]_2$ can be used to represent $-(N)_2$.

- $[(N)_2]_2 = [(010111)_2]_2 = (1000000)_2 - (010111)_2 = (101001)_2 = (N)_2$.
- Two's complement of $(N)_2 = (1010)_2$ for $n = 6$

$$[N]_2 = (1000000)_2 - (1010)_2 = (110110)_2.$$



Radix Complement Number Systems (2)

■ **Algorithm 1.4 Find $[N]_r$ given $(N)_r$.**

- Copy the digits of N , beginning with the LSD and proceeding toward the MSD until the first nonzero digit, a_i , has been reached
- Replace a_i with $r - a_i$.
- Replace each remaining digit a_j , of N by $(r - 1) - a_j$ until the MSD has been replaced.

■ **Example:** 10's complement of $(56700)_{10}$ is $(43300)_{10}$

■ **Example:** 2's complement of $(10100)_2$ is $(01100)_2$.

■ **Example:** 2's complement of $N = (10110)_2$ for $n = 8$.

- Put three zeros in the MSB position and apply algorithm 1.4
- $N = 00010110$
- $[N]_2 = (11101010)_2$

■ The same rule applies to the case when N contains a radix point.



Radix Complement Number Systems (3)

- **Algorithm 1.5 Find $[N]_r$ given $(N)_r$.**
 - First replace each digit, a_k , of $(N)_r$ by $(r - 1) - a_k$ and then add 1 to the resultant.
- For binary numbers ($r = 2$), complement each digit and add 1 to the result.

- **Example:** Find 2's complement of $N = (01100101)_2$.

$$N = 01100101$$

$$\begin{array}{r} 10011010 \\ \text{Complement the bits} \\ +1 \text{ Add 1} \end{array}$$

$$[N]_2 = (10011011)_{10}$$

- **Example:** Find 10's complement of $N = (40960)_{10}$

$$N = 40960$$

$$\begin{array}{r} 59039 \\ \text{Complement the bits} \\ +1 \text{ Add 1} \end{array}$$

$$[N]_2 = (59040)_{10}$$



Radix Complement Number Systems (4)

■ ***Two's complement number system :***

$$0 \leq N \leq 2^{n-1} - 1$$

■ Positive number :

- $N = +(a_{n-2}, \dots, a_0)_2 = (0, a_{n-2}, \dots, a_0)_{2cns}$,

where

■ Negative number:

- $N = (a_{n-1}, a_{n-2}, \dots, a_0)_2$

- $-N = [a_{n-1}, a_{n-2}, \dots, a_0]_2$ (two's complement of N),

where

■ ***Example:*** Two's complement number system representation of $\pm (N)_2$

when $(N)_2 = (1011001)_2$ for $n = 8$:

- $+(N)_2 = (0, 1011001)_{2cns}$

15-Jun-2021 - arithmetic
 $(N)_2 = [+(N)_2]_2 = [0, 1011001]_2 = (1, 0100111)_{2cns}$



Radix Complement Number Systems (5)

■ **Example:** Two's complement number system representation of $-(18)_{10}$, $n = 8$:

- $+(18)_{10} = (0, 0010010)_{2cns}$
- $-(18)_{10} = [0, 0010010]_2 = (1, 1101110)_{2cns}$

■ **Example:** Decimal representation of $N = (1, 1101000)_{2cns}$

- $N = (1, 1101000)_{2cns} = -[1, 1101000]_2 = -(0, 0011000)_{2cns} = -(24)_2$.



Radix Complement Arithmetic (1)

- Radix complement number systems are used to convert subtraction to addition, which reduces hardware requirements (only adders are needed).
- $A - B = A + (-B)$ (add r 's complement of B to A)
- Range of numbers in two's complement number system:
 $-2^{n-1} \leq N \leq 2^{n-1} - 1$, where n is the number of bits.
- $2^{n-1} - 1 = (0, 11 \dots 1)_{2\text{cns}}$ and $-2^{n-1} = (1, 00 \dots 0)_{2\text{cns}}$
- If the result of an operation falls outside the range, an ***overflow condition*** is said to occur and the result is not valid.
- Consider three cases:
 - $A = B + C$,
 - $A = B - C$,
 - $A = -B - C$,



Radix Complement Arithmetic (2)

■ Case 1: $A = B + C$

- $(A)_2 = (B)_2 + (C)_2$
- If $A > 2^{n-1} - 1$ (**overflow**), it is detected by the n th bit, which is set to 1.

■ **Example:** $(7)_{10} + (4)_{10} = ?$ using 5-bit two's complement arithmetic.

- $+ (7)_{10} = +(0111)_2 = (0, 0111)_{2cns}$
- $+ (4)_{10} = +(0100)_2 = (0, 0100)_{2cns}$
- $(0, 0111)_{2cns} + (0, 0100)_{2cns} = (0, 1011)_{2cns} = +(1011)_2 = +(11)_{10}$
- No overflow.

■ **Example:** $(9)_{10} + (8)_{10} = ?$

- $+ (9)_{10} = +(1001)_2 = (0, 1001)_{2cns}$
- $+ (8)_{10} = +(1000)_2 = (0, 1000)_{2cns}$
- $(0, 1001)_{2cns} + (0, 1000)_{2cns} = (1, 0001)_{2cns}$ (**overflow**)



Radix Complement Arithmetic (3)

■ Case 2: $A = B - C$

- $A = (B)_2 + (-C)_2 = (B)_2 + [C]_2 = (B)_2 + 2^n - (C)_2 = 2^n + (B - C)_2$
- If $B \geq C$, then $A \geq 2^n$ and the carry is discarded.
- So, $(A)_2 = (B)_2 + [C]_2$ | carry discarded
- If $B < C$, then $A = 2^n - (C - B)_2 = [C - B]_2$ or $A = -(C - B)_2$ (no carry in this case).
- No overflow for Case 2.

- **Example:** $(14)_{10} - (9)_{10} = ?$
 - Perform $(14)_{10} + (-9)_{10}$
 - $(14)_{10} = +(1110)_2 = (0, 1110)_{2cns}$
 - $-(9)_{10} = -(1001)_2 = (1, 0111)_{2cns}$
 - $(14)_{10} - (9)_{10} = (0, 1110)_{2cns} + (1, 0111)_{2cns} = (0, 0101)_{2cns} + \text{carry}$
 $= +(0101)_2 = +(5)_{10}$



Radix Complement Arithmetic (4)

- **Example:** $(9)_{10} - (14)_{10} = ?$
 - Perform $(9)_{10} + (-14)_{10}$
 - $(9)_{10} = +(1001)_2 = (0, 1001)_{2cns}$
 - $-(14)_{10} = -(1110)_2 = (1, 0010)_{2cns}$
 - $(9)_{10} - (14)_{10} = (0, 1001)_{2cns} + (1, 0010)_{2cns} = (1, 1011)_{2cns}$
 $= -(0101)_2 = -(5)_{10}$

- **Example:** $(0,0100)_{2cns} - (1,0110)_{2cns} = ?$
 - Perform $(0,0100)_{2cns} + (-1,0110)_{2cns}$
 - $-(1,0110)_{2cns} = \text{two's complement of } (1,0110)_{2cns}$
 $= (0,1010)_{2cns}$
 - $(0,0100)_{2cns} - (1,0110)_{2cns} = (0,0100)_{2cns} + (0,1010)_{2cns}$
 $= (0,1110)_{2cns} = +(1110)_2 = +(14)_{10}$
 - $+(4)_{10} - (-(10)_{10}) = +(14)_{10}$



Radix Complement Arithmetic (5)

■ Case 3: $A = -B - C$

- $A = [B]_2 + [C]_2 = 2^n - (B)_2 + 2^n - (C)_2 = 2^n + 2^n - (B + C)_2 = 2^n + [B + C]_2$
- The carry bit (2^n) is discarded.
- An overflow can occur, in which case the sign bit is 0.

■ Example: $-(7)_{10} - (8)_{10} = ?$

- Perform $-(7)_{10} + -(8)_{10}$
- $-(7)_{10} = -(0111)_2 = (1, 1001)_{2cns}$, $-(8)_{10} = -(1000)_2 = (1, 1000)_{2cns}$
- $-(7)_{10} - (8)_{10} = (1, 1001)_{2cns} + (1, 1000)_{2cns} = (1, 0001)_{2cns} + \text{carry}$
 $= -(1111)_2 = -(15)_{10}$

■ Example: $-(12)_{10} - (5)_{10} = ?$

- Perform $-(12)_{10} + -(5)_{10}$
- $-(12)_{10} = -(1100)_2 = (1, 0100)_{2cns}$, $-(5)_{10} = -(0101)_2 = (1, 1011)_{2cns}$
- $-(7)_{10} - (8)_{10} = (1, 0100)_{2cns} + (1, 1011)_{2cns} = (0, 1111)_{2cns} + \text{carry}$

15-Jun-2021 | arithmetic1 | Overflow, because the sign bit is 0.



Radix Complement Arithmetic (6)

■ **Example:** $A = (25)_{10}$ and $B = -(46)_{10}$

- $A = +(25)_{10} = (0,0011001)_{2cns}$, $-A = (1,1100111)_{2cns}$
- $B = -(46)_{10} = -(0,0101110)_2 = (1,1010010)_{2cns}$, $-B = (0,0101110)_{2cns}$

- $A + B = (0,0011001)_{2cns} + (1,1010010)_{2cns} = (1,1101011)_{2cns} = -(21)_{10}$
- $A - B = A + (-B) = (0,0011001)_{2cns} + (0,0101110)_{2cns}$
 $= (0,1000111)_{2cns} = +(71)_{10}$
- $B - A = B + (-A) = (1,1010010)_{2cns} + (1,1100111)_{2cns}$
 $= (1,0111001)_{2cns} + carry = -(0,1000111)_{2cns} = -(71)_{10}$
- $-A - B = (-A) + (-B) = (1,1100111)_{2cns} + (0,0101110)_{2cns}$
 $= (0,0010101)_{2cns} + carry = +(21)_{10}$
- Note: Carry bit is discarded.



Radix Complement Arithmetic (7)

■ Summary

Case	Carry	Sign Bit	Condition	Overflow ?
$B + C$	0	0	$B + C \leq 2^{n-1} - 1$	No
	0	1	$B + C > 2^{n-1} - 1$	Yes
$B - C$	1	0	$B \leq C$	No
	0	1	$B > C$	No
$-B - C$	1	1	$-(B + C) \geq -2^{n-1}$	No
	1	0	$-(B + C) < -2^{n-1}$	Yes

- When numbers are represented using two's complement number system:
 - Addition: Add two numbers.
 - Subtraction: Add two's complement of the subtrahend to the minuend.
 - Carry bit is discarded, and overflow is detected as shown above.
 - Radix complement arithmetic can be used for any radix.



Diminished Radix Complement Number systems (1)

- **Diminished radix complement** $[N]_{r-1}$ of a number $(N)_r$ is:

$$[N]_{r-1} = r^n - (N)_r - 1 \quad (1.10)$$

- **One's complement** ($r = 2$):

$$[N]_{2-1} = 2^n - (N)_2 - 1 \quad (1.11)$$

- **Example:** One's complement of $(01100101)_2$

$$\begin{aligned}[N]_{2-1} &= 2^8 - (01100101)_2 - 1 \\&= (10000000)_2 - (01100101)_2 - (00000001)_2 \\&= (10011011)_2 - (00000001)_2 \\&= (10011010)_2\end{aligned}$$



Diminished Radix Complement Number systems (2)

- ***Example:*** Nine's complement of (40960)

$$\begin{aligned}
 [N]_{2-1} &= 10^5 - (40960)_{10} - 1 \\
 &= (100000)_{10} - (40960)_{10} - (00001)_{10} \\
 &= (59040)_{10} - (00001)_{10} \\
 &= (59039)_{10}
 \end{aligned}$$

- ***Algorithm 1.6 Find $[N]_{r-1}$ given $(N)_r$.***

Replace each digit a_i of $(N)_r$ by $r - 1 - a$. Note that when $r = 2$, this simplifies to complementing each individual bit of $(N)_r$.

- Radix complement and diminished radix complement of a number (N) :

$$[N]_r = [N]_{r-1} + 1 \quad (1.12)$$



Diminished Radix Complement Arithmetic (1)

- Operands are represented using diminished radix complement number system.
- The carry, if any, is added to the result (**end-around carry**).
- **Example:** Add $+(1001)_2$ and $-(0100)_2$.
One's complement of $+(1001) = 01001$
One's complement of $-(0100) = 11011$
 $01001 + 11011 = 100100$ (carry)
Add the carry to the result: correct result is 00101.
- **Example:** Add $+(1001)_2$ and $-(1111)_2$.
One's complement of $+(1001) = 01001$
One's complement of $-(1111) = 10000$
 $01001 + 10000 = 11001$ (no carry, so this is the correct result).



Diminished Radix Complement Arithmetic (2)

- **Example:** Add $-(1001)_2$ and $-(0011)_2$.

One's complement of the operands are: 10110 and 11100

$$10110 + 11100 = 110010 \text{ (carry)}$$

Correct result is $10010 + 1 = 10011$.

- **Example:** Add $+(75)_{10}$ and $-(21)_{10}$.

Nine's complements of the operands are: 075 and 978

$$075 + 978 = 1053 \text{ (carry)}$$

Correct result is $053 + 1 = 054$

- **Example:** Add $+(21)_{10}$ and $-(75)_{10}$.

Nine's complements of the operands are: 021 and 924

$$021 + 924 = 945 \text{ (no carry, so this is the correct result).}$$

Example (3):

Using 2's complement, subtract $1010100 - 1000011$

$$\begin{array}{r} X \quad - \quad Y \\ \hline X \quad = \quad 1010100 \\ 2\text{'s complement of } Y \quad = \quad + \quad \underline{0111101} \\ \text{Sum} \quad = \quad \textcolor{red}{10010001} \\ \text{Discard end carry } 2^7 \quad = \quad - \quad \underline{10000000} \\ \text{Answer: } X - Y \quad = \quad \textcolor{red}{0010001} \end{array}$$

Example (4):

Using 2's complement, subtract $1000011 - 1010100$

$$\begin{array}{r} Y \quad - \quad X \\ \hline Y \quad = \quad 1000011 \\ 2\text{'s complement of } X \quad = \quad + \quad \underline{0101100} \\ \text{Sum} \quad = \quad \textcolor{red}{1101111} \\ \text{No end carry.} \\ \text{Answer: } Y - X - (\text{2's complement of } 1101111) \quad = \quad \textcolor{red}{-0010001} \end{array}$$

Example (5): Using 1's complement, subtract $X - Y = 1010100 - 1000011$

$$\begin{array}{r} X \quad = \quad 1010100 \\ 1\text{'s complement of } Y \quad = \quad + \quad \underline{0111100} \text{ (+1 End-around carry)} \\ \text{Sum} \quad = \quad \boxed{} \quad 10010000 \\ \qquad \qquad \qquad \longrightarrow \qquad \qquad \qquad \textcolor{red}{+1} \\ \text{Answer: } X - Y \quad = \quad \textcolor{red}{0010001} \end{array}$$

Example (6): Using 1's complement, subtract $Y - X$ $1000011 - 1010100$

$$\begin{array}{r} Y \quad = \quad 1000011 \\ 1\text{'s complement of } X \quad = \quad + \quad \underline{0101011} \\ \text{Sum} \quad = \quad \textcolor{red}{1101110} \\ \text{No end carry.} \\ \text{Answer: } Y - X - (\text{1's complement of } 1101110) \quad = \quad \textcolor{red}{-0010001} \end{array}$$



Multiplication

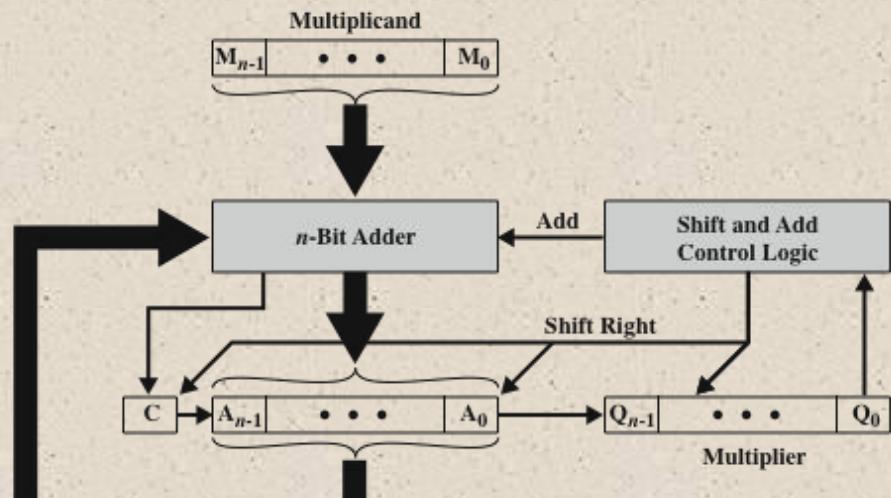
$$\begin{array}{r} 1011 \\ \times 1101 \\ \hline 1011 \\ 0000 \\ 1011 \\ 1011 \\ \hline 10001111 \end{array}$$

Multiplicand (11)
Multiplier (13)
Partial products
Product (143)

Figure 10.7 Multiplication of Unsigned Binary Integers



Hardware Implementation of Unsigned Binary Multiplication



(a) Block Diagram

C	A	Q	M		Initial Values
0	0000	1101	1011		
0	1011	1101	1011	Add	First Cycle
0	0101	1110	1011	Shift	
0	0010	1111	1011	Shift	Second Cycle
0	1101	1111	1011	Add	
0	0110	1111	1011	Shift	Third Cycle
1	0001	1111	1011	Add	
0	1000	1111	1011	Shift	Fourth Cycle

(b) Example from Figure 9.7 (product in A, Q)

Figure 10.8 Hardware Implementation of Unsigned Binary Multiplication

Flowchart for Unsigned Binary Multiplication

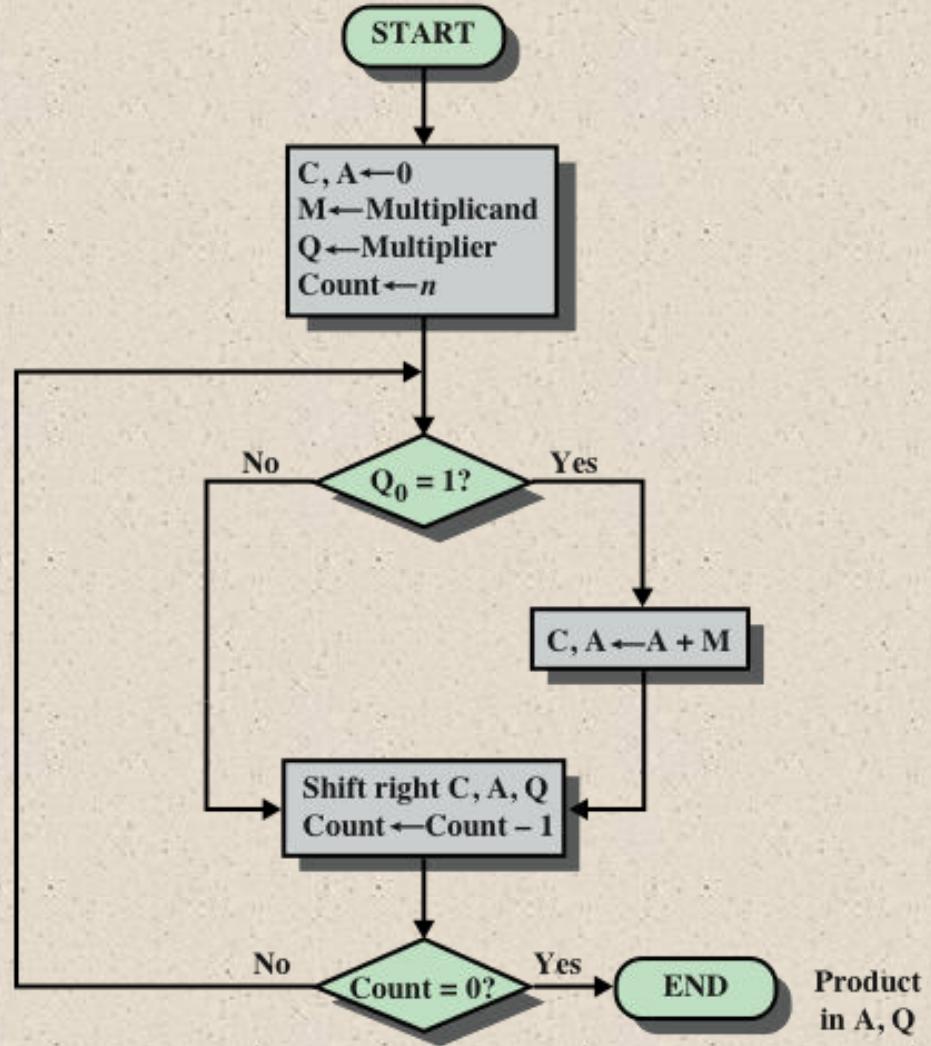


Figure 10.9 Flowchart for Unsigned Binary Multiplication

Twos Complement Multiplication

1011	
$\times 1101$	
<hr/>	
00001011	$1011 \times 1 \times 2^0$
00000000	$1011 \times 0 \times 2^1$
00101100	$1011 \times 1 \times 2^2$
<hr/> <u>01011000</u>	$1011 \times 1 \times 2^3$
10001111	

Figure 10.10 Multiplication of Two Unsigned 4-Bit Integers Yielding an 8-Bit Result

Comparison

$\begin{array}{r} 1001 \quad (9) \\ \times 0011 \quad (3) \\ \hline 00001001 \quad 1001 \times 2^0 \\ 00010010 \quad 1001 \times 2^1 \\ \hline 00011011 \quad (27) \end{array}$	$\begin{array}{r} 1001 \quad (-7) \\ \times 0011 \quad (3) \\ \hline 11111001 \quad (-7) \times 2^0 = (-7) \\ 11110010 \quad (-7) \times 2^1 = (-14) \\ \hline 11101011 \quad (-21) \end{array}$
---	--

(a) Unsigned integers

(b) Twos complement integers

Figure 10.11 Comparison of Multiplication of Unsigned and Twos Complement Integers

Booth's

Algorithm

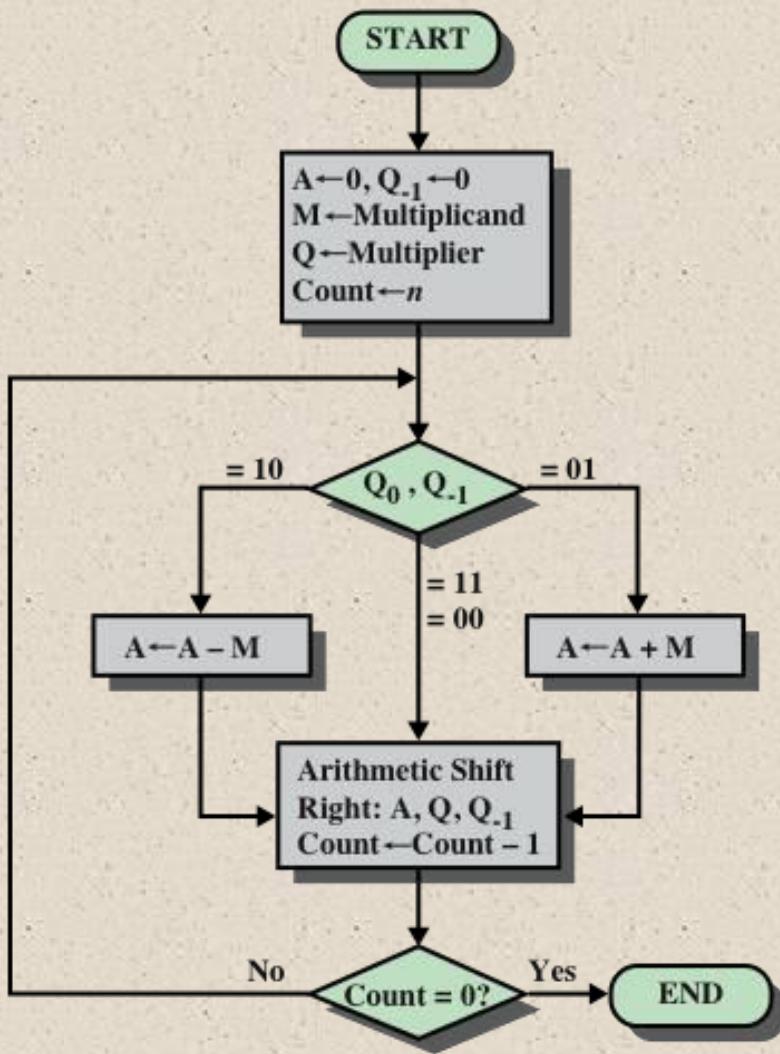


Figure 10.12. Booth's Algorithm for Twos Complement Multiplication

Example of Booth's Algorithm

A	Q	Q_{-1}	M		
0000	0011	0	0111	Initial Values	
1001	0011	0	0111	$A \leftarrow A - M$	First Cycle
1100	1001	1	0111	Shift	
1110	0100	1	0111	Shift	Second Cycle
0101	0100	1	0111	$A \leftarrow A + M$	Third Cycle
0010	1010	0	0111	Shift	
0001	0101	0	0111	Shift	Fourth Cycle

Examples Using Booth's Algorithm

$$\begin{array}{r} 0111 \\ \times 0011 \\ \hline 11111001 \\ 00000000 \\ \hline 000111 \\ \hline 00010101 \end{array} \quad (0) \quad (21)$$

$$\begin{array}{r} 0111 \\ \times 1101 \\ \hline 11111001 \\ 0000111 \\ \hline 111001 \\ \hline 11101011 \end{array} \quad (0) \quad (-21)$$

(a) $(7) \times (3) = (21)$

(b) $(7) \times (-3) = (-21)$

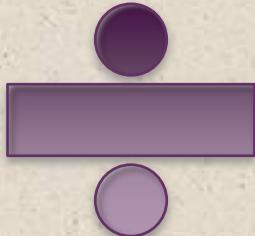
$$\begin{array}{r} 1001 \\ \times 0011 \\ \hline 00000111 \\ 00000000 \\ \hline 111001 \\ \hline 11101011 \end{array} \quad (0) \quad (-21)$$

$$\begin{array}{r} 1001 \\ \times 1101 \\ \hline 00000111 \\ 1111001 \\ 000111 \\ \hline 00010101 \end{array} \quad (0) \quad (21)$$

(c) $(-7) \times (3) = (-21)$

(d) $(-7) \times (-3) = (21)$

Figure 10.14 Examples Using Booth's Algorithm



Division

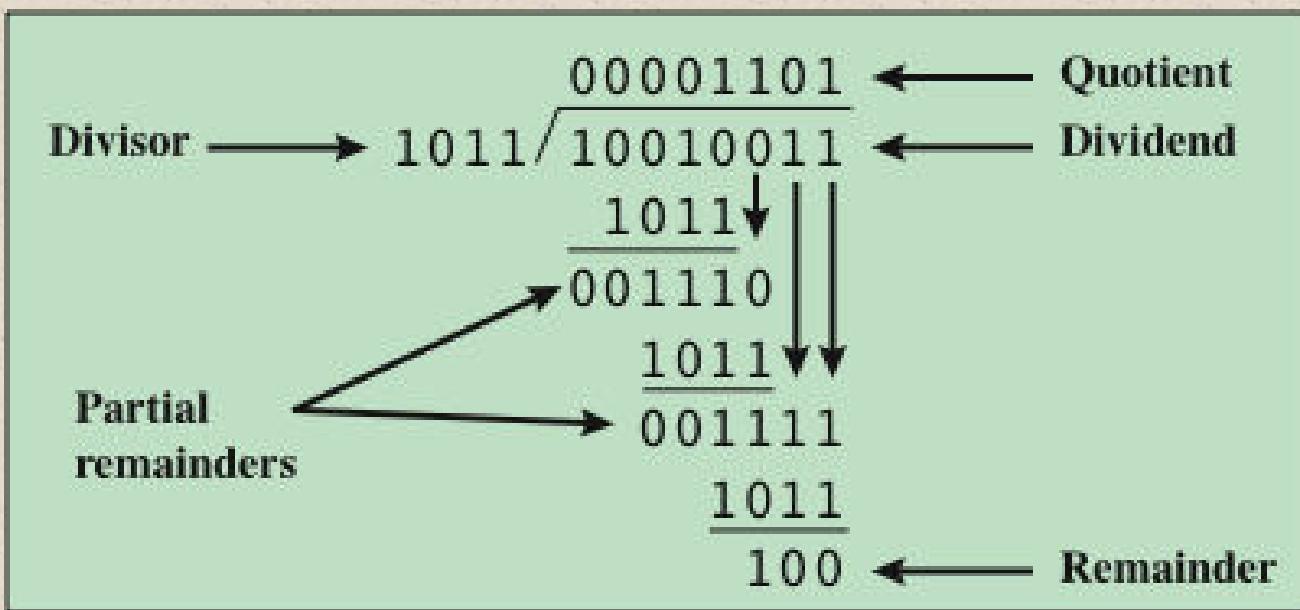


Figure 10.15 Example of Division of Unsigned Binary Integers

Flowchart for Unsigned Binary Division

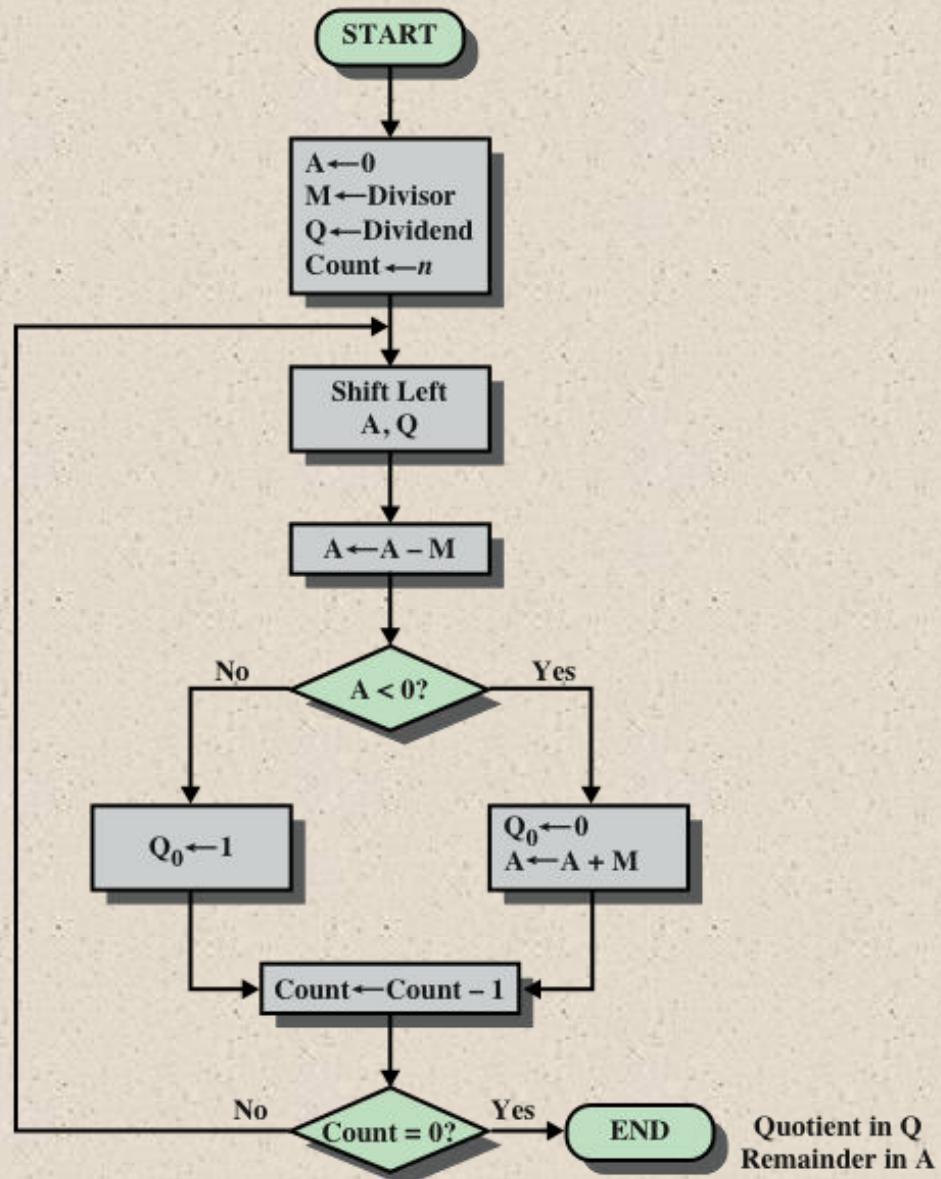


Figure 10.16 Flowchart for Unsigned Binary Division

Example of Restoring Twos Complement Division

A	Q	
0000	0111	Initial value
0000 <u>1101</u> 1101 0000	1110	Shift Use twos complement of 0011 for subtraction Subtract Restore, set $Q_0 = 0$
0001 <u>1101</u> 1110 0001	1100	Shift Subtract Restore, set $Q_0 = 0$
0011 <u>1101</u> 0000	1000	Shift
0001 <u>1101</u> 1110 0001	1001	Subtract, set $Q_0 = 1$
0001 <u>1101</u> 1110 0001	0010	Shift Subtract Restore, set $Q_0 = 0$

15-Jun-2021_aithafee Figure 10.17 Example of Restoring Twos Complement Division (7/3)



Floating-Point Representation Principles

- With a fixed-point notation it is possible to represent a range of positive and negative integers centered on or near 0
- By assuming a fixed binary or radix point, this format allows the representation of numbers with a fractional component as well
- Limitations:
 - Very large numbers cannot be represented nor can very small fractions
 - The fractional part of the quotient in a division of two large numbers could be lost

Typical 32-Bit Floating-Point Format



$$\begin{array}{lll} 1.1010001 \times 2^{10100} & = 0 & 10010011 \ 10100010000000000000000 \\ -1.1010001 \times 2^{10100} & = 1 & 10010011 \ 10100010000000000000000 \\ 1.1010001 \times 2^{-10100} & = 0 & 01101011 \ 10100010000000000000000 \\ -1.1010001 \times 2^{-10100} & = 1 & 01101011 \ 10100010000000000000000 \end{array} = \begin{array}{lll} 1.6328125 \times 2^{20} \\ -1.6328125 \times 2^{20} \\ 1.6328125 \times 2^{-20} \\ -1.6328125 \times 2^{-20} \end{array}$$

(b) Examples

The closest binary number to Y that can be stored by computer in 32 bits is:

$$(-1)^s \frac{J(\text{in binary})}{2^{23}} \times 2^P$$

where

$$s=0 \text{ if } y \leq 0 \text{ and } s=1 \text{ if } y > 1$$

$$\rightarrow z = |y|$$

$$\rightarrow P = \text{Floor}(\log_2 z) = \text{Floor}\left(\frac{\log z}{\log 2}\right)$$

$$J = \text{Round}(z \times 2^{23-P})$$

a) Convert the given IEEE 754 formatted 32-bit floating point number in to decimal

1 10111011 10110000000000000000000000

b) Define Normalization. Give two examples.

Give the flow chart for addition and subtraction of two floating-point numbers.

Show the IEEE 754 binary representation of the number $(-0.4375)_{\text{ten}}$ in single precision.



Floating-Point Significand



- The final portion of the word
 - Any floating-point number can be expressed in many ways
-

The following are equivalent, where the significand is expressed in binary form:

$$0.110 * 2^5$$

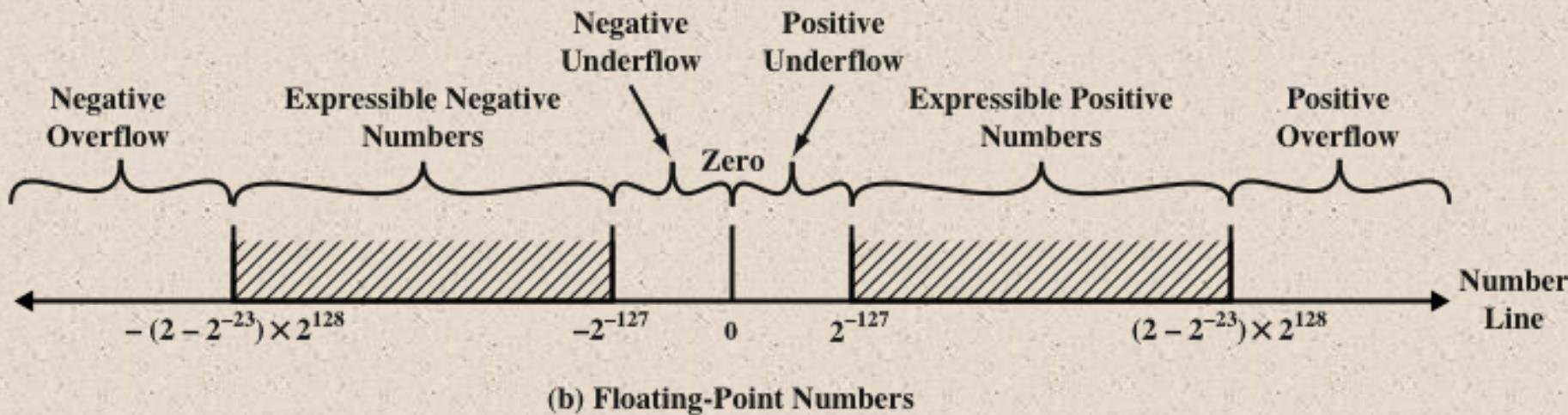
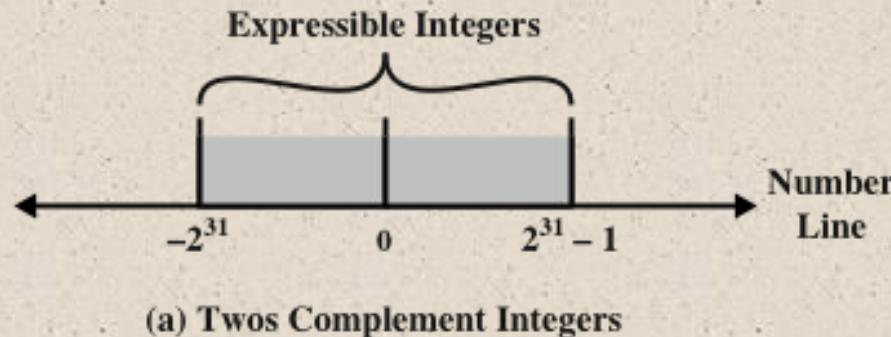
$$110 * 2^2$$

$$0.0110 * 2^6$$

- *Normal number*
 - The most significant digit of the significand is nonzero



Expressible Numbers



Density of Floating-Point Numbers

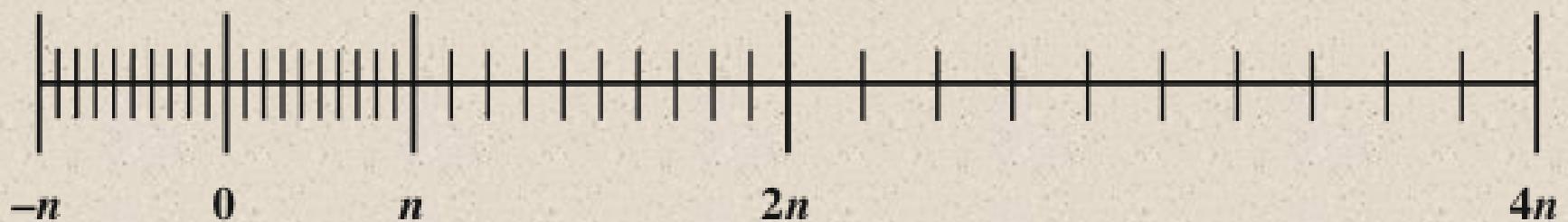


Figure 10.20 Density of Floating-Point Numbers

IEEE Standard 754

Most important floating-point representation is defined

Standard was developed to facilitate the portability of programs from one processor to another and to encourage the development of sophisticated, numerically oriented programs

Standard has been widely adopted and is used on virtually all contemporary processors and arithmetic coprocessors

IEEE 754-2008 covers both binary and decimal floating-point representations

IEEE 754-2008

- Defines the following different types of floating-point formats:
- Arithmetic format
 - All the mandatory operations defined by the standard are supported by the format. The format may be used to represent floating-point operands or results for the operations described in the standard.
- Basic format
 - This format covers five floating-point representations, three binary and two decimal, whose encodings are specified by the standard, and which can be used for arithmetic. At least one of the basic formats is implemented in any conforming implementation.
- Interchange format
 - A fully specified, fixed-length binary encoding that allows data interchange between different platforms and that can be used for storage.



IEEE 754 Formats

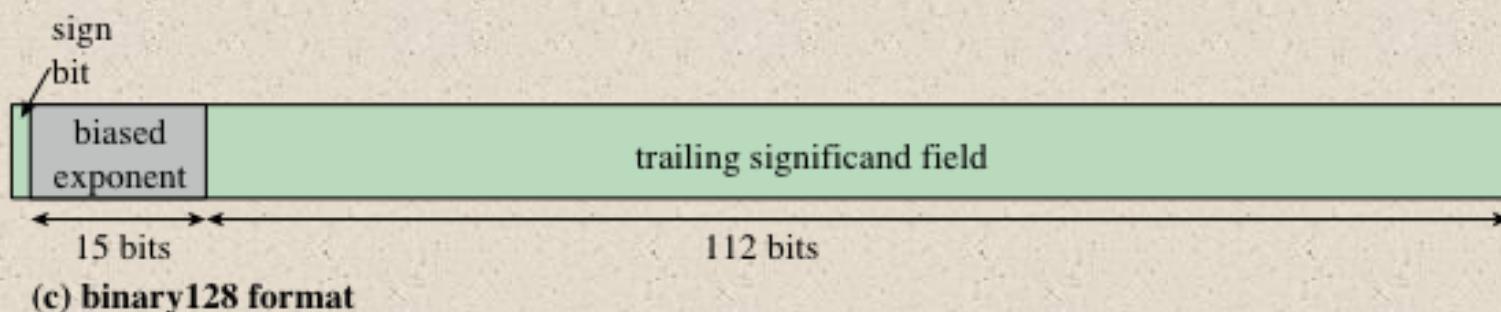
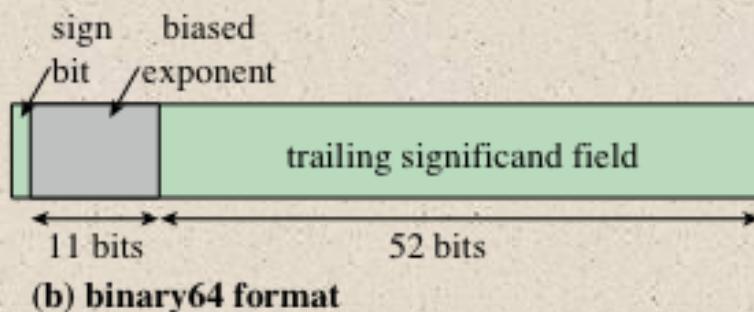
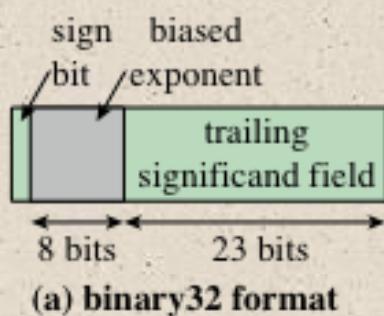


Figure 10.21 IEEE 754 Formats

Table 10.3

IEEE 754

Format
Parameters

Parameter	Format		
	binary32	binary64	binary128
Storage width (bits)	32	64	128
Exponent width (bits)	8	11	15
Exponent bias	127	1023	16383
Maximum exponent	127	1023	16383
Minimum exponent	-126	-1022	-16382
Approx normal number range (base 10)	10_{-38} , 10_{+38}	10_{-308} , 10_{+308}	10_{-4932} , 10_{+4932}
Trailing significand width (bits)*	23	52	112
Number of exponents	254	2046	32766
Number of fractions	2_{23}	2_{52}	2_{112}
Number of values	$1.98 \times 2_{31}$	$1.99 \times 2_{63}$	$1.99 \times 2_{128}$
Smallest positive normal number	2_{-126}	2_{-1022}	2_{-16382}
Largest positive normal number	$2_{128} - 2_{104}$	$2_{1024} - 2_{971}$	$2_{16384} - 2_{16271}$
Smallest subnormal magnitude	2_{-149}	2_{-1074}	2_{-16494}



Additional Formats

Extended Precision Formats

- Provide additional bits in the exponent (extended range) and in the significand (extended precision)
- Lessens the chance of a final result that has been contaminated by excessive roundoff error
- Lessens the chance of an intermediate overflow aborting a computation whose final result would have been representable in a basic format
- Affords some of the benefits of a larger basic format without incurring the time penalty usually associated with higher

Extendable Precision Format

- Precision and range are defined under user control
- May be used for intermediate calculations but the standard places no constraint or format or length



Table 10.4

IEEE Formats

Format	Format Type		
	Arithmetic Format	Basic Format	Interchange Format
binary16			X
binary32	X	X	X
binary64	X	X	X
binary128	X	X	X
binary $\{k\}$ $(k = n \times 32 \text{ for } n > 4)$	X		X
decimal64	X	X	X
decimal128	X	X	X
decimal $\{k\}$ $(k = n \times 32 \text{ for } n > 4)$	X		X
extended precision	X		
extendable precision	X		

Table 10.4 IEEE Formats

Interpretation of IEEE 754 Floating-Point Numbers

(a) binary 32 format

	Sign	Biased exponent	Fraction	Value
positive zero	0	0	0	0
negative zero	1	0	0	-0
plus infinity	0	all 1s	0	∞
Minus infinity	1	all 1s	0	$-\infty$
quiet NaN	0 or 1	all 1s	$\neq 0$; first bit = 1	qNaN
signaling NaN	0 or 1	all 1s	$\neq 0$; first bit = 0	sNaN
positive normal nonzero	0	$0 < e < 255$	f	$2_{e-127}(1.f)$
negative normal nonzero	1	$0 < e < 255$	f	$-2_{e-127}(1.f)$
positive subnormal	0	0	$f \neq 0$	$2_{e-126}(0.f)$
negative subnormal	1	0	$f \neq 0$	$-2_{e-126}(0.f)$

Table 10.5 Interpretation of IEEE 754 Floating-Point Numbers (page 1 of 3)

Interpretation of IEEE 754 Floating-Point Numbers

(b) binary 64 format

	Sign	Biased exponent	Fraction	Value
positive zero	0	0	0	0
negative zero	1	0	0	-0
plus infinity	0	all 1s	0	∞
Minus infinity	1	all 1s	0	$-\infty$
quiet NaN	0 or 1	all 1s	$\neq 0$; first bit = 1	qNaN
signaling NaN	0 or 1	all 1s	$\neq 0$; first bit = 0	sNaN
positive normal nonzero	0	$0 < e < 2047$	f	$2_{e-1023}(1.f)$
negative normal nonzero	1	$0 < e < 2047$	f	$-2_{e-1023}(1.f)$
positive subnormal	0	0	$f \neq 0$	$2_{e-1022}(0.f)$
negative subnormal	1	0	$f \neq 0$	$-2_{e-1022}(0.f)$

Table 10.5 Interpretation of IEEE 754 Floating-Point Numbers (page 2 of 3)

Interpretation of IEEE 754 Floating-Point Numbers

(c) binary 128 format

	Sign	Biased exponent	Fraction	Value
positive zero	0	0	0	0
negative zero	1	0	0	-0
plus infinity	0	all 1s	0	∞
minus infinity	1	all 1s	0	$-\infty$
quiet NaN	0 or 1	all 1s	$\neq 0$; first bit = 1	qNaN
signaling NaN	0 or 1	all 1s	$\neq 0$; first bit = 0	sNaN
positive normal nonzero	0	all 1s	f	$2_{e-16383}(1.f)$
negative normal nonzero	1	all 1s	f	$-2_{e-16383}(1.f)$
positive subnormal	0	0	$f \neq 0$	$2_{e-16383}(0.f)$
negative subnormal	1	0	$f \neq 0$	$-2_{e-16383}(0.f)$

Table 10.5 Interpretation of IEEE 754 Floating-Point Numbers (page 3 of 3)
15-Jun-2021_arithmetic1

Table 10.6 Floating-Point Numbers and Arithmetic Operations

Floating Point Numbers	Arithmetic Operations
$X = X_s \times B^{X_E}$ $Y = Y_s \times B^{Y_E}$	$X + Y = \left(X_s \times B^{X_E - Y_E} + Y_s \right) \times B^{Y_E}$ $X - Y = \left(X_s \times B^{X_E - Y_E} - Y_s \right) \times B^{Y_E} \quad X_E \leq Y_E$ $X \times Y = (X_s \times Y_s) \times B^{X_E + Y_E}$ $\frac{X}{Y} = \left(\frac{X_s}{Y_s} \right) \times B^{X_E - Y_E}$

Examples:

$$X = 0.3 \times 10^2 = 30$$

$$Y = 0.2 \times 10^3 = 200$$

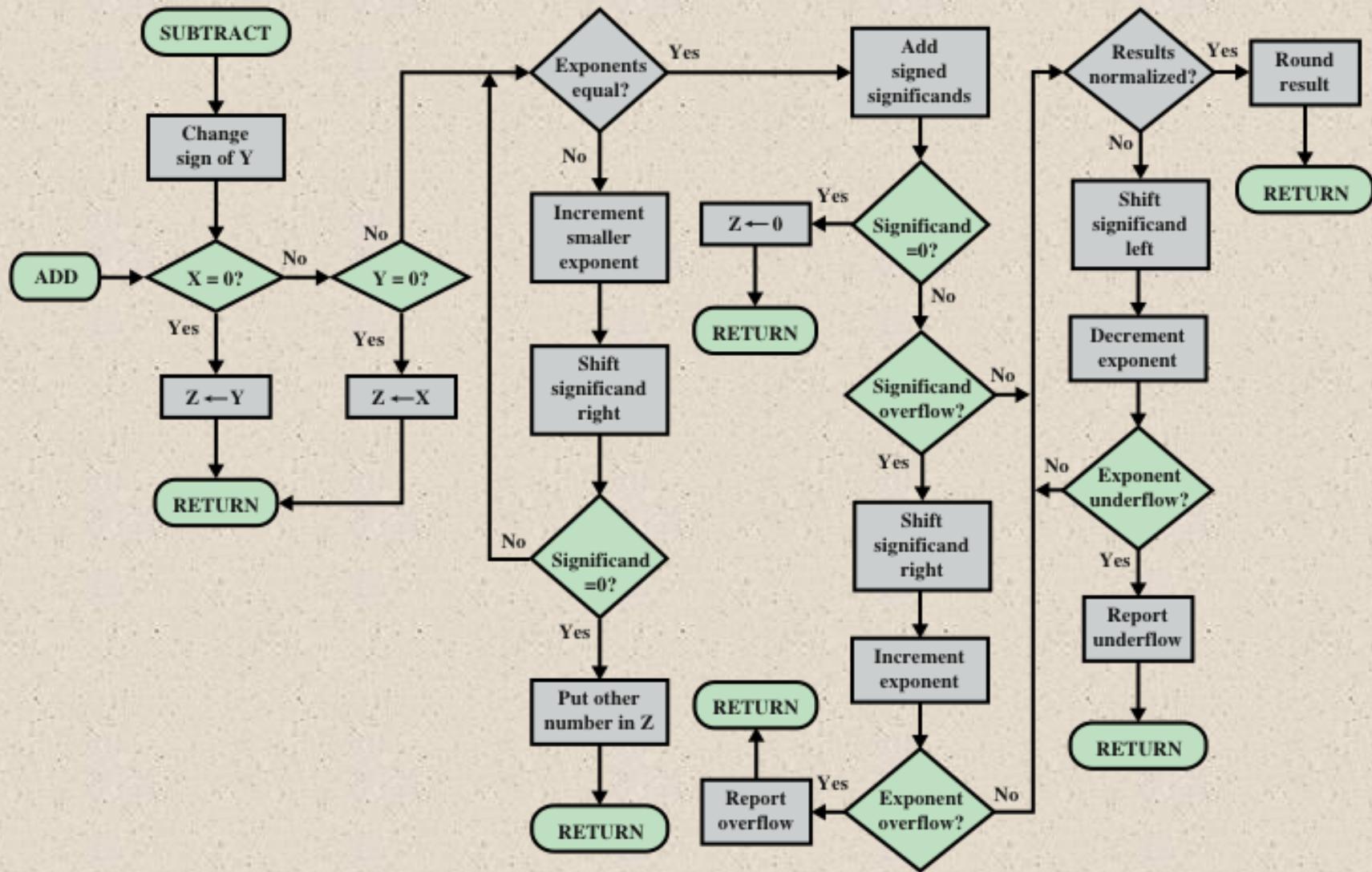
$$X + Y = (0.3 \times 10^{2-3} + 0.2) \times 10^3 = 0.23 \times 10^3 = 230$$

$$X - Y = (0.3 \times 10^{2-3} - 0.2) \times 10^3 = (-0.17) \times 10^3 = -170$$

$$X \times Y = (0.3 \times 0.2) \times 10^{2+3} = 0.06 \times 10^5 = 6000$$

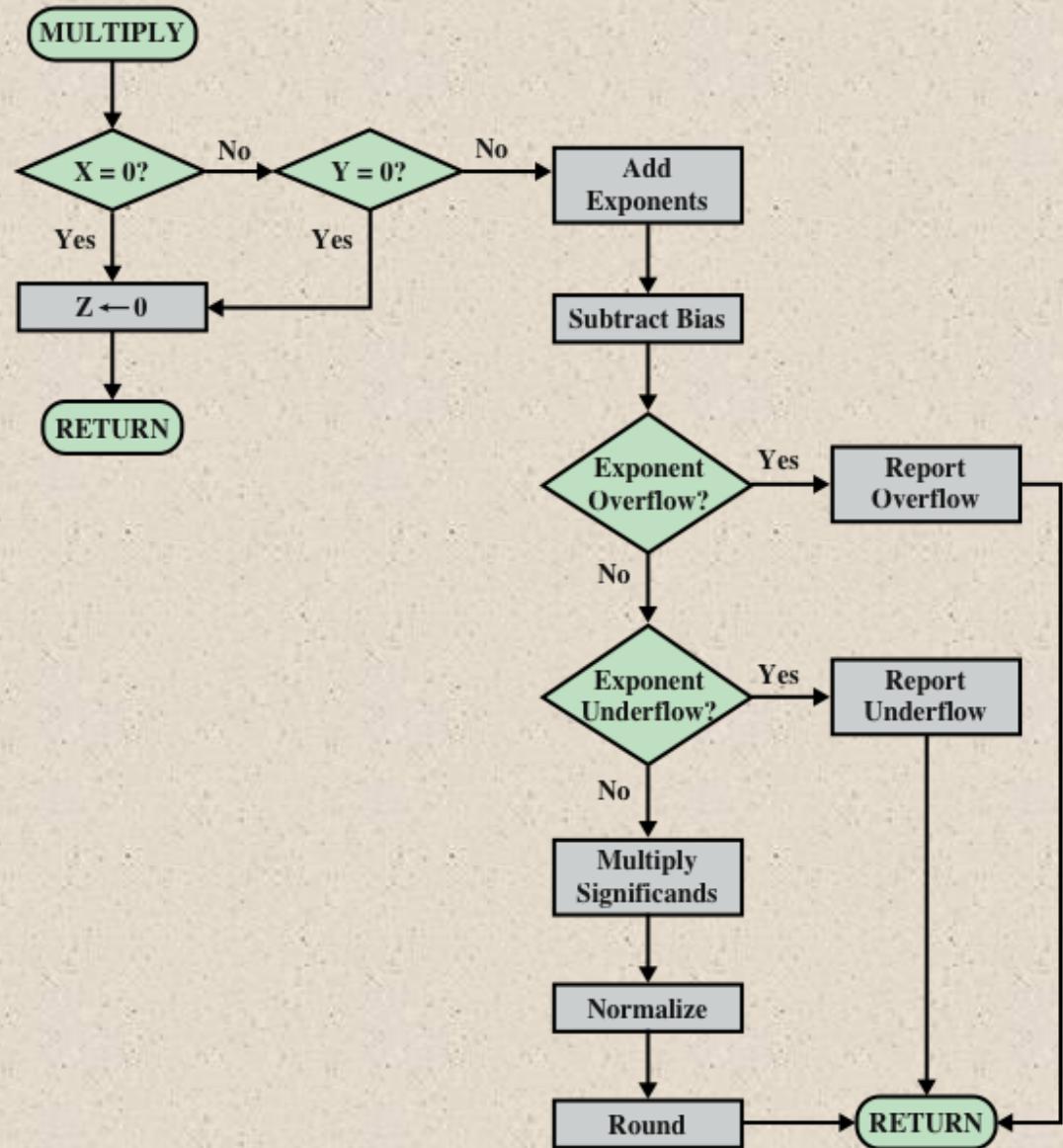
$$X \div Y = (0.3 \div 0.2) \times 10^{2-3} = 1.5 \times 10^{-1} = 0.15$$

Floating-Point Addition and Subtraction





Floating-Point Multiplication





Floating-Point Division

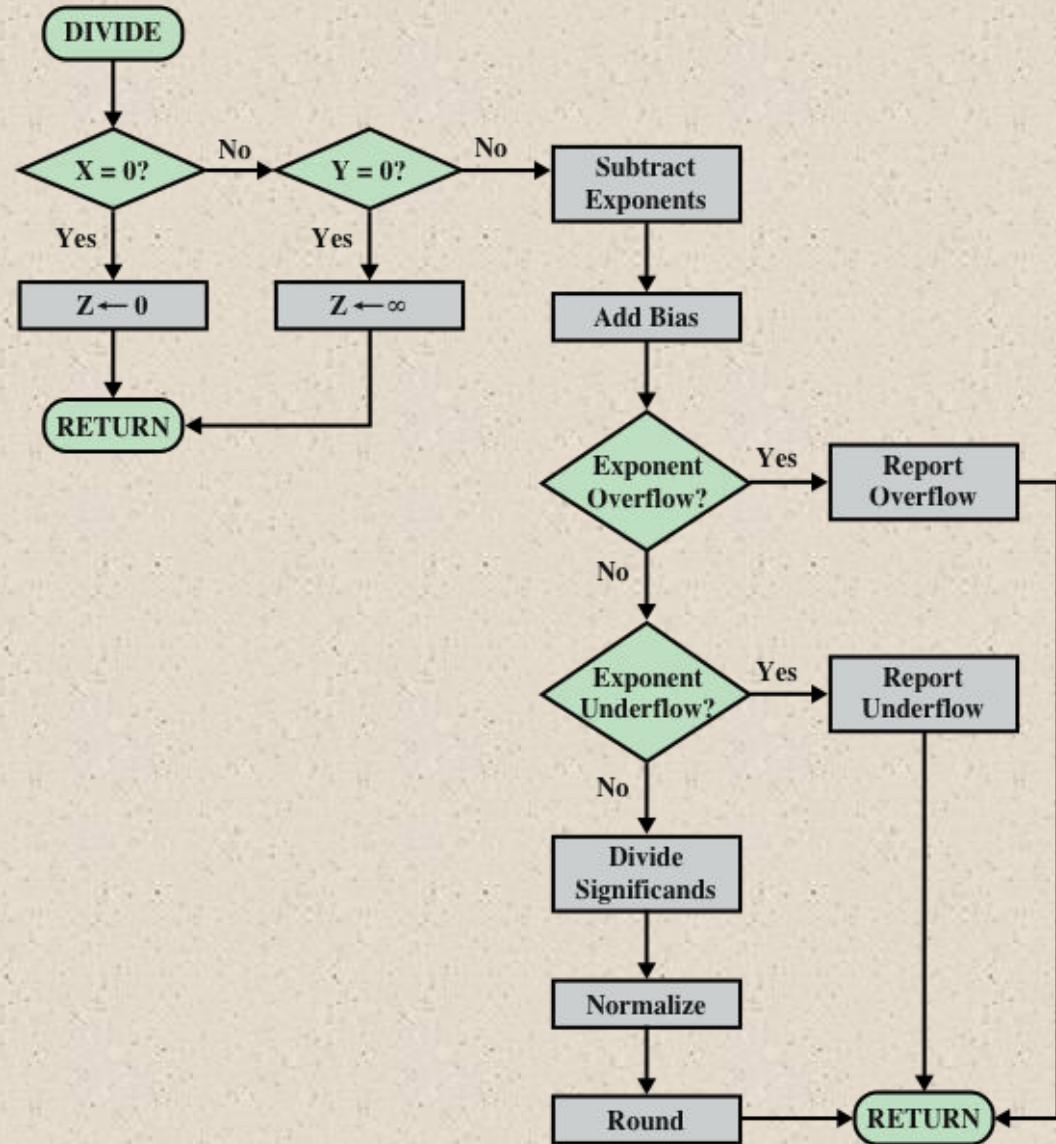


Figure 10.24 Floating-Point Division ($Z \leftarrow X/Y$)

Precision Considerations

Guard Bits

$$\begin{aligned}
 x &= 1.000\ldots00 \times 2^1 \\
 -y &= \underline{0.111\ldots11} \times 2^1 \\
 z &= 0.000\ldots01 \times 2^1 \\
 &= 1.000\ldots00 \times 2^{-22}
 \end{aligned}$$

(a) Binary example, without guard bits

$$\begin{aligned}
 x &= .100000 \times 16^1 \\
 -y &= \underline{.0FFFFFFF} \times 16^1 \\
 z &= .000001 \times 16^1 \\
 &= .100000 \times 16^{-4}
 \end{aligned}$$

(c) Hexadecimal example, without guard bits

$$\begin{aligned}
 x &= 1.000\ldots00 0000 \times 2^1 \\
 -y &= \underline{0.111\ldots11 1000} \times 2^1 \\
 z &= 0.000\ldots00 1000 \times 2^1 \\
 &= 1.000\ldots00 0000 \times 2^{-23}
 \end{aligned}$$

(b) Binary example, with guard bits

$$\begin{aligned}
 x &= .100000 00 \times 16^1 \\
 -y &= \underline{.0FFFFFFF F0} \times 16^1 \\
 z &= .000000 10 \times 16^1 \\
 &= .100000 00 \times 16^{-5}
 \end{aligned}$$

(d) Hexadecimal example, with guard bits

Figure 10.25 The Use of Guard Bits



Precision Considerations

Rounding

- IEEE standard approaches:
 - Round to nearest:
 - The result is rounded to the nearest representable number.
 - Round toward $+\infty$:
 - The result is rounded up toward plus infinity.
 - Round toward $-\infty$:
 - The result is rounded down toward negative infinity.
 - Round toward 0:
 - The result is rounded toward zero.

Interval Arithmetic

- Provides an efficient method for monitoring and controlling errors in floating-point computations by producing two values for each result
- The two values correspond to the lower and upper endpoints of an interval that contains the true result
- The width of the interval indicates the accuracy of the result
- If the endpoints are not representable then the interval endpoints are rounded down and up respectively
- If the range between the upper and lower bounds is sufficiently narrow then a sufficiently accurate result has been obtained

- *Minus infinity and rounding to plus* are useful in implementing interval arithmetic

Truncation

- Round toward zero
- Extra bits are ignored
- Simplest technique
- A consistent bias toward zero in the operation
- Serious bias because it affects every operation for which there are nonzero extra bits



IEEE Standard for Binary Floating-Point Arithmetic

Infinity

Is treated as the limiting case of real arithmetic, with the infinity values given the following interpretation:

$$-\infty < (\text{every finite number}) < +\infty$$

For example:

$$5 + (+\infty) = +\infty$$

$$5 \div (+\infty) = +0$$

$$5 - (+\infty) = -\infty$$

$$(+\infty) + (+\infty) = +\infty$$

$$5 + (-\infty) = -\infty$$

$$(-\infty) + (-\infty) = -\infty$$

$$5 - (-\infty) = +\infty$$

$$(-\infty) - (+\infty) = -\infty$$

$$5 * (+\infty) = +\infty$$

$$(+\infty) - (-\infty) = +\infty$$



IEEE Standard for Binary Floating-Point Arithmetic

Quiet and Signaling NaNs

- Signaling NaN signals an invalid operation exception whenever it appears as an operand
- Quiet NaN propagates through almost every arithmetic operation without signaling an exception

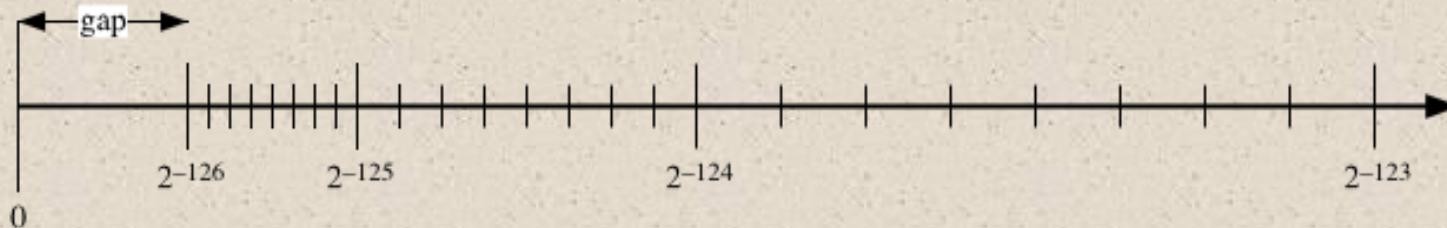
Operation	Quiet NaN Produced by
Any	Any operation on a signaling NaN
Add or subtract	Magnitude subtraction of infinities: $(+\infty) + (-\infty)$ $(-\infty) + (+\infty)$ $(+\infty) - (+\infty)$ $(-\infty) - (-\infty)$
Multiply	$0 \times \infty$
Division	$\frac{0}{0}$ or $\frac{\infty}{\infty}$
Remainder	$x \text{ REM } 0$ or $\infty \text{ REM } y$
Square root	\sqrt{x} where $x < 0$

Operations that
Produce a
Quiet NaN

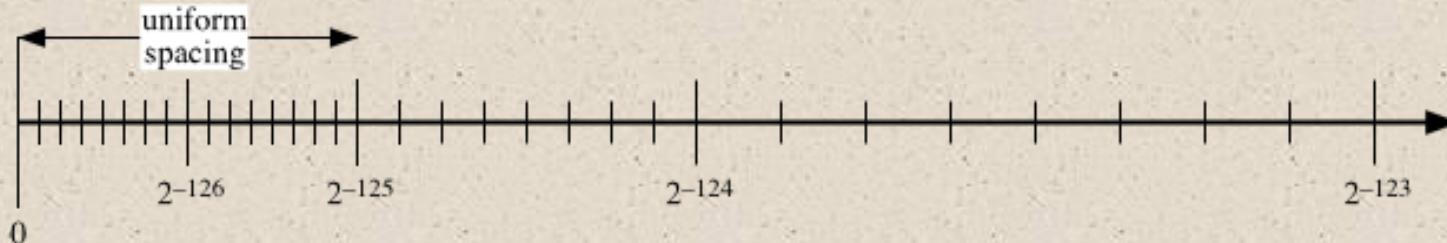


IEEE Standard for Binary Floating-Point Arithmetic

Subnormal Numbers



(a) 32-bit format without subnormal numbers



(b) 32-bit format with subnormal numbers

-6X-5 0101(5) 1011(-5)

0110(6) 1100(6X2)

1010(-6) 0100(-6X2)

4/2 → even 2clock cycle

2's Arithmetic shift

	A						Q	Q0	Q-1	
0	0	0	0		1	0	1	1	0	
0	1	1	0		1	0	1	1	0	-M
0	0	1	1		0	1	0	1	1	1 st s
0	0	0	1		1	0	1	0	1	2 nd s
0	1	1	0							-M
0	1	1	1		1	0	1	0	1	
0	0	1	1		1	1	0	1	0	1 st
0	0	0	1		1	1	1	0	1	2 nd
			16		8	4	2	0		30

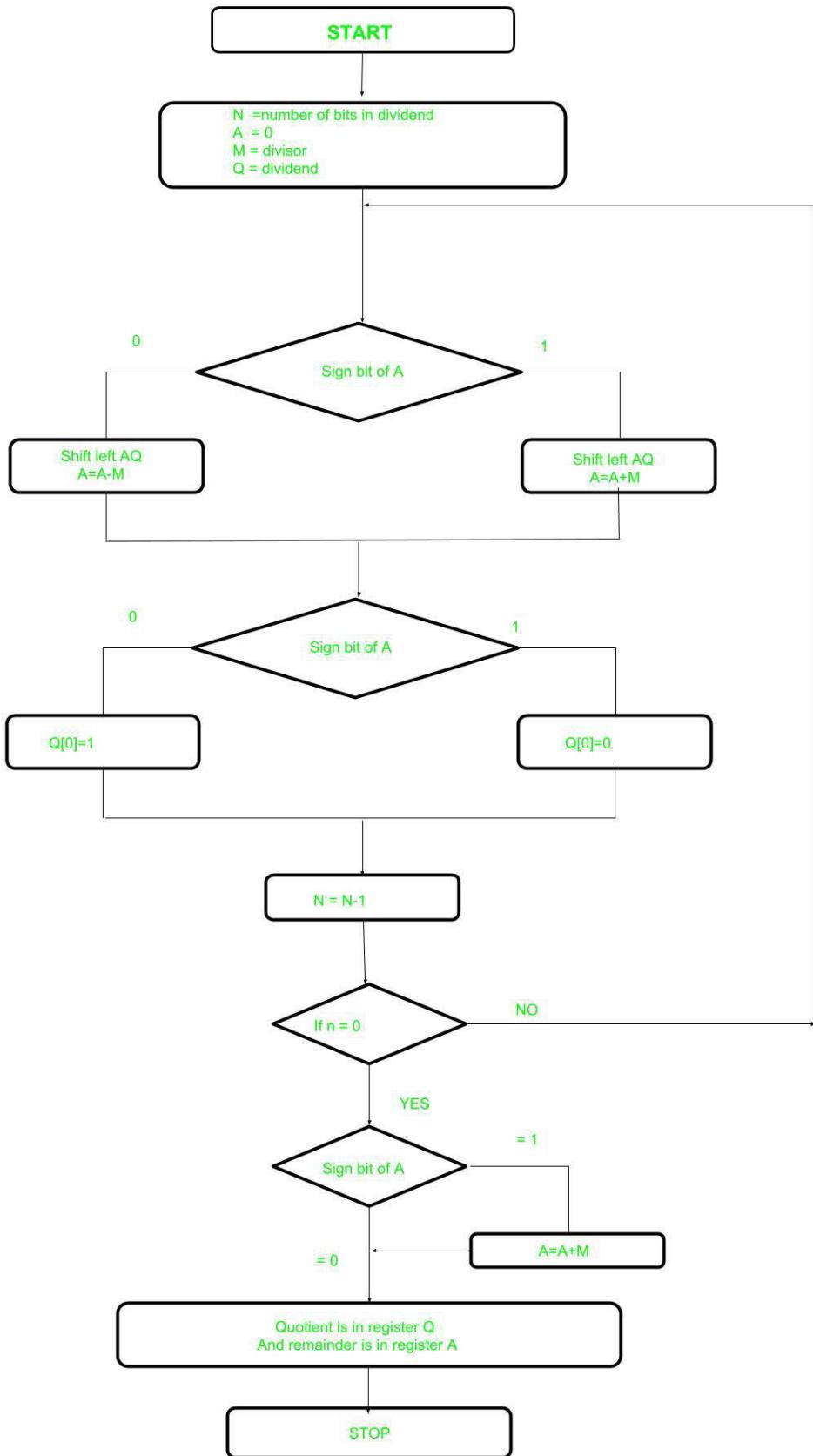
	A								Q				Q-1	
0	0	0	0	0	0		1	1	0	1	1	1	0	
1	1	0	1	0	1		1	1	0	1	1	1	0	-m
1	1	1	0	1	0		1	1	1	0	1	1	1	1 st s
1	1	1	1	0	1		0	1	1	1	0	1	1	2 nd s T1
0	1	0	1	1	0									
0	1	0	0	1	1		0	1	1	1	0	1	1	(2m)
0	0	1	0	0	1		1	0	1	1	1	0	1	1 st s
0	0	0	1	0	0		1	1	0	1	1	1	0	2 nd s T2
1	1	0	1	0	1									
1	1	1	0	0	1		1	1	0	1	1	1	0	(-m)
1	1	1	1	0	0		1	1	1	0	1	1	1	1 st
1	1	1	1	1	0		0	1	1	1	0	1	1	
						1	1	0	0	0	1	1		
						64	32	16	8	4	2	1		-99

11X-9

001011(11) 110101(-11) 010110(11X2) 101010(-11X2)
 9(001001) 110111(-9)

Booth recoding table for radix-4

Multiplier Bits Block			Recoded 1-bit pair		2 bit booth	
i+1	i	i-1	i+1	i	Multiplier Value	Partial Product
0	0	0	0	0	0	Mx0
0	0	1	0	1	1	Mx1
0	1	0	1	-1	1	Mx1
0	1	0	1	0	2	Mx2
1	0	0	-1	0	-2	Mx-2
1	0	1	-1	1	-1	Mx-1
1	1	0	0	-1	-1	Mx-1
1	1	0	0	0	0	Mx0



		A	N+1						Q	n→4		
0	0	0	0	0		1	0	1	1			
0	0	0	0	1		0	1	1		ls	a-m	
1	1	1	0	1								
1	1	1	1	0		0	1	1	0		1 st	
1	1	1	0	0		1	1	0				
0	0	0	1	1								
1	1	1	1	1		1	1	0	0		2 nd	
1	1	1	1	1		1	0	0				
0	0	0	1	1							A+(-m)	
0	0	0	1	0		1	0	0	1		3 rd	
0	0	1	0	1		0	0	1				
1	1	1	0	1								
0	0	0	1	0		0	0	1	1		4th	

11/3

1011(end)
 $3 \rightarrow 00011$ (m) $-3 \rightarrow 11101$

		A	N+1						Q	n→4		
0	0	0	0	0		1	1	0	1			
0	0	0	0	1		1	0	1			ls	
1	1	1	0	0								
1	1	1	0	1		1	0	1	0			1st
1	1	0	1	1		0	1	0			ls	
0	0	1	0	0							+	
1	1	1	1	1		0	1	0	0			2 nd
1	1	1	1	0		1	0	0				
0	0	1	0	0								
0	0	0	1	0		1	0	0	1			3 rd
0	0	1	0	1		0	0	1			ls	
1	1	1	0	0							-	
0	0	0	0	1		0	0	1	1			4th

13/4

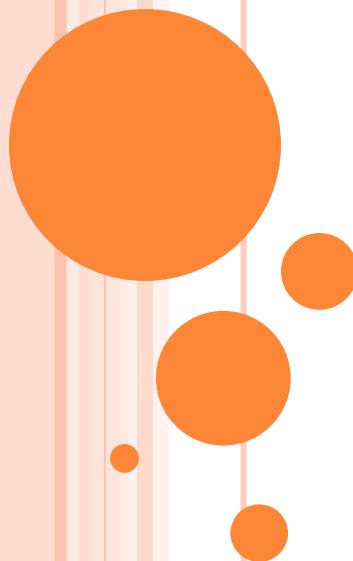
1101(13)

00100(4)

11100(-4)

FUNDAMENTALS OF COMPUTER ARCHITECTURE

Unit - I



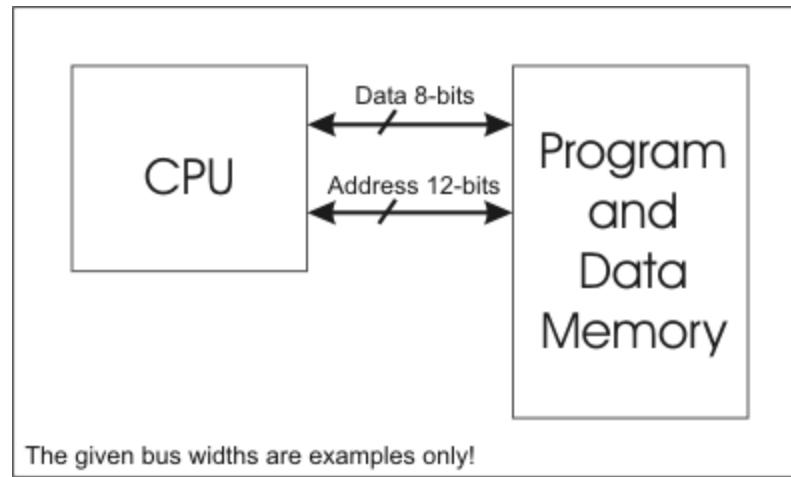
TOPICS OF UNIT - I

- Organization of the von Neumann machine
- Instruction formats
- The fetch/execute cycle, instruction decoding and execution;
- Registers and register files;
- Instruction types and addressing modes;
- Subroutine call and return mechanisms
- Programming in assembly language
- I/O techniques and interrupts
- Other design issues.



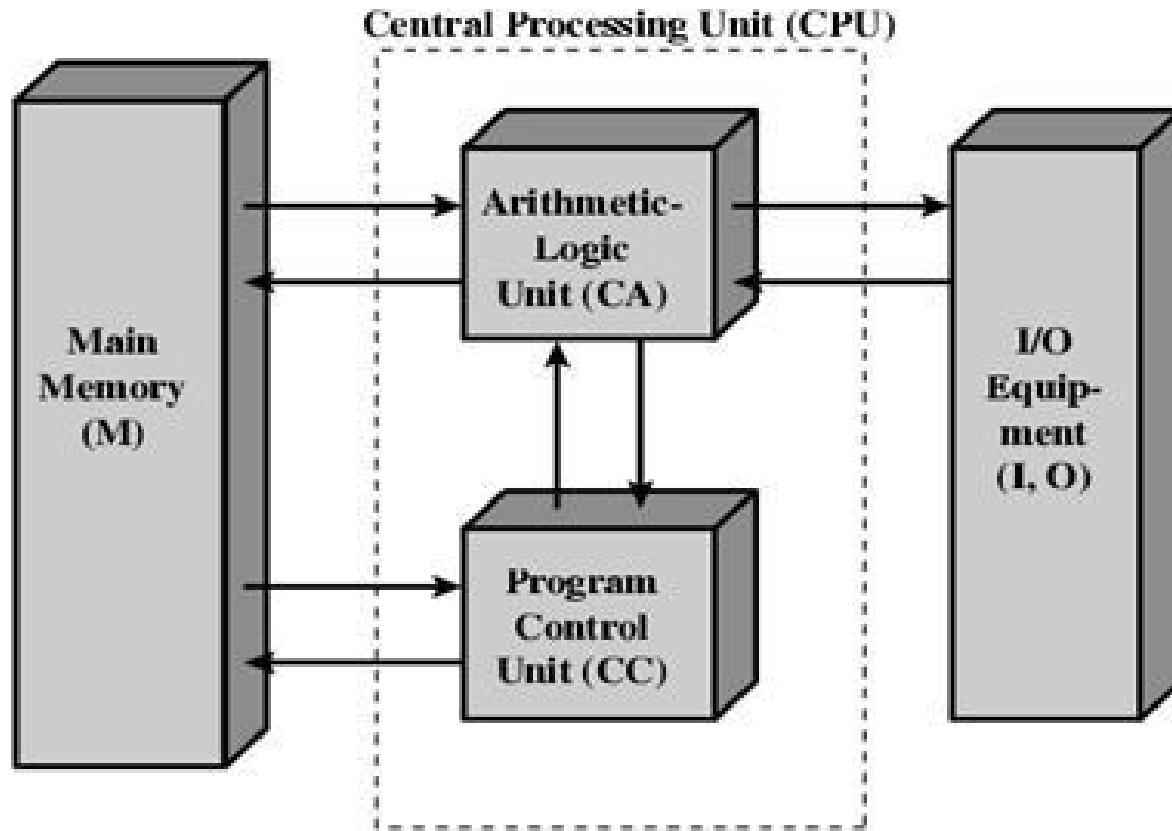
VON NEUMANN ARCHITECTURE:

- Computer has **single** storage system(memory) for storing data as well as program to be executed.
- ***Processor needs two clock cycles*** to complete an instruction (Query and Reply)



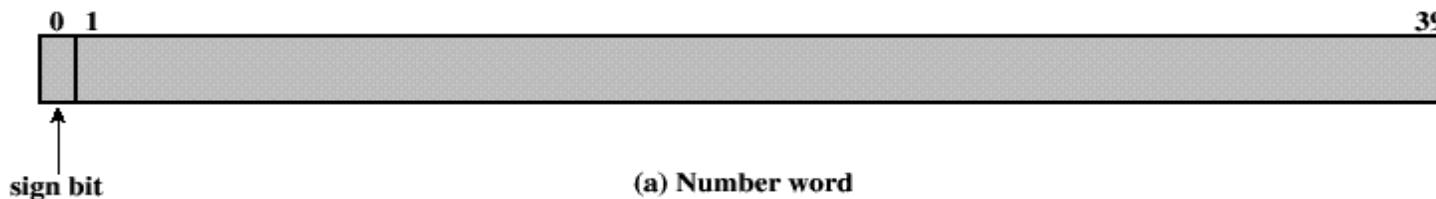
- Pipelining is not possible
- This is a relatively older architecture and was replaced by Harvard architecture.

Structure of Von Neumann Machine

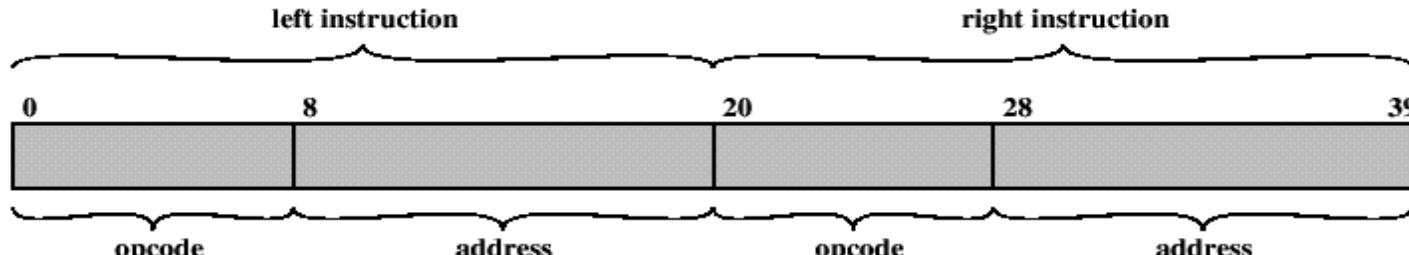


MEMORY OF THE IAS

- 1000 storage locations called words. (Institute for Advanced Studies (IAS)).
- Each word 40 bits.
- A word may contain:
 - A numbers stored as 40 binary digits (bits) – sign bit + 39 bit value
 - An instruction-pair. Each instruction:
 - An opcode (8 bits)
 - An address (12 bits) – designating one of the 1000 words in memory.

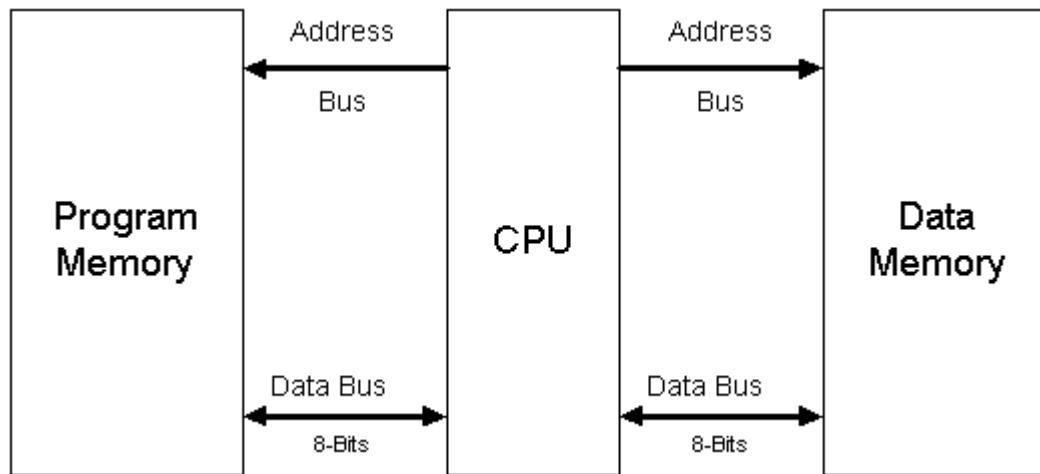


(a) Number word



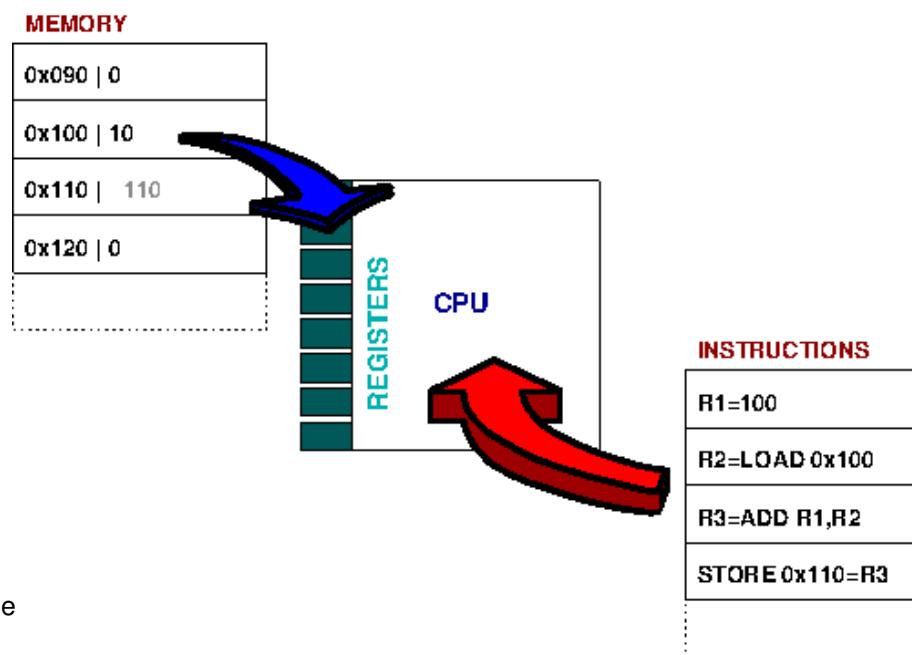
HARVARD ARCHITECTURE

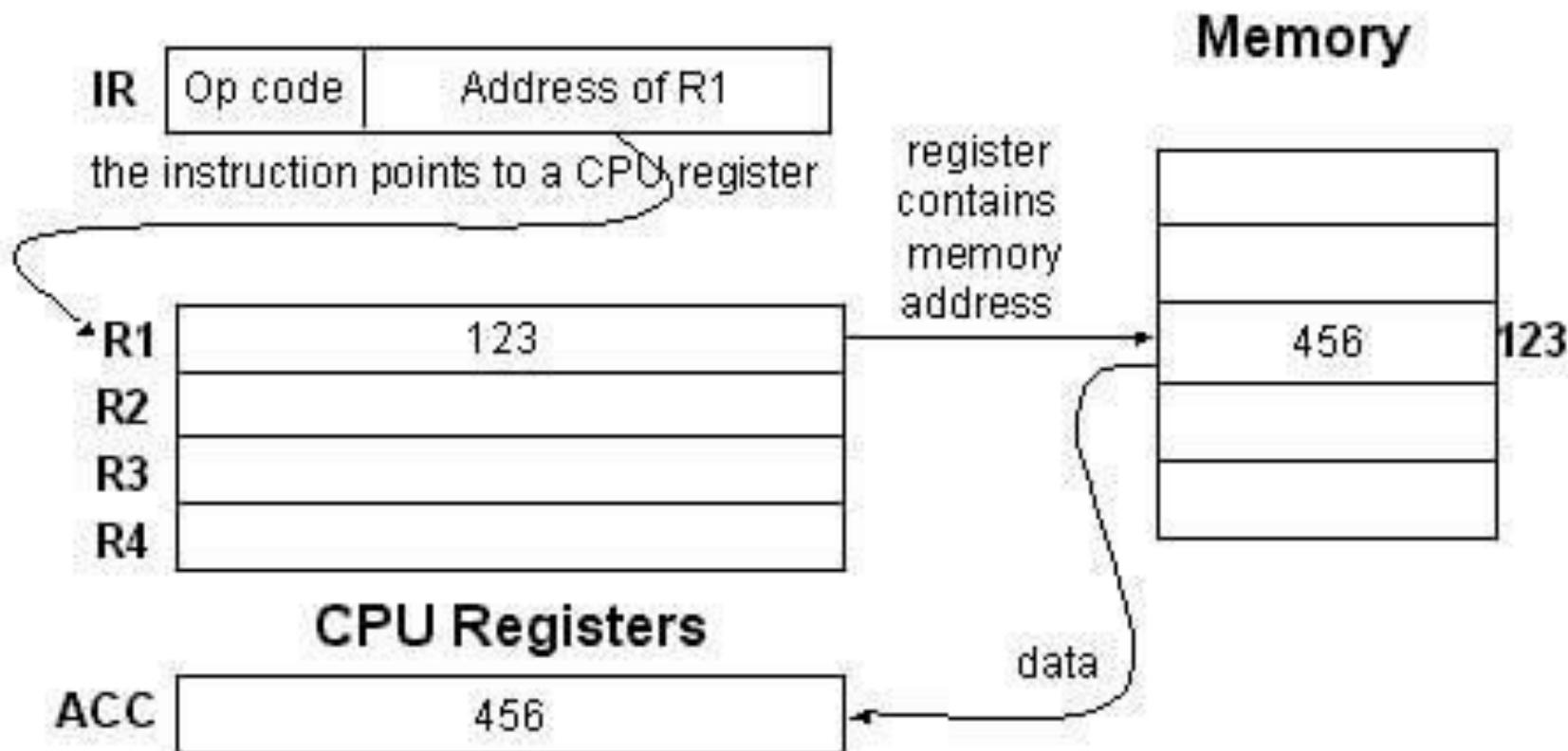
- Computer has two separate memories for storing data and program
- Processor can complete an instruction in one cycle
- Pipelining is possible



INSTRUCTION FORMAT

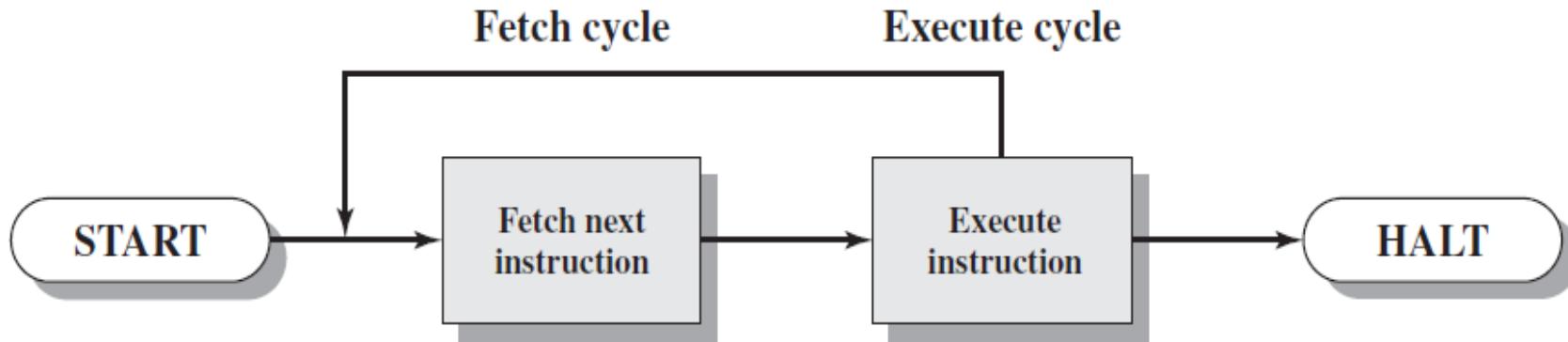
- An instruction set, or instruction set architecture (ISA), is the part of the computer architecture related to programming.



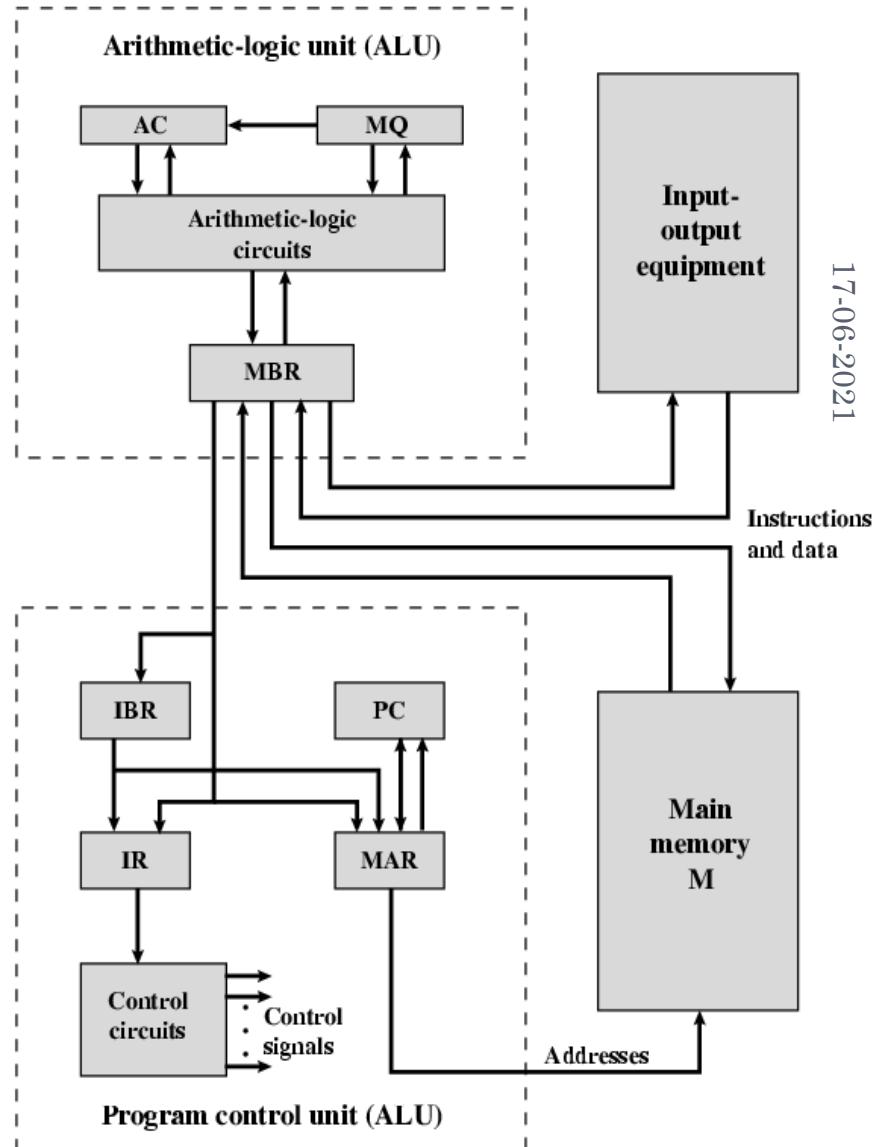


COMPUTER FUNCTION

- The basic function performed by a computer is execution of a program, which consists of a set of instructions stored in memory.
- Instruction fetch-decode-execute



- MBR: Memory Buffer Register**
 - contains the word to be stored in memory or just received from memory.
- MAR: Memory Address Register**
 - specifies the address in memory of the word to be stored or retrieved.
- IR: Instruction Register** - contains the 8-bit opcode currently being executed.
- IBR: Instruction Buffer Register**
 - temporary store for RHS instruction from word in memory.
- PC: Program Counter** - address of next instruction-pair to fetch from memory.
- AC: Accumulator & MQ: Multiplier quotient** - holds operands and results of ALU ops.

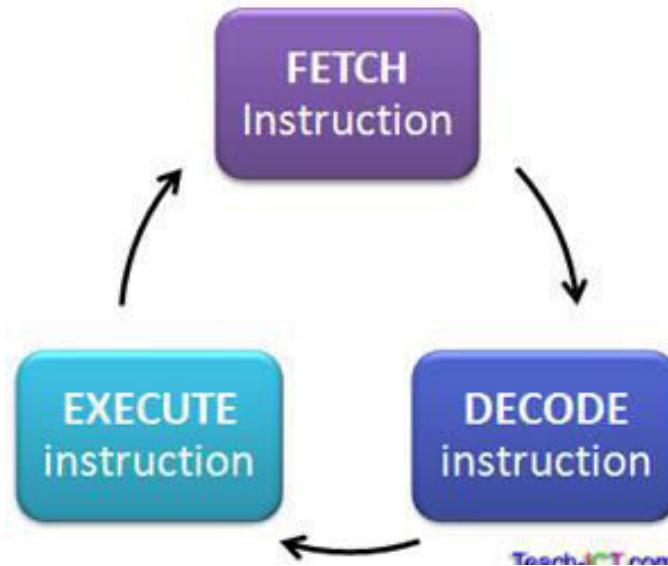


THE FETCH/EXECUTE CYCLE

- Standard process.

Also called as

- fetch-and-execute cycle,
- fetch-decode-execute cycle
- FDX



QUESTIONS:

- MBR –
- MAR –
- AC –
- IBR –
- IR –
- PC –
- MQ –
- IAS –
- What is Computer Architecture?
- What is Computer Organization?
- Number of words in IAS machine?
- Number of bits per word in IAS machine?
- Data is represented in _____ form in IAS machine
- Explain Stored program concept.

REGISTERS AND REGISTER FILES

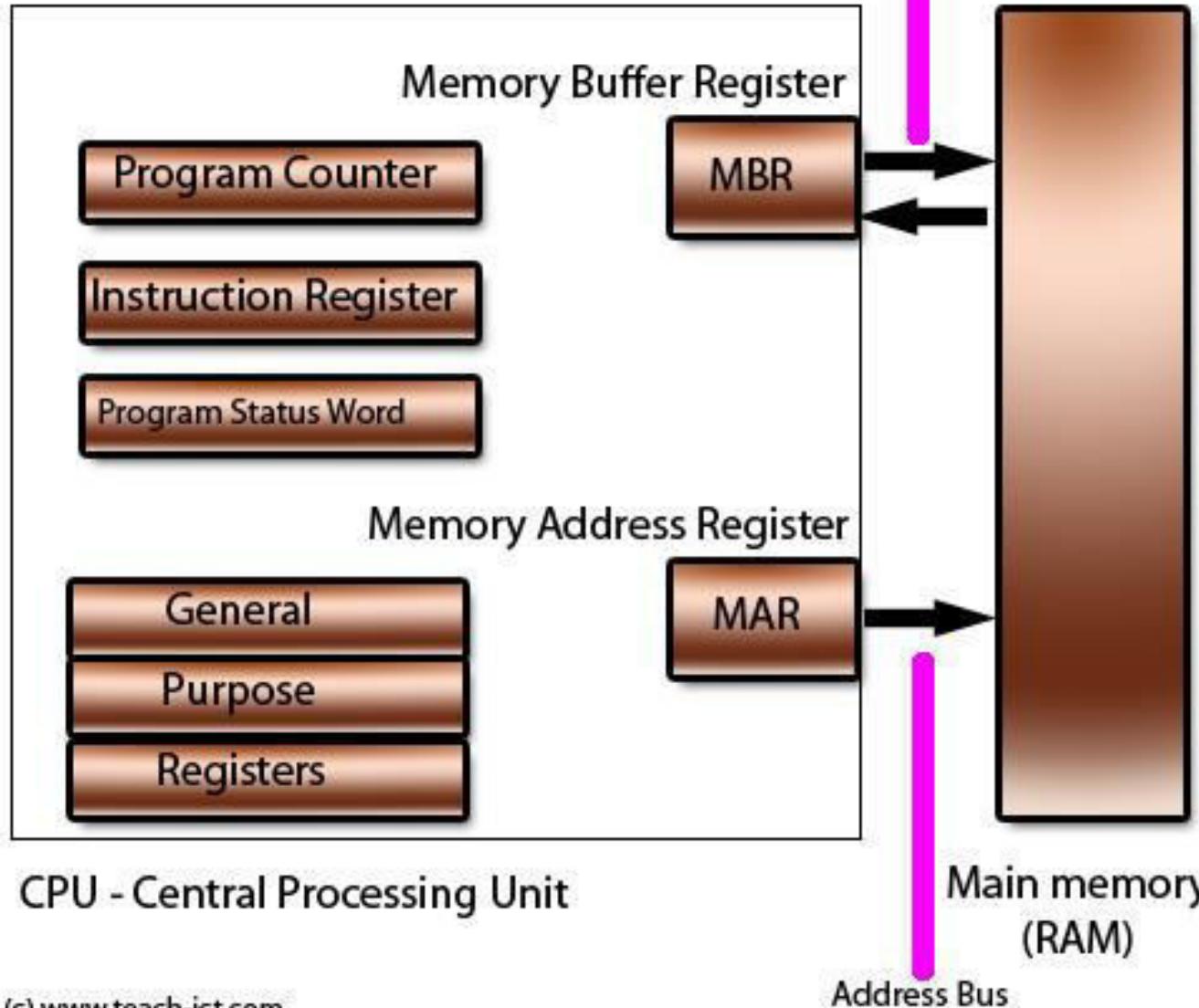
- Registers?
 - Group of Flip-flops capable of storing one bit of information.
 - N bit registers consists of a group of N flip-flops capable of storing N bits.
 - Provides storage internal to the CPU.

As the instructions are interpreted and executed by the CPU, there is a **movement of information between the various units** of the computer system. In order to handle this process satisfactorily, and to **speed up the rate of information transfer**, the computer uses a number of special memory units, called **registers**. These registers are used to hold information on temporary basis, and are part of the CPU (not main memory).

CONT..

- The length of a register equals the number of bits it can store.
- Hence, a register that can store 8 bits is normally referred to as 8-bit register.
- Most CPU sold today, have 32-bit or 64-bit registers.
- The size of the registers is sometimes called the world size.
- The bigger the world size, the faster the computer can process a set of data.
- With all other parameters being same, a CPU with 32-bit registers, can process data twice as fast as one with 16-bit registers.

Registers within a CPU



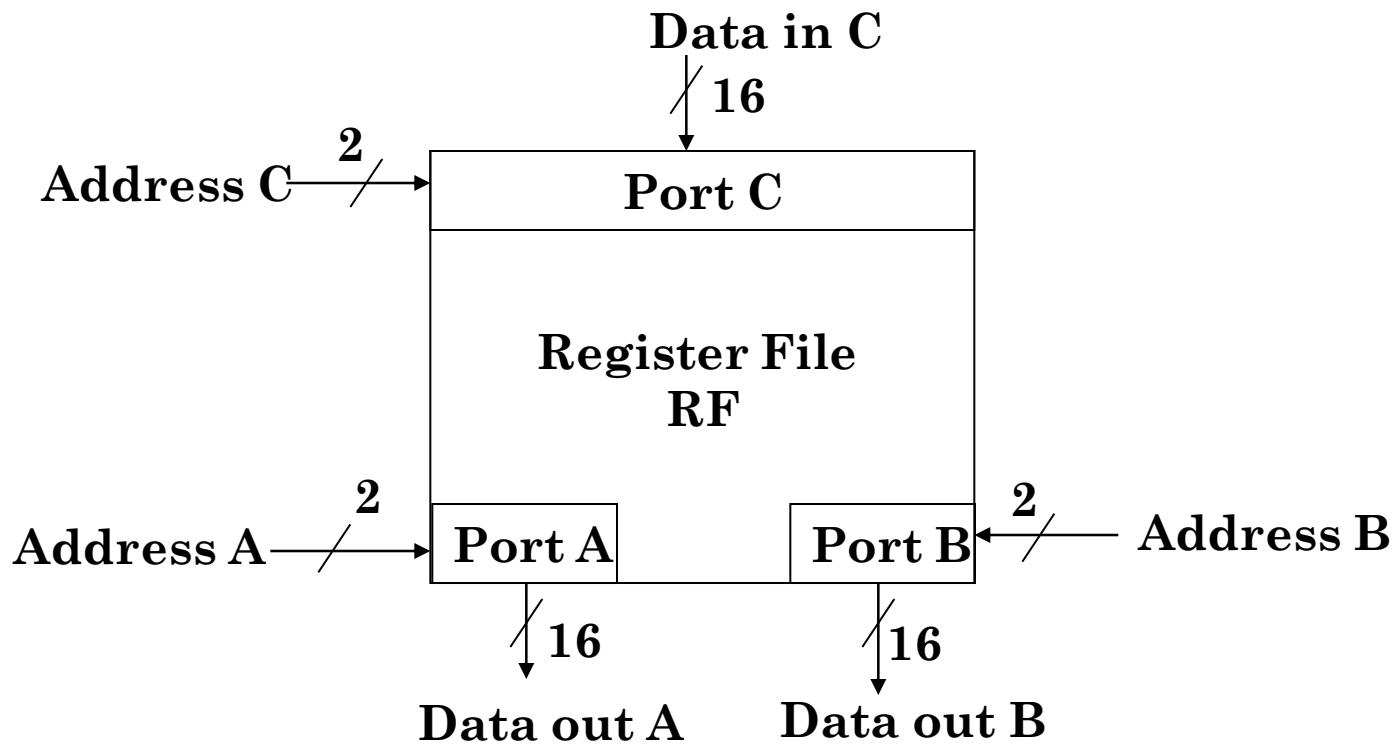
CPU - Central Processing Unit

Main memory
(RAM)

REGISTER FILES (RF)

- Set of general purpose registers.
- It functions as small RAM and implemented using fast RAM technology.
- RF needs several access ports for simultaneously reading from or writing to several different registers. Hence RF is realized as **multiport RAM**.
- A standard RAM has just one access port with an associated address bus and data bus.

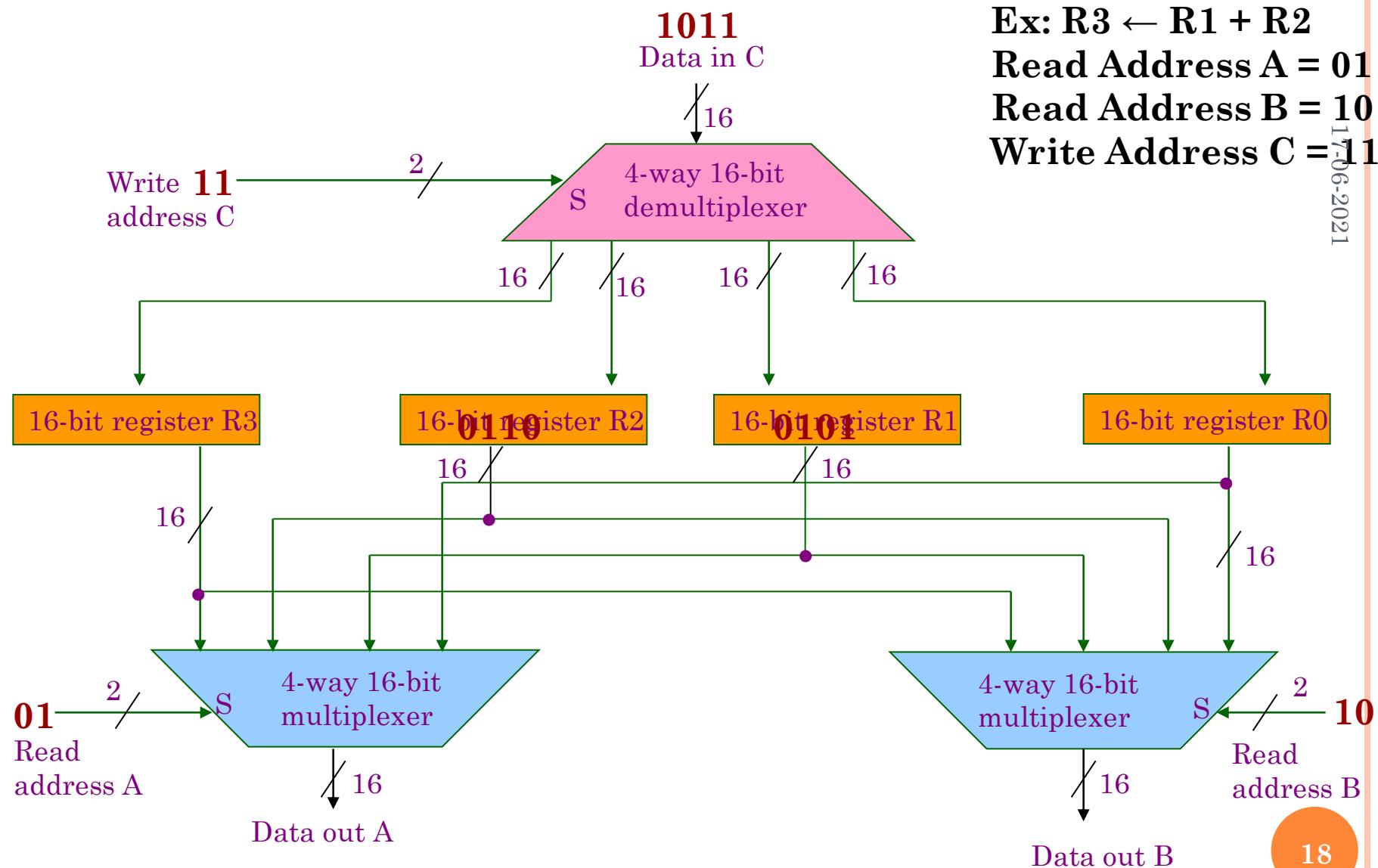
A REGISTER FILE WITH THREE ACCESS PORTS - SYMBOL



A REGISTER FILE WITH THREE ACCESS PORTS – LOGIC DIAGRAM

Ex: $R3 \leftarrow R1 + R2$
Read Address A = 01
Read Address B = 10
Write Address C = 11

17-Jun-2021



INSTRUCTION TYPES

- Data transfer instructions
- Data manipulation instructions
 - Arithmetic instructions
 - Logical and bit manipulation instructions
 - Shift instructions
- Program control instructions

DATA TRANSFER INSTRUCTIONS

- Move data from one place to another without changing the data content in the computer.
- Different data transfers:
 - Memory \leftrightarrow processor registers
 - Processor registers \leftrightarrow input or output
 - Processor register \leftrightarrow processor register

SET OF DATA TRANSFER INSTRUCTIONS

- Load – transfer from memory to a processor register
- Store – transfer from processor register into memory
- Move – transfer from one register to another, transfer between register and memory or between two memory words.
- Exchange – swaps information between two registers or a register and a memory word
- Input – transfer data among registers/memory and input terminal
- Output – transfer data among register/memory and output terminal
- Push – transfer data from register/memory to memory stack
- Pop – transfer data from stack to register/memory

DATA MANIPULATION INSTRUCTIONS

- Perform operations on data and provide the computational capabilities for the computer.
- Arithmetic instructions
 - Increment
 - Decrement
 - Add
 - Subtract
 - Multiply
 - Divide
 - Add with carry
 - Subtract with borrow
 - Negate (2's complement) – change the sign of the operand
 - Absolute – replace operand by its absolute value
 - Arithmetic shift left
 - Arithmetic shift right

Arithmetic Operations

Mnemonic		Description	Byte	Cyc
ADD	A,Rn	Add register to Accumulator	1	1
ADD	A,direct	Add direct byte to Accumulator	2	1
ADD	A,@Ri	Add indirect RAM to Accumulator	1	1
ADD	A,#data	Add immediate data to Accumulator	2	1
ADDC	A,Rn	Add register to Accumulator with Carry	1	1
ADDC	A,direct	Add direct byte to A with carry flag	2	1
ADDC	A,@Ri	Add indirect RAM to A with Carry flag	1	1
ADDC	A,#data	Add immediate data to A with Carry flag	2	1
SUBB	A,Rn	Subtract register from A with Borrow	1	1
SUBB	A,direct	Subtract direct byte from A with Borrow	2	1
SUBB	A,@Ri	Subtract indirect RAM from A with borrow	1	1
SUBB	A,#data	Subtract immed data from A with Borrow	2	1
INC	A	Increment Accumulator	1	1
INC	Rn	Increment register	1	1
INC	direct	Increment direct byte	2	1
INC	@Ri	Increment indirect RAM	1	1
DEC	A	Decrement Accumulator	1	1
DEC	Rn	Decrement register	1	1
DEC	direct	Decrement direct byte	2	1
DEC	@Ri	Decrement indirect RAM	1	1
INC	DPTR	Increment data Pointer	1	2
MUL	AB	Multiply A and B	1	4
DIV	AB	Divide A by B	1	4
DA	A	Decimal Adjust Accumulator	1	1

Logic Operations

ANL	A,Rn	AND register to accumulator	1	1
ANL	A,direct	AND direct byte to accumulator	2	1
ANL	A,@Ri	AND indirect RAM to accumulator	1	1
ANL	A,#data	AND immediate data to accumulator	2	1
ANL	direct,A	AND accumulator to direct byte	2	1
ANL	direct,#data	AND immediate data to direct byte	3	2
ORL	A,Rn	OR register to accumulator	1	1
ORL	A,direct	OR direct byte to accumulator	2	1
ORL	A,@Ri	OR indirect RAM to accumulator	1	1
ORL	A,#data	OR immediate data to accumulator	2	1
ORL	direct,A	OR accumulator to direct byte	2	1
ORL	direct,#data	OR immediate data to direct byte	3	2
XRL	A,Rn	Exclusive OR register to accumulator	1	1
XRL	A direct	Exclusive OR direct byte to accumulator	2	1
XRL	A,@Ri	Exclusive OR indirect RAM to accumulator	1	1
XRL	A,#data	Exclusive OR immediate data to accumulator	2	1
XRL	direct,A	Exclusive OR accumulator to direct byte	2	1
XRL	direct,#data	Exclusive OR immediate data to direct byte	3	2
CLR	A	Clear accumulator	1	1
CPL	A	Complement accumulator	1	1
RL	A	Rotate accumulator left	1	1
RLC	A	Rotate accumulator left through carry	1	1
RR	A	Rotate accumulator right	1	1
RRC	A	Rotate accumulator right through carry	1	1
SWAP	A	Swap nibbles within the accumulator	1	1

○ Arithmetic shift left

0	0	0	1	1	0	1	0
---	---	---	---	---	---	---	---

↑
Sign bit

0	0	0	1	1	0	1	0
---	---	---	---	---	---	---	---

↑
Sign bit

0

Shift by 1 bit towards left

0	0	1	1	0	1	0	0
---	---	---	---	---	---	---	---

↑
Sign bit

0

After shifting two times

0	1	1	0	1	0	0	0
---	---	---	---	---	---	---	---

↑
Sign bit

0

After shifting three times

**Overflow occurs as
sign bit changes**

○ Arithmetic shift Right

0	0	0	1	1	0	1	0
---	---	---	---	---	---	---	---

↑
Sign bit

0	0	0	1	1	0	1	0
---	---	---	---	---	---	---	---

↑
Sign bit

0	0	0	0	1	1	0	1
---	---	---	---	---	---	---	---

↑
Sign bit

0	0	0	0	0	1	1	0
---	---	---	---	---	---	---	---

↑
Sign bit

Shift by 1 bit towards right

After shifting two times

After shifting three times

DATA MANIPULATION INSTRUCTIONS

○ Logical and Bit manipulation instructions

- Clear (can also be included in data transfer instruction based on the way the operation is performed – 0's transferred to the destination)
- Complement
- AND- to clear a bit
- OR – set a bit
- Ex-Or –to complement a bit
- Clear carry
- Set carry
- Complement carry
- Enable interrupt – flip-flop that controls the interrupt facility is enabled
- Disable interrupt – flip-flop that controls the interrupt facility is disabled

DATA MANIPULATION INSTRUCTIONS

- Shift Instructions

- Logical left shift
- Logical right shift
- Arithmetic shift left
- Arithmetic shift right
- Rotate right
- Rotate left
- Rotate right through carry
- Rotate left through carry

○ Logical shift left

0	0	0	1	1	0	1	0
---	---	---	---	---	---	---	---

0	0	0	1	1	0	1	0
---	---	---	---	---	---	---	---

Shift by 1 bit towards left

0

0	0	1	1	0	1	0	0
---	---	---	---	---	---	---	---

After shifting two times

0

0	1	1	0	1	0	0	0
---	---	---	---	---	---	---	---

After shifting three times

○ Logical shift Right

0	0	0	1	1	0	1	0
---	---	---	---	---	---	---	---

0

0	0	0	1	1	0	1	0
---	---	---	---	---	---	---	---

Shift by 1 bit towards right

0

0	0	0	0	1	1	0	1
---	---	---	---	---	---	---	---

After shifting two times

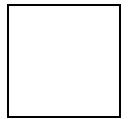
0

0	0	0	0	0	1	1	0
---	---	---	---	---	---	---	---

After shifting three times

- Rotate left

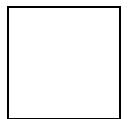
0	0	0	1	1	0	1	0
---	---	---	---	---	---	---	---



Buffer

0	0	0	1	1	0	1	0
---	---	---	---	---	---	---	---

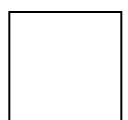
Rotate by 1 bit towards left



Buffer

0	0	1	1	0	1	0	0
---	---	---	---	---	---	---	---

After rotating two times



Buffer

0	1	1	0	1	0	0	0
---	---	---	---	---	---	---	---

After rotating three times

- Rotate right

0	0	0	1	1	0	1	0
---	---	---	---	---	---	---	---

rotate by 1 bit towards right

0	0	0	1	1	0	1	0
---	---	---	---	---	---	---	---



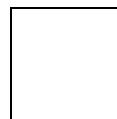
Buffer

0	0	0	0	1	1	0	1
---	---	---	---	---	---	---	---



After rotating two times

1	0	0	0	0	1	1	0
---	---	---	---	---	---	---	---



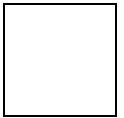
Buffer

After rotating three times

- Rotate left through carry

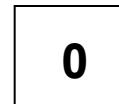
0	0	0	1	1	0	1	0
---	---	---	---	---	---	---	---

Rotate by 1 bit towards left

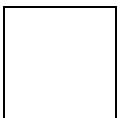


Buffer

0	0	0	1	1	0	1	0
---	---	---	---	---	---	---	---

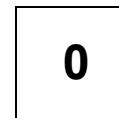


Carry



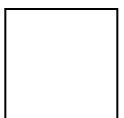
Buffer

0	0	1	1	0	1	0	0
---	---	---	---	---	---	---	---



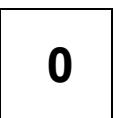
Carry

After rotating two times



Buffer

0	1	1	0	1	0	0	0
---	---	---	---	---	---	---	---



Carry

After rotating three times

- Rotate right through carry

0	0	0	1	1	0	1	0
---	---	---	---	---	---	---	---

0
Carry

0	0	0	1	1	0	1	0
---	---	---	---	---	---	---	---

rotate by 1 bit towards right

Buffer

0
Carry

0	0	0	0	1	1	0	1
---	---	---	---	---	---	---	---

After rotating two times
Buffer

0
Carry

1	0	0	0	0	1	1	0
---	---	---	---	---	---	---	---

After rotating three times
Buffer

PROGRAM CONTROL INSTRUCTIONS

- Branch
- Jump
- Skip
- Call
- Return
- Compare (by subtraction)
- Test (by ANDing)

ADDRESSING MODES

The way the operands are chosen during the program execution is dependent on the addressing mode of the instruction.

DIFFERENT TYPES

- Implied Addressing Mode
- Immediate Addressing Mode
- Direct Addressing Mode
- Indirect Addressing Mode
- Register Direct Addressing Mode
- Register Indirect Addressing Mode
- Displacement Addressing Mode (combines the direct addressing and register addressing modes)
 - Relative Addressing Mode
 - Indexed Addressing Mode
 - Base Addressing Mode
- Auto Increment and Auto Decrement Addressing Mode

IMPLIED ADDRESSING MODE

- No address field is required
- Operand is implied / implicit
- Ex:
 - Complementing Accumulator
 - Set or Clearing the flag bits (CLC, STC etc.)
- 0 – address instructions in a stack organized computer are implied mode instructions.
- Effective Address (EA) = AC or Stack[SP]
- Ex: Tomorrow, I am on leave (implies that there is no CAO class)
- Come to my cabin (implies to come to 313A-07 SJT)

IMMEDIATE ADDRESSING MODE

- Operand is specified in the instruction itself
- Useful for initializing the registers with constant value
- Operand = address field
- Ex: Mov Dx, #0034H
- Advantage: No memory Reference, fast
- Disadvantage: Limited operand magnitude
- Ex: Come to my cabin: 313A-07 SJT

DIRECT ADDRESSING MODE

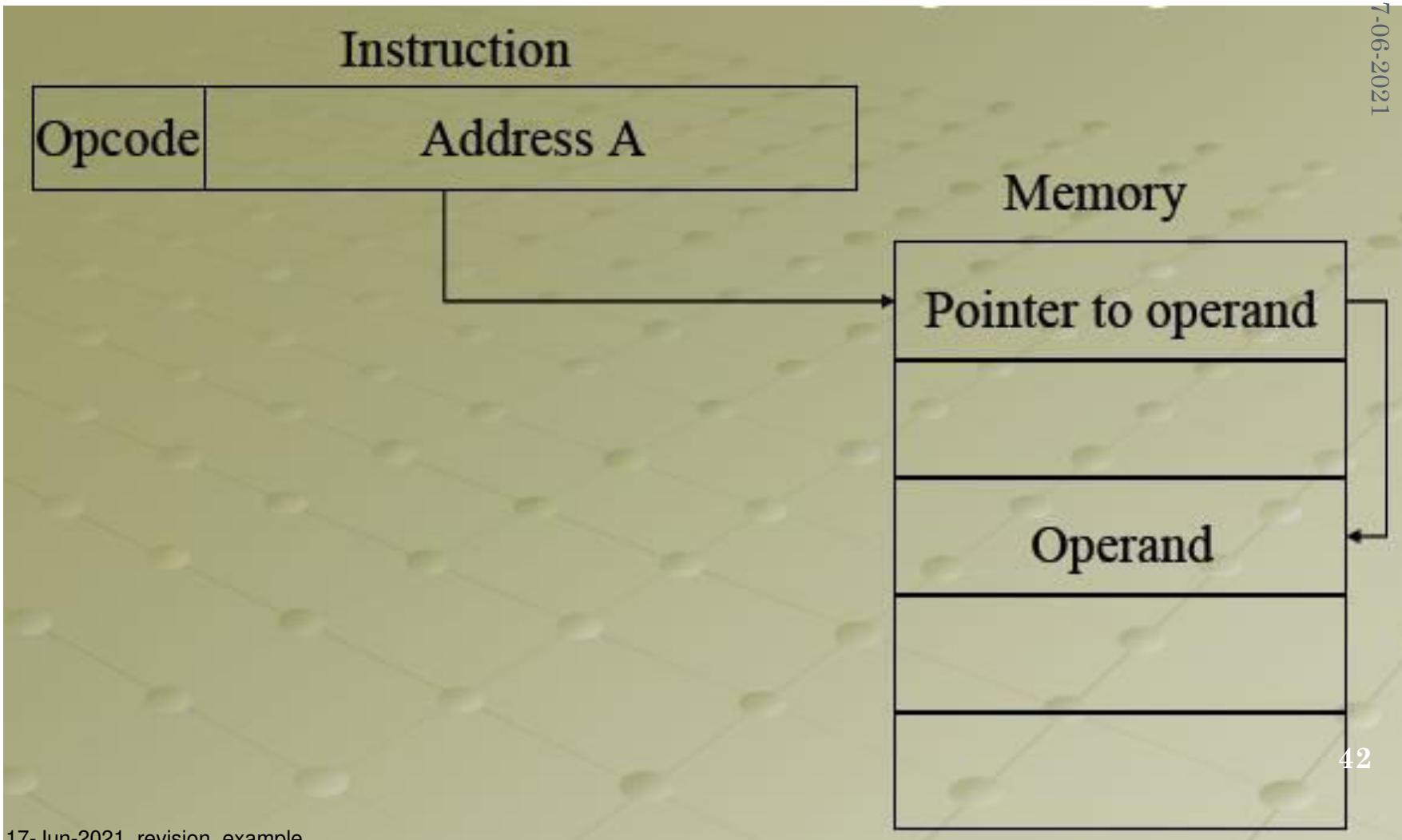
- Effective address is the address part of the instruction
- EA (effective address) = A
- Ex:
 - Mov CX, [4200]H
- Advantage: Simple memory reference to access data, no additional calculations to work out effective address
- Disadvantage: Limited address space
- Ex: Aashiq, please bring my laptop from my cabin (cabin is known to Aashiq)

INDIRECT ADDRESSING MODE

- The address field of the instruction gives the address of the effective address of the operand stored in the memory.
- $EA = (A)$
- Ex: `Mov CX, [BX]`
- Advantage: Large address space, may be nested, multilevel or cascaded
- Disadvantage: Multiple memory accesses to find the operand, hence slower

INDIRECT ADDRESSING MODE DIAGRAM

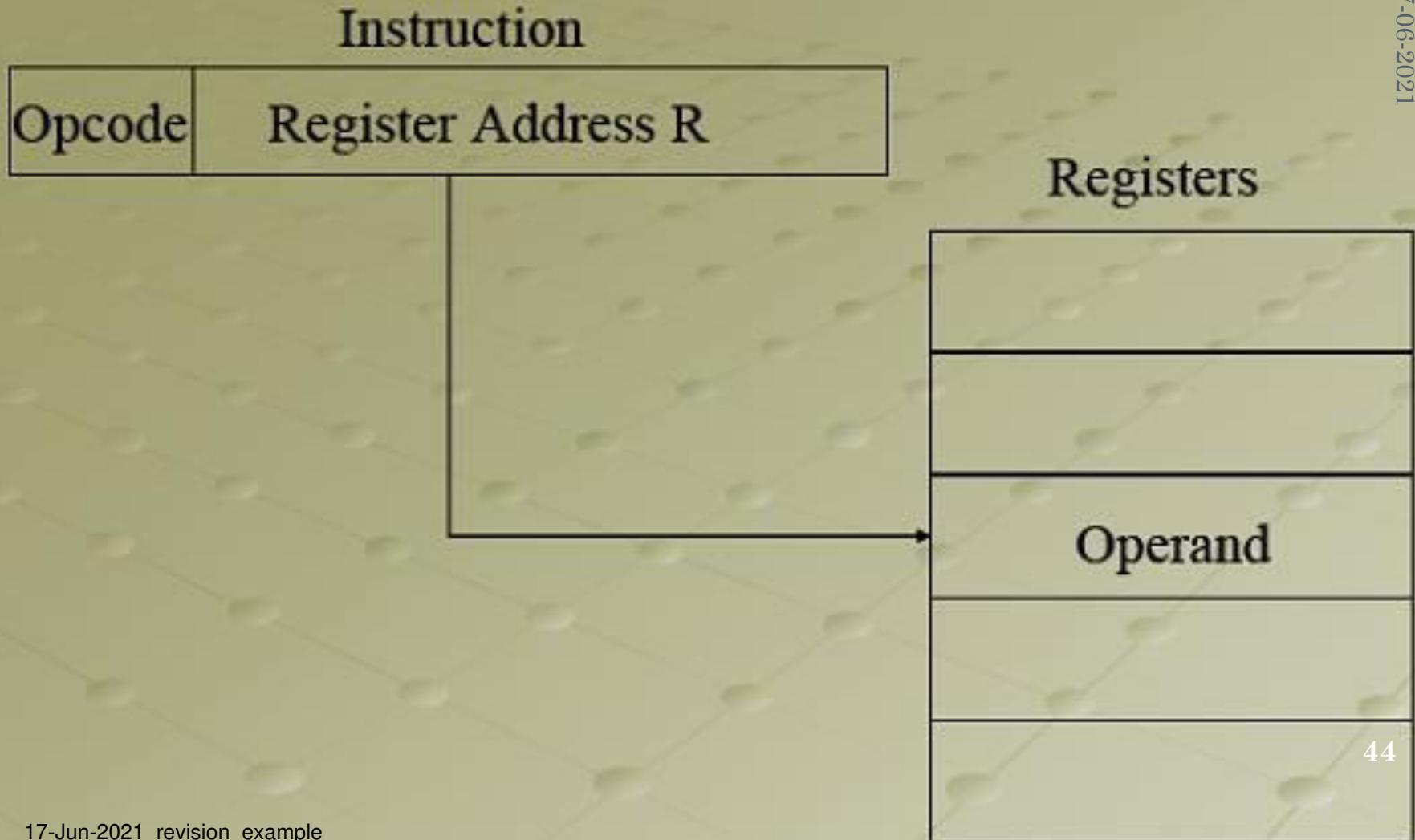
17-06-2021



REGISTER DIRECT ADDRESSING MODE

- Operand is in the register specified in the address part of the instruction
- $EA = R$
- Ex: `Mov AX, [BX]`
- Special case of direct addressing
- Advantage: No memory reference, shorter instructions, faster instruction fetch, very fast execution
- Disadvantage: Limited address space as limited number of registers

REGISTER ADDRESSING DIAGRAM

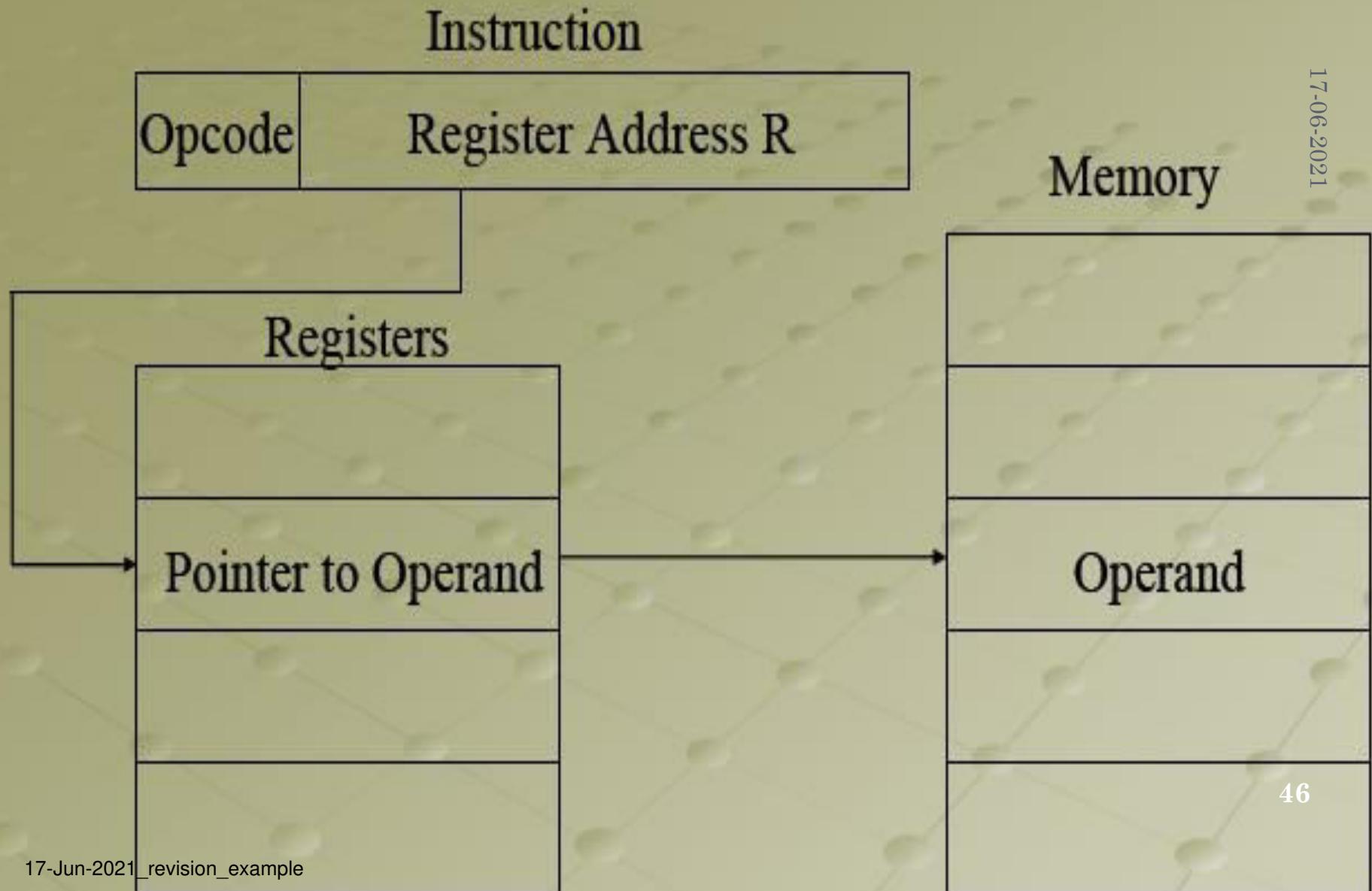


REGISTER INDIRECT ADDRESSING MODE

- Address part of the instruction specifies the register which gives the address of the operand in memory
- Special case of indirect addressing
- $EA = (R)$
- Ex: `Mov BX, [DX]`
- Advantage: Large address space
- Disadvantage: Extra memory reference

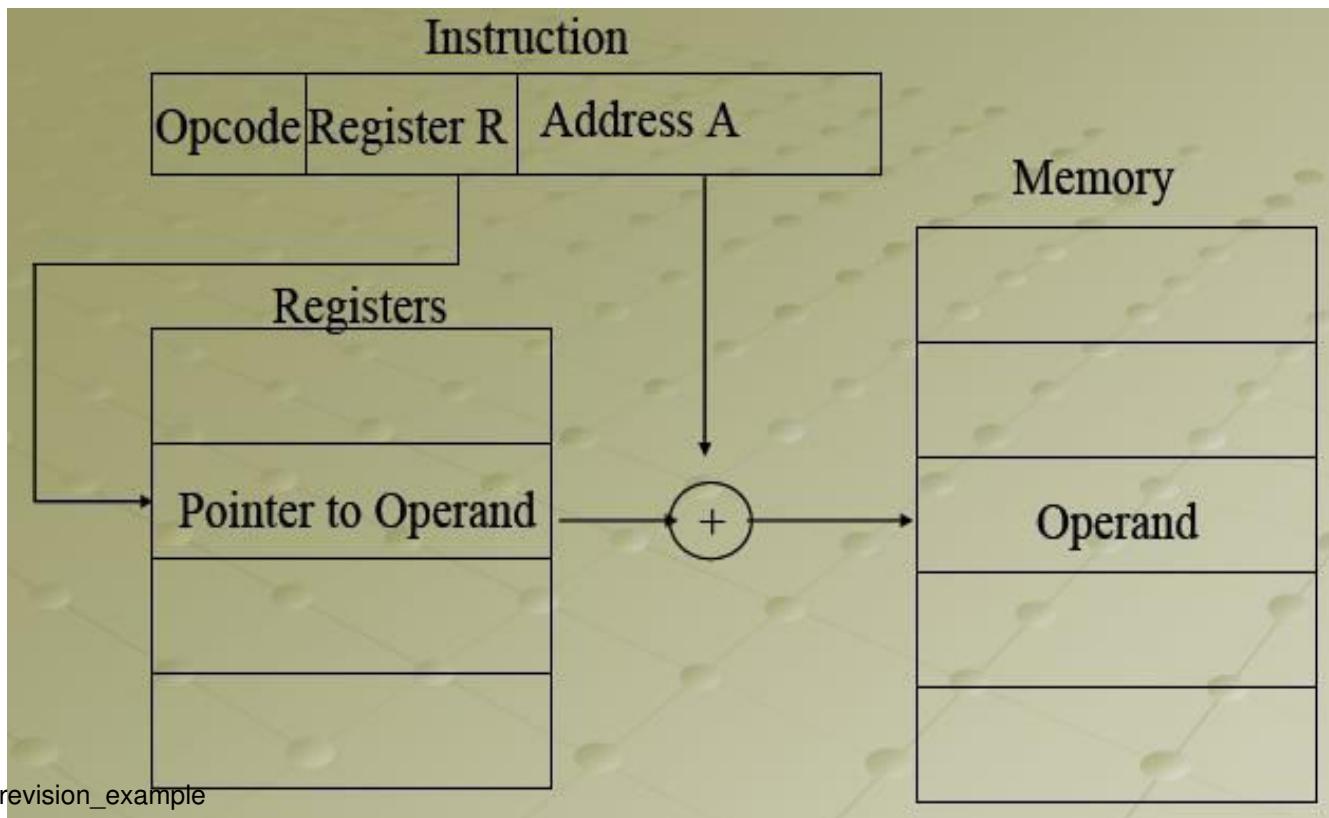
Register Indirect Addressing Diagram

17-06-2021



DISPLACEMENT ADDRESSING MODE

- $EA = A + (R)$
- Address field holds two values
 - A = Base value
 - R = register that holds displacement
 - Or vice-versa



RELATIVE ADDRESSING MODE

- Version of the displacement addressing
- R = program counter, PC
- Content of PC is added to address part of the instruction to obtain the effective address of the operand
- $EA = A + (PC)$
- It is often used in branch (conditional and unconditional) instructions, locality of reference and cache usage
- Advantage: Flexibility
- Disadvantage: Complexity

INDEXED ADDRESSING MODE

- A holds base address
- R holds displacement, may be explicit or implicit (segment registers in 8086)
- Content of the index register is added to the address part of the instruction to obtain effective address of the operand.
- Used in performing iterative operations
- $EA = A + (SI)$
- Ex: Mov CX, [SI] 2400H
- Advantage: Flexibility, good for accessing arrays
- Disadvantage: Complexity

BASE REGISTER ADDRESSING MODE

- The content of the base register is added to the address part of the instruction to obtain the effective address of the operand.
- Used to facilitate the relocation of programs in memory.
- $EA = A + (BX)$
- Ex: Mov 2345H [BX], 0AC24H
- Advantage: Flexibility
- Disadvantage: Complexity

AUTO INCREMENT AND AUTO DECREMENT ADDRESSING MODES

- This addressing mode is used when the address stored in the register refers to a table of data in memory, it is necessary to increment or decrement the register after every access to the table.
- Ex: Mov AX, (BX)+, Mov AX, -(BX)
- Used mostly in Motorola 680X0 series of computers

Basic Addressing Modes differences :

	Mode	Algorithm	Advantage	Disadvantage	
1	Immediate	Operand=1	No memory Reference	Limited operand magnitude	17-06-2021
2	Direct	EA = A	Simple	Limited address space	.
3	Indirect	EA =(A)	Large Address space	Multiple Memory References	.
4	Register	EA = R	No memory Reference	Limited address space	.
5	Register Indirect	EA = (R)	Large address space	Extra memory space	.
6	Displacement	EA = A+(R)	Flexibility	Complexity	.
7	Stack	EA= Top of Stack	No memory Reference	Limited Applicability	.

16 BIT ADDITION

ADDRESS	LABEL	OPCODE	MNEMONICS	OPERAND	COMMENTS
4100		C3	CLR	C	Clear carry
4101		74,04	MOV	A, #DATA1	Move data1 to acc
4103		24,02	ADD	A, #DATA2	Add data2 with acc
4105		90,41,50	MOV	DPTR, #4150h	Move content in 4500 to DPTR.
4108		FO	MOVX	@DPTR, A	Move data to DPTR location
4109		A3	INC	DPTR	Increment DPTR
410A		74,12	MOV	A, #DATA1	Move data1 to acc
410C		34,56	ADDC	A, #DATA2	Add with carry
410E		FO	MOVX	@DPTR, A	Move data to dp location
410F	HERE	80, FE	SJMP	HERE	End of program

Find the Output?

ADDRESS	LABEL	OPCODE	MNEMONICS	OPERAND	COMMENTS
4100		16,00	MVI	D, 00	Clear d register
4102		1E, 00	MVI	E, 00	Clear e register
4104		3A, 53,42	LDA	4253	Load data to acc
4107	HUND	FE, 64	CPI	64H	Compare data with acc
4109		DA, 12,41	JC	TEN	Jump on carry to adder
410C		D6, 64	SUI	64	Subtract data from acc
410E		1C	INR	E	Increment e register
410F		C3, 07,41	JMP	HUND	Jump to address
4112	TEN	FE, 0A	CPI	0AH	Compare data with acc
4114		DA, 1D, 41	JC	UNIT	Jump on carry to adder
4117		D6, 0A	SUI	0AH	Subtract data with acc
4119		14	INR	D	Increment d register
411A		C3, 12,41	JMP	TEN	Jump to address
411D	UNIT	4F	MOV	C, A	Move acc to c register
411E		7A	MOV	A, D	Move data to acc
411F		07	RLC		Rotate left without cy
4120		07	RLC		Rotate left without cy
4121		07	RLC		Rotate left without cy
4122		07	RLC		Rotate left without cy
4123		81	ADD	C	Add data to acc
4124		32,50,42	STA	4250	Store the result
4127	revision_example	7B	MOV	A, E	Move data to acc
4128		32, 51,42	STA	4251	Store the result

PROBLEMS

1. Find the effective address and the content of AC for the given data.

PC = 200

R1 = 400

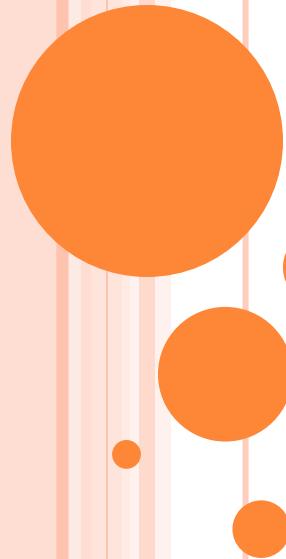
XR = 100

AC

Address	Memory	
200	Load to AC	Mode
201	Address = 500	
202	Next instruction	
399	450	
400	700	
500	800	
600	900	
702	325	
800	300	

Addressing Mode	Effective Address		Content of AC
Direct Address	500	$AC \leftarrow (500)$	800
Immediate operand	201	$AC \leftarrow 500$	500
Indirect address	800	$AC \leftarrow ((500))$	300
Relative address	702	$AC \leftarrow (PC + 500)$	325
Indexed address	600	$AC \leftarrow (XR + 500)$	900
Register	-	$AC \leftarrow R1$	400
Register Indirect	400	$AC \leftarrow (R1)$	700
Autoincrement	400	$AC \leftarrow (R1)+$	700
Autodecrement	399	$AC \leftarrow -(R1)$	450

- 2. An instruction is stored at location 300 with its address field at location 301. The address field has the value 400. A processor register R1 contains the number 200. Evaluate the effective address if the addressing mode of the instruction is (a) direct; (b) immediate (c) relative (d) register indirect; (e) index with R1 as the index register.



SUBROUTINE CALL AND RETURN MECHANISMS

SUBROUTINE

- Subroutine is a self-contained sequence of instructions that performs a given computational task.
- It may be called many times at various points in the main program
- When called, branches to 1st line of subroutine and at the end, returned to main program.
- Different names to the instruction that transfers program control to a subroutine
 - Call subroutine
 - Jump to subroutine
 - Branch to subroutine
 - Branch and save address



CONTROL TRANSFER FROM CALLED TO CALLER

- Subroutine instruction – Opcode + starting address of the subroutine
- Execution:
 - PC content (return address) is stored in a temporary location
 - Control is transferred to the subroutine
- when return
 - Transfers the return address from the temporary location to the PC.
 - Control is transferred back to the called routine



LOCATIONS TO STORE THE RETURN ADDRESS

- First memory location of the subroutine
- Fixed location in memory
- Processor registers
- Memory stack – best option
 - Adv: In the case of sequential calls to subroutines. So, the top of the stack always has the return address of the subroutine which to be returned first.



MICRO-OPERATIONS

Call:

```
SP ← SP – 1      // decrement stack pointer  
M[SP] ← PC       // push content of PC onto the stack  
PC ← effective address /* transfer control to the  
subroutine */
```

Return:

```
PC ← M[SP] // pop stack and transfer to PC  
SP ← SP + 1 // increment stack pointer
```

RECURSIVE SUBROUTINES

- Subroutine that calls itself
- If only one register or memory location is used to hold the return address, when subroutine is called recursively, it destroys the previous return address.
- So, stack is the good solution for this problem



ASSIGNMENT

1. Write an assembly language program using IAS instruction set for performing all arithmetic operations (+, -, *, /)
2. Show the register transfer operations using IAS machine registers for division operation.
3. Given the memory contents of the IAS computer shown below. Show the assembly language code for the program, starting at address 08A. Explain what this program does. Given the memory contents of the IAS computer shown below. Show the assembly language code for the program, starting at address 08A. Explain what this program does.

Address	Contents
08A	010FA210FB
08B	010FA0F08D
08C	020FA210FB

4. Write an Assembly language programming for the following expressions using IAS computer Instruction set and interpret to the flow of IAS computer [Any one]

1. $A = (B - C) * D$
2. $A = B * (C + D)$
3. $A = (B - C) / D$
4. $A = B / (C + D)$
5. $A = -(B + C - D)$
6. $A = (B * 2) / 2$

Make necessary assumptions.

5. On the IAS, describe in English the process that the CPU must undertake to read a value from memory and to write a value to memory in terms of what is put into the MAR, MBR, address bus, data bus, and control bus.
6. Find out the difference between Multicomputer, Multiprocessor, Distributed computer, Multicores

7. A two-word instruction is stored in memory at an address designated by the symbol W. The address field of the instruction (stored at $W + 1$) is designated by the symbol Y. The operand used during the execution of the instruction is stored at an address symbolized by Z. An index register contains the value X. State how Z is calculated from the other addresses if the addressing mode of the instruction is
- Direct
 - Indirect
 - Relative
 - Indexed
8. A relative mode branch type of instruction is stored in memory at an address equivalent to decimal 750. The branch is made to an address equivalent to decimal 500. What should be the value of the relative address field of the instruction (in decimal)?

9. How many times does the control unit refer to memory when it fetches and executes an indirect addressing mode instruction if the instruction is (a) a computational type requiring an operand from memory; (b) a branch type.
10. What must the address field of an indexed addressing mode instruction be to make it the same as a register indirect mode instruction?
11. An instruction is stored at location 300 with its address field at location 301. The address field has the value 400. A processor register R1 contains the number 200. Evaluate the effective address if the addressing mode of the instruction is (a) direct; (b) immediate (c) relative (d) register indirect; (e) index with R1 as the index register.

12. Assume that in a certain byte-addressed machine all instructions are 32 bits long. Assume the following state of affairs for the machine: Fill in the following table:

Address	Value	Instruction	Addressing mode	Value in R0
PC	100	Load r0, #200	Immediate	
R0	200	Load r0, 200	Direct	
R1	300	Load r0, (200)	Indirect	
100	200	Load r0,r1	Register	
104	300	Load r0, [r1]	Register Indirect	
108	400	Load r0, -100[r1]	Based	
200	500	Load r0, 200[PC]	Relative	
300	600			
500	700			

13. Given the following memory values and a one-address machine with an accumulator, what values do the following instructions load into the accumulator?

- Word 20 contains 40
- Word 30 contains 50
- Word 40 contains 60
- Word 50 contains 70
 - Load immediate 20
 - Load direct 20
 - Load indirect 20
 - Load immediate 30
 - Load direct 30
 - Load indirect 30

14. Let the address stored in the program counter be designated by the symbol X1. The instruction stored in X1 has the address part (operand reference) X2. The operand needed to execute the instruction is stored in the memory word with address X3. An index register contains the value X4. What is the relationship between these various quantities if the addressing mode of the instruction is (a) direct (b) indirect (c) PC relative (d) indexed?



15. An address field in an instruction contains decimal value 14. where is the corresponding operand located for:

- Immediate addressing?
- Direct addressing?
- Indirect addressing?
- Register addressing?
- Register indirect addressing?

16. A PC-relative mode branch instruction is stored in memory at address 620_{10} . The branch is made to location 530_{10} . The address field in the instruction is 10 bits long. What is the binary value in the instruction?

REFERENCES

- William Stallings “Computer Organization and architecture” 8th edition:
 - History of computing :pg 35-56
 - IAS organization : pg 36 -42
 - Instruction fetch & execute : pg 87 – 91
 - Addressing Modes : pg 419 -426

M. M. Mano, Computer System Architecture,
Prentice-Hall

Instruction format : pg 255-260

subroutine call & return statement : pg 278 -279

Vincent .P. Heuring, Harry F. Jordan “ Computer System design and Architecture” Pearson, 2nd Edition, 2003

Instruction format calculation



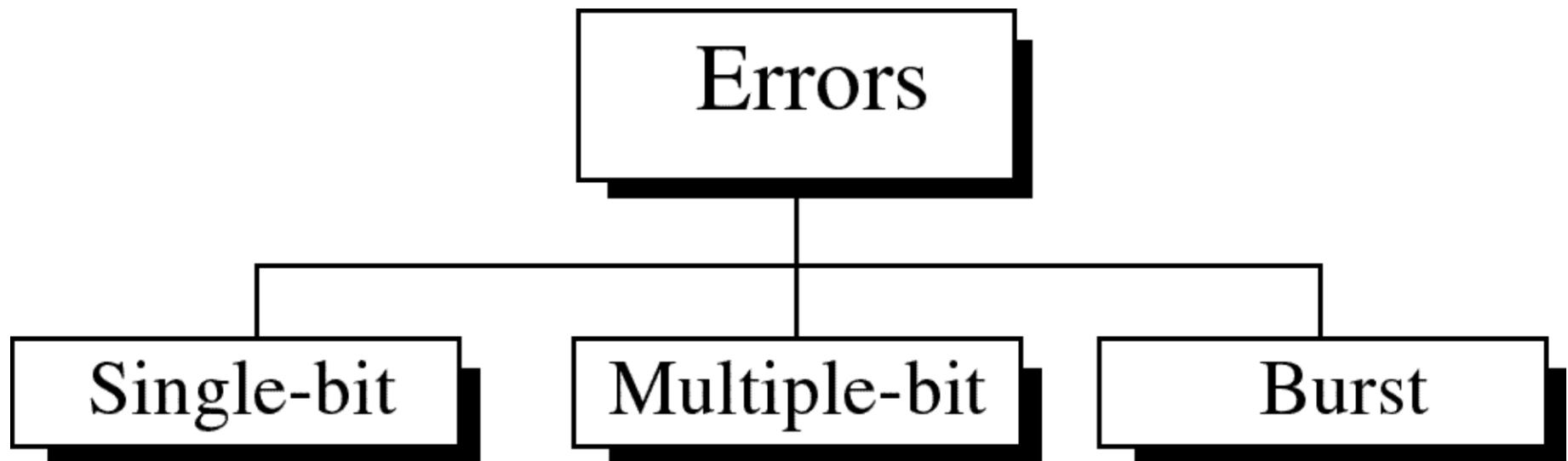
Error Detection and Correction

- Types of Errors
- Detection
- Correction

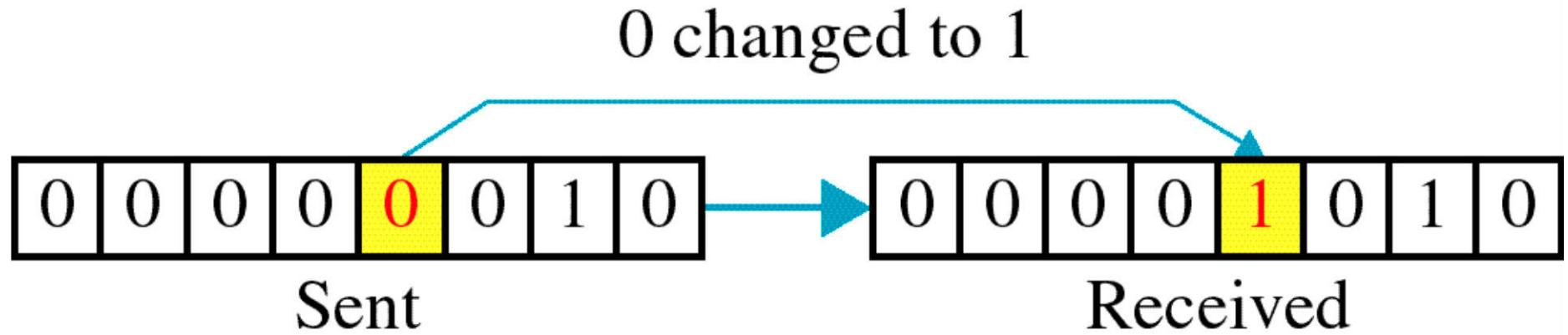
Basic concepts

- ★ Networks must be able to transfer data from one device to another with complete accuracy.
- ★ Data can be corrupted during transmission.
- ★ For reliable communication, errors must be detected and corrected.
- ★ **Error detection and correction** are implemented either at the **data link layer** or the **transport layer** of the OSI model.

Types of Errors



Single-bit error

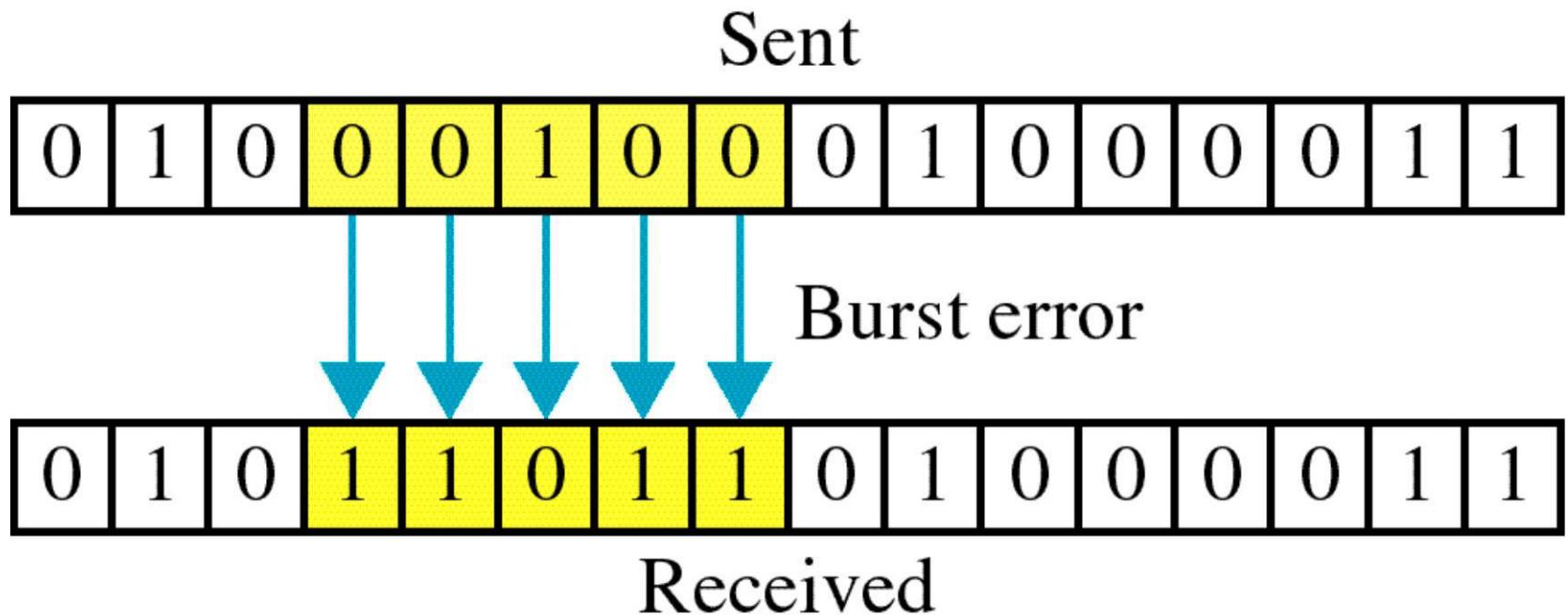


Single bit errors are the **least likely** type of errors in serial data transmission because the noise must have a very short duration which is very rare. However this kind of errors can happen in parallel transmission.

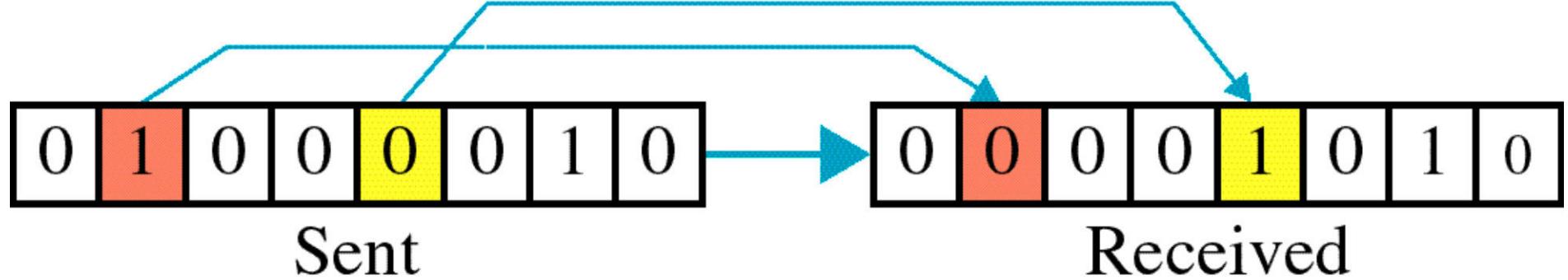
Example:

- ★ If data is sent at 1Mbps then each bit lasts only $1/1,000,000$ sec. or $1 \mu\text{s}$.
- ★ For a single-bit error to occur, the noise must have a duration of only $1 \mu\text{s}$, which is very rare.

Burst error



Two errors



The term **burst error** means that two or more bits in the data unit have changed from 1 to 0 or from 0 to 1.

Burst errors does not necessarily mean that the errors occur in consecutive bits, the length of the burst is measured from the first corrupted bit to the last corrupted bit. Some bits in between may not have been corrupted.

- ★ **Burst error is most likely to happen in serial transmission** since the duration of noise is normally longer than the duration of a bit.
- ★ The number of bits affected depends on the data rate and duration of noise.

Example:

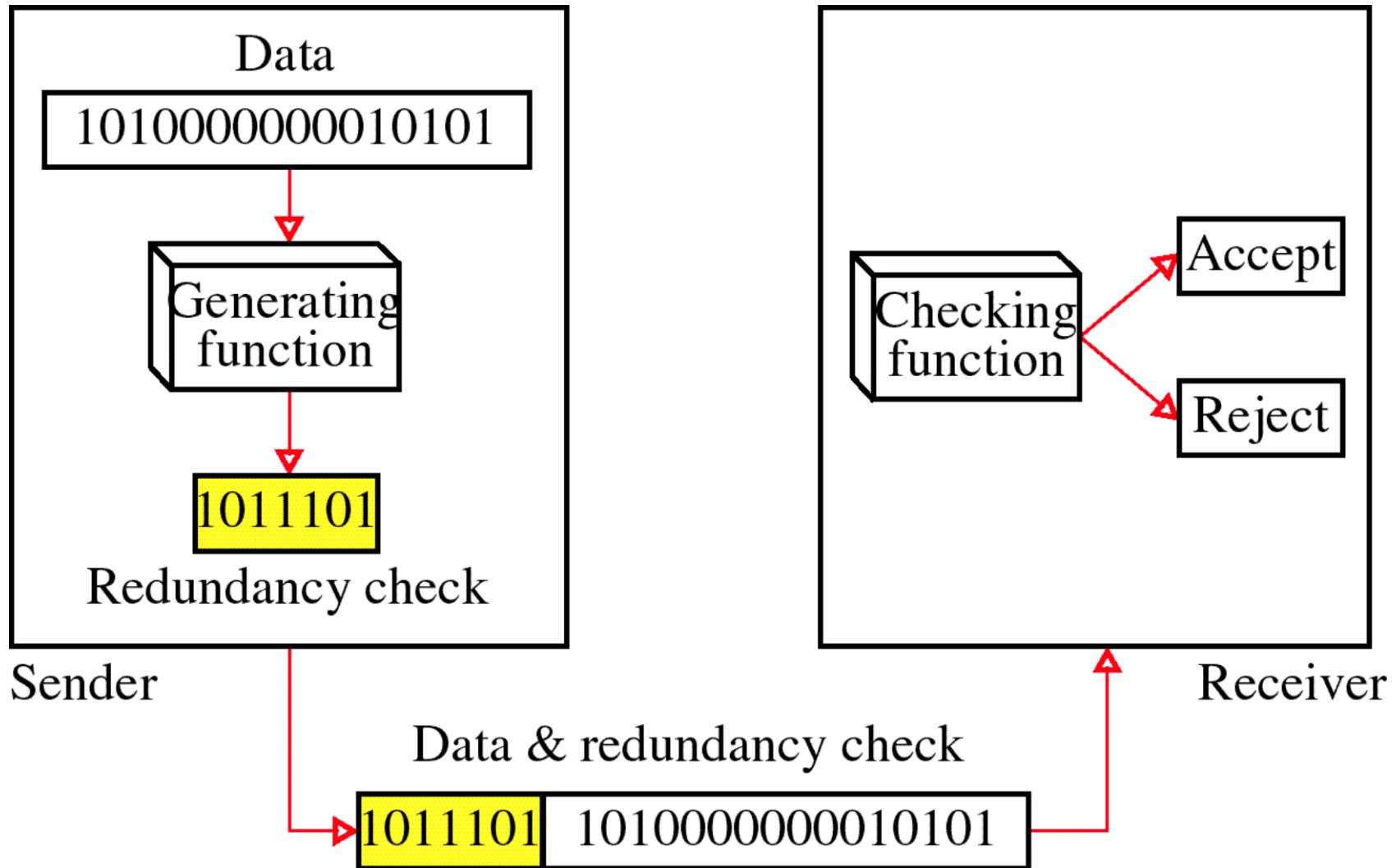
- ➔ If data is sent at rate = 1Kbps then a noise of 1/100 sec can affect 10 bits. $(1/100 * 1000)$
- ➔ If same data is sent at rate = 1Mbps then a noise of 1/100 sec can affect 10,000 bits. $(1/100 * 10^6)$

Error detection

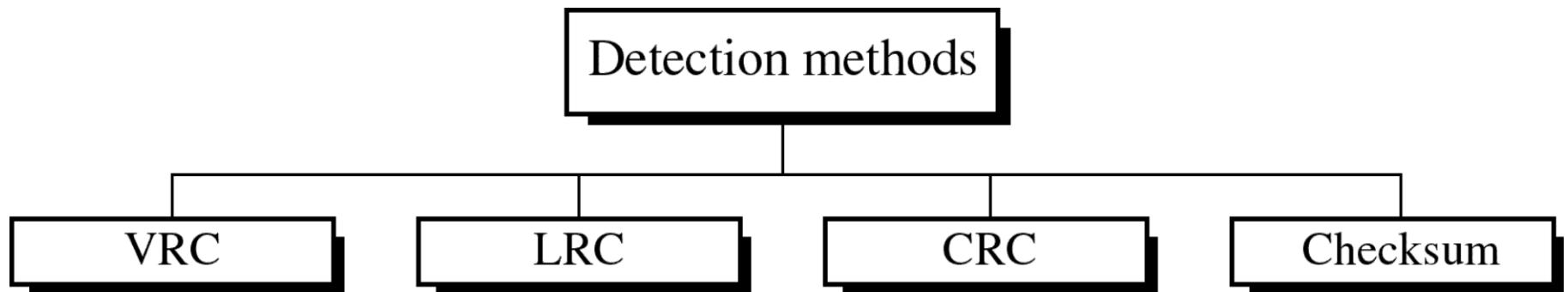
Error detection means to decide whether the received data is correct or not without having a copy of the original message.

Error detection **uses the concept of redundancy, which means** adding extra bits for detecting errors at the destination.

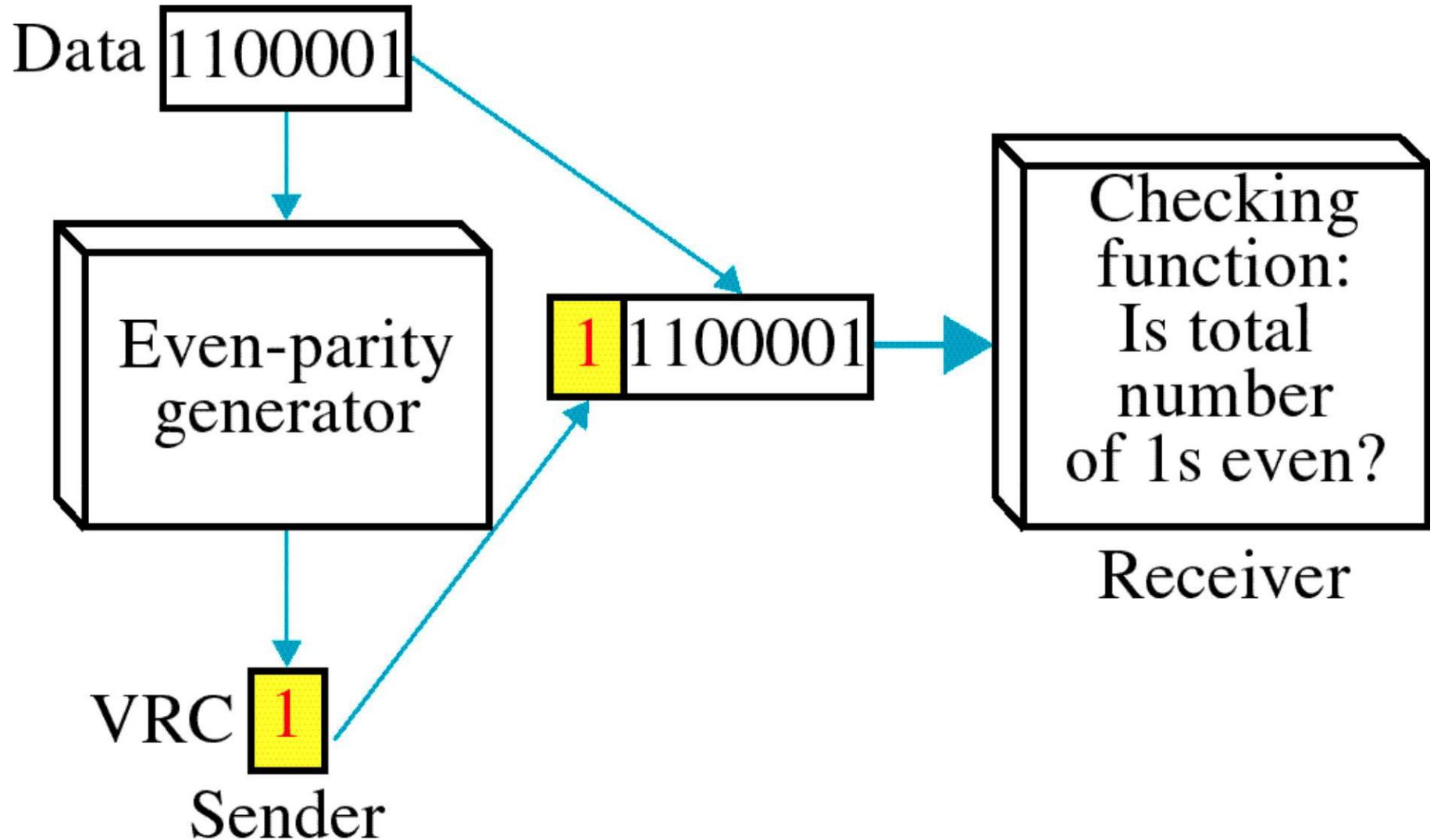
Redundancy



Four types of redundancy checks are used in data communications



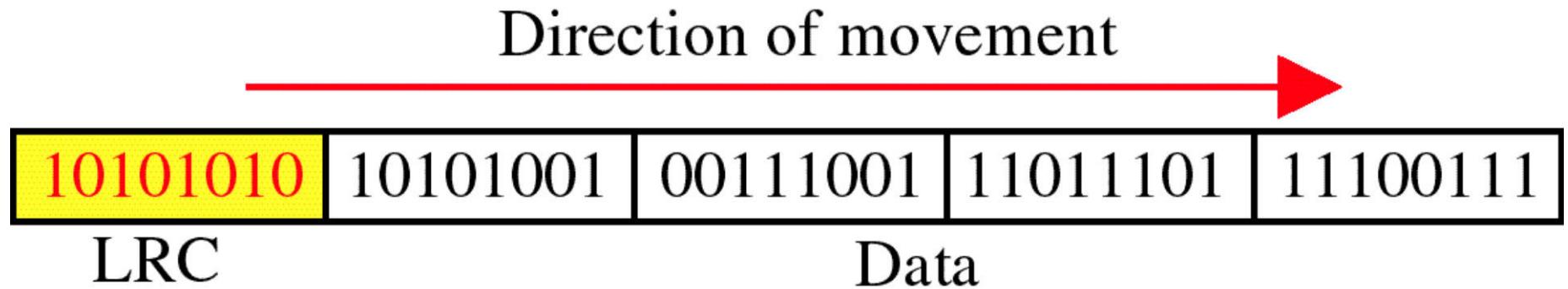
Vertical Redundancy Check VRC



Performance

- ➔ It can detect single bit error
- ➔ It can detect burst errors only if the total number of errors is odd.

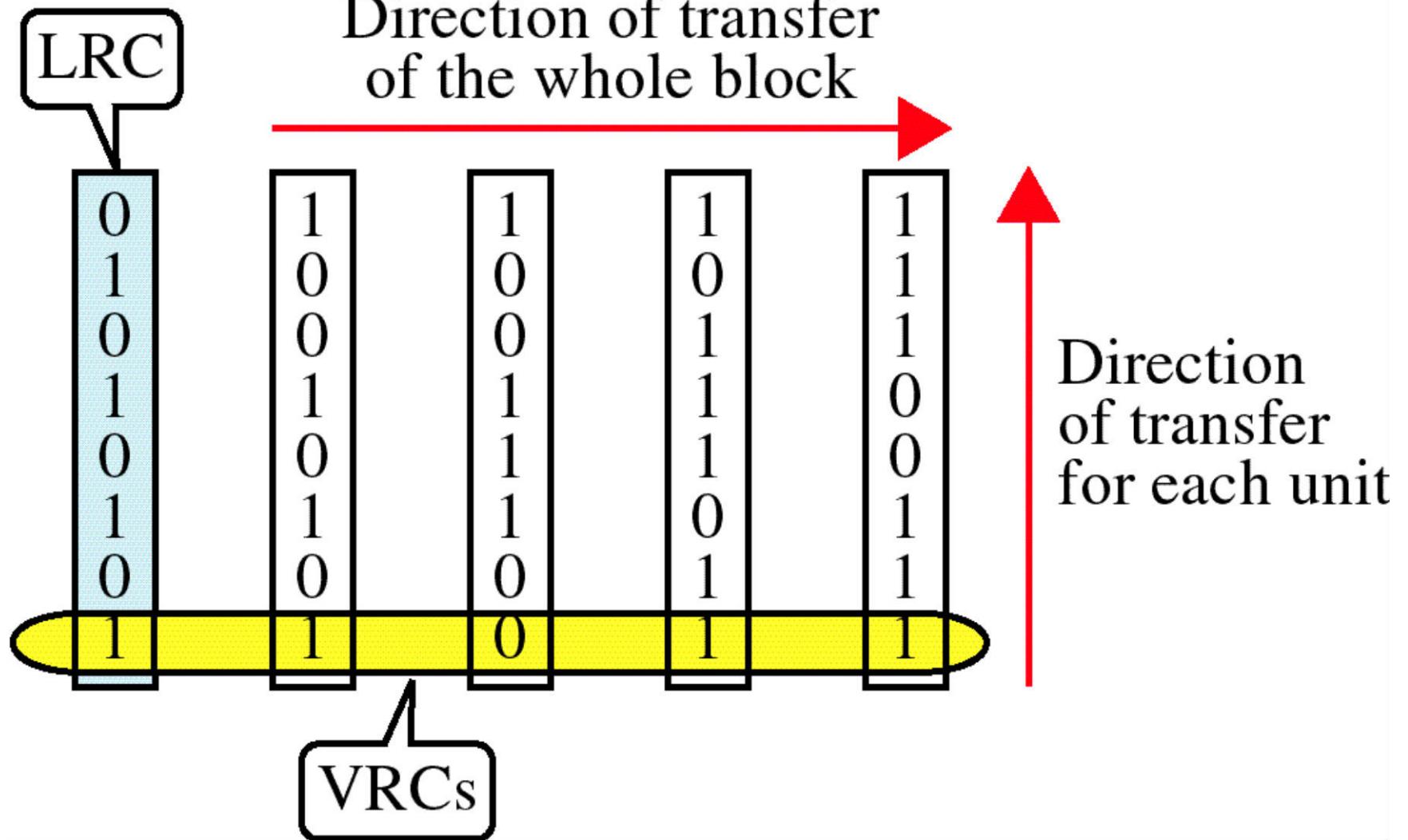
Longitudinal Redundancy Check LRC



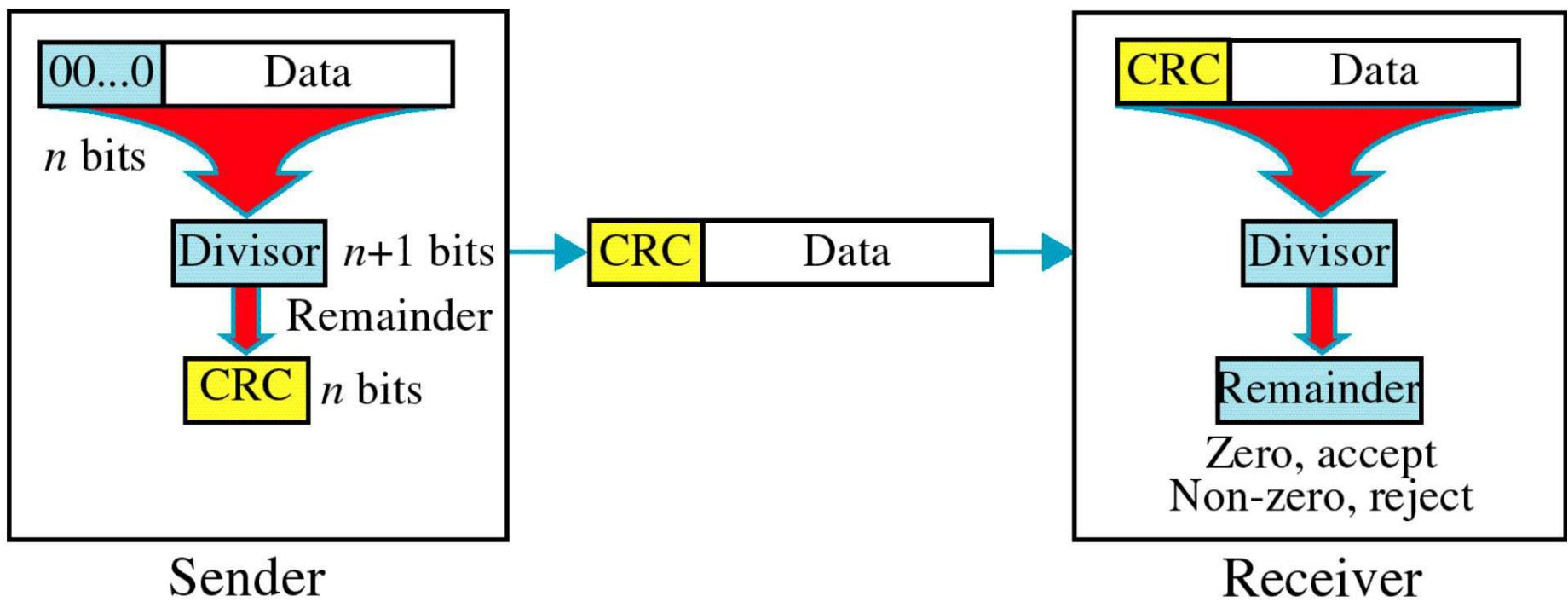
Performance

- LCR increases the likelihood of detecting burst errors.
- If two bits in one data units are damaged and two bits in exactly the same positions in another data unit are also damaged, the LRC checker will not detect an error.

VRC and LRC



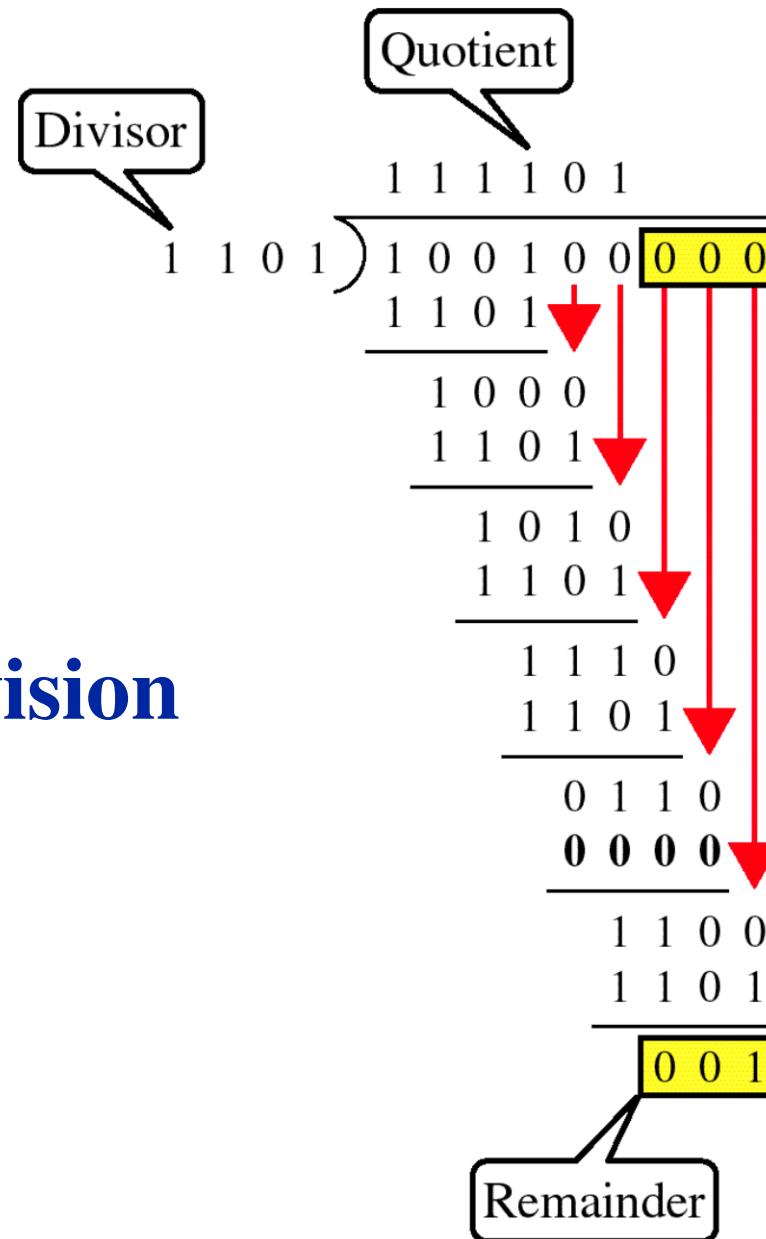
Cyclic Redundancy Check CRC



Cyclic Redundancy Check

- Given a k -bit frame or message, the transmitter generates an n -bit sequence, known as a *frame check sequence (FCS)*, so that the resulting frame, consisting of $(k+n)$ bits, is exactly divisible by some predetermined number.
- The receiver then divides the incoming frame by the same number and, if there is no remainder, assumes that there was no error.

Binary Division



Polynomial

$$x^7 + x^5 + x^2 + x + 1$$

Polynomial and Divisor

Polynomial

$$x^7 + x^5 + x^2 + x + 1$$

$$x^6 \quad x^4 \quad x^3$$

1	0	1	0	0	1	1	1
---	---	---	---	---	---	---	---

Divisor

Standard Polynomials

CRC-12

$$x^{12} + x^{11} + x^3 + x + 1$$

CRC-16

$$x^{16} + x^{15} + x^2 + 1$$

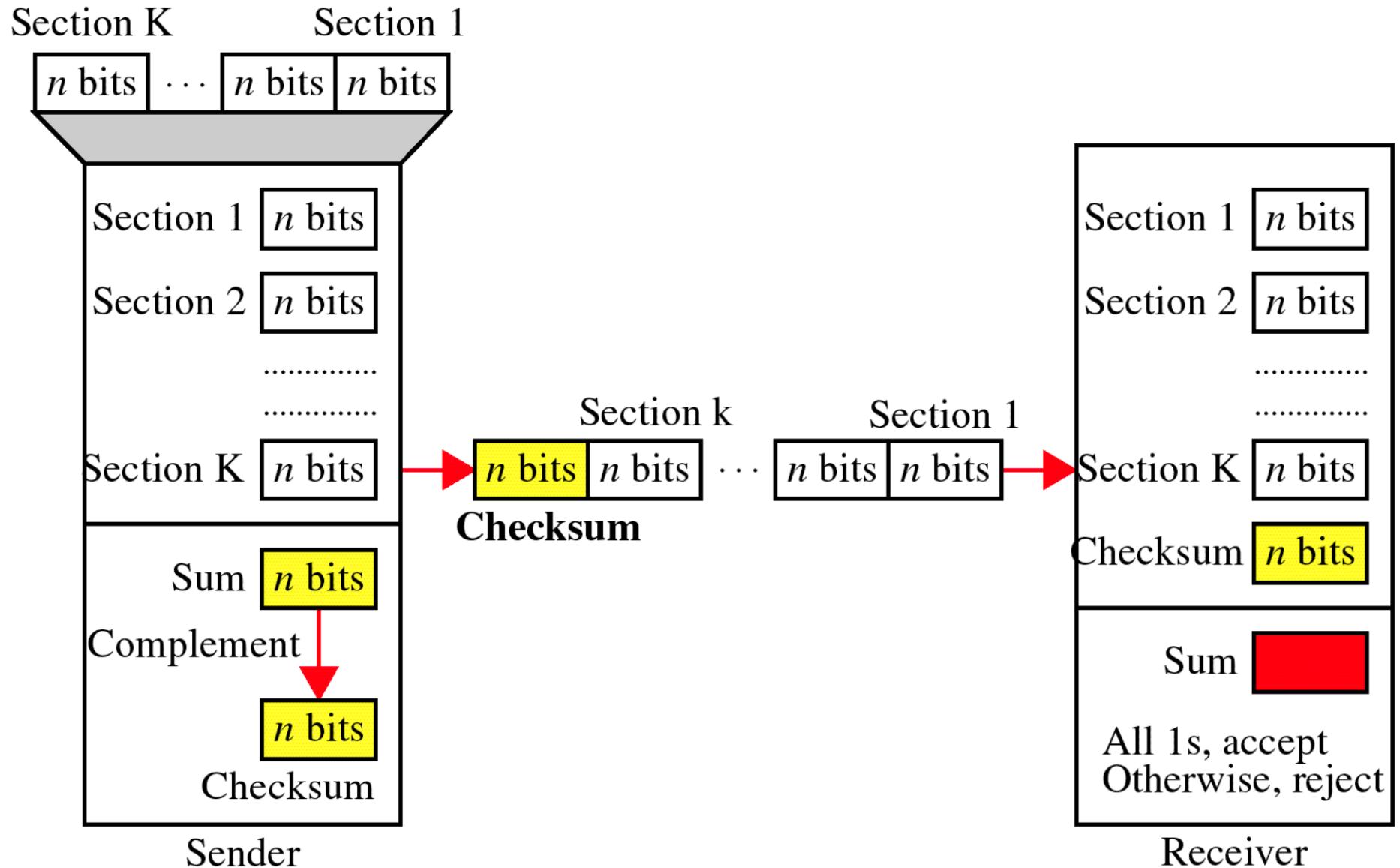
CRC-ITU

$$x^{16} + x^{12} + x^5 + 1$$

CRC-32

$$x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$$

Checksum



At the sender

- The unit is divided into k sections, each of n bits.
- All sections are added together using one's complement to get the sum.
- The sum is complemented and becomes the checksum.
- The checksum is sent with the data

At the receiver

- ➔ The unit is divided into k sections, each of n bits.
- ➔ All sections are added together using one's complement to get the sum.
- ➔ The sum is complemented.
- ➔ If the result is zero, the data are accepted: otherwise, they are rejected.

Example:

$$k=4, m=8$$

10110011

10101011

$$\begin{array}{r} \text{01011110} \\ \text{1} \end{array}$$

01011111

01011010

10111001

11010101

$$\begin{array}{r} \text{10001110} \\ \text{1} \end{array}$$

Sum : $\begin{array}{r} \text{10001111} \\ \hline \end{array}$

Checksum $\underline{\text{01110000}}$

(a)

Example: Received data

10110011

10101011

01011110

1

01011111

01011010

10111001

11010101

10001110

1

10001111

01110000

Sum: 11111111

Complement = 00000000

Conclusion = Accept data

(b)

Performance

- ➔ The checksum detects all errors involving an odd number of bits.
- ➔ It detects most errors involving an even number of bits.
- ➔ If one or more bits of a segment are damaged and the corresponding bit or bits of opposite value in a second segment are also damaged, the sums of those columns will not change and the receiver will not detect a problem.

Error Correction

It can be handled in two ways:

- 1) receiver can have the sender retransmit the entire data unit.
- 2) The receiver can use an error-correcting code, which automatically corrects certain errors.

Single-bit error correction

To correct an error, the receiver reverses the value of the altered bit. To do so, it must know which bit is in error.

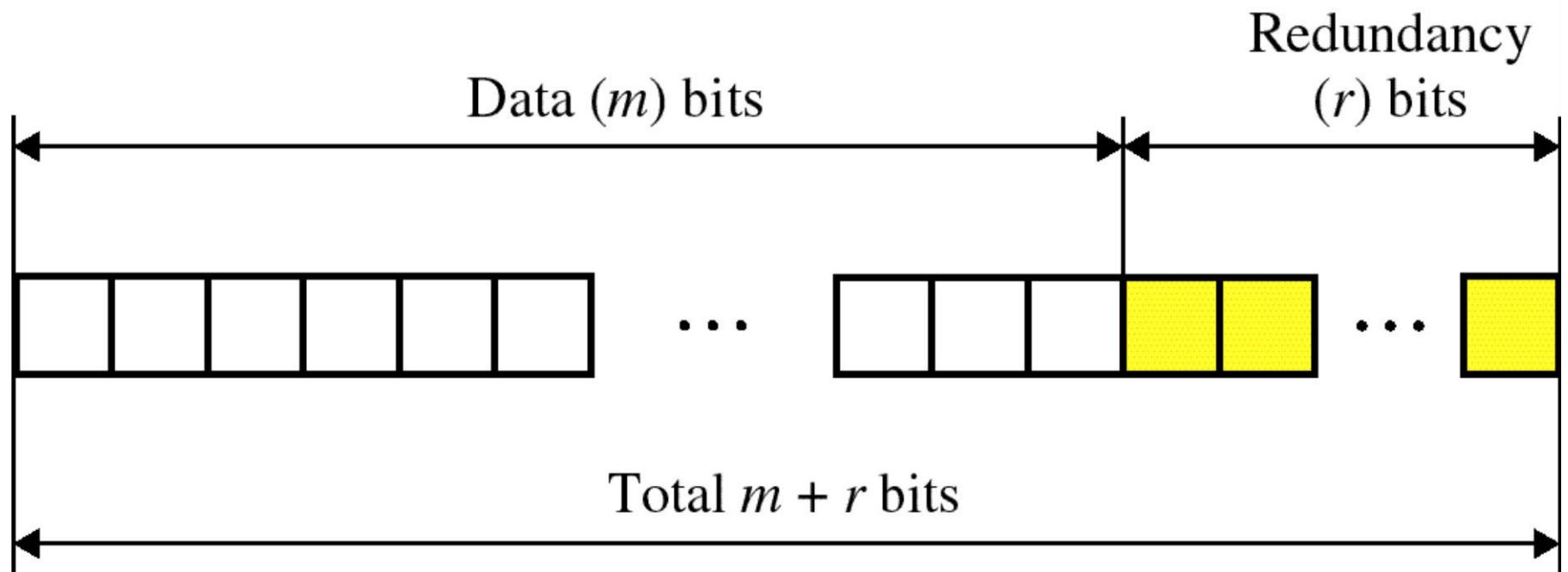
Number of redundancy bits needed

- Let data bits = m
 - Redundancy bits = r
- ∴ Total message sent = $m+r$

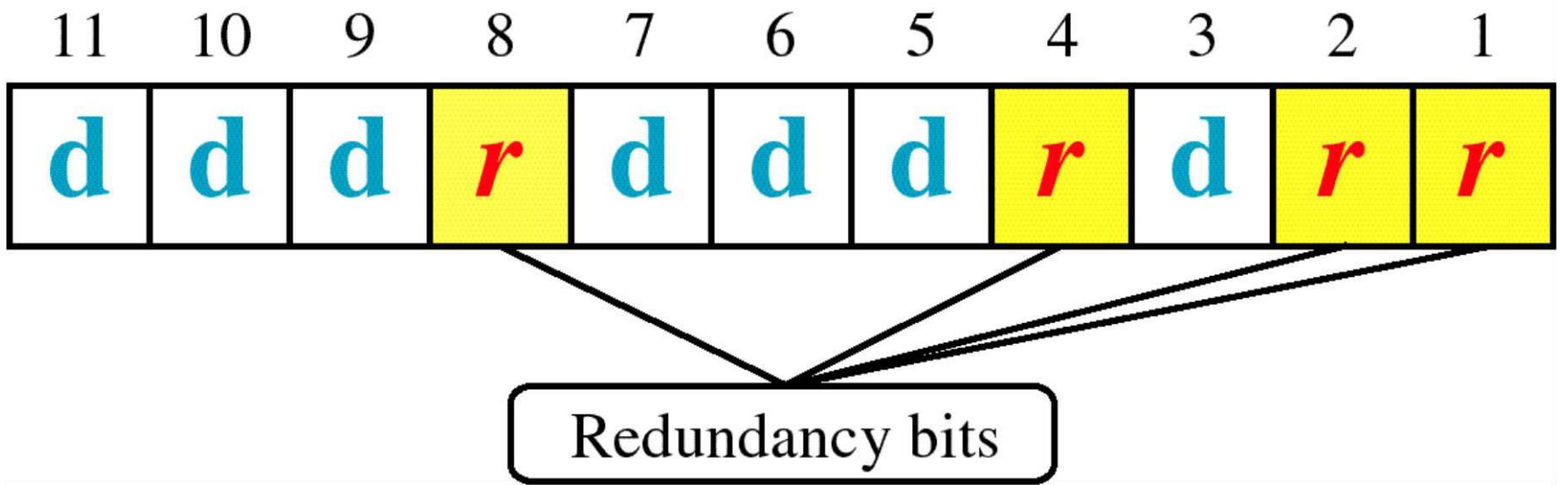
The value of r must satisfy the following relation:

$$2^r \geq m+r+1$$

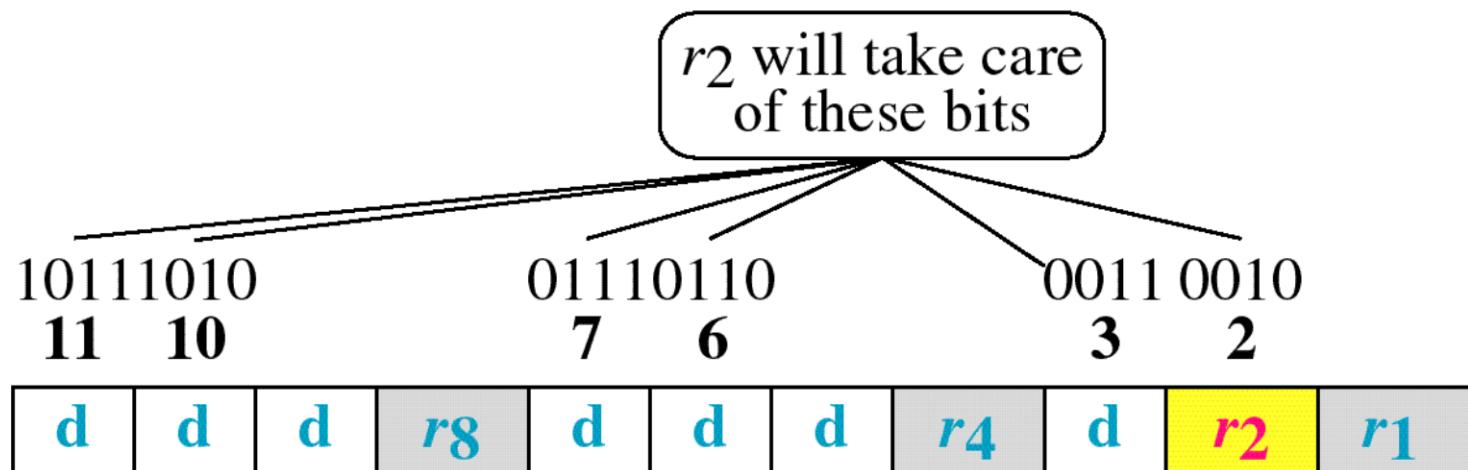
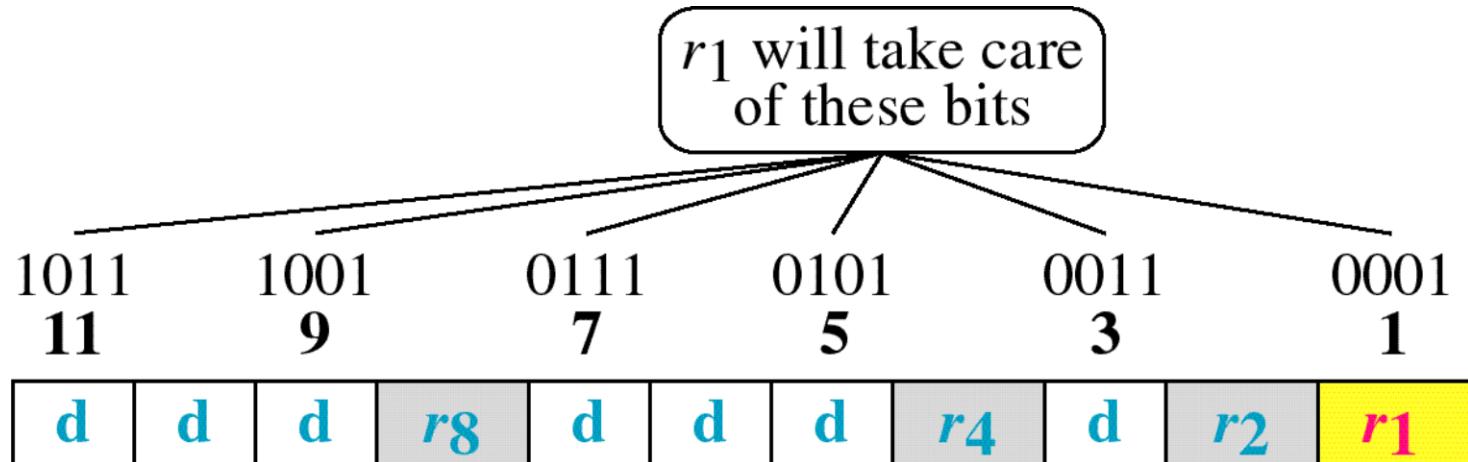
Error Correction



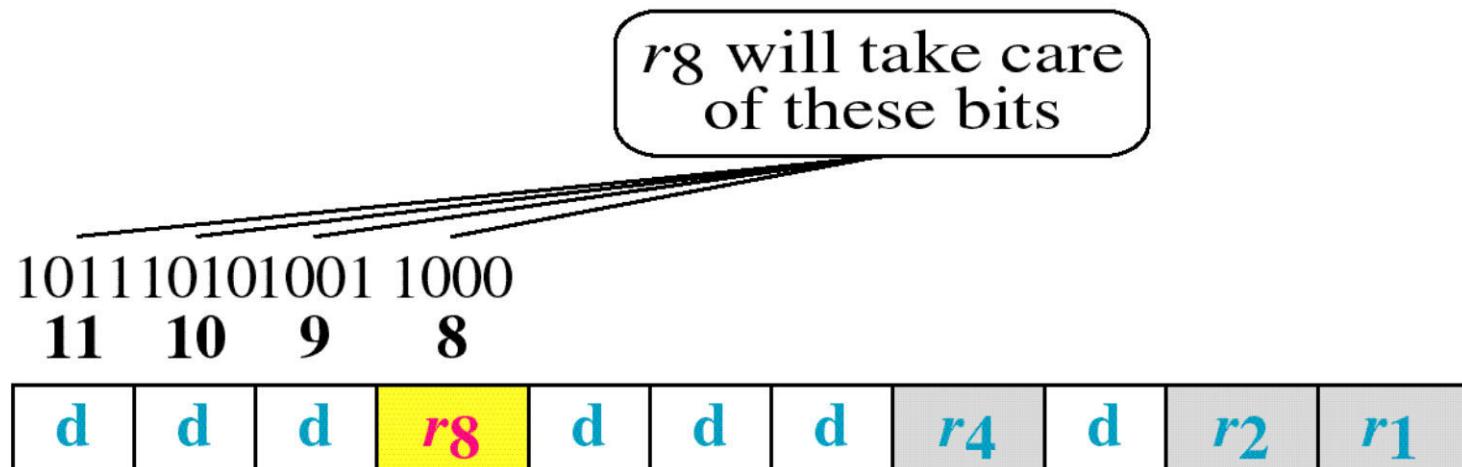
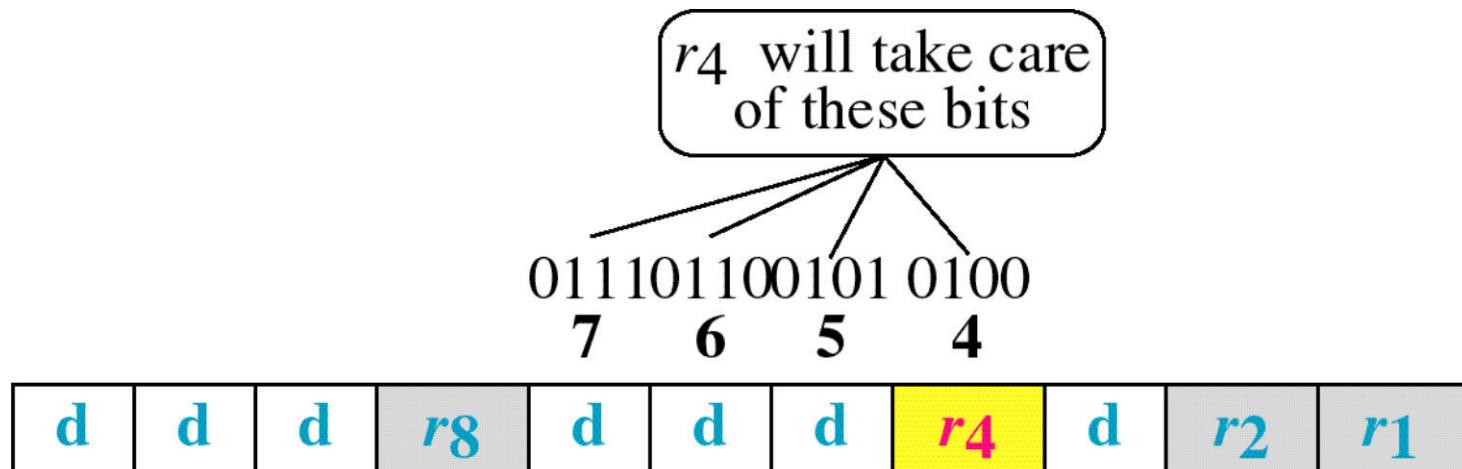
Hamming Code



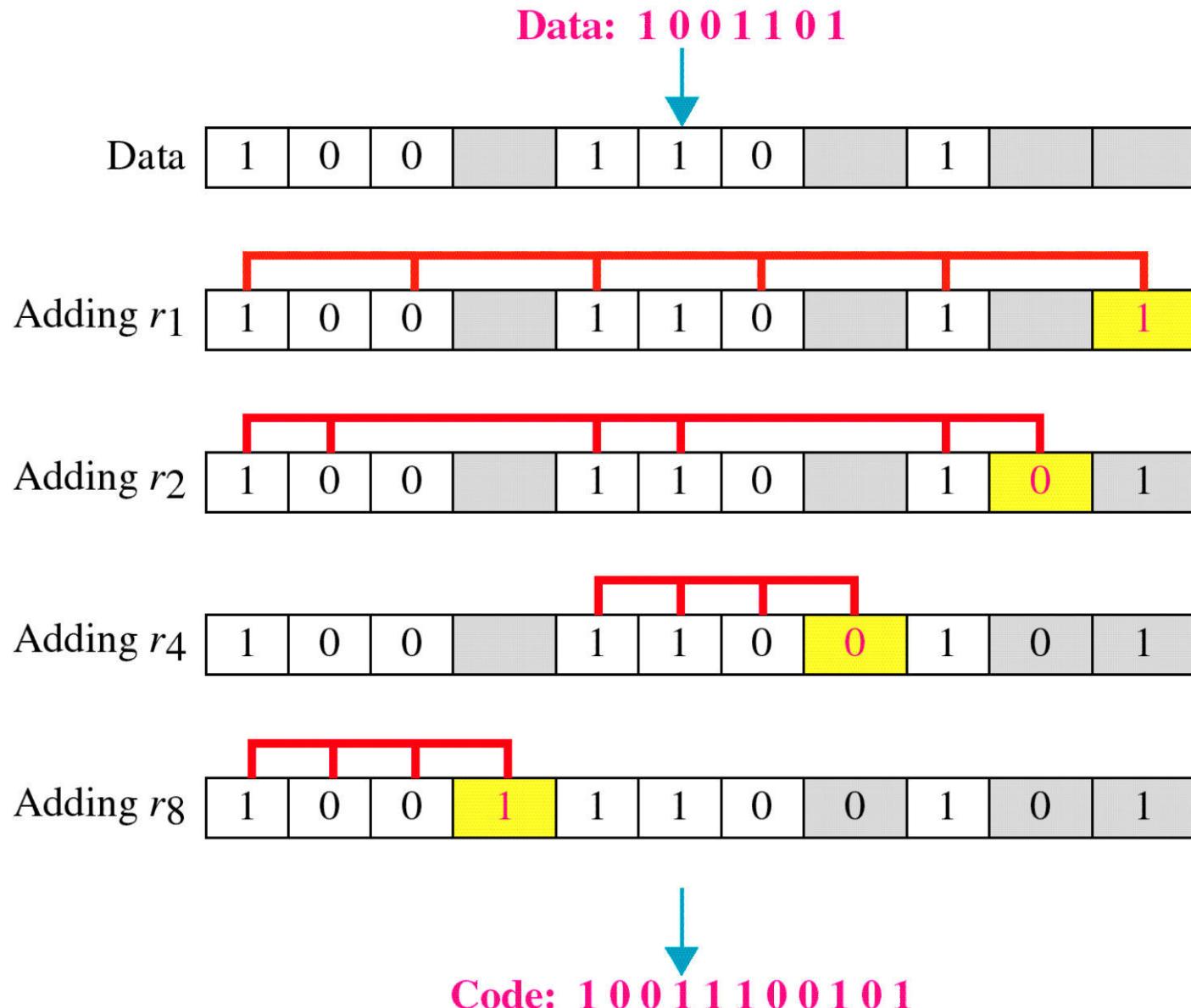
Hamming Code



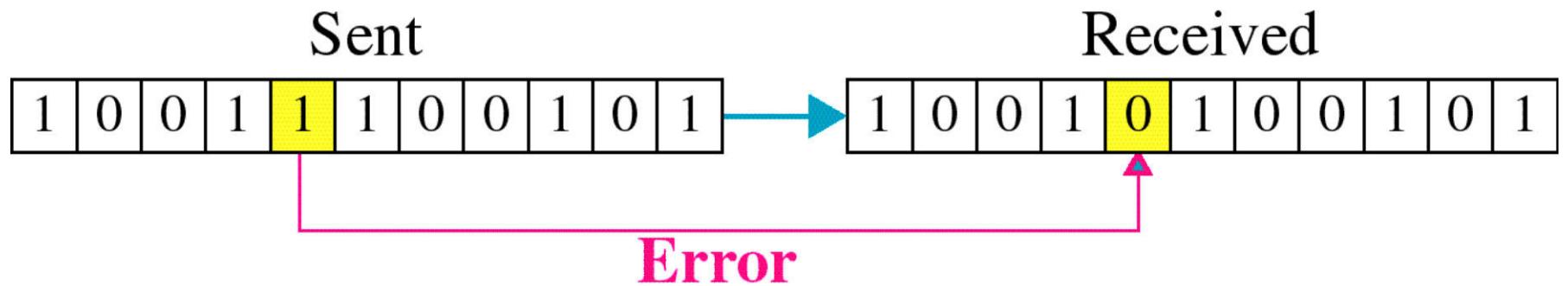
Hamming Code



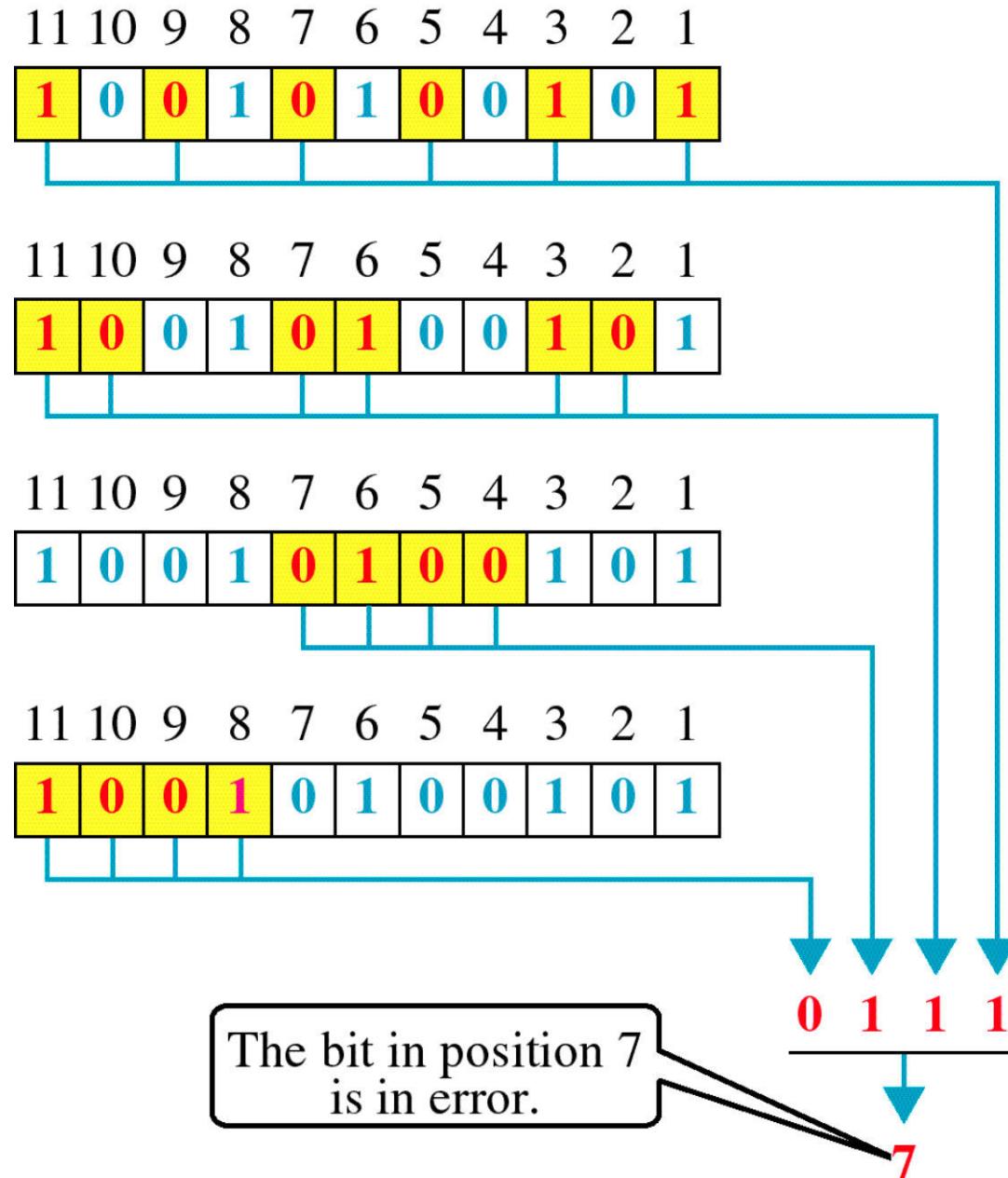
Example of Hamming Code



Single-bit error



Error Detection



Input/Output Organization

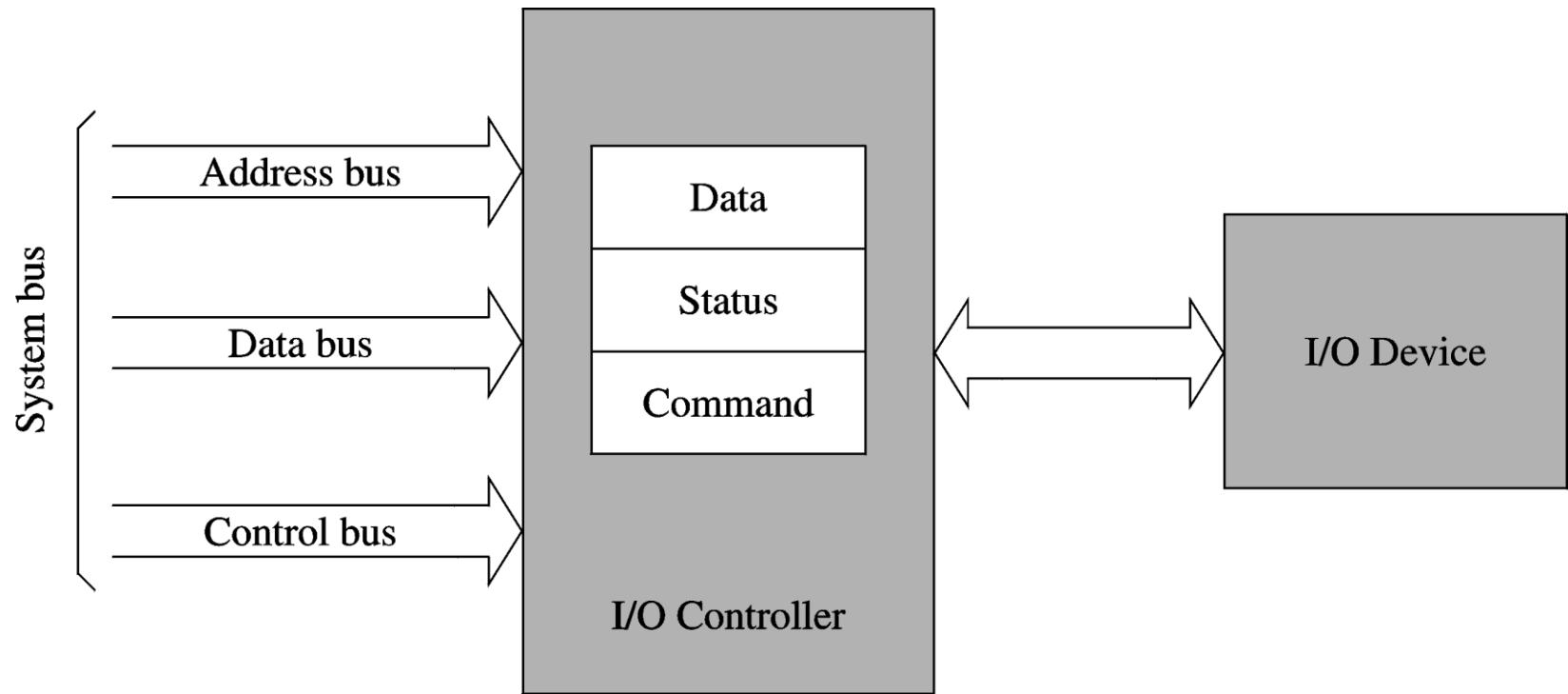
Outline

- Introduction
- Accessing I/O devices
- An example I/O device
 - * Keyboard
- I/O data transfer
 - * Programmed I/O
 - * DMA
- Error detection and correction
 - * Parity encoding
 - * Error correction
 - * CRC
- External interface
 - * Serial transmission
 - * Parallel interface
- USB
 - * Motivation
 - * USB architecture
 - * USB transactions
- IEEE 1394
 - * Advantages
 - * Transactions
 - * Bus arbitration
 - * Configuration

Introduction

- I/O devices serve two main purposes
 - * To communicate with outside world
 - * To store data
- I/O controller acts as an interface between the systems bus and I/O device
 - * Relieves the processor of low-level details
 - * Takes care of electrical interface
- I/O controllers have three types of registers
 - * Data
 - * Command
 - * Status

Introduction (cont'd)



Introduction (cont'd)

- To communicate with an I/O device, we need
 - * Access to various registers (data, status,...)
 - » This access depends on I/O mapping
 - Two basic ways
 - Memory-mapped I/O
 - Isolated I/O
 - * A protocol to communicate (to send data, ...)
 - » Three types
 - Programmed I/O
 - Direct memory access (DMA)
 - Interrupt-driven I/O

Accessing I/O Devices

- I/O address mapping
 - * Memory-mapped I/O
 - » Reading and writing are similar to memory read/write
 - » Uses same memory read and write signals
 - » Most processors use this I/O mapping
 - * Isolated I/O
 - » Separate I/O address space
 - » Separate I/O read and write signals are needed
 - » Pentium supports isolated I/O
 - 64 KB address space
 - Can be any combination of 8-, 16- and 32-bit I/O ports
 - Also supports memory-mapped I/O

Accessing I/O Devices (cont'd)

- Accessing I/O ports in Pentium

- * Register I/O instructions

- `in accumulator, port8 ; direct format`

- Useful to access first 256 ports

- `in accumulator,DX ; indirect format`

- DX gives the port address

- * Block I/O instructions

- » `ins` and `outs`

- Both take no operands---as in string instructions

- » `ins`: port address in DX, memory address in ES:(E)DI

- » `outs`: port address in DX, memory address in ES:(E)SI

- » We can use `rep` prefix for block transfer of data

An Example I/O Device

- Keyboard
 - * Keyboard controller scans and reports
 - Key depressions and releases
 - » Supplies key identity as a scan code
 - Scan code is like a sequence number of the key
 - Key's scan code depends on its position on the keyboard
 - No relation to the ASCII value of the key
 - * Interfaced through an 8-bit parallel I/O port
 - » Originally supported by 8255 programmable peripheral interface chip (PPI)

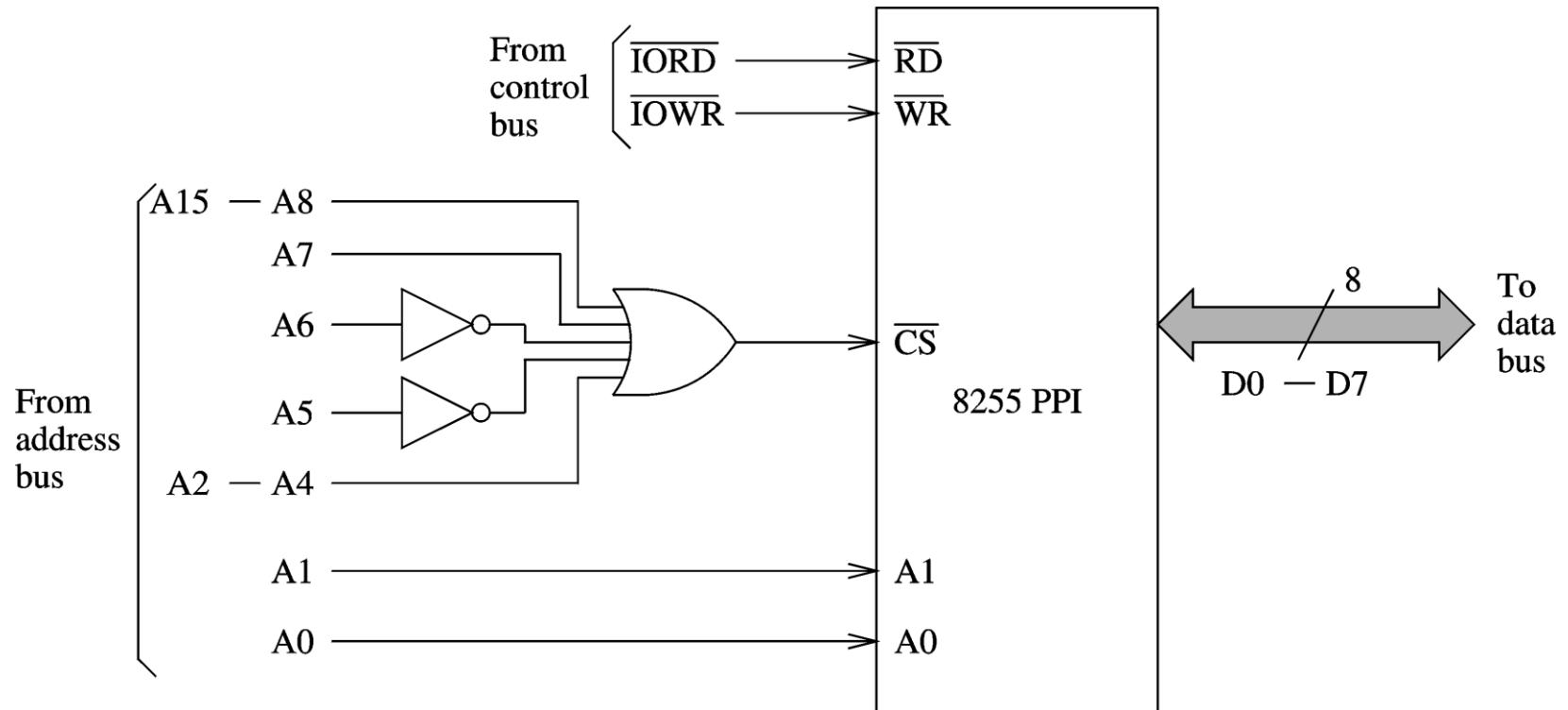
An Example I/O Device (cont'd)

- 8255 PPI has three 8-bit registers
 - » Port A (PA)
 - » Port B (PB)
 - » Port C (PC)
- * These ports are mapped as follows

8255 register	Port address
PA (input port)	60H
PB (output port)	61H
PC (input port)	62H
Command register	63H

An Example I/O Device (cont'd)

Mapping of 8255 I/O ports



An Example I/O Device (cont'd)

- Mapping I/O ports is similar to mapping memory
 - * Partial mapping
 - * Full mapping
- Keyboard scan code and status can be read from port 60H
 - * 7-bit scan code is available from
 - » PA0 – PA6
 - * Key status is available from PA7
 - » PA7 = 0 – key depressed
 - » PA0 = 1 – key released

I/O Data Transfer

- Data transfer involves two phases
 - * A data transfer phase
 - » It can be done either by
 - Programmed I/O
 - DMA
 - * An end-notification phase
 - » Programmed I/O
 - » Interrupt
 - Three basic techniques
 - * Programmed I/O
 - * DMA
 - * Interrupt-driven I/O
-

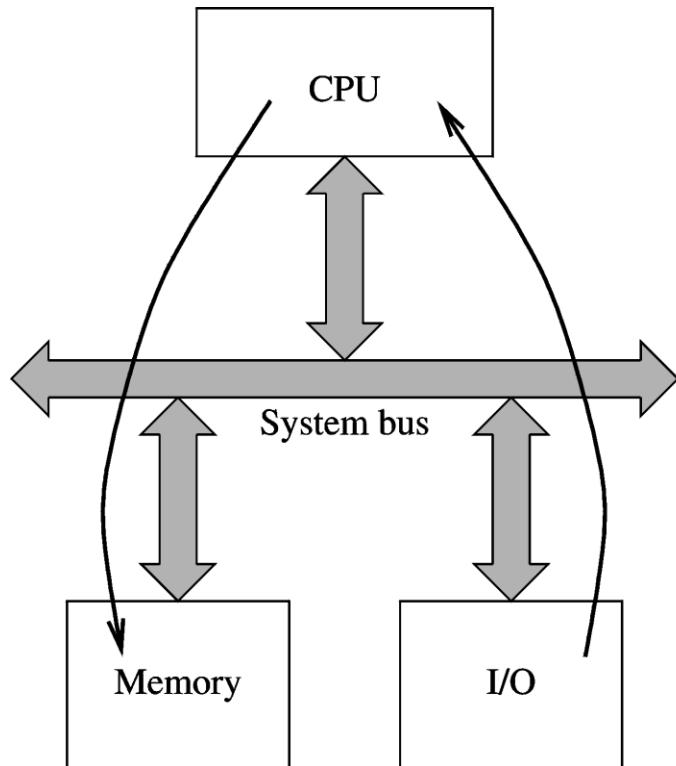
I/O Data Transfer (cont'd)

- Programmed I/O
 - * Done by busy-waiting
 - » This process is called *polling*
- Example
 - * Reading a key from the keyboard involves
 - » Waiting for PA7 bit to go low
 - Indicates that a key is pressed
 - » Reading the key scan code
 - » Translating it to the ASCII value
 - » Waiting until the key is released

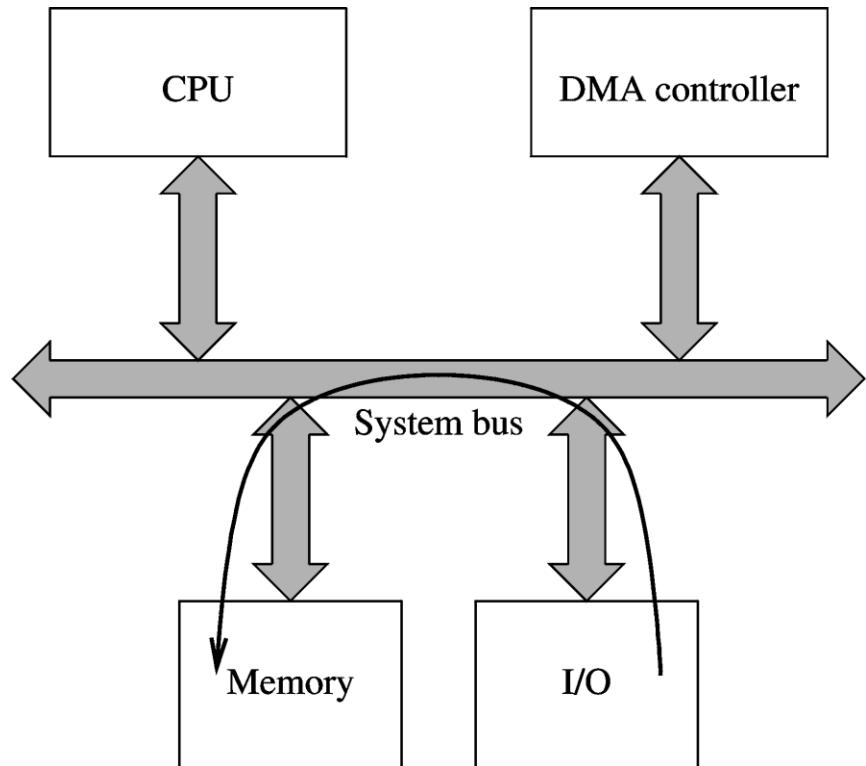
I/O Data Transfer (cont'd)

- Direct memory access (DMA)
 - * Problems with programmed I/O
 - » Processor wastes time polling
 - In our example
 - Waiting for a key to be pressed,
 - Waiting for it to be released
 - » May not satisfy timing constraints associated with some devices
 - Disk read or write
 - * DMA
 - » Frees the processor of the data transfer responsibility

I/O Data Transfer (cont'd)



(a) Programmed I/O transfer



(b) DMA transfer

I/O Data Transfer (cont'd)

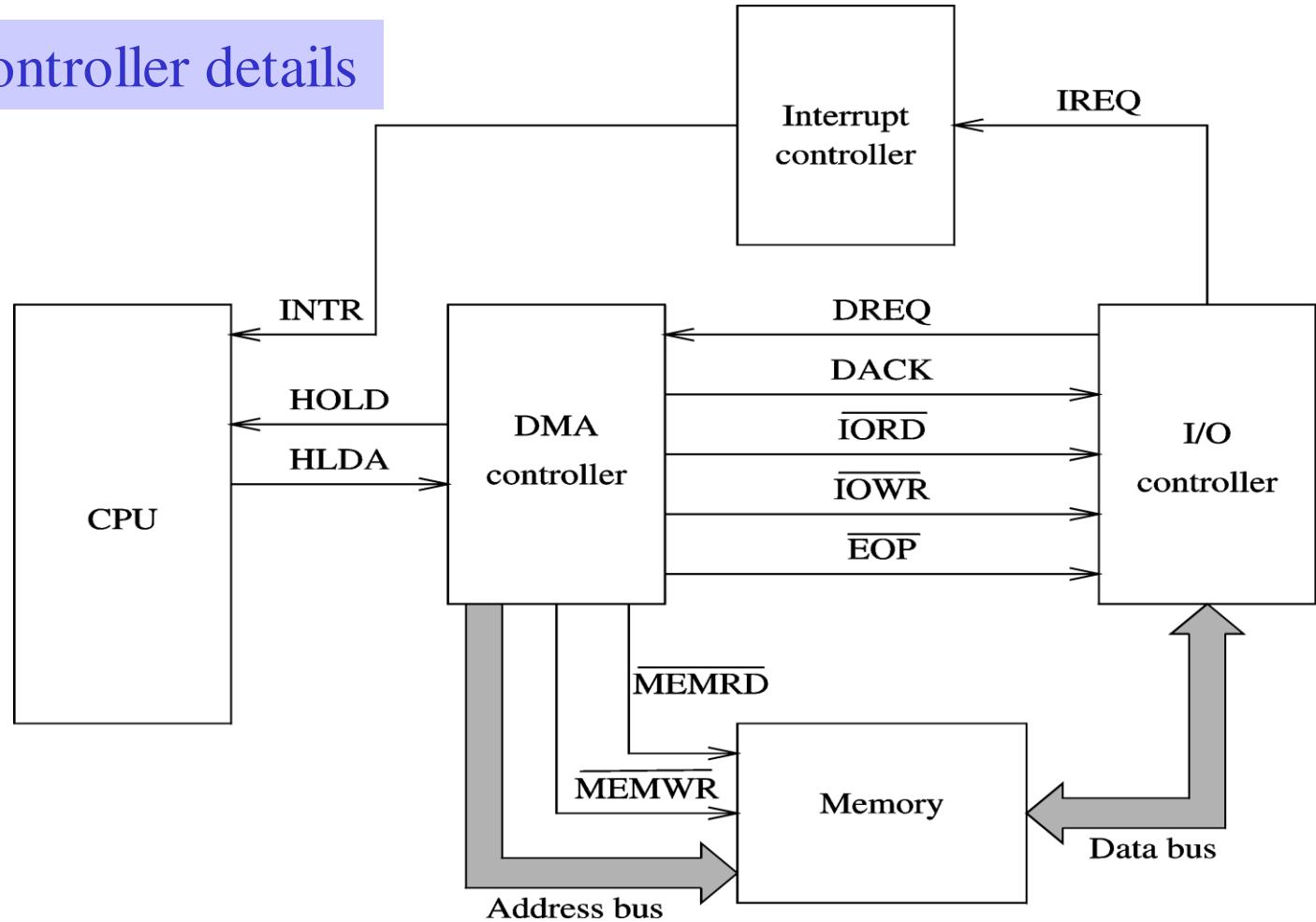
- DMA is implemented using a DMA controller
 - * DMA controller
 - » Acts as slave to processor
 - » Receives instructions from processor
 - » Example: Reading from an I/O device
 - Processor gives details to the DMA controller
 - I/O device number
 - Main memory buffer address
 - Number of bytes to transfer
 - Direction of transfer (memory → I/O device, or vice versa)

I/O Data Transfer (cont'd)

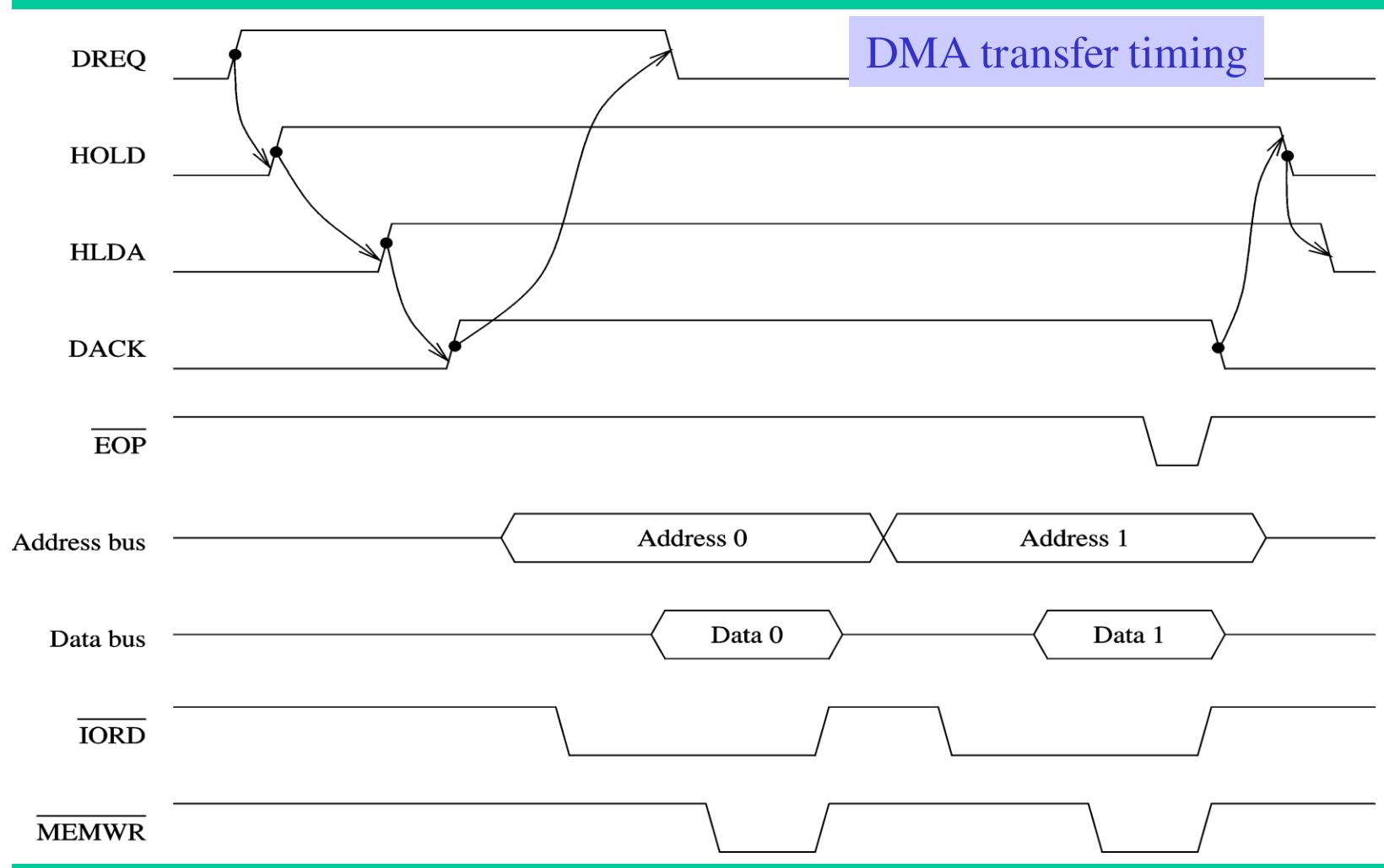
- Steps in a DMA operation
 - * Processor initiates the DMA controller
 - » Gives device number, memory buffer pointer, ...
 - Called ***channel initialization***
 - » Once initialized, it is ready for data transfer
 - * When ready, I/O device informs the DMA controller
 - » DMA controller starts the data transfer process
 - Obtains bus by going through bus arbitration
 - Places memory address and appropriate control signals
 - Completes transfer and releases the bus
 - Updates memory address and count value
 - If more to read, loops back to repeat the process
 - * Notify the processor when done
 - » Typically uses an interrupt

I/O Data Transfer (cont'd)

DMA controller details

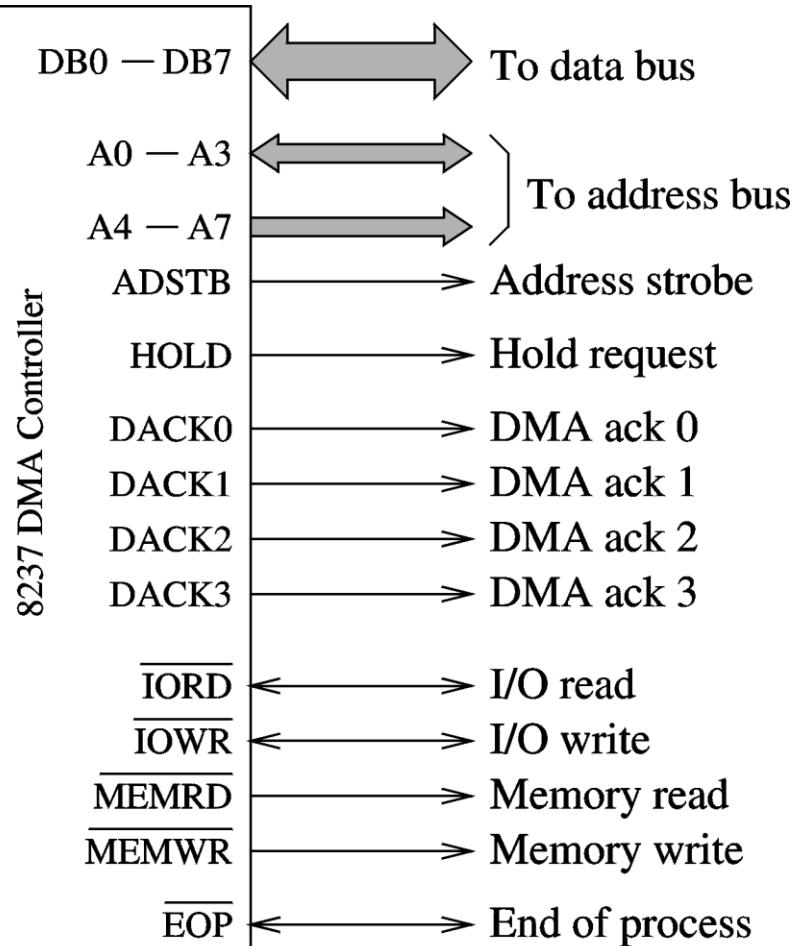
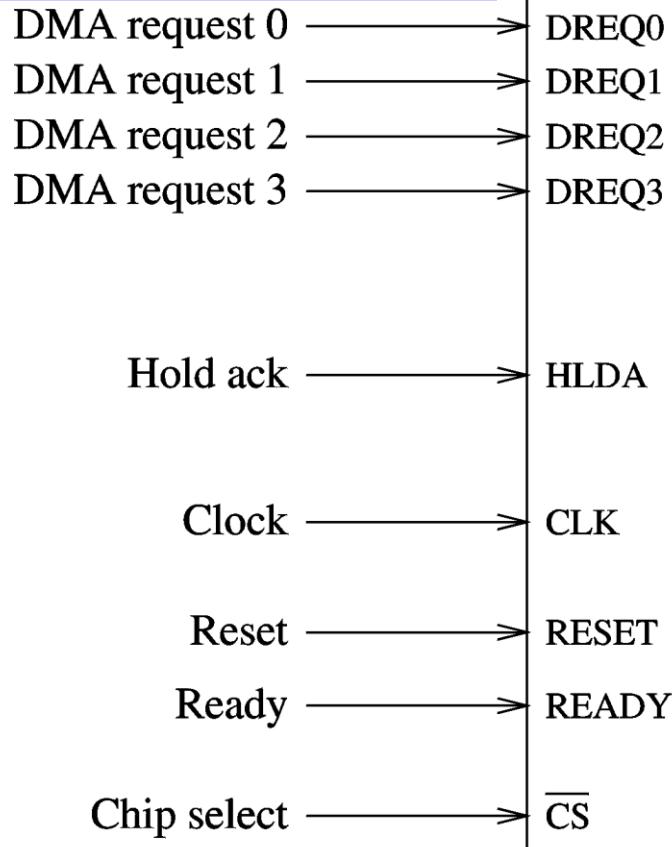


I/O Data Transfer (cont'd)



I/O Data Transfer (cont'd)

8237 DMA controller



I/O Data Transfer (cont'd)

- 8237 supports four DMA channels
- It has the following internal registers
 - * Current address register
 - » One 16-bit register for each channel
 - » Holds address for the current DMA transfer
 - * Current word register
 - » Keeps the byte count
 - » Generates terminal count (TC) signal when the count goes from zero to FFFFH
 - * Command register
 - » Used to program 8257 (type of priority, ...)

I/O Data Transfer (cont'd)

- * Mode register
 - » Each channel can be programmed to
 - Read or write
 - Autoincrement or autodecrement the address
 - Autoinitiate the channel
- * Request register
 - » For software-initiated DMA
- * Mask register
 - » Used to disable a specific channel
- * Status register
- * Temporary register
 - » Used for memory-to-memory transfers

I/O Data Transfer (cont'd)

- 8237 supports four types of data transfer
 - * Single cycle transfer
 - » Only single transfer takes place
 - » Useful for slow devices
 - * Block transfer mode
 - » Transfers data until TC is generated or external EOP signal is received
 - * Demand transfer mode
 - » Similar to the block transfer mode
 - » In addition to TC and EOP, transfer can be terminated by deactivating DREQ signal
 - * Cascade mode
 - » Useful to expand the number channels beyond four

External Interface

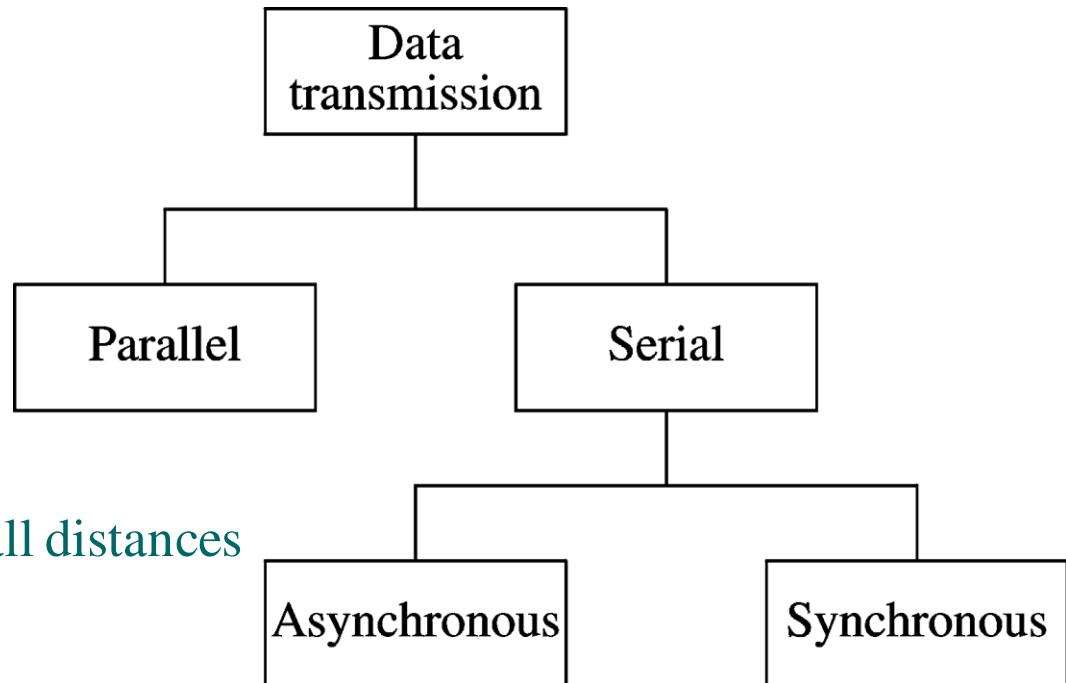
- Two ways of interfacing I/O devices

- * Serial

- » Cheaper
 - » Slower

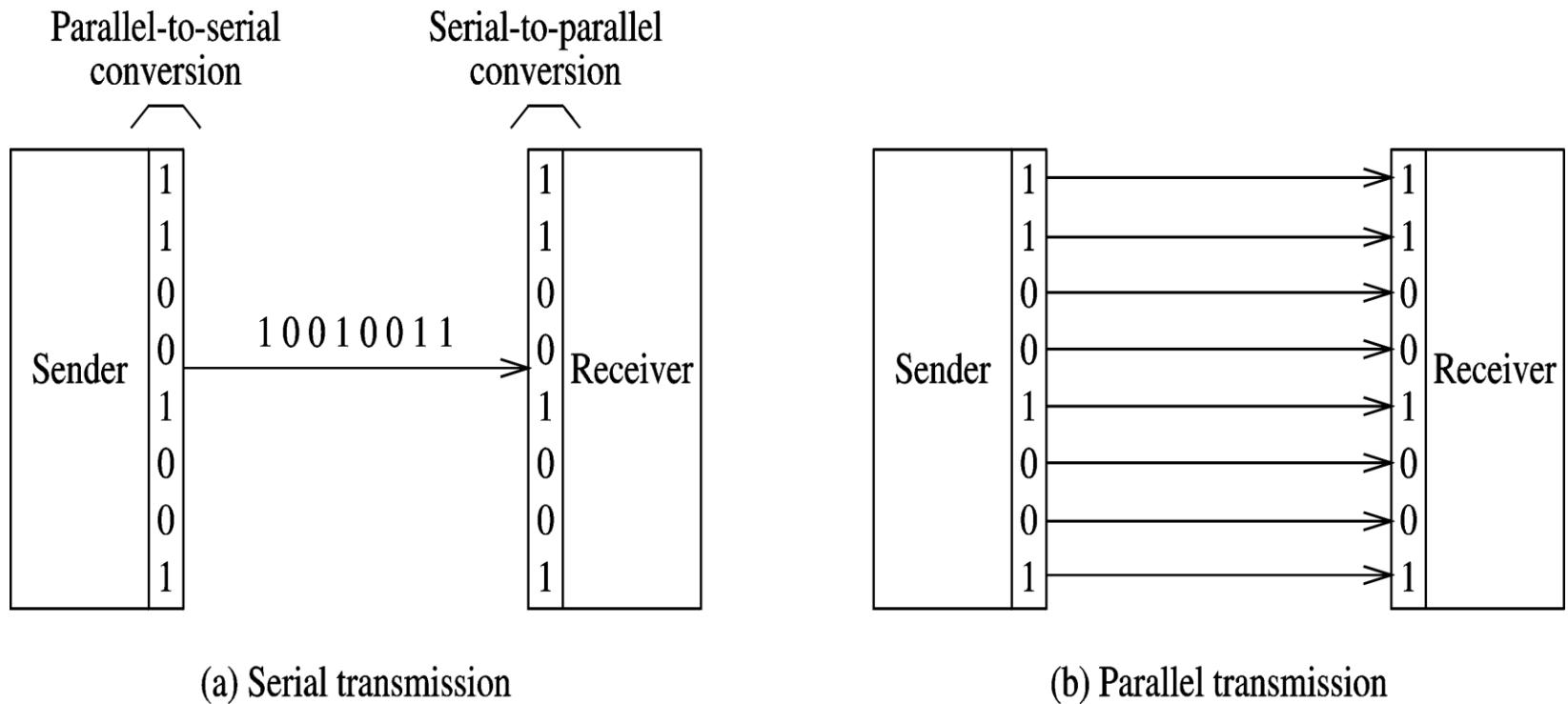
- * Parallel

- » Faster
 - » Data skew
 - » Limited to small distances



External Interface (cont'd)

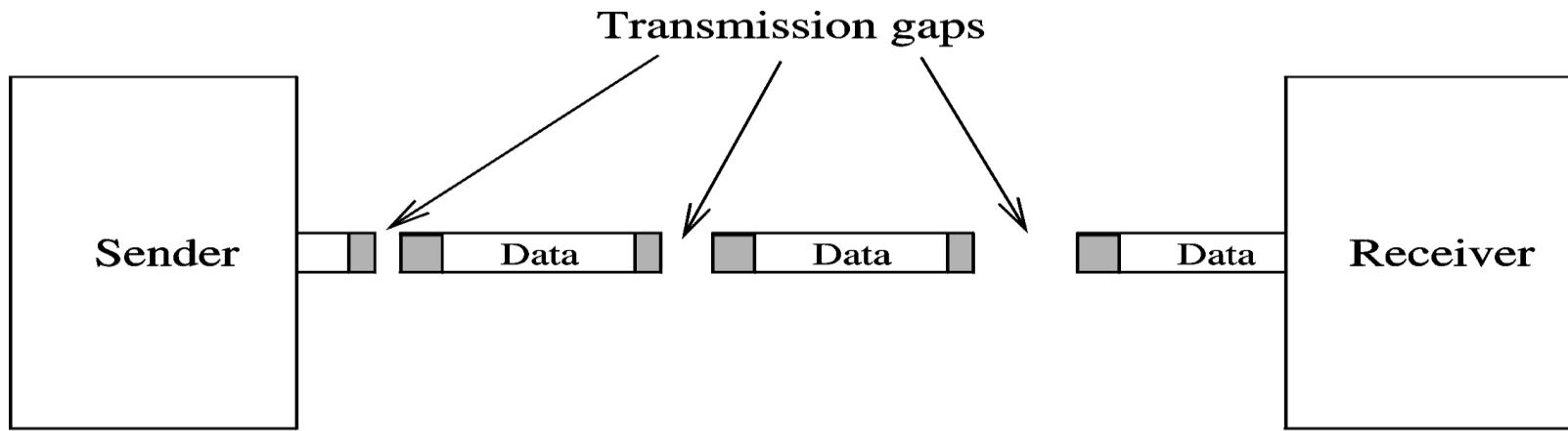
Two basic modes of data transmission



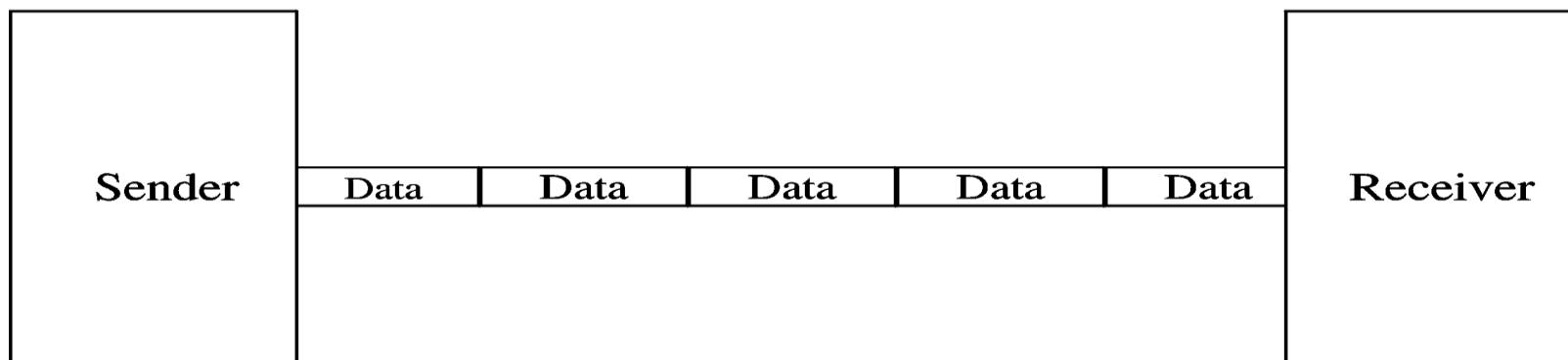
External Interface (cont'd)

- Serial transmission
 - * Asynchronous
 - » Each byte is encoded for transmission
 - Start and stop bits
 - » No need for sender and receiver synchronization
 - * Synchronous
 - » Sender and receiver must synchronize
 - Done in hardware using phase locked loops (PLLs)
 - » Block of data can be sent
 - » More efficient
 - Less overhead than asynchronous transmission
 - » Expensive

External Interface (cont'd)

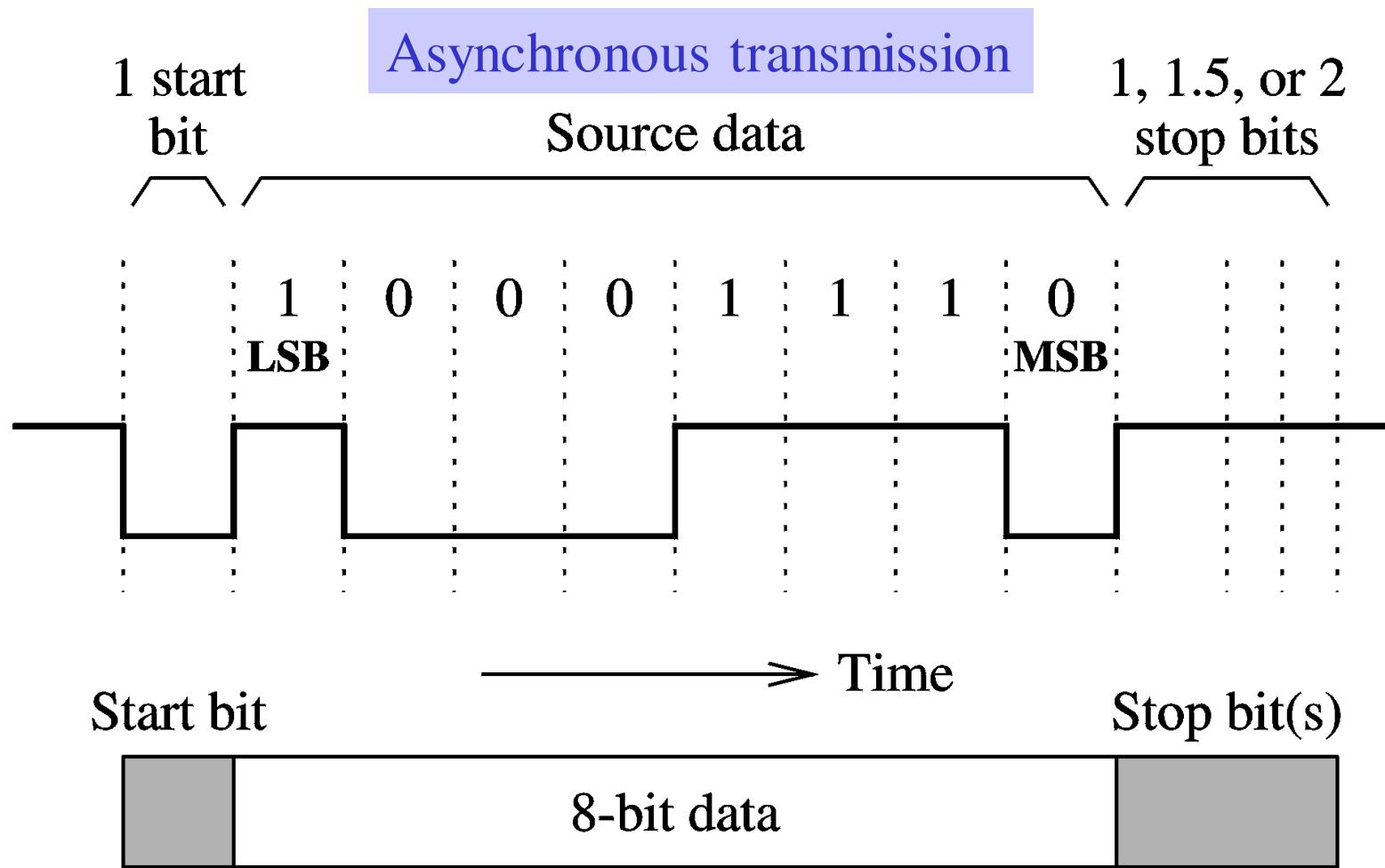


(a) Asynchronous transmission



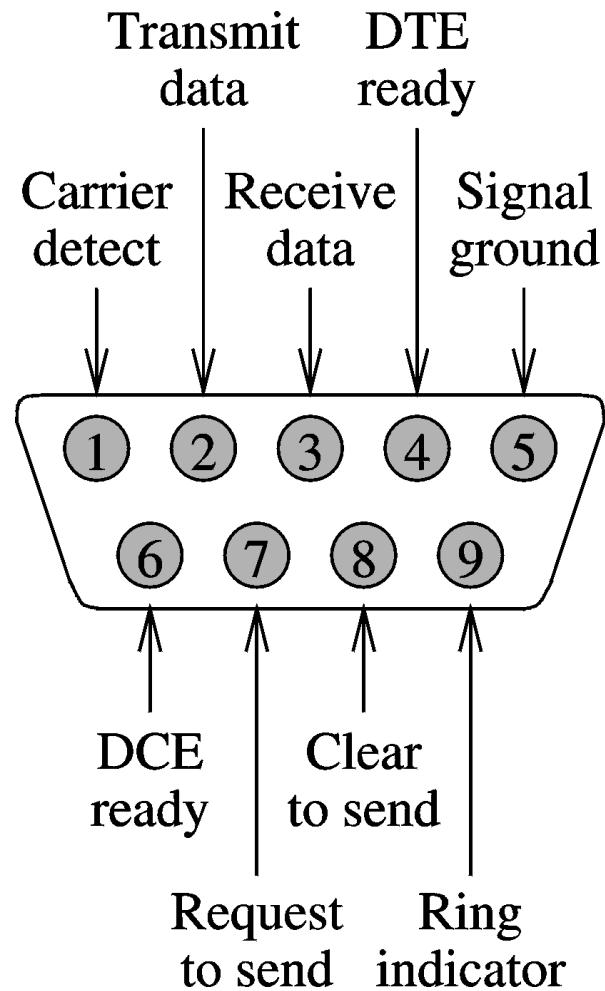
(b) Synchronous transmission

External Interface (cont'd)



External Interface (cont'd)

- EIA-232 serial interface
 - * Low-speed serial transmission
 - * Adopted by Electronics Industry Association (EIA)
 - » Popularly known by its predecessor RS-232
 - * It uses a 9-pin connector DB-9
 - » Uses 8 signals
 - * Typically used to connect a modem to a computer



External Interface (cont'd)

- Transmission protocol uses three phases
 - * Connection setup
 - » Computer A asserts DTE Ready
 - Transmits phone# via Transmit Data line (pin 2)
 - » Modem B alerts its computer via Ring Indicator (pin 9)
 - Computer B asserts DTE Ready (pin 4)
 - Modem B generates carrier and turns its DCE Ready
 - » Modem A detects the carrier signal from modem B
 - Modem A alters its computer via Carrier Detect (pin 1)
 - Turns its DCE Ready
 - * Data transmission
 - » Done by handshaking using
 - request-to-send (RTS) and clear-to-send (CTS) signals
 - * Connection termination
 - » Done by deactivating RTS

External Interface (cont'd)

- Parallel printer interface
 - * A simple parallel interface
 - * Uses 25-pin DB-25
 - » 8 data signals
 - Latched by strobe (pin 1)
 - » Data transfer uses simple handshaking
 - Uses acknowledge (CK) signal
 - After each byte, computer waits for ACK
 - » 5 lines for printer status
 - Busy, out-of-paper, online/offline, autofeed, and fault
 - » Can be initialized with INIT
 - Clears the printer buffer and resets the printer

External Interface (cont'd)

Table 19.3 Parallel printer interface signals

Pin #	Signal	Signal direction	Signal function
1	STROBE	PC \Rightarrow printer	Clock used to latch data
2	Data 0	PC \Rightarrow printer	Data bit 0 (LSB)
3	Data 1	PC \Rightarrow printer	Data bit 1
4	Data 2	PC \Rightarrow printer	Data bit 2
5	Data 3	PC \Rightarrow printer	Data bit 3
6	Data 4	PC \Rightarrow printer	Data bit 4
7	Data 5	PC \Rightarrow printer	Data bit 5
8	Data 6	PC \Rightarrow printer	Data bit 6
9	Data 7	PC \Rightarrow printer	Data bit 7 (MSB)
10	ACK	printer \Rightarrow PC	Printer acknowledges receipt of data
11	BUSY	printer \Rightarrow PC	Printer is busy
12	POUT	printer \Rightarrow PC	Printer is out of paper
13	SEL	printer \Rightarrow PC	Printer is online
14	AUTO FEED	printer \Rightarrow PC	Autofeed is on
15	FAULT	printer \Rightarrow PC	Printer fault
16	INIT	PC \Rightarrow printer	Clears printer buffer and resets printer
17	SLCT IN	PC \Rightarrow printer	TTL high level
18–25	Ground	N/A	Ground reference

External Interface (cont'd)

- **SCSI**
 - * Pronounced “scuzzy”
 - * Small Computer System Interface
 - » Supports both internal and external connection
 - * Comes in two bus widths
 - » 8 bits
 - Known as *narrow SCSI*
 - Uses a 50-pin connector
 - Device id can range from 0 to 7
 - » 16 bits
 - Known as *wide SCSI*
 - Uses a 68-pin connector
 - Device id can range from 0 to 15

External Interface (cont'd)

Table 19.4 Types of SCSI

SCSI type	Bus width (bits)	Transfer rate MB/s
SCSI 1	8	5
Fast SCSI	8	10
Ultra SCSI	8	20
Ultra 2 SCSI	8	40
Wide Ultra SCSI	16	40
Wide Ultra 2 SCSI	16	80
Ultra 3 (Ultra 160) SCSI	16	160
Ultra 4 (Ultra 320) SCSI	16	320

External Interface (cont'd)

Table 19.5 Narrow SCSI signals

Description	Signal	Pin	Pin	Signal	Description
Twisted pair ground	GND	1	26	D0	Data 0
Twisted pair ground	GND	2	27	D1	Data 1
Twisted pair ground	GND	3	28	D2	Data 2
Twisted pair ground	GND	4	29	D3	Data 3
Twisted pair ground	GND	5	30	D4	Data 4
Twisted pair ground	GND	6	31	D5	Data 5
Twisted pair ground	GND	7	32	D6	Data 6
Twisted pair ground	GND	8	33	D7	Data 7
Twisted pair ground	GND	9	34	DP	Data parity bit
Ground	GND	10	35	GND	Ground
Ground	GND	11	36	GND	Ground
Reserved		12	37		Reserved
No connection		13	38	TermPwr	Termination power (+5 V) cont'd

External Interface (cont'd)

Reserved		14	39		Reserved
Ground	GND	15	40	GND	Ground
Twisted pair ground	GND	16	41	ATN	Attention
Ground	GND	17	42	GND	Ground
Twisted pair ground	GND	18	43	BSY	Busy
Twisted pair ground	GND	19	44	ACK	Acknowledge
Twisted pair ground	GND	20	45	RST	Reset
Twisted pair ground	GND	21	46	MSG	Message
Twisted pair ground	GND	22	47	SEL	Selection
Twisted pair ground	GND	23	48	C/D	Command/data
Twisted pair ground	GND	24	49	REQ	Request
Twisted pair ground	GND	25	50	I/O	Input/output

External Interface (cont'd)

- SCSI uses client-server model
 - * Uses terms *initiator* and *target* for client and server
 - » Initiator issues commands to targets to perform a task
 - Initiators are typically SCSI host adaptors
 - » Targets receive the command and perform the task
 - Targets are SCSI devices like disk drives
- SCSI transfer proceeds in phases
 - * Command
 - * Message in
 - * Message out
 - * Data in
 - * Data out
 - * Status

IN and OUT from
the initiator point
of view

External Interface (cont'd)

- * SCSI uses asynchronous mode for all bus negotiations
 - » Uses handshaking using REQ and ACK signals for each byte of data
- * On a synchronous SCSI
 - » Data are transferred synchronously
 - » REQ-ACK signals are not used for each byte
 - » A number of bytes (e.g., 8) can be sent without waiting for ACK
 - Improves throughput
 - Minimizes adverse impact of cable propagation delay

USB

- Universal Serial Bus
 - * Originally developed in 1995 by a consortium including
 - » Compaq, HP, Intel, Lucent, Microsoft, and Philips
 - * USB 1.1 supports
 - » Low-speed devices (1.5 Mbps)
 - » Full-speed devices (12 Mbps)
 - * USB 2.0 supports
 - » High-speed devices
 - Up to 480 Mbps (a factor of 40 over USB 1.1)
 - » Uses the same connectors
 - Transmission speed is negotiated on device-by-device basis

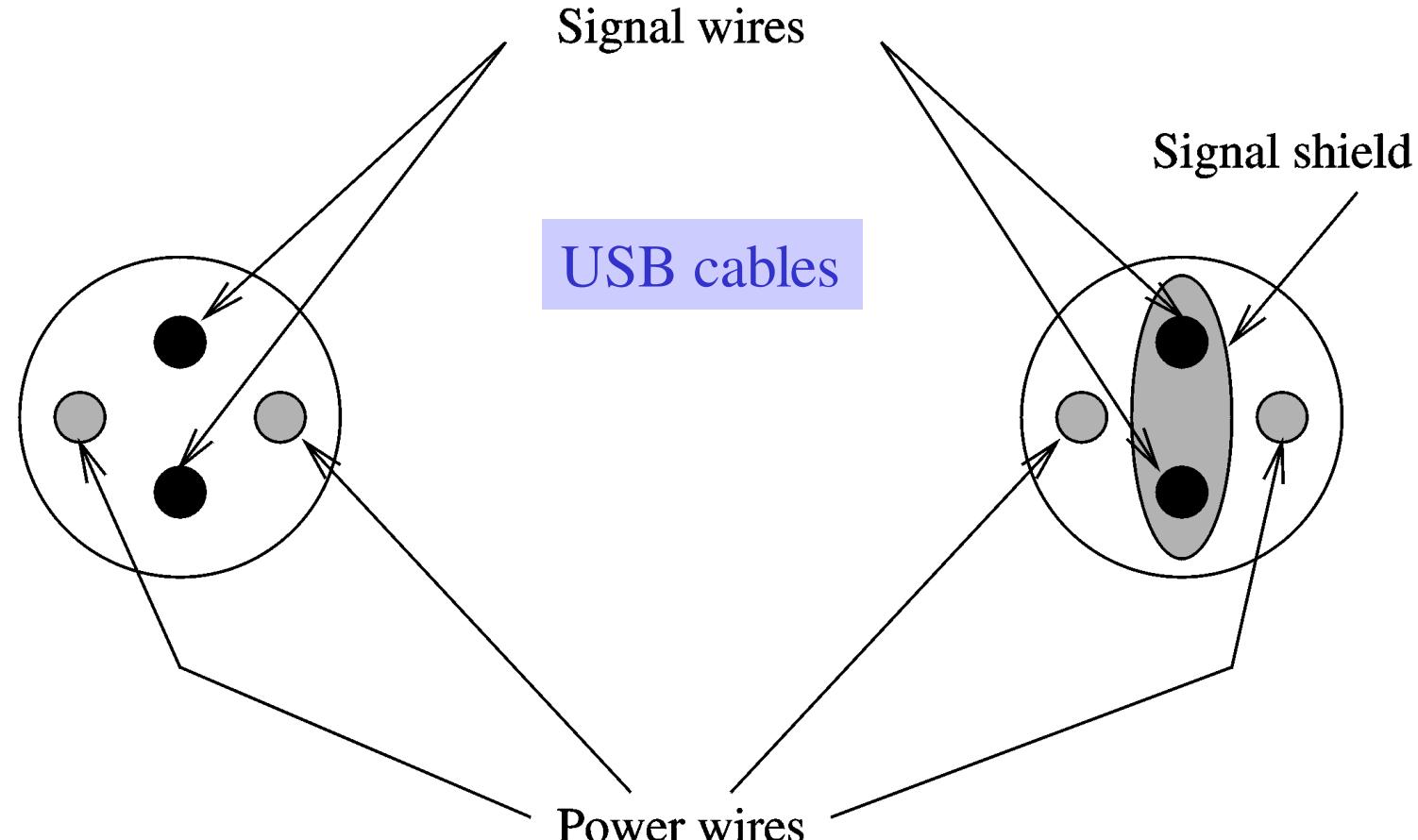
USB (cont'd)

- Motivation for USB
 - * Avoid device-specific interfaces
 - » Eliminates multitude of interfaces
 - PS/2, serial, parallel, monitor, microphone, keyboard,...
 - * Avoid non-shareable interfaces
 - » Standard interfaces support only one device
 - * Avoid I/O address space and IRQ problems
 - » USB does not require memory or address space
 - * Avoid installation and configuration problems
 - » Don't have to open the box to install and configure jumpers
 - * Allow hot attachment of devices

USB (cont'd)

- Additional advantages of USB
 - * Power distribution
 - » Simple devices can be bus-powered
 - Examples: mouse, keyboards, floppy disk drives, wireless LANs, ...
 - * Control peripherals
 - » Possible because USB allows data to flow in both directions
 - * Expandable through hubs
 - * Power conservation
 - » Enters suspend state if there is no activity for 3 ms
 - * Error detection and recovery
 - » Uses CRC

USB (cont'd)

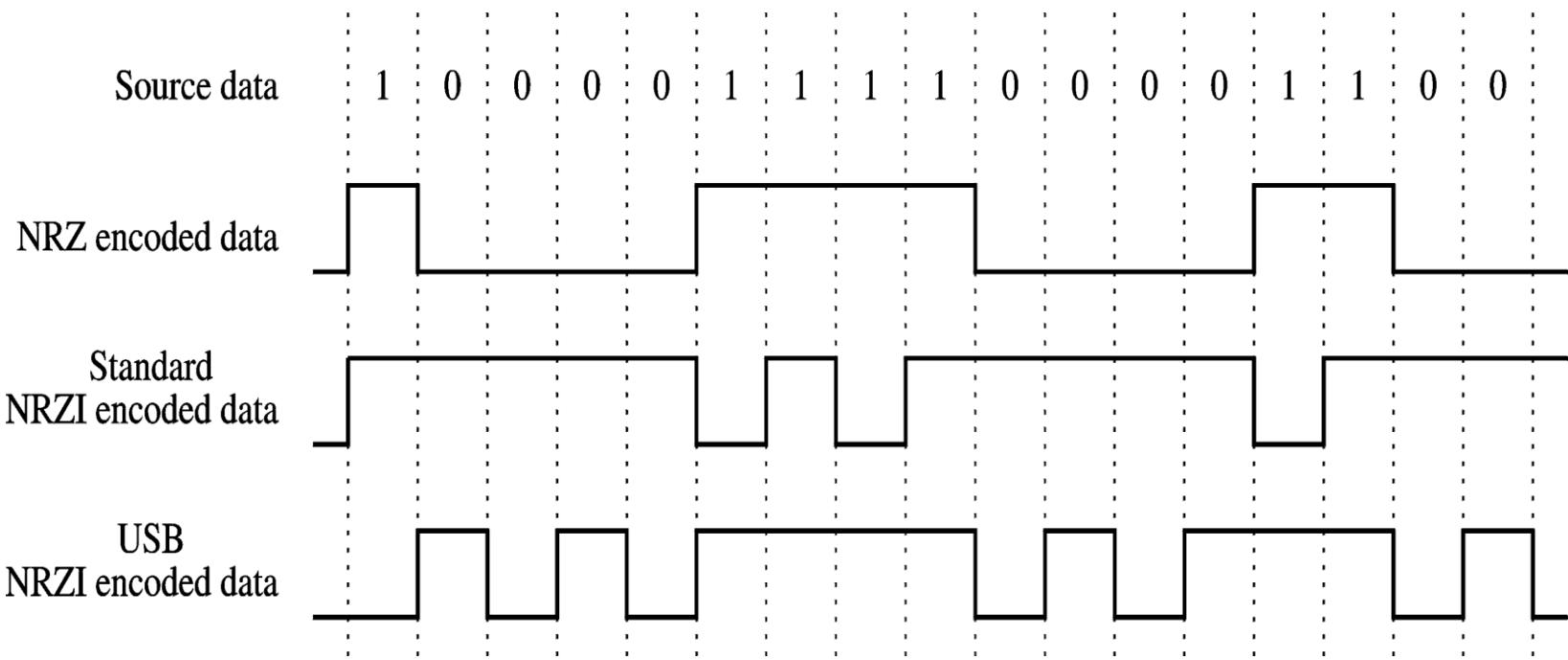


(a) Low-speed cable

(b) Full-speed/High-speed cable

USB (cont'd)

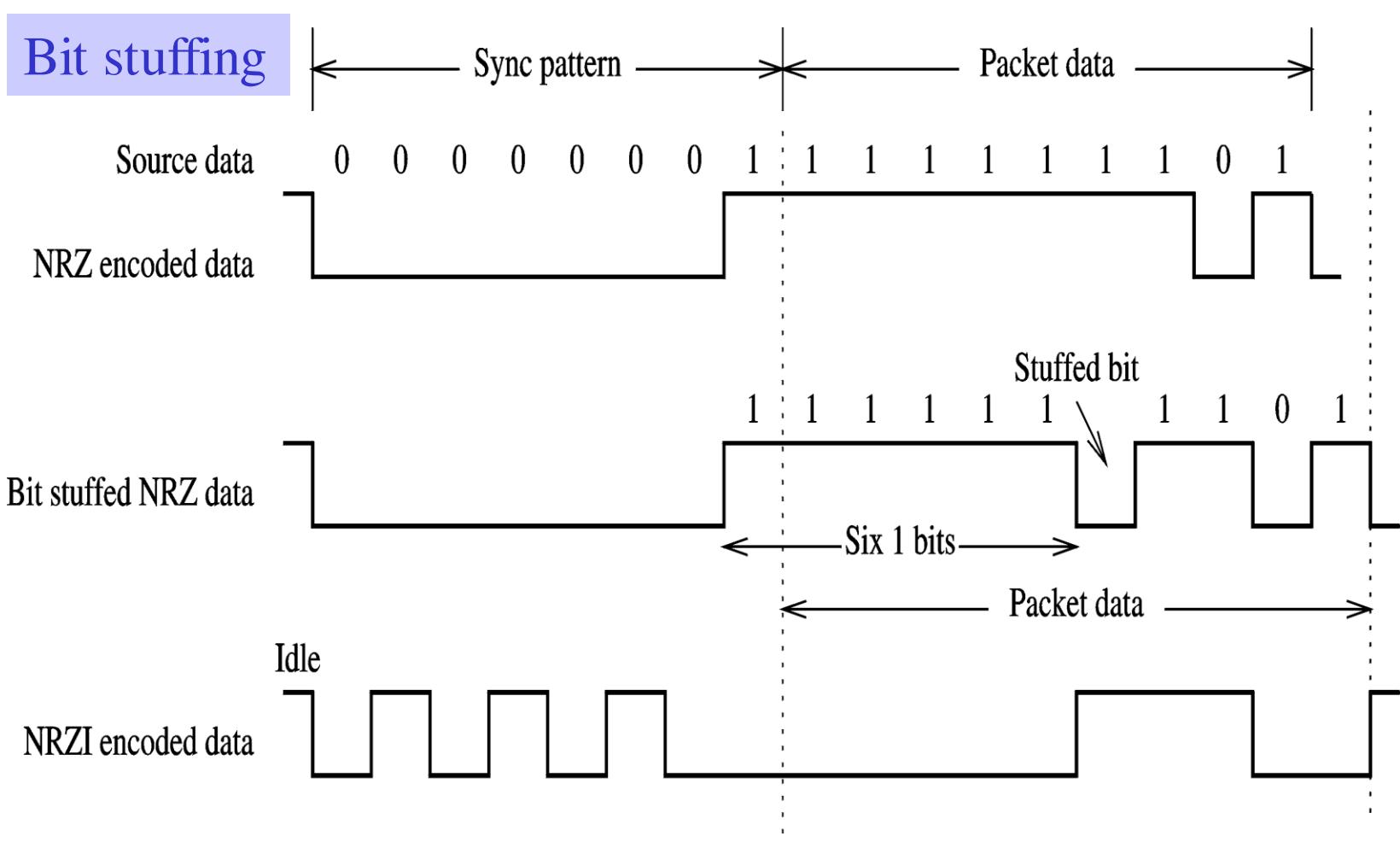
- USB encoding
 - * Uses NRZI encoding
 - » Non-Return to Zero-Inverted



USB (cont'd)

- NRZI encoding
 - * A signal transition occurs if the next bit is zero
 - » It is called *differential encoding*
 - * Two desirable properties
 - » Signal transitions, not levels, need to be detected
 - » Long string of zeros causes signal changes
 - * Still a problem
 - » Long strings of 1s do not cause signal change
 - * To solve this problem
 - » Uses *bit stuffing*
 - A zero is inserted after every six consecutive 1s

USB (cont'd)



USB (cont'd)

- Transfer types
 - » Four types of transfer
 - * Interrupt transfer
 - » Uses polling
 - Polling interval can range from 1 ms to 255 ms
 - * Isochronous transfer
 - » Used in real-time applications that require constant data transfer rate
 - Example: Reading audio from CD-ROM
 - » These transfers are scheduled regularly
 - » Do not use error detection and recovery

USB (cont'd)

- * Control transfer

- » Used to configure and set up USB devices
- » Three phases
 - Setup stage
 - Conveys type of request made to target device
 - Data stage
 - Optional stage
 - Control transfers that require data use this stage
 - Status stage
 - Checks the status of the operation
- » Allocates a guaranteed bandwidth of 10%
- » Error detection and recovery are used
 - Recovery is by means of retries

USB (cont'd)

* Bulk transfer

- » For devices with no specific data transfer rate requirements
 - Example: sending data to a printer
- » Lowest priority bandwidth allocation
- » If the other three types of transfers take 100% of the bandwidth
 - Bulk transfers are deferred until load decreases
- » Error detection and recovery are used
 - Recovery is by means of retries

USB (cont'd)

- USB architecture
 - * USB host controller
 - » Initiates transactions over USB
 - * Root hub
 - » Provides connection points
 - * Two types of host controllers
 - » Open host controller (OHC)
 - Defined by Intel
 - » Universal host controller (UHC)
 - Specified by National Semiconductor, Microsoft, Compaq
 - » Difference between the two
 - How they schedule the four types of transfers

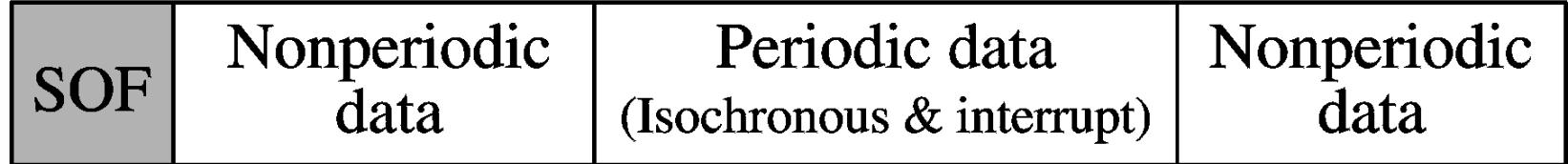
USB (cont'd)

- UHC scheduling
 - * Schedules periodic transfers first
 - » Periodic transfers: isochronous and interrupts
 - » Can take up to 90% of bandwidth
 - * These transfers are followed by control and bulk transfers
 - » Control transfers are guaranteed 10% of bandwidth
 - * Bulk transfers are scheduled only if there is bandwidth available

USB (cont'd)



(a) UHC scheduling



(b) OHC scheduling

USB (cont'd)

- OHC scheduling
 - * Different from UHC scheduling
 - * Reserves space for non-periodic transfers first
 - » Non-periodic transfers: control and bulk
 - » 10% bandwidth reserved
 - * Next periodic transfers are scheduled
 - » Guarantees 90% bandwidth
 - * Left over bandwidth is allocated to non-periodic transfers

USB (cont'd)

- Bus powered devices
 - * Low-power
 - » Less than 100 mA
 - » Can be bus-powered
 - * High-powered
 - » Between 100 mA and 500 mA
 - Full-powered ports can power these devices
 - » Can be designed to have their own power
 - » Operate in three modes
 - Configured (500 mA)
 - Unconfigured (100 mA)
 - Suspended (about 2.5 mA)

USB (cont'd)

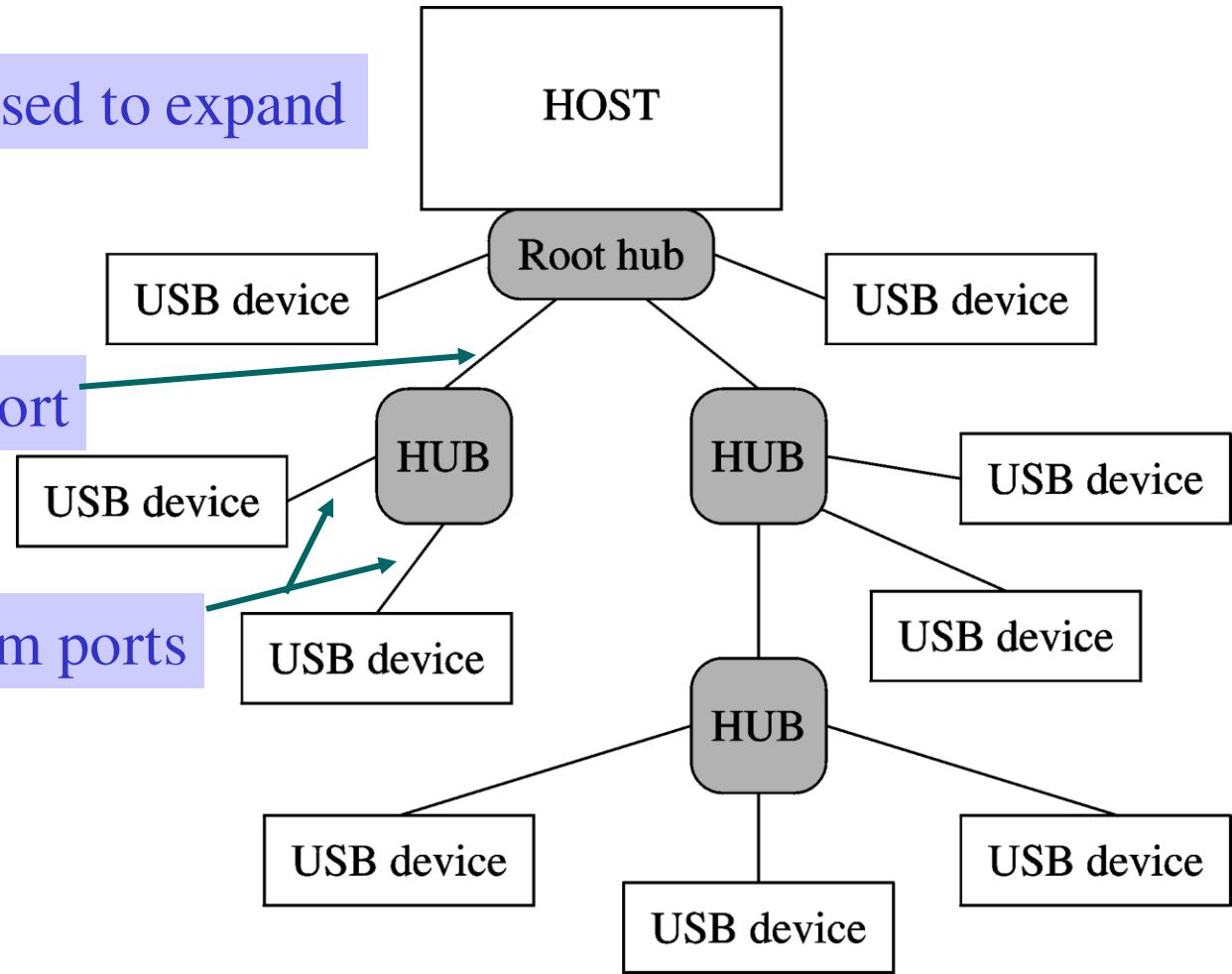
- USB hubs
 - * Bus-powered
 - » No extra power supply required
 - » Must be connected to an upstream port that can supply 500 mA
 - » Downstream ports can only supply 100 mA
 - Number of ports is limited to four
 - Support only low-powered devices
 - * Self-powered
 - » Support 4 high-powered devices
 - » Support 4 bus-powered USB hubs
 - * Most 4-port hubs are dual-powered

USB (cont'd)

Hubs can be used to expand

Upstream port

Downstream ports

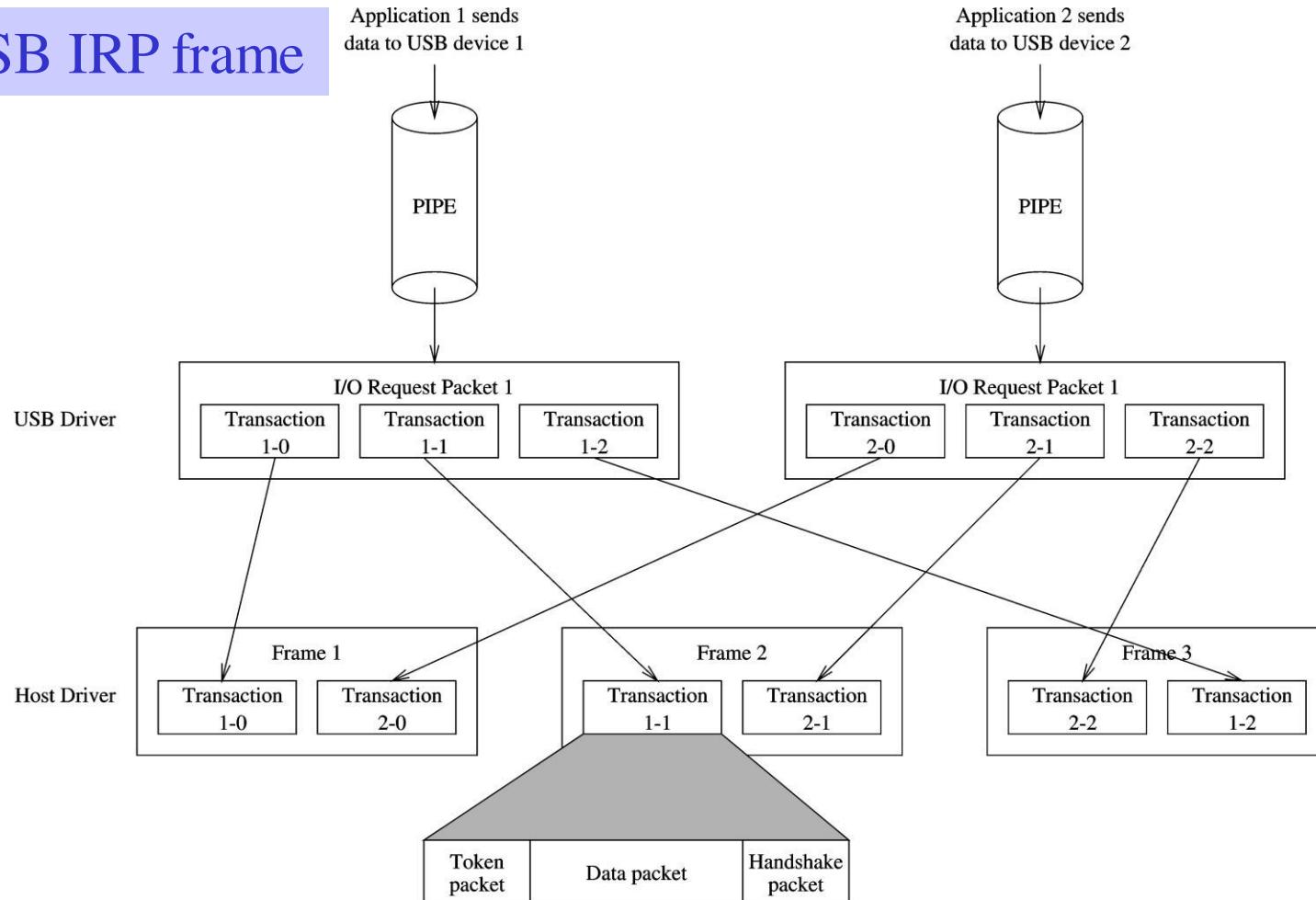


USB (cont'd)

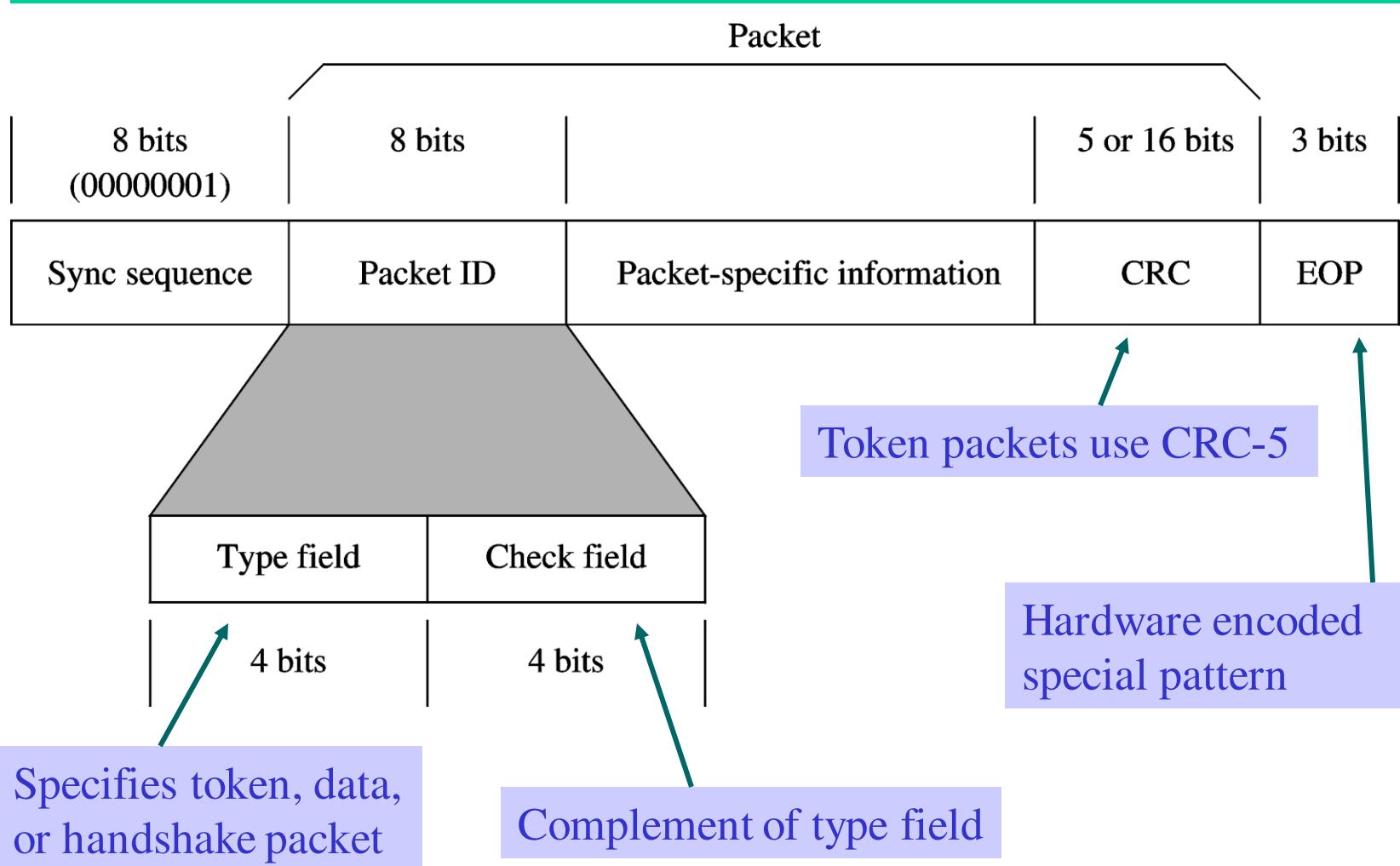
- USB transactions
 - * Transfers are done in one or more transactions
 - » Each transaction consists of several packets
 - * Transactions may have between 1 and 3 phases
 - » Token packet phase
 - Specifies transaction type and target device address
 - » Data packet phase (optional)
 - Maximum of 1023 bytes are transferred
 - » Handshake packet phase
 - Except for isochronous transfers, others use error detection for guaranteed delivery
 - Provides feedback on whether data has been received without error

USB (cont'd)

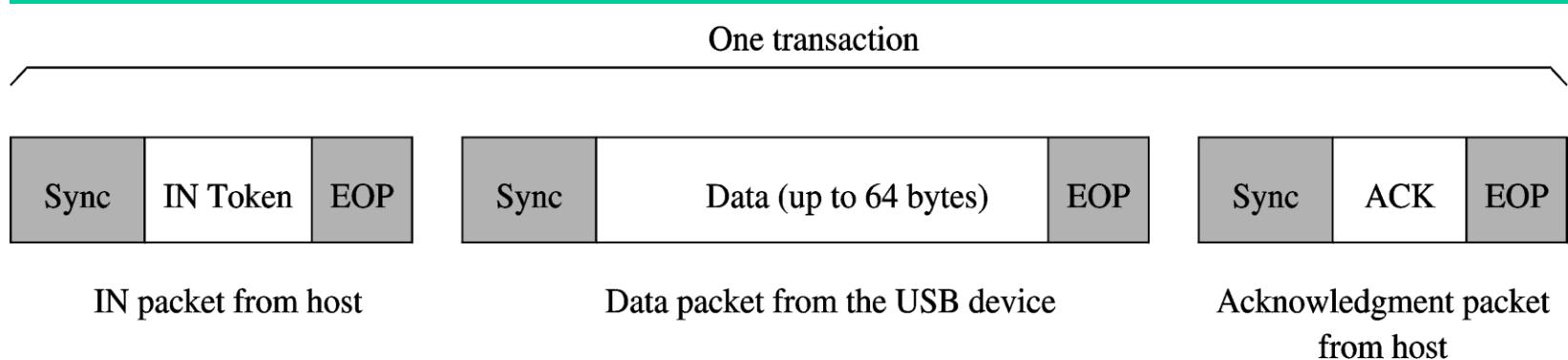
USB IRP frame



USB (cont'd)

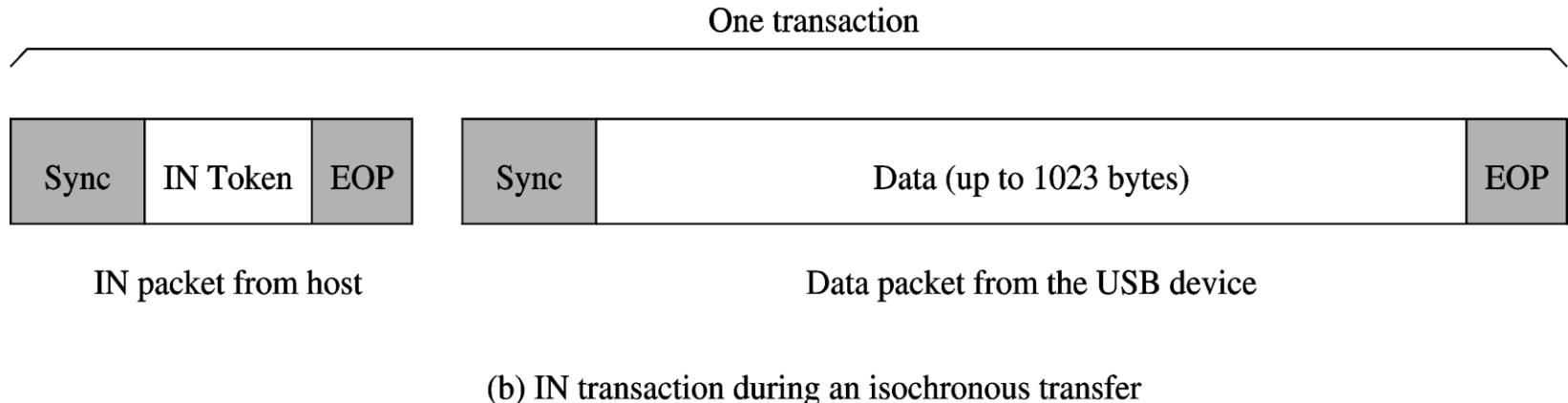


USB (cont'd)



USB 1.1 transactions

(a) IN transaction without errors



USB (cont'd)

- USB 2.0
 - * USB 1.1 uses 1 ms frames
 - * USB 2.0 uses 125 µs frames
 - » 1/8 of USB 1.1
 - * Supports 40X data rates
 - » Up to 480 Mbps
 - * Competitive with
 - » SCSI
 - » IEEE 1394 (FireWire)
 - * Widely available now

IEEE 1394

- Apple originally developed this standard for high-speed peripherals
 - * Known by a variety of names
 - » Apple: FireWire
 - » Sony: i.LINK
 - * IEEE standardized it as IEEE 1394
 - » First released in 1995 as IEEE 1394-1995
 - » A slightly revised version as 1394a
 - » Next version 1394b
 - * Shares many of the features of USB

IEEE 1394 (cont'd)

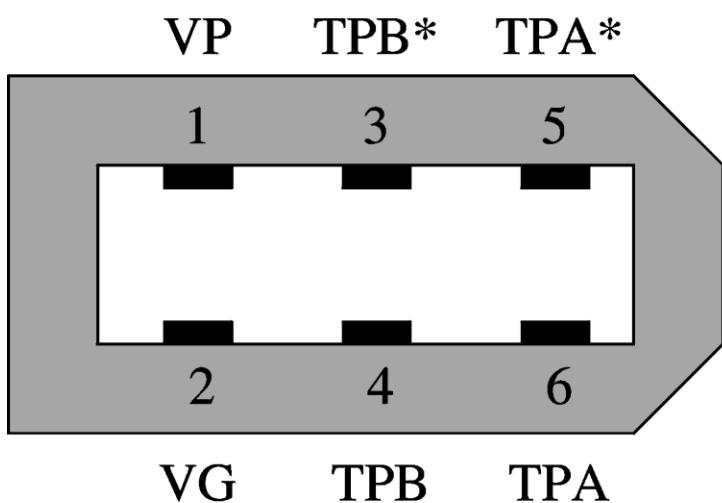
- Advantages
 - * High speed
 - » Supports three speeds
 - 100, 200, 400 Mbps
 - Competes with USB 2.0
 - Plans to boost it to 3.2 Gbps
 - * Hot attachment
 - » Like USB
 - » No need to shut down power to attach devices
 - * Peer-to-peer support
 - » USB is processor-centric
 - » Supports peer-to-peer communication without involving the processor
-

IEEE 1394 (cont'd)

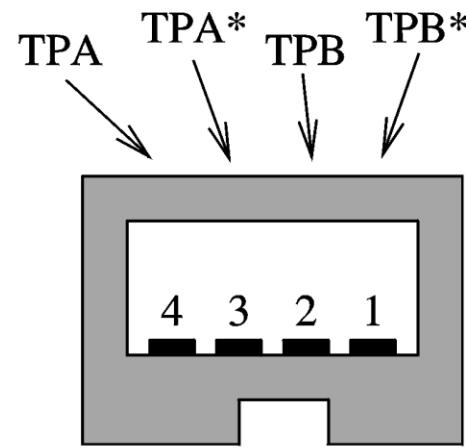
- * Expandable bus
 - » Devices can be connected in daisy-chain fashion
 - » Hubs can be used to expand
- * Power distribution
 - » Like the USB, cables distribute power
 - Much higher power than USB
 - Voltage between 8 and 33 V
 - Current can be up to 1.5 Amps
- * Error detection and recovery
 - » As in USB, uses CRC
 - » Uses retransmission in case of error
- * Long cables
 - » Like the USB

IEEE 1394 (cont'd)

IEEE 1394 6-pin and 4-pin connectors



(a) 6-pin connector

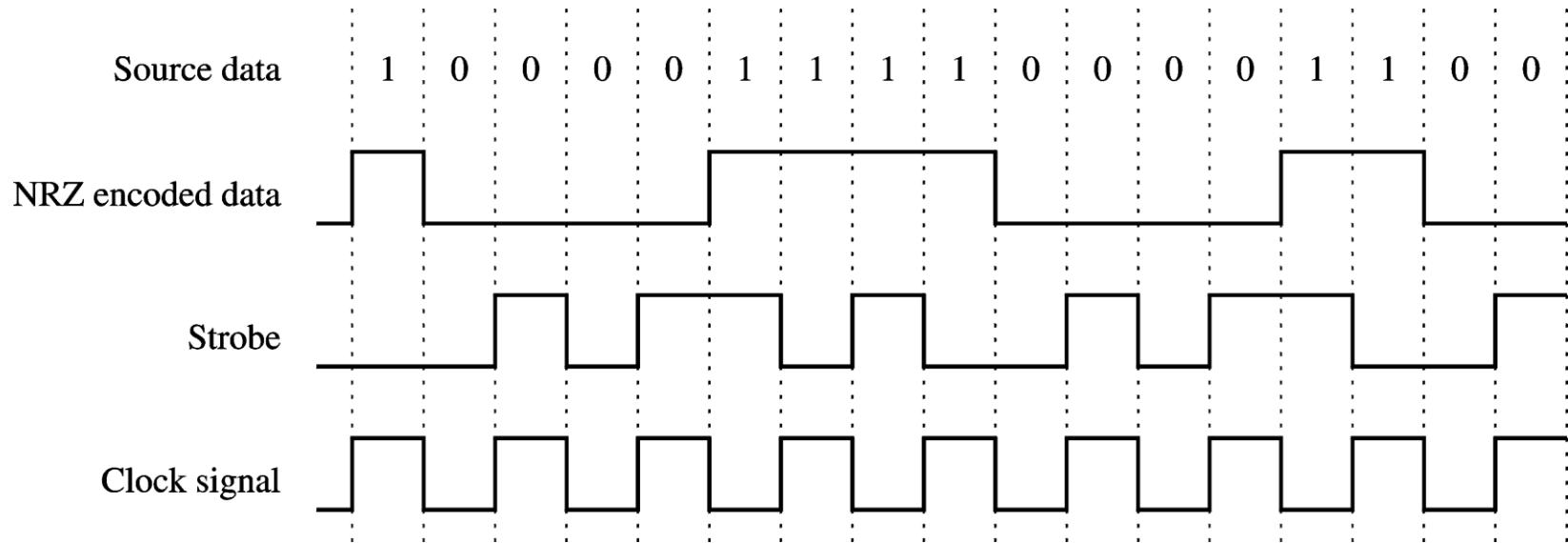


(b) 4-pin connector

4-pin connector does
not distribute power

IEEE 1394 (cont'd)

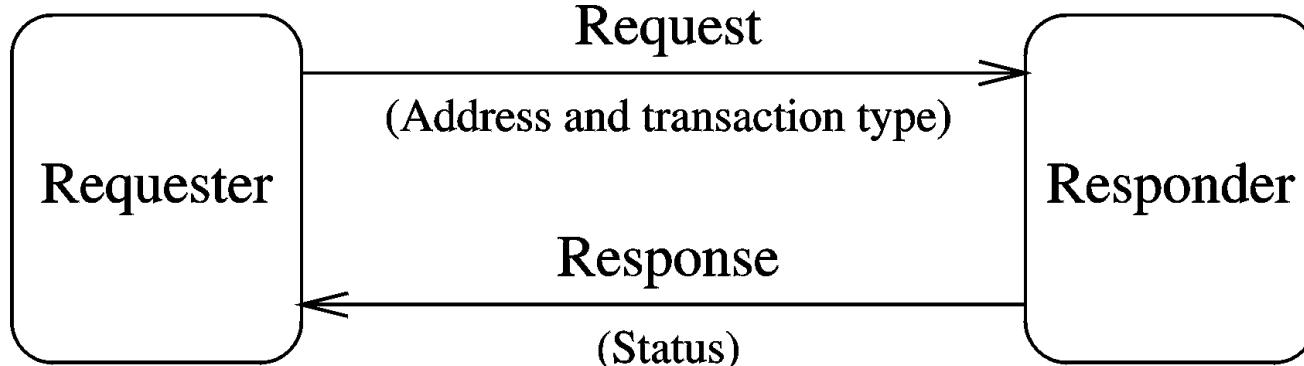
- Encoding
 - * Uses a simple NRZ encoding
 - * Strobe signal is encoded
 - » Changes the signal even if successive bits are the same



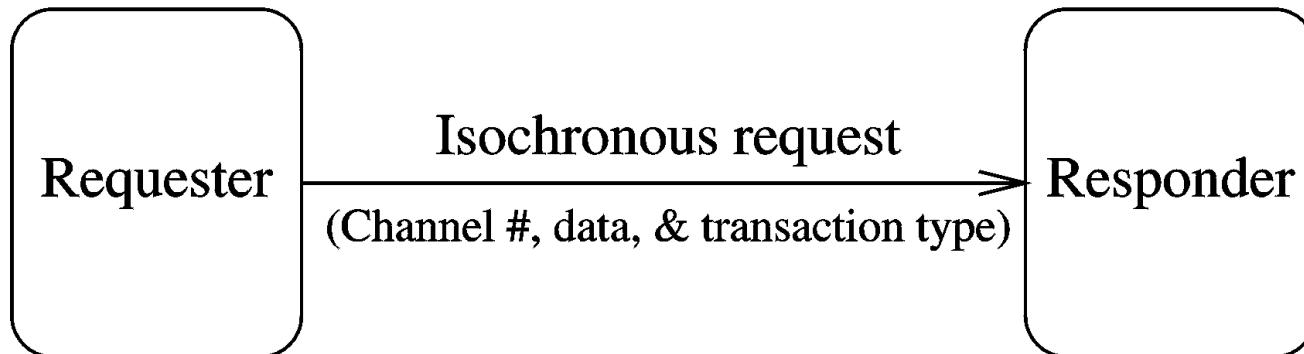
IEEE 1394 (cont'd)

- Transfer types
 - * Asynchronous
 - » For applications that require correct delivery of data
 - Example: writing a file to a disk drive
 - » Uses an acknowledgement to confirm delivery
 - » Guaranteed bandwidth of 20%
 - * Isochronous
 - » For real-time applications
 - » No acknowledgement
 - » Up to 80% of bandwidth allocated
 - * Bandwidth allocation on a cycle-by-cycle basis
 - » Cycle time: 125 µs

IEEE 1394 (cont'd)



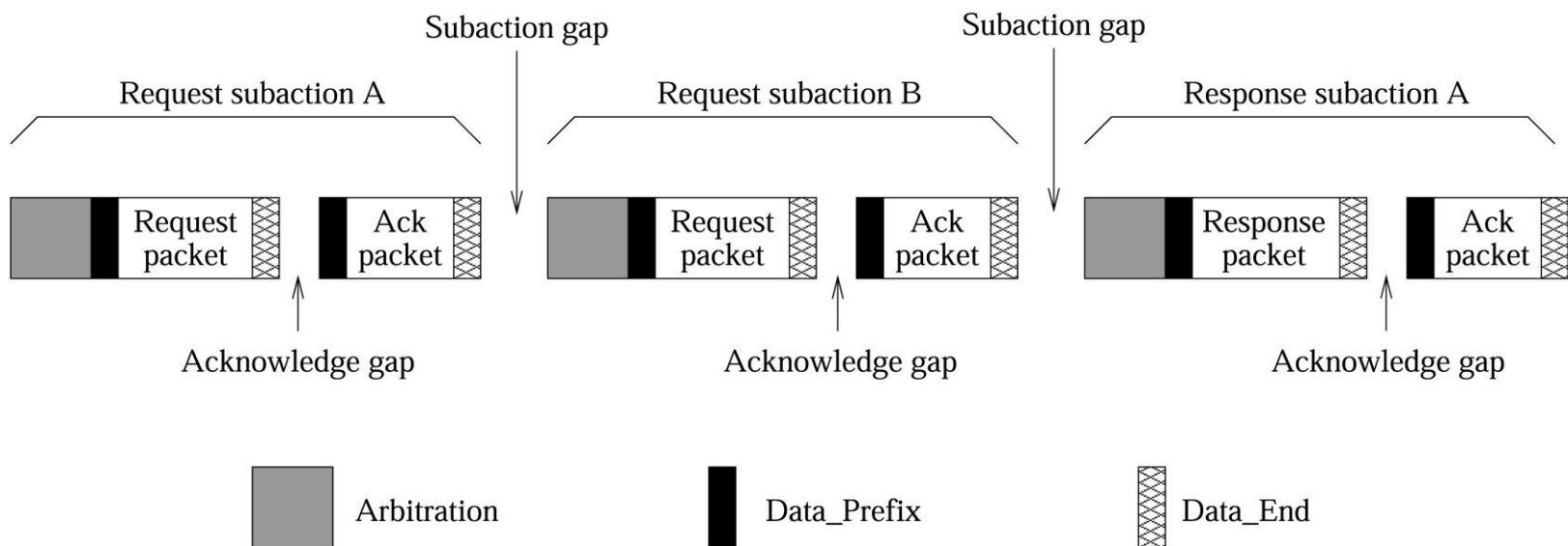
(a) Asynchronous transfer



(b) Isochronous transfer

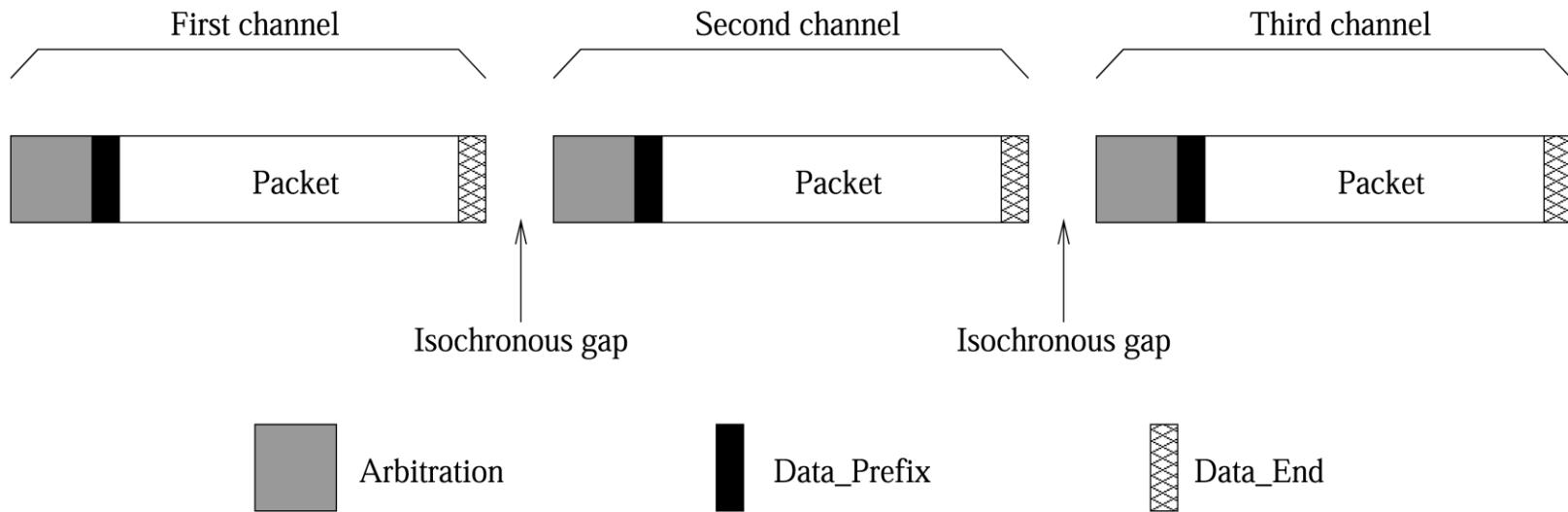
IEEE 1394 (cont'd)

- Transactions
 - * Follow request and reply format
 - * Each packet is encapsulated between **Data_Prefix** and **Data_end**



IEEE 1394 (cont'd)

- Isochronous transactions
 - * Similar to asynchronous transactions
 - * Main difference:
 - » No acknowledgement packets



IEEE 1394 (cont'd)

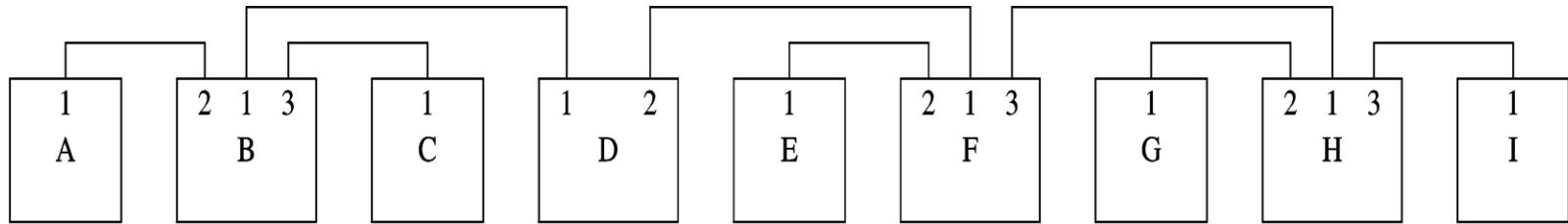
- Bus arbitration
 - * Needed because of peer-to-peer communication
 - * Arbitration must respect
 - » Bandwidth allocation to isochronous channels
 - » Fairness-based allocation for asynchronous channels
 - * Uses fairness interval
 - » During each interval
 - All nodes with pending asynchronous transaction are allowed bus ownership once
 - * Nodes with pending isochronous transactions go through arbitration during each cycle
 - * IRM is used for isochronous bandwidth allocations

IEEE 1394 (cont'd)

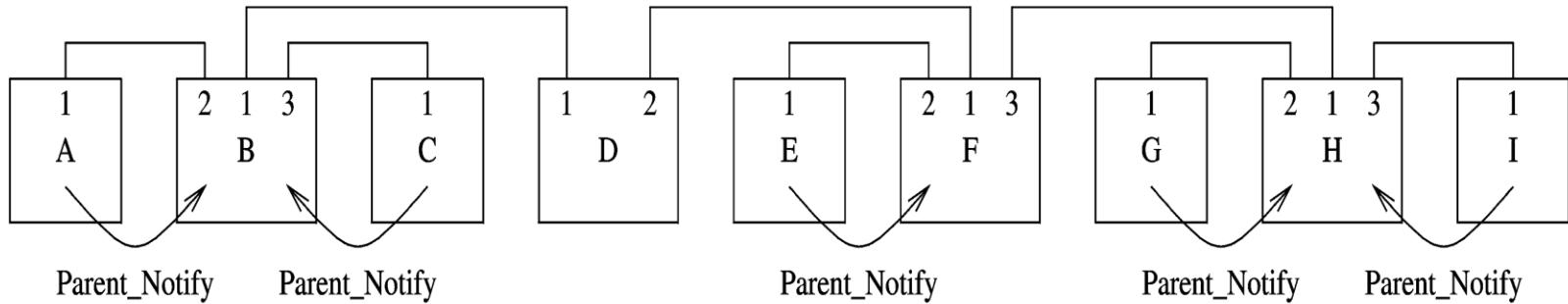
- Configuration
 - * Does not require the host system
 - * Consists of two main phases
 - » Tree identification
 - Used to find the network topology
 - Uses two special signals
 - **Parent_notify** and **Child_Notify**
 - » Self-identification
 - Done after the tree identification
 - Assigns unique ids to nodes

IEEE 1394 (cont'd)

Tree identification



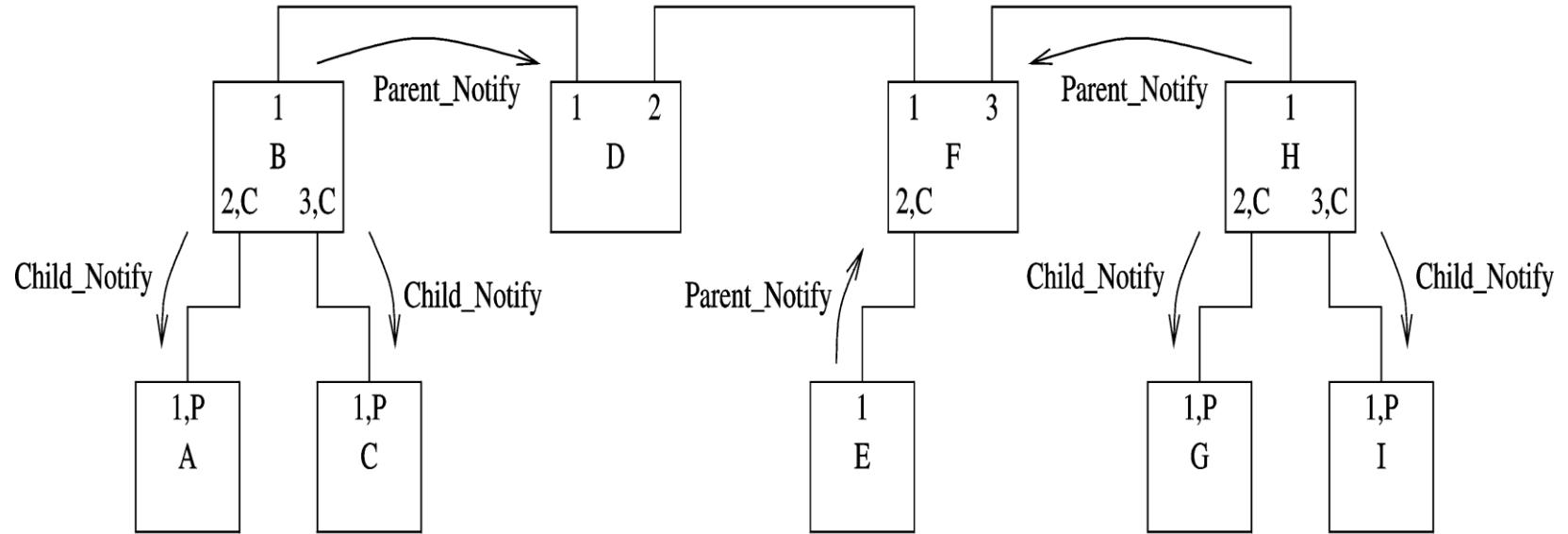
(a) Original unconfigured network



(b) Leaf nodes send Parent_Notify signal to their parent nodes

IEEE 1394 (cont'd)

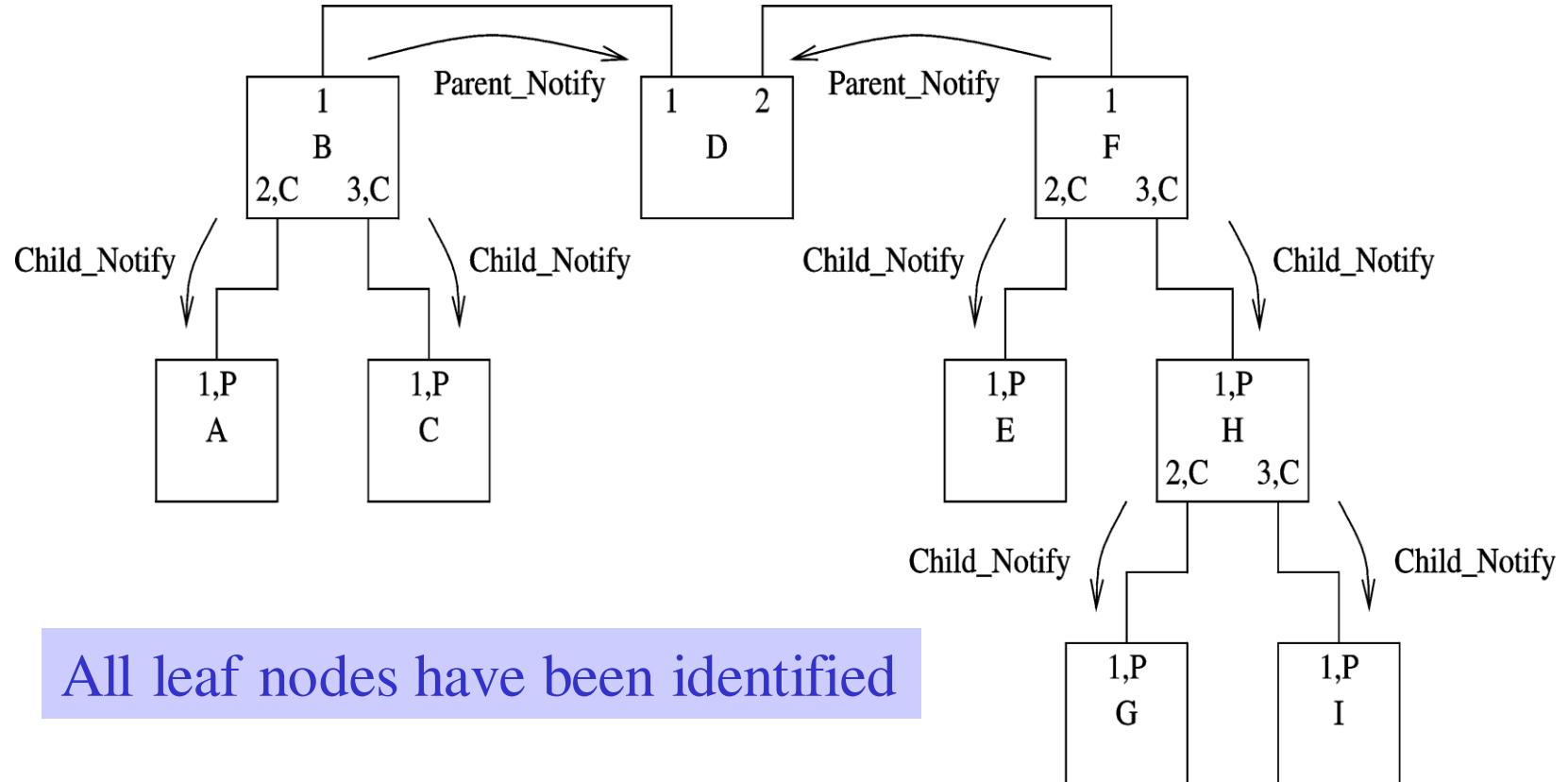
Tree identification



(c) Topology starts to take shape with nodes A, C, E, G, and I identified as leaf nodes

IEEE 1394 (cont'd)

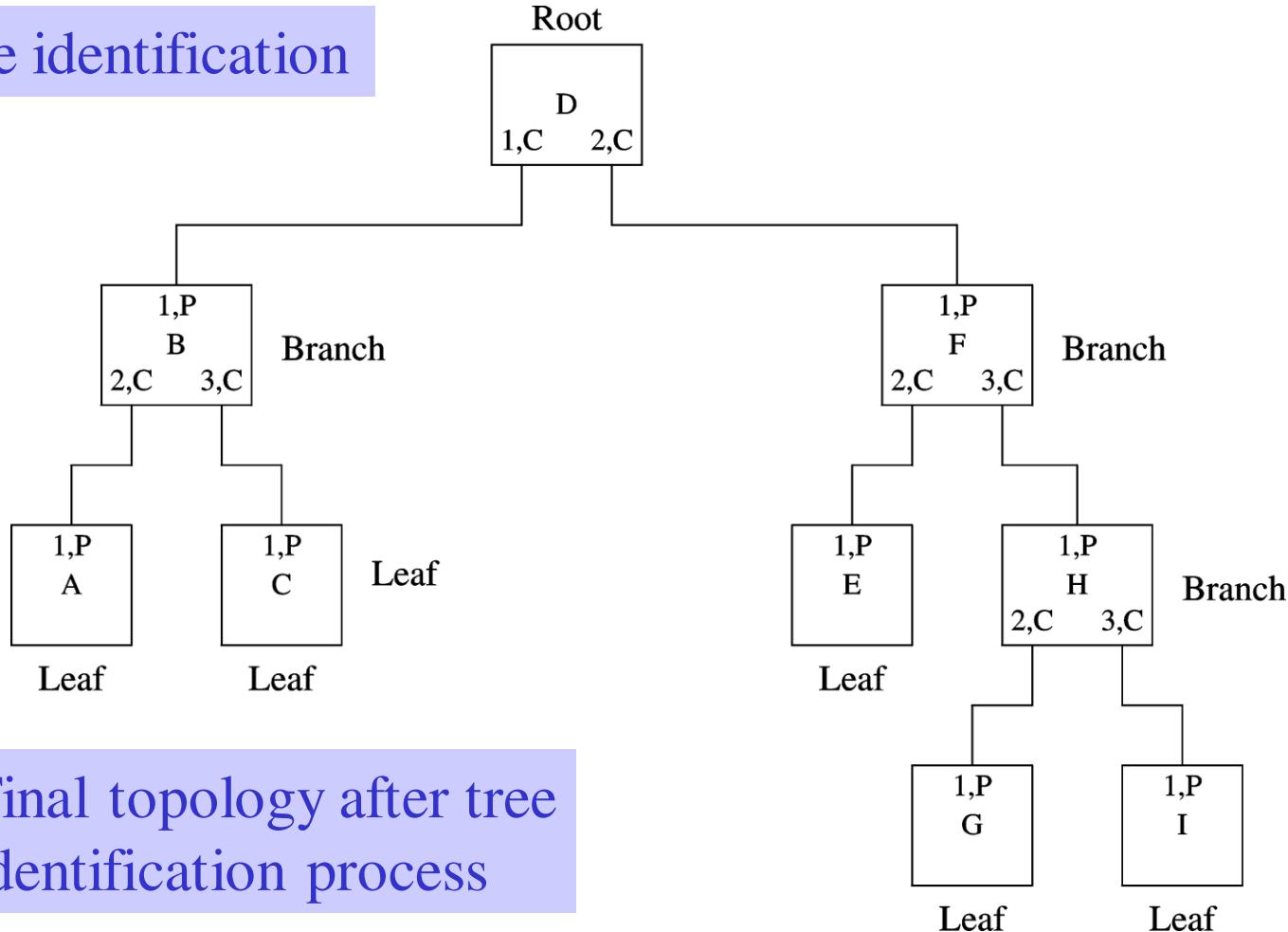
Tree identification



All leaf nodes have been identified

IEEE 1394 (cont'd)

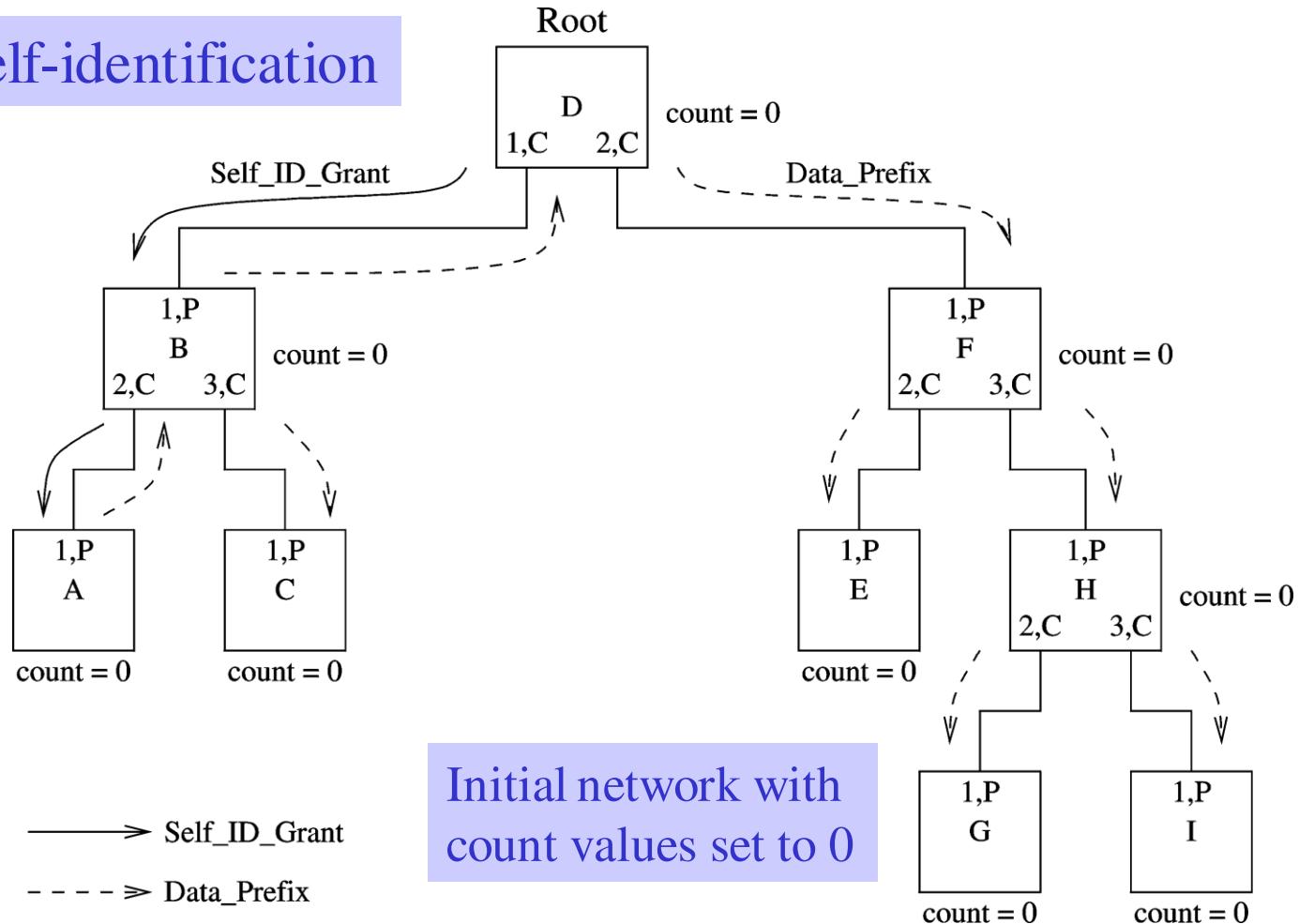
Tree identification



Final topology after tree identification process

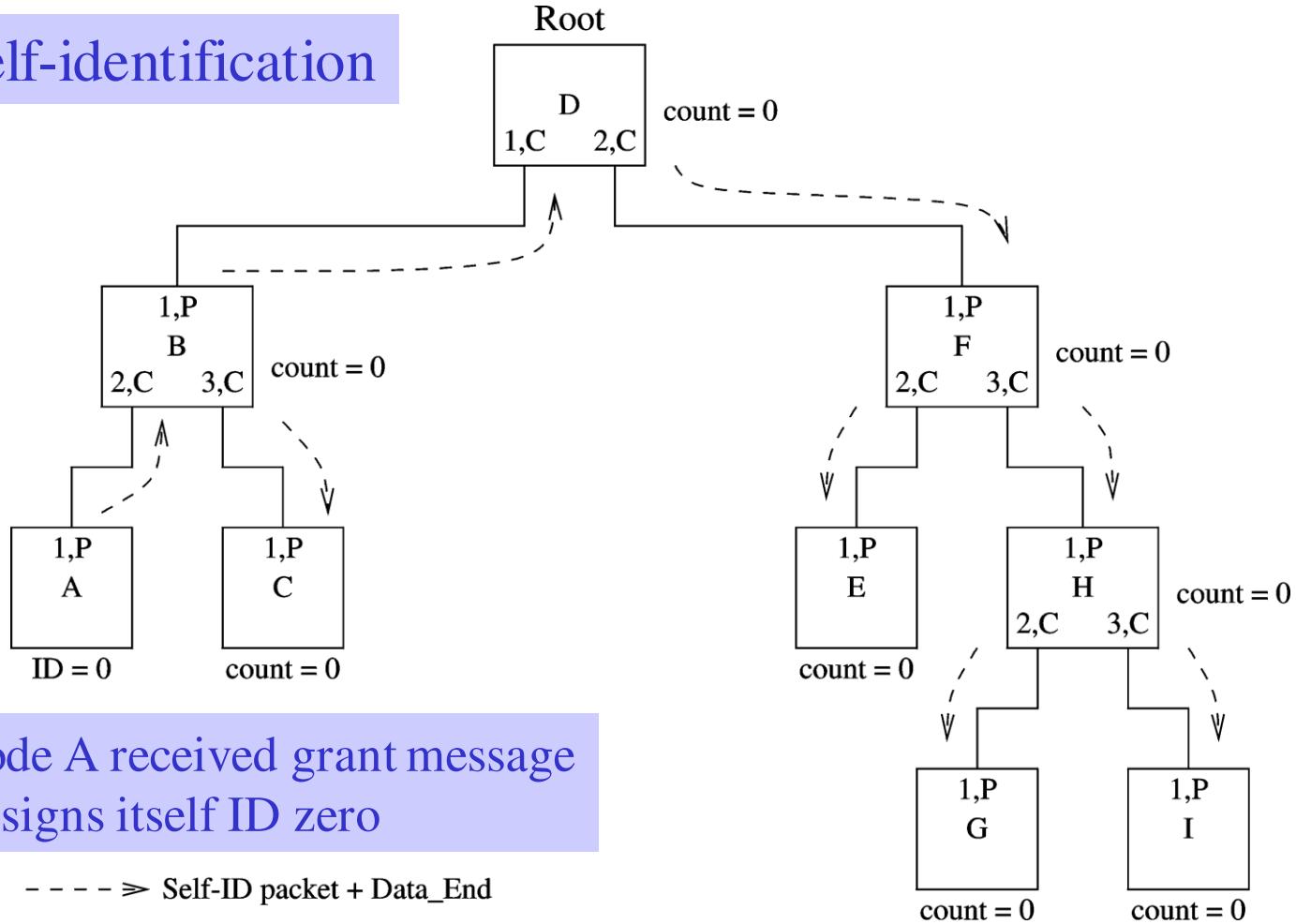
IEEE 1394 (cont'd)

Self-identification



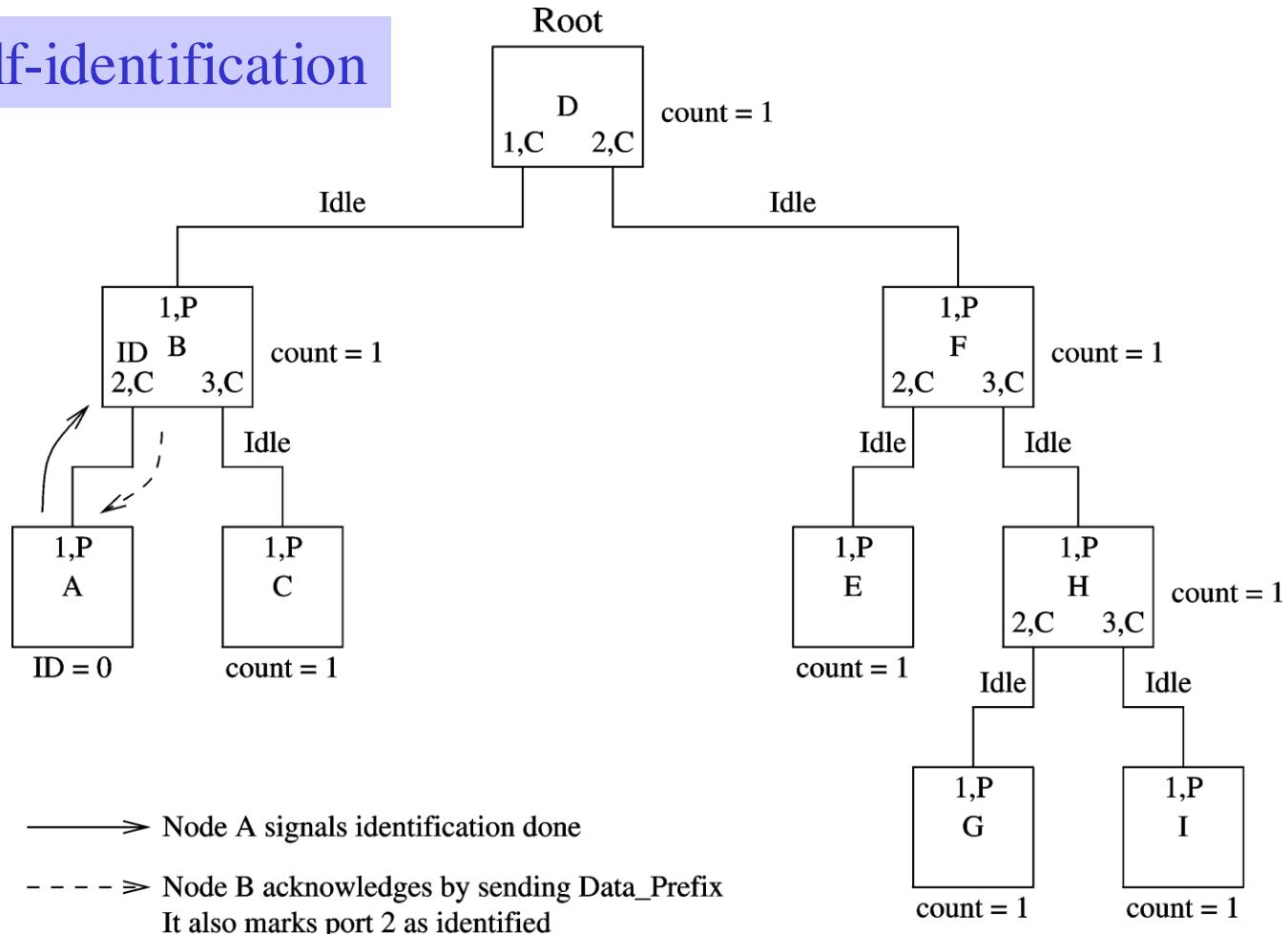
IEEE 1394 (cont'd)

Self-identification



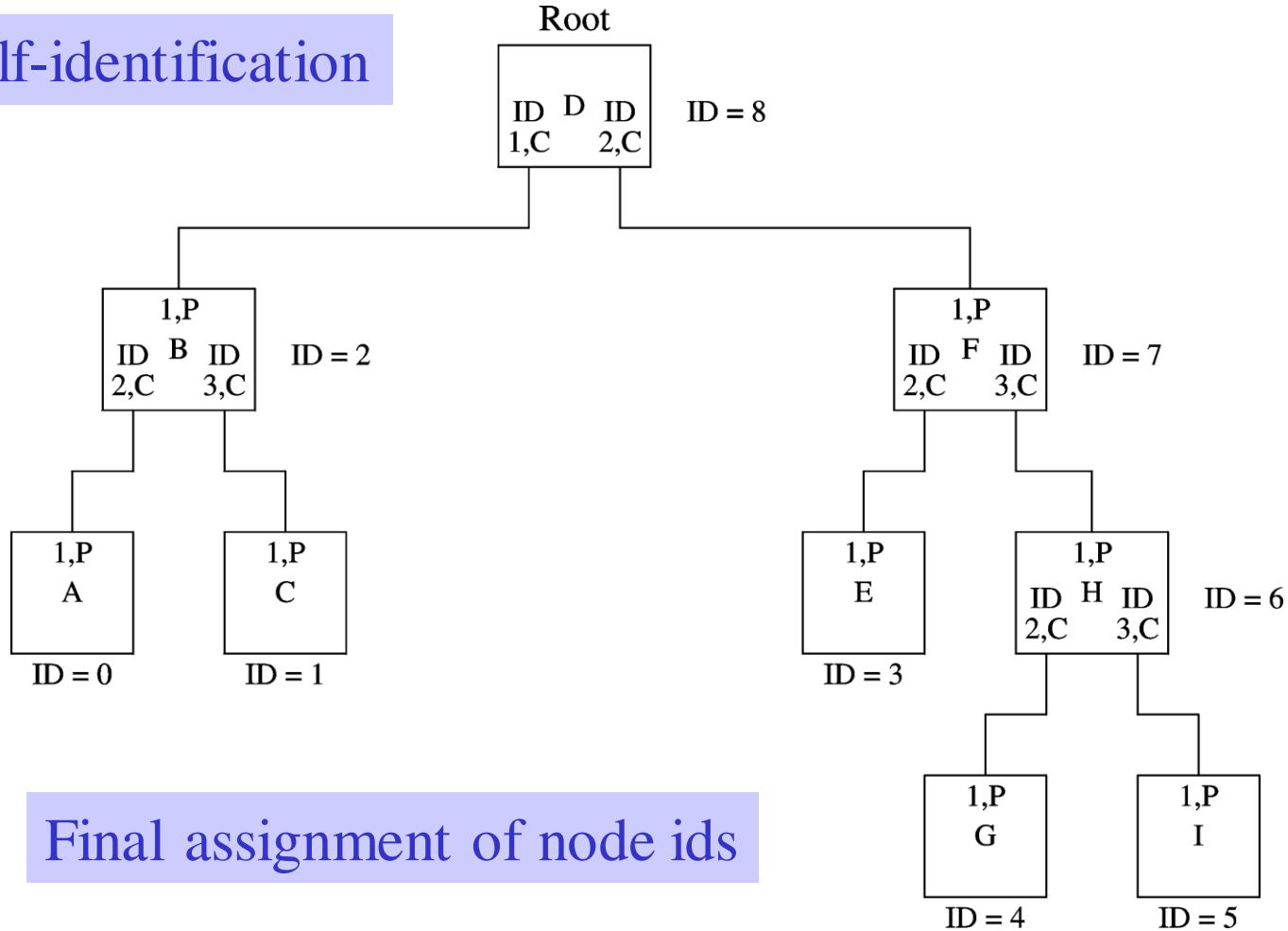
IEEE 1394 (cont'd)

Self-identification



IEEE 1394 (cont'd)

Self-identification



Final assignment of node ids

Bus Wars

- SCSI is dominant in disk and storage device interfaces
 - * Parallel interface
 - * Its bandwidth could go up to 640 MB/s
- IEEE 1394
 - * Serial interface
 - * Supports peer-to-peer applications
 - * Dominant in video applications
- USB
 - * Useful in low-cost, host-to-peripheral applications
 - * USB 2.0 provides high-speed support

Last slide

Input/Output Organization

- | The computer system's *input/output* (I/O) architecture is its interface to the outside world.
- | Till now we have discussed the two important modules of the computer system -
 - m **The processor** and
 - m **The memory** module.
- | The third key component of a computer system is a set of **I/O modules**
- | Each I/O module interfaces to the system bus and controls one or more peripheral devices.

There are several reasons why an I/O device or peripheral device is not directly connected to the system bus. Some of them are as follows -

- I There are a wide variety of peripherals with various methods of operation. It would be impractical to include the necessary logic within the processor to control several devices.
 - I The data transfer rate of peripherals is often much slower than that of the memory or processor. Thus, it is impractical to use the high-speed system bus to communicate directly with a peripheral.
 - I Peripherals often use different data formats and word lengths than the computer to which they are attached.
- Thus, an I/O module is required.

Input/Output Modules

The major functions of an I/O module are categorized as follows –

- m Control and timing
- m Processor Communication
- m Device Communication
- m Data Buffering
- m Error Detection

During any period of time, the processor may communicate with one or more external devices in unpredictable manner, depending on the program's need for I/O.

The internal resources, such as main memory and the system bus, must be shared among a number of activities, including data I/O.

<<

Control & timings:

The I/O function includes a control and timing requirement to co-ordinate the flow of traffic between internal resources and external devices.

For example, the control of the transfer of data from an external device to the processor might involve the following sequence of steps –

- a. The processor interacts with the I/O module to check the status of the attached device.
- b. The I/O module returns the device status.
- c. If the device is operational and ready to transmit, the processor requests the transfer of data, by means of a command to the I/O module.
- d. The I/O module obtains a unit of data from external device.
- e. The data are transferred from the I/O module to the processor.

If the system employs a bus, then each of the interactions between the processor and the I/O module involves one or more bus arbitrations.

Processor & Device Communication

During the I/O operation, the I/O module must communicate with the processor and with the external device.

Processor communication involves the following -

Command decoding :

The I/O module accepts command from the processor, typically sent as signals on control bus.

Data :

Data are exchanged between the processor and the I/O module over the data bus.

Status Reporting :

Because peripherals are so slow, it is important to know the status of the I/O module. For example, if an I/O module is asked to send data to the processor(read), it may not be ready to do so because it is still working on the previous I/O command. This fact can be reported with a status signal. Common status signals are **BUSY** and **READY**.

Address Recognition :

Just as each word of memory has an address, so thus each of the I/O devices. Thus an I/O module must recognize one unique address for each peripheral it controls.

On the other hand, the I/O must be able to perform device communication. This communication involves command,
status information and data.

Data Buffering:

An essential task of an I/O module is data buffering. The data buffering is required due to the mismatch of the speed of CPU, memory and other peripheral devices. In general, the speed of CPU is higher than the speed of the other peripheral devices. So, the I/O modules store the data in a data buffer and regulate the transfer of data as per the speed of the devices.

In the opposite direction, data are buffered so as not to tie up the memory in a slow transfer operation.

Thus the I/O module must be able to operate at both device and memory speed.

Error Detection:

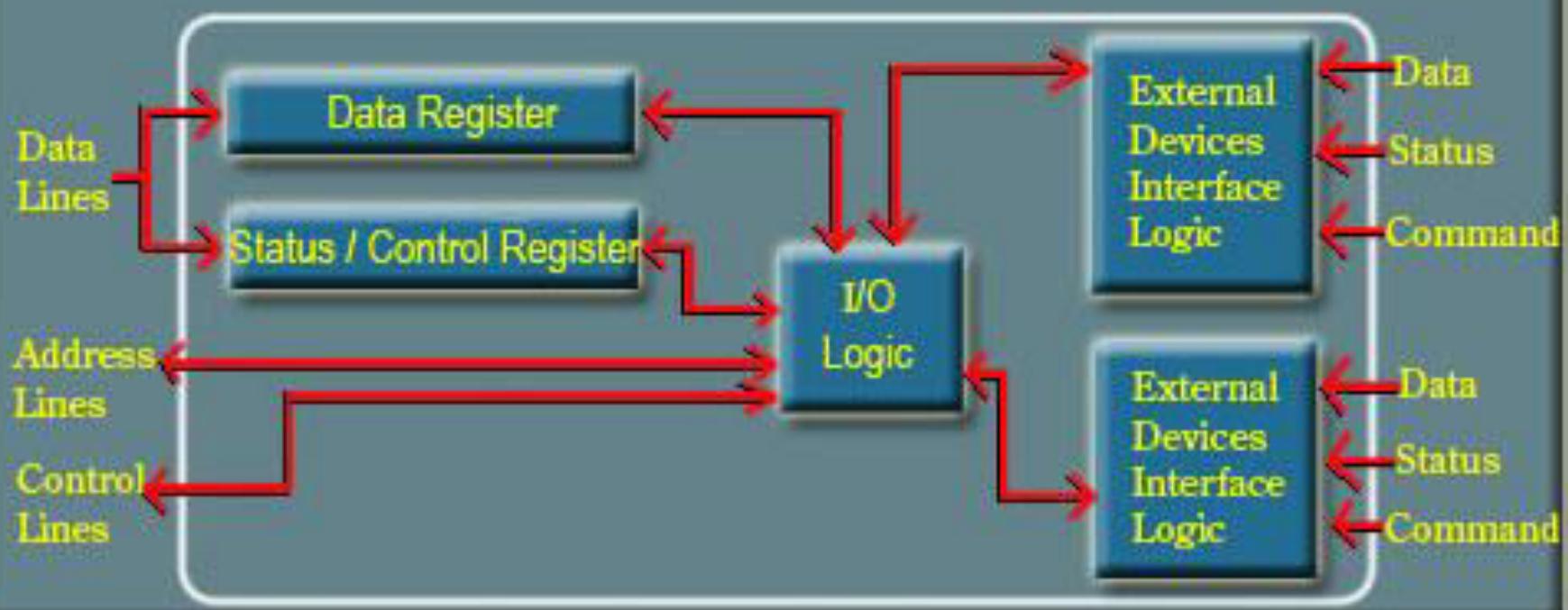
Another task of I/O module is error detection and for subsequently reporting error to the processor. One

class or error includes mechanical and electrical malfunctions reported by the device (e.g. paper jam).

Another class consists of unintentional changes to the bit pattern as it is transmitted from devices to the I/O module.

Interface to System Bus

Interface to External Devices



There will be many I/O devices connected through I/O modules to the system. Each device will be identified by a unique address.

When the processor issues an I/O command, the command contains the address of the device that is used by the command. The I/O module must interpret the address lines to check if the command is for itself.

Generally in most of the processors, the processor, main memory and I/O share a common bus(data address and control bus).

Two types of addressing are possible -

- | Memory-mapped I/O
- | Isolated or I/O mapped I/O

Memory-mapped I/O:

There is a single address space for memory locations and I/O devices.

The processor treats the status and address register of the I/O modules as memory location. For example, if the size of address bus of a processor is 16, then there are 2^{16} combinations and all together 2^{16} address locations can be addressed with these 16 address lines.

Out of these 2^{16} address locations, some address locations can be used to address I/O devices and other locations are used to address memory locations.

Since I/O devices are included in the same memory address space, so the status and address registers of I/O modules are treated as memory location by the processor. Therefore, the same machine instructions are used to access both memory and I/O devices.

Isolated or I/O -mapped I/O:

In this scheme, the full range of addresses may be available for both.

The address refers to a memory location or an I/O device is specified with the help of a command line.

In general command line is used to identify a memory location or an I/O device.

if $\text{IO/M}' = 1$, it indicates that the address present in address bus is the address of an I/O device.

if $\text{IO/M}' = 0$, it indicates that the address present in address bus is the address of a memory location.

Since full range of address is available for both memory and I/O devices, so, with 16 address lines, the system may now support both 2^{16} memory locations and 2^{16} I/O addresses.

Input / Output Subsystem

There are three basic forms of input and output systems –

- m **Programmed I/O**
- m **Interrupt driven I/O**

Direct Memory Access(DMA)

With programmed I/O, the processor executes a program that gives its direct control of the I/O operation, including

sensing device status, sending a read or write command, and transferring the data.

With interrupt driven I/O, the processor issues an I/O command, continues to execute other instructions, and is

interrupted by the I/O module when the I/O module completes its work.

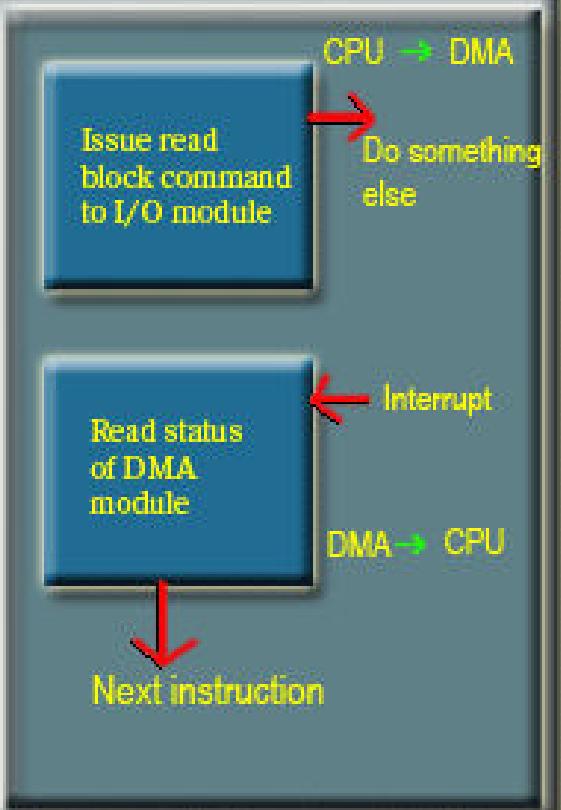
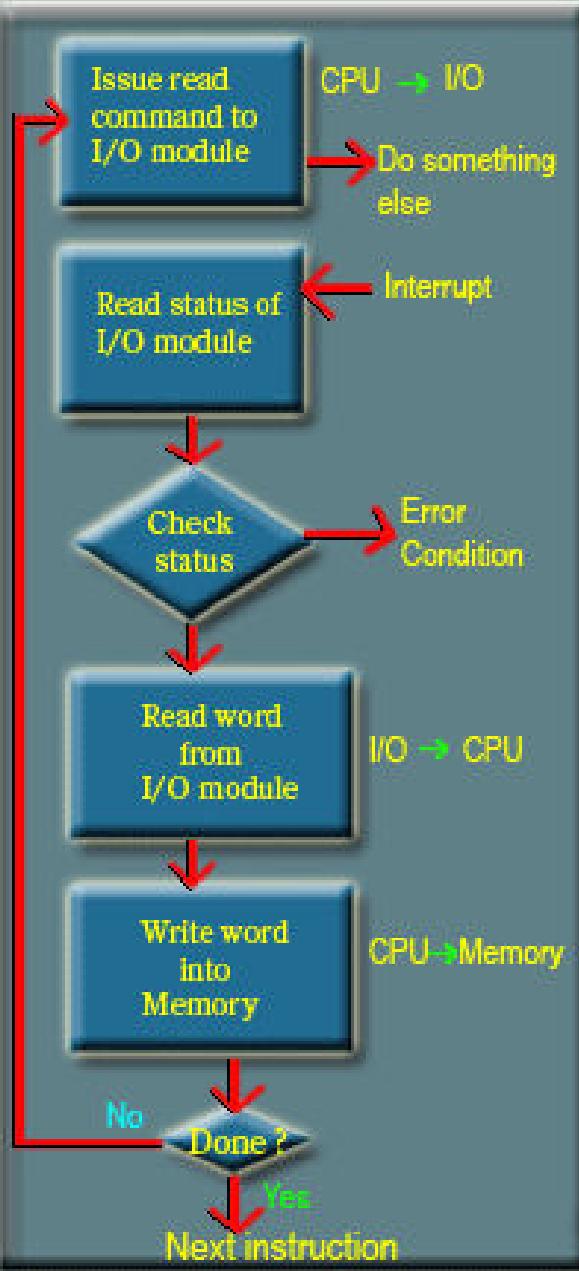
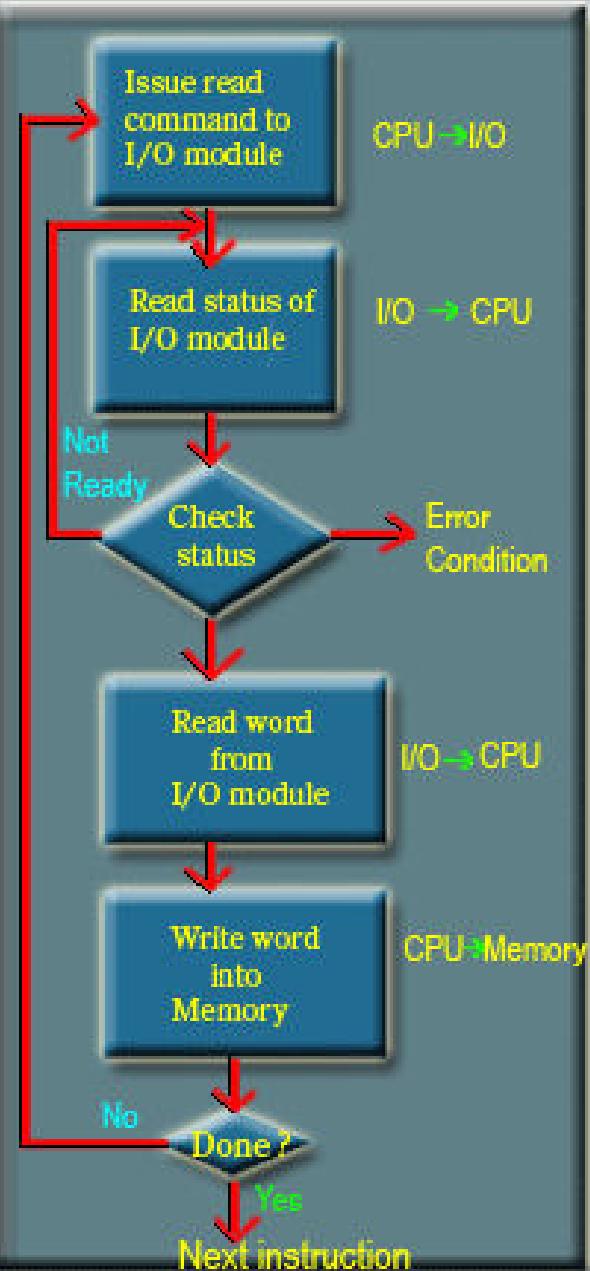
In Direct Memory Access (DMA), the I/O module and main memory exchange data directly without processor involvement.

With both programmed I/O and Interrupt driven I/O, the processor is responsible for extracting data from main memory for output operation and storing data in main memory for input operation. To send data to an output device, the CPU simply moves that data to a *special memory location* in the I/O address space if I/O mapped input/output is used or to an address in the memory address space if memory mapped I/O is used.

Data I/O Address Space (in memory) if I/O mapped input/output is used
memory address space if memory mapped I/O is used

To read data from an input device, the CPU simply moves data from the address (I/O or memory) of that device into the CPU.

Input/Output Operation: The input and output operation looks very similar to a memory read or write operation except it usually takes *more time* since peripheral devices are slow in speed than main memory modules.



(a) Programmed I/O
16-Jul-2021 - Organization

(b) Interrupt Instruction

Three techniques
for input of
block of data

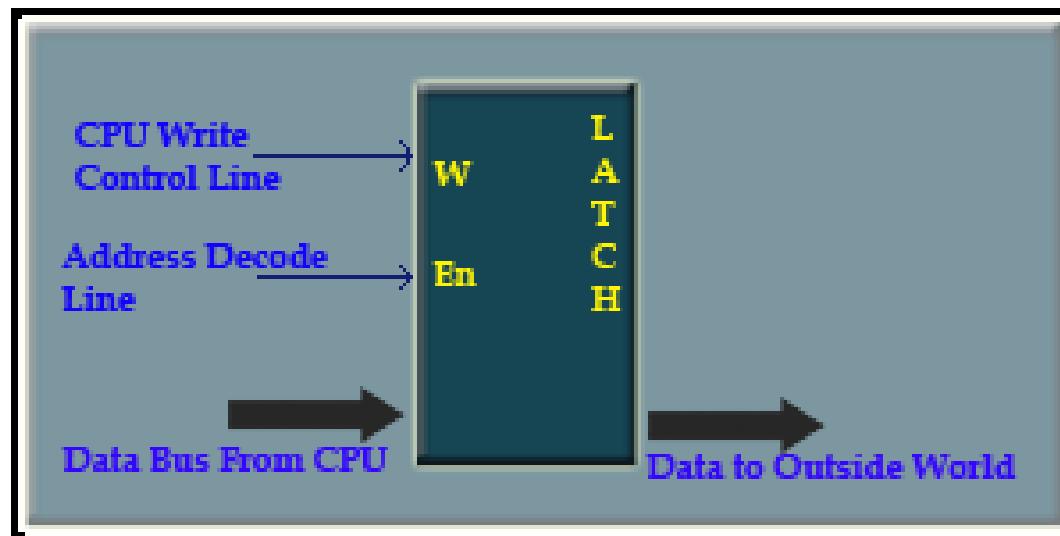
Input/Output Port

An *I/O port* is a device that looks like a memory cell to the computer but contains connection to the outside world.

An *I/O port* typically uses a *latch*. When the CPU writes to the address associated with the latch, the latch device

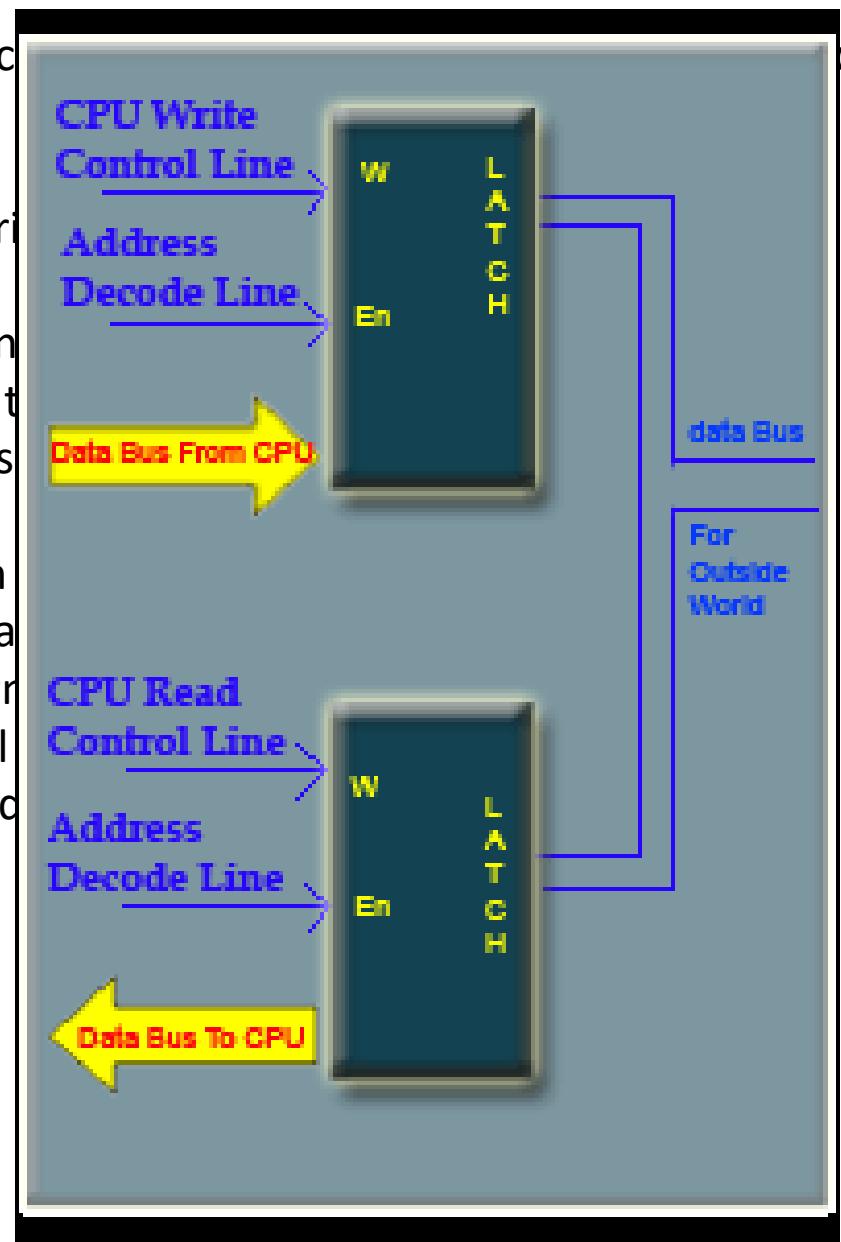
captures the data and makes it available on a set of wires external to the CPU and memory system.

The *I/O ports* can be *read-only*, *write-only*, or *read/write*. The *write-only* port is shown in the figure.



First, the CPU will place the address of the device in the *address decoder* a signal is generated which will enable the latch. Next, the CPU will indicate the operation is a write by placing an appropriate signal in CPU write control line. Then the data to be transferred will be placed in the latch. Finally, the data will be stored in the latch for the onward transmission to the output device. Both the address decode and write control lines must be asserted for the latch to operate.

The *read/write* or *input/output* port is shown in the figure. The device is identified by putting the appropriate address in the address decoder. The address decoder will generate the signal required for the operation. According to the operation, *read* or *write*, it will enable the appropriate latch. If it is a write operation, then data will be placed in the latch. Finally, the data will be stored in the latch for the onward transmission to the output device.



If it is in a read operation, the data that are already stored in the latch will be transferred to the CPU.

A *read only* (input) port is simply the lower half of the figure.

In case of I/O mapped I/O, a different address space is used for I/O devices. The address space for memory is

different. In case of memory mapped I/O, same address space is used for both memory and I/O devices. Some of the

memory address space are kept reserved for I/O devices.

To the programmer, the difference between I/O-mapped and memory-mapped input/output operation is the

instruction to be used.

For memory-mapped I/O, any instruction that accessed memory can access a memory-mapped I/O port.

I/O-mapped input/output uses special instruction to access I/O port.

Generally, a given peripheral device will use more than a single I/O port. A typical PC parallel printer interface, for example, uses three ports, a *read/write port*, and *input port* and an *output port*. The read/write port is the data port (it is read/write to allow the CPU to read the last ASCII character it wrote to the printer port).

The input port returns control signals from the printer.

- These signals indicate whether the printer is ready to accept another character, is off-line, is out of paper, etc.

The output port transmits control information to the printer such as

- whether data is available to print.

Memory-mapped I/O subsystems and I/O-mapped subsystems both require the CPU to move data between the peripheral device and main memory.

For example, to input a sequence of 20 bytes from an input port and store these bytes into memory, the CPU must send each value and store it into memory.

Programmed I/O:

In programmed I/O, the data transfer between CPU and I/O device is carried out with the help of a software routine.

When a processor is executing a program and encounters an instruction relating to I/O, it executes that I/O instruction by issuing a command to the appropriate I/O module.

The I/O module will perform the requested action and then set the appropriate bits in the I/O status register.

The I/O module takes no further action to alert the processor.

It is the responsibility of the processor to check periodically the status of the I/O module until it finds that the operation is complete.

In programmed I/O, when the processor issues a command to a I/O module, it must wait until the I/O operation is complete.

Generally, the I/O devices are slower than the processor, so in this scheme CPU time is wasted. CPU is checking the status of the I/O module periodically without doing any other work.

I/O Commands

To execute an I/O-related instruction, the processor issues an address, specifying the particular I/O module and external device, and an I/O command. There are four types of I/O commands that an I/O module will receive when it is addressed by a processor –

I Control : Used to activate a peripheral device and instruct it what to do. For example, a magnetic tape unit may be instructed to rewind or to move forward one record. These commands are specific to a particular type of peripheral device.

I Test : Used to test various status conditions associated with an I/O module and its peripherals. The processor will want to know if the most recent I/O operation is completed or any error has occurred.

I Read : Causes the I/O module to obtain an item of data from the peripheral and place it in the internal buffer.

I Write : Causes the I/O module to take an item of data (byte or word) from the data bus and subsequently transmit the data item to the peripheral.

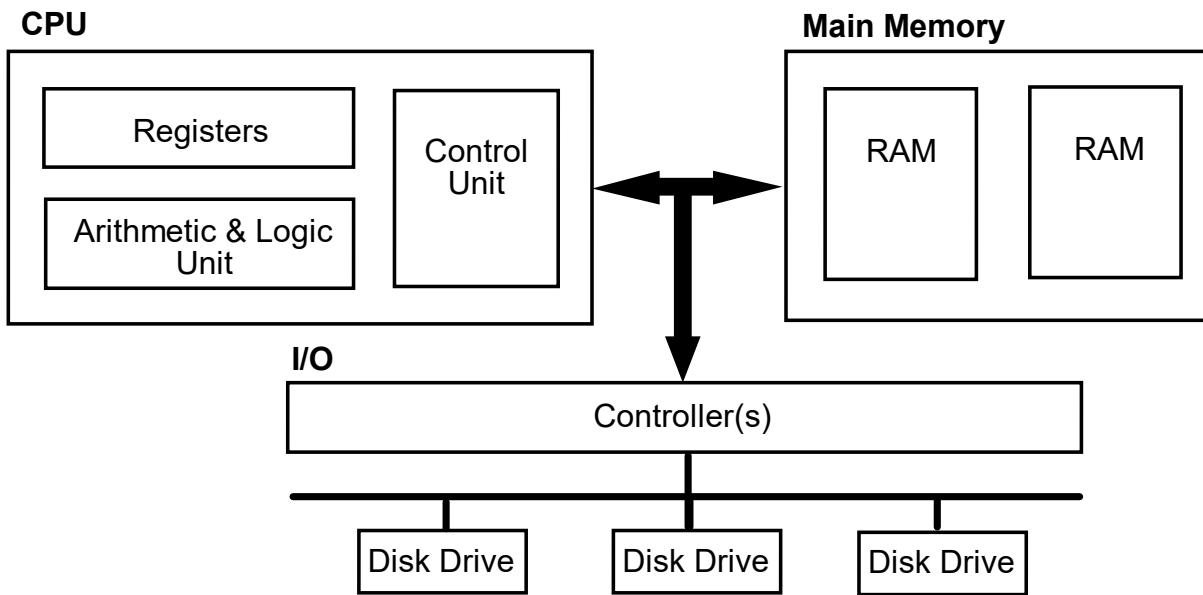
Memory Organisation

CPU Organisation and Operation

Introduction to Assembly Programming

Main Memory (RAM) Organisation

Computers employ many different types of memory (semi-conductor, magnetic disks and tapes, DVDs etc.) to hold data and programs. Each type has its own characteristics and uses. We will look at the way that Main Memory (RAM) is organised and very briefly at the characteristics of Register Memory and Disk Memory. Let's locate these 3 types of memory in an abstract computer:



Register Memory

Registers are memories located within the Central Processing Unit (CPU). They are few in number (there are rarely more than 64 registers) and also small in size, typically a register is less than 64 bits; 32-bit and more recently 64-bit are common in desktops.

The contents of a register can be “read” or “written” very quickly¹ however, often an order of magnitude faster than main memory and several orders of magnitude faster than disk memory.

Different kinds of register are found within the CPU. General Purpose Registers² are available for general³ use by the programmer. Unless the context implies otherwise we'll use the term "register" to refer to a General Purpose Register within the CPU. Most modern CPU's have between 16 and 64 general purpose registers. Special Purpose Registers have specific uses and are either non-programmable and internal to the CPU or accessed with special instructions by the programmer. Examples of such registers that we will encounter later in the course include: the Program Counter register (PC), the Instruction Register (IR), the ALU Input & Output registers, the Condition Code (Status/Flags) register, the Stack Pointer register (SP). The size (the number of bits in the register) of

¹ e.g. less than a nanosecond (10^{-9} sec)

² Occasionally called Working Registers

³ Used for performing calculations, moving and manipulating data etc.

the these registers varies according to register type. The Word Size of an architecture is often (but not always!) defined by the size of the general purpose registers.

In contrast to main memory and disk memory, registers are referenced directly by specific instructions or by encoding a register number within a computer instruction. At the programming (assembly) language level of the CPU, registers are normally specified with special identifiers (e.g. R0, R1, R7, SP, PC)

As a final point, the contents of a register are lost if power to the CPU is turned off, so registers are unsuitable for holding long-term information or information that is needed for retention after a power-shutdown or failure. Registers are however, the fastest memories, and if exploited can result in programs that execute very quickly.

Main Memory (RAM)

If we were to sum all the bits of all registers within CPU, the total amount of memory probably would not exceed 5,000 bits. Most computational tasks undertaken by a computer require a lot more memory. Main memory is the next⁴ fastest memory within a computer and is much larger in size. Typical main memory capacities for different kinds of computers are: PC 512MB⁵, fileserver 2GB, database server 8GB. Computer architectures also impose an architectural constraint on the maximum allowable RAM. This constraint is normally equal to 2^{WordSize} memory locations.

RAM⁶ (Random⁷ Access Memory) is the most common form of Main Memory. RAM is normally located on the motherboard and so is typically less than 12 inches from the CPU. ROM (Read Only Memory) is like RAM except that its contents cannot be overwritten and its contents are not lost if power is turned off (ROM is non-volatile).

Although slower than register memory, the contents of any location⁸ in RAM can still be “read” or “written” very quickly⁹. The time to read or write is referred to as the **access time** and is constant for all RAM locations.

In contrast to register memory, RAM is used to hold both program code (instructions) and data (numbers, strings etc). Programs are “loaded” into RAM from a disk prior to execution by the CPU.

Locations in RAM are identified by an **addressing scheme** e.g. numbering the bytes in RAM from 0 onwards¹⁰. Like registers, the contents of RAM are lost if the power is turned off.

⁴ Actually many computers systems also include Cache memory, which is faster than Main memory, but slower than register memory. We will ignore Cache memories in this course.

⁵ $1K = 2^{10} = 1024$, $1M = 2^{20}$, $1G = 2^{30}$, ‘B’ will be used for Bytes, and ‘b’ or ‘bit’ for bits, cf. 1MB and 1Mbit

⁶ There are many types of RAM technologies.

⁷ Random is a Misnomer. Direct Access Memory would have been a better term.

⁸ Typically a byte multiple.

⁹ e.g. less than 10 nanoseconds (10×10^{-9} sec)

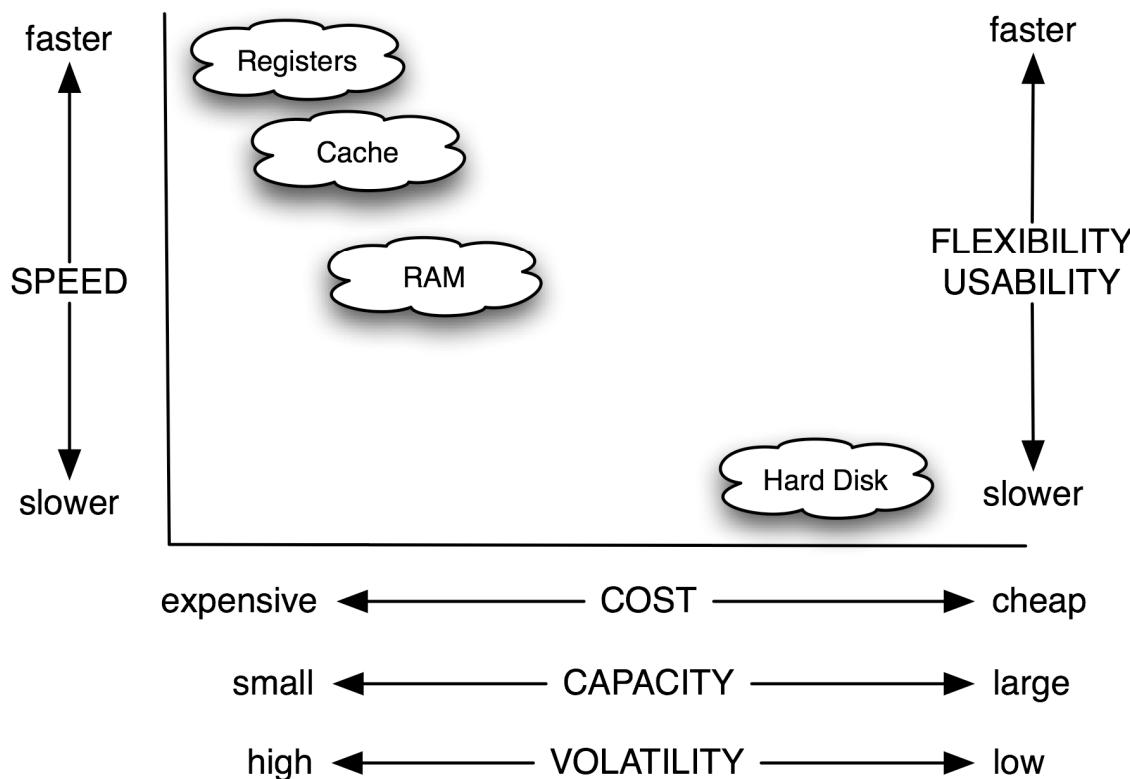
¹⁰ Some RAM locations (typically those with the lowest & highest addresses) may cause side-effects, e.g. cause data to be transferred to/from external devices

Disk Memory

Disk memory¹¹ is used to hold programs and data over the longer term. The **contents of a disk are NOT lost if the power is turned off**. Typical hard disk capacities range from 40GB to over 500 GB (5×10^{29}). Disks are much slower than register and main memory, the access-time (known as the seek-time) to data on disk is typically between 2 and 4 milli-seconds, although disk drives can transfer thousands of bytes in one go achieving transfer rates from 25MB/s to 500MB/s.

Disks can be housed internally within a computer “box” or externally in an enclosure connected by a fast USB or firewire cable¹². Disk locations are identified by special disk addressing schemes (e.g. track and sector numbers).

Summary of Characteristics



¹¹ Some authors refer to disk memory as disk storage.

¹² For details about how disks and other storage devices work, check out Tanenbaum or Stallings.

SRAM, DRAM, SDRAM, DDR SDRAM

There are many kinds of RAM and new ones are invented all the time. One of aims is to make RAM access as fast as possible in order to keep up with the increasing speed of CPUs.

SRAM (Static RAM) is the fastest form of RAM but also the most expensive. Due to its cost it is not used as main memory but rather for cache memory. Each bit requires a 6-transistor circuit.

DRAM (Dynamic RAM) is not as fast as SRAM but is cheaper and is used for main memory. Each bit uses a single capacitor and single transistor circuit. Since capacitors lose their charge, DRAM needs to be refreshed every few milliseconds. The memory system does this transparently. There are many implementations of DRAM, two well-known ones are SDRAM and DDR SDRAM.

SDRAM (Synchronous DRAM) is a form of DRAM that is synchronised with the clock of the CPU's system bus, sometimes called the front-side bus (FSB). As an example, if the system bus operates at 167Mhz over an 8-byte (64-bit) data bus , then an SDRAM module could transfer $167 \times 8 \sim 1.3\text{GB/sec}$.

DDR SDRAM (Double-Data Rate DRAM) is an optimisation of SDRAM that allows data to be transferred on both the rising edge and falling edge of a clock signal. Effectively doubling the amount of data that can be transferred in a period of time. For example a PC-3200 DDR-SDRAM module operating at 200Mhz can transfer $200 \times 8 \times 2 \sim 3.2\text{GB/sec}$ over an 8-byte (64-bit) data bus.

ROM, PROM, EPROM, EEPROM, Flash

In addition to RAM, they are also a range of other semi-conductor memories that retain their contents when the power supply is switched off.

ROM (Read Only Memory) is a form of semi-conductor that can be written to once, typically in bulk at a factory. ROM was used to store the “boot” or start-up program (so called firmware) that a computer executes when powered on, although it has now fallen out-of-favour to more flexible memories that support occasional writes. ROM is still used in systems with fixed functionalities, e.g. controllers in cars, household appliances etc.

PROM (Programmable ROM) is like ROM but allows end-users to write their own programs and data. It requires a special PROM writing equipment. Note: users can only write-once to PROM.

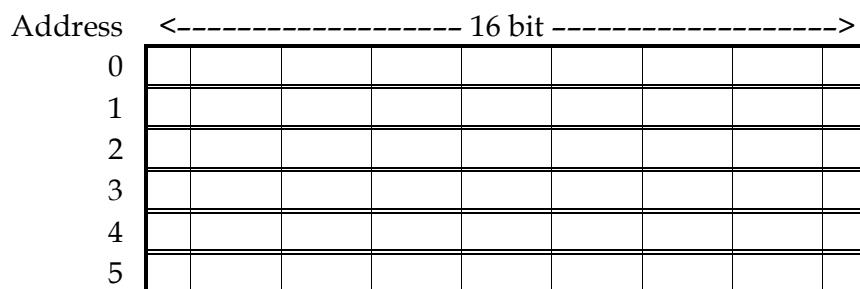
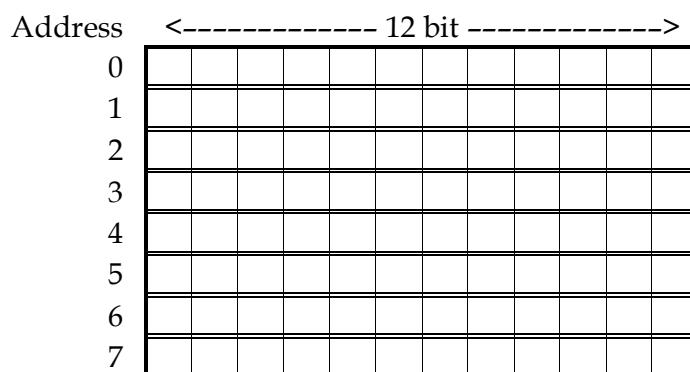
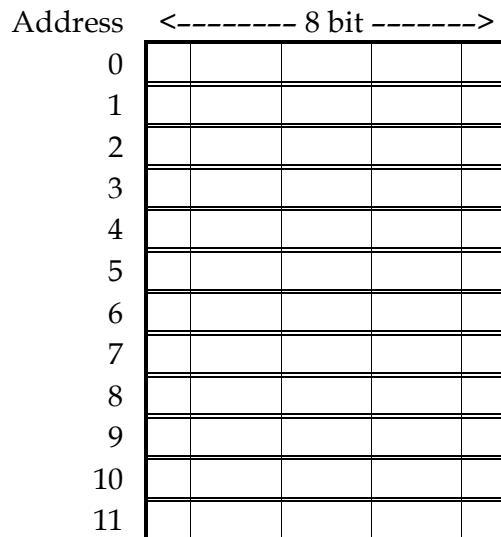
EPROM (Erasable PROM). With EPROM we can erase (using strong ultra-violet light) the contents of the chip and rewrite it with new contents, typically several thousand times. It is commonly used to store the “boot” program of a computer, known as the firmware. PCs call this firmware, the BIOS (Basic I/O System). Other systems use Open Firmware. Intel-based Macs use EFI (Extensible Firmware Interface).

EEPROM (Electrically Erasable PROM). As the name implies the contents of EEPROMs are erased electrically. EEPROMSs are also limited to the number of erase-writes that can be performed (e.g, 100,000) but support updates (erase-writes) to individual bytes whereas EPROM updates the whole memory and only supports around 10,000 erase-write cycles.

FLASH memory is a cheaper form of EEPROM where updates (erase-writes) can only be performed on blocks of memory, not on individual bytes. Flash memories are found in USB sticks, flash cards and typically range in size from 32M to 2GB. The number of erase/write cycles to a block is typically several hundred thousand before the block can no longer be written.

Main Memory Organisation

Main memory can be considered to be organised as a matrix of bits. Each row represents a memory location, typically this is equal to the word size of the architecture, although it can be a word multiple (e.g. 2xWordsize) or a partial word (e.g. half the wordsize). **For simplicity we will assume that data within main memory can only be read or written a single row (memory location) at a time.** For a 96-bit memory we could organise the memory as 12x8 bits, or 8x12 bits or, 6x16 bits, or even as 96x1 bits or 1x96 bits. Each row also has a natural number called its **address**¹³ which is used for selecting the row:



¹³ The concept of an address is very important to properly understanding how CPUs work.

Byte Addressing

Main-memories generally store and recall rows, which are multi-byte in length (e.g. 16-bit word = 2 bytes, 32-bit word = 4 bytes). Many architectures, however, make main memory **byte-addressable** rather than **word addressable**. In such architectures the CPU and/or the main memory hardware is capable of reading/writing any individual byte. Here is an example of a main memory with 16-bit memory locations¹⁴. Note how the memory locations (rows) have even addresses.

Word Address	16 bit = 2 bytes							
0								
2								
4								
6								
8								
10								
12								
14								
16								
18								
20								

Byte Ordering

The ordering of bytes within a **multi-byte** data item defines the endian-ness of the architecture.

In BIG-ENDIAN systems the most significant byte of a multi-byte data item always has the lowest address, while the least significant byte has the highest address.

In LITTLE-ENDIAN systems, the least significant byte of a multi-byte data item always has the lowest address, while the most significant byte has the highest address.

In the following example, table cells represent bytes, and the cell numbers indicate the address of that byte in main memory. Note: by convention we draw the bytes within a memory word left-to-right for big-endian systems, and right-to-left for little-endian systems.

Word Address	Big-Endian				Word Address	Little-Endian			
0	0	1	2	3	0	3	2	1	0
4	4	5	6	7	4	7	6	5	4
8	8	9	10	11	8	11	10	9	8
12	12	13	14	15	12	15	14	13	12

MSB —————> LSB MSB —————> LSB

¹⁴ To avoid confusion we will use the term **memory word** for a word-sized memory location.

Note: an N-character ASCII string value is not treated as one large multi-byte value, but rather as N byte values, i.e. the first character of the string always has the lowest address, the last character has the highest address. This is true for both big-endian and little-endian. An N-character Unicode string would be treated as N two-byte value and each two-byte value would require suitable byte-ordering.

Example: Show the contents of memory at word address 24 if that word holds the number given by 122E 5F01H in both the big-endian and the little-endian schemes?

Big Endian				Little Endian					
MSB	----->			LSB	MSB	----->			LSB
24	25	26	27		27	26	25	24	
Word 24	12	2E	5F	01	Word 24	12	2E	5F	01

Example: Show the contents of main memory from word address 24 if those words hold the text JIM SMITH.

Big Endian				Little Endian			
+0	+1	+2	+3	+3	+2	+1	+0
Word 24	J	I	M	Word 24	M	I	J
Word 28	S	M	I	Word 28	T	I	M
Word 32	H	?	?	Word 32	?	?	H

The bytes labelled with ? are unknown. They could hold important data, or they could be don't care bytes – the interpretation is left up to the programmer.

Unfortunately computer systems¹⁵, in use today are split between those that are big-endian, and those that are little-endian¹⁶. This leads to problems when a big-endian computer wants to transfer data to a little-endian computer. Some architectures, for example the PowerPC and ARM, allow the endianness of the architecture to be changed programmatically.

Word Alignment

Although main-memories are generally organised as byte-addressed rows of words and accessed a row at a time, some architectures, allow the CPU to access any word-sized bit-group regardless of its byte address. We say that accesses that begin on a memory word boundary are **aligned accesses** while accesses that do not begin on word boundaries are **unaligned accesses**.

¹⁵ The interested student might want to read the paper, “On Holy Wars and a Plea for Peace”, D. Cohen, IEEE Computer, Vol 14, Pages 48-54, October 1981.

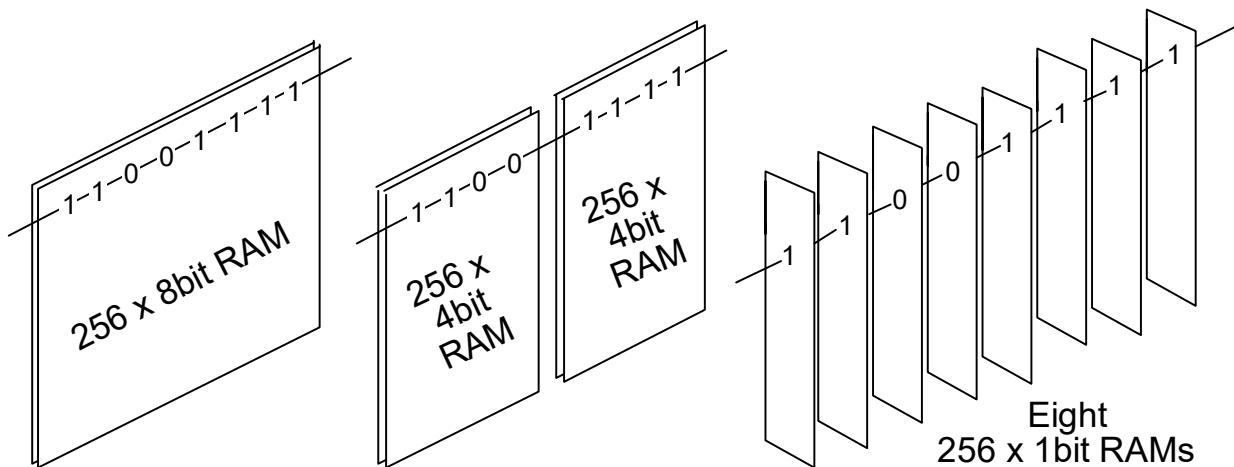
¹⁶ The Motorola 68000 architecture is big-endian, while the Intel Pentium architecture is little-endian.

Address	Memory (16-bit) word	
0	MSB LSB	Word starting at Address 0 is Aligned
2		
4		MSB
6	LSB	Word starting at Address 5 is Unaligned

Reading an unaligned word from RAM requires (i) reading of adjacent words, (ii) selecting the required bytes from each word and (iii) concatenating those bytes together => SLOW. Writing an unaligned word is more complex and slower¹⁷. For this reason some architectures prohibit unaligned word accesses. e.g. on the 68000 architecture, words must not be accessed starting from an odd-address (e.g. 1, 3, 5, 7 etc), on the SPARC architecture, 64-bit data items must have a byte address that is a multiple of 8.

Memory Modules, Memory Chips

So far, we have looked at the logical organisation of main memory. Physically RAM comes on small memory modules (little green printed circuit-boards about the size of a finger). A typical memory module holds 512MB to 2GB. The computer's motherboard will have slots to hold 2, 4 maybe 8 memory modules. Each memory module is itself comprised of several memory chips. For example here are 3 ways of forming a 256x8 bit memory module.



In the first case, main memory is built with a single memory chip. In the second, we use two memory chips, one gives us the most significant 4 bits, the other, the least significant 4 bits. In the third we use 8 memory chips, each chip gives us 1 bit - to read an 8 bit memory word, we would have to access all 8 memory chips simultaneously and concatenate the bits.

On PCs, memory modules are known as DIMMs (dual inline memory modules) and support 64-bit transfers. The previously generation of modules were called SIMMs (single inline memory modules) and supported 32-bit data transfers.

Example: Given Main Memory = 1M x 16 bit (word addressable),

RAM chips = 256K x 4 bit

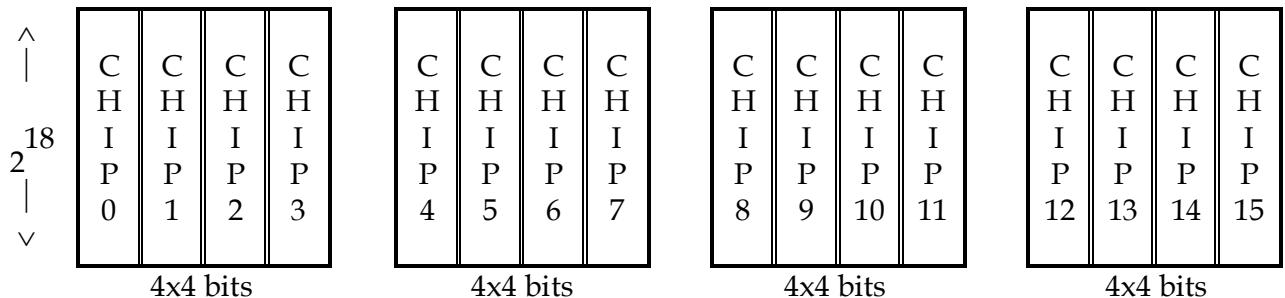
Module 0

Module 1

Module 2

Module 3

¹⁷ Describe a method for doing an unaligned word write operation.



$$\text{RAM chips per memory module} = \frac{\text{Width of Memory Word}}{\text{Width of RAM Chip}} = 16/4 = 4$$

18 bits are required to address a RAM chip (since $256K = 2^{18}$ = Length of RAM Chip)

A $1M \times 16$ bit word-addressed memory requires 20 address bits (since $1M = 2^{20}$)

Therefore 2 bits (=20–18) are needed to select a module.

The total number of RAM Chips = $(1M \times 16) / (256K \times 4) = 16$.

Total number of Modules = Total number of RAM chips / RamChipsPerModule = $16/4 = 4$

Interleaved Memory

When memory consists of several memory modules, some address bits will select the module, and the remaining bits will select a row within the selected module.

When the module selection bits are the least significant bits of the memory address we call the resulting memory a **low-order interleaved** memory.

When the module selection bits are the most significant bits of the memory address we call the resulting memory a **high-order interleaved** memory.

Interleaved memory can yield performance advantages if more than one memory module can be read/written at a time:-

- (I) for low-order interleave if we can read the same row in each module. This is good for a single multi-word access of sequential data such as program instructions, or elements in a vector,
- (ii) for high-order interleave, if different modules can be independently accessed by different units. This is good if the CPU can access rows in one module, while at the same time, the hard disk (or a second CPU) can access different rows in another module.

Example: Given that Main Memory = $1M \times 8$ bits, RAM chips = $256K \times 4$ bit. For this memory we would require $4 \times 2 = 8$ RAM chips. Each chip would require 18 address bits (ie. $2^{18} = 256K$) and the full $1M \times 16$ bit memory would require 20 address bits (ie. $2^{20} = 1M$)

CPU Organisation & Operation

The Fetch-Execute Cycle

The operation of the CPU¹⁸ is usually described in terms of the Fetch-Execute cycle¹⁹.

Fetch-Execute Cycle	The cycle raises many interesting questions, e.g.
Fetch the <i>Instruction</i>	What is an Instruction? Where is the Instruction? Why does it need to be fetched? Isn't it okay where it is? How does the computer keep track of instructions? Where does it put the instruction it has just fetched?
Increment the <i>Program Counter</i>	What is the Program Counter? What does the Program Counter count? Increment by how much? Where does the Program Counter point to after it is incremented?
Decode the Instruction	Why does the instruction need to be decoded? How does it get decoded?
Fetch the <i>Operands</i>	What are operands? What does it mean to fetch? Is this fetching distinct from the fetching in Step 1 above? Where are the operands? How many are there? Where do we put the operands after we fetch them?
Perform the Operation	Is this the main step? Couldn't the computer simply have done this part? What part of the CPU performs this operation?
Store the results	What results? Where from? Where to?
Repeat forever	Repeat what? Repeat from where? Is it really an infinite loop? Why? How do these steps execute any instructions at all?

In order to appreciate the operation of a computer we need to answer such questions and to consider in more detail the organisation of the CPU.

Representing Programs

Each complex task carried out by a computer needs to be broken down into a sequence of simpler tasks and a **binary machine instruction** is needed for the most primitive tasks. Consider a task that adds two numbers²⁰, held in memory locations designated by B and C²¹ and stores the result in memory location designated by A.

$$A = B + C$$

¹⁸ Central Processing Unit.

¹⁹ Sometimes called the Fetch-Decide-Execute Cycle.

²⁰ Let's assume they are held in two's complement form.

²¹ A, B and C are actually main memory **addresses**, i.e. natural binary numbers.

This assignment can be broken down (compiled) into a sequence of simpler tasks or **assembly instructions**, e.g:

Assembly Instruction	Effect
LOAD R2, B	Copy the contents of memory location designated by B into Register 2
ADD R2, C	Add the contents of the memory location designated by C to the contents of Register 2 and put the result back into Register 2
STORE R2, A	Copy the contents of Register 2 into the memory location designated by A.

Each of these assembly instructions needs to be encoded into binary for execution by the Central Processing Unit (CPU). Let's try this encoding for a simple architecture called TOY1.

TOY1 Architecture

TOY1 is a fictitious architecture with the following characteristics:

1024 x 16-bit words of RAM maximum. RAM is word-addressable.

4 general purpose registers R0, R1, R2 and R3. Each general purpose register is 16-bits (the same size as a memory location).

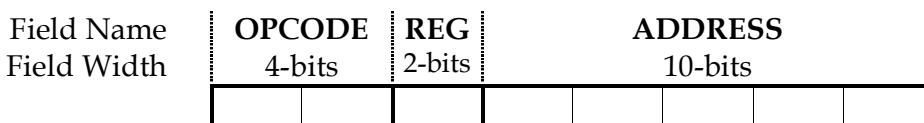
16 different instructions that the CPU can decode and execute, e.g. LOAD, STORE, ADD, SUB and so on. These different instructions constitute the **Instruction Set** of the Architecture.

The representation for integers will be two's complement.

For this architecture, the architect (us) needs to define a coding scheme²² for instructions. This is termed the **Instruction Format**. Lets look at an example before we consider how we arrived at it. Here's our instruction format for TOY1:

TOY1 Instruction Format

TOY1 instructions are 16-bits (so they will fit into a main-memory word). Each instruction is divided into a number of **instruction fields** that encode a different piece of information for the CPU.



The **OPCODE**²³ field identifies the CPU operation required. Since TOY1 only supports 16 instructions, these can be encoded as a 4-bit natural number. For TOY1, opcodes 1 to 4 will be²⁴:

0001 = LOAD

0010 = STORE

0011 = ADD

0100 = SUB

²² Most architectures actually have different instruction formats for different categories of instruction.

²³ Operation Code

²⁴ The meaning of CPU operations is defined in the Architecture's Instruction Set Manual.

The **REG** field defines a General CPU Register. Arithmetic operations will use 1 register **operand** and 1 main memory **operand**, results will be written back to the register. Since TOY1 has 4 registers; these can be encoded as a 2-bit natural number:

00 = Register 0 01 = Register 1 10 = Register 2 11 = Register 3

The **ADDRESS** field defines the address of a word in RAM. Since TOY1 can have upto 1024 memory locations; a memory address can be encoded as a 10-bit natural number.

If we define addresses 200H, 201H and 202H for A, B and C, we can encode the example above as:

Assembly Instruction	Machine Instruction
LOAD R2, [201H]	0001 10 10 0000 0001
ADD R2, [202H]	0011 10 10 0000 0010
STORE R2, [200H]	0010 10 10 0000 0000

Memory Placement of Program and Data

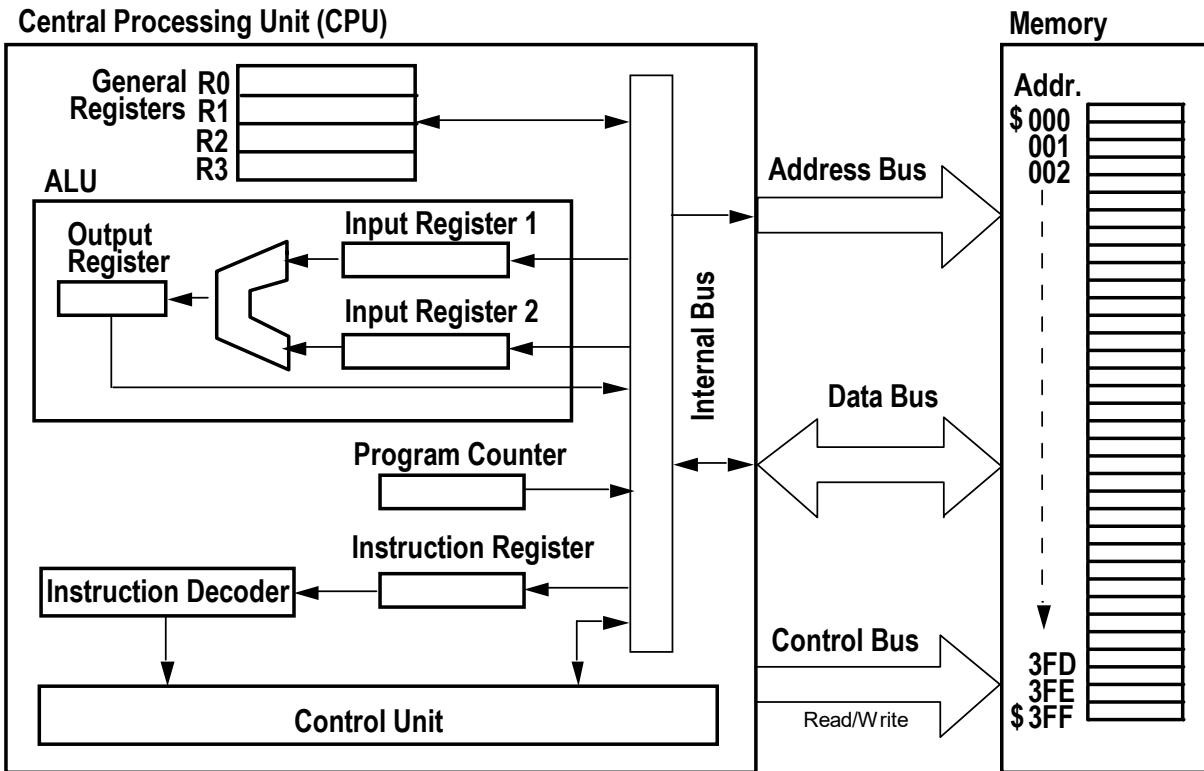
In order to execute a TOY1 program, its instructions and data needs to placed within main memory²⁵. We'll place our 3-instruction program in memory starting at address 080H and we'll place the variables A, B and C at memory words 200H, 201H, and 202H respectively. Such placement results in the following memory layout prior to program execution. For convenience, memory addresses and memory contents are also given in hex.

Memory Address in binary & hex	Machine Instruction					Assembly Instruction
	OP	Reg	Address			
0000 1000 0000 0 8 0	0001	10	10	0000	0001	LOAD R2, [201H]
	1	A	0	1		
0000 1000 0001 0 8 1	0011	10	10	0000	0010	ADD R2, [202H]
	3	A	0	2		
0000 1000 0010 0 8 2	0010	10	10	0000	0000	STORE R2, [200H]
	2	A	0	0		
Etc		Etc			Etc	
0010 0000 0000 2 0 0	0000	0000	0000	0000	0000	A = 0
	0	0	0	0	0	
0010 0000 0001 2 0 1	0000	0000	0000	1001		B = 9
	0	0	0	9		
0010 0000 0010 2 0 2	0000	0000	0000	0110		C = 6
	0	0	0	6		

Of course, the big question is “How is such a program executed by the TOY1 CPU?”

²⁵ The Operating System software is normally responsible for undertaking this task.

CPU Organisation



The **Program Counter (PC)** is a special register that holds the **address** of the next instruction to be fetched from Memory (for TOY1, the PC is 10-bits wide). The PC is incremented²⁶ to "point to" the next instruction while an instruction is being fetched from main memory.

The **Instruction Register (IR)** is a special register that holds each instruction after it is fetched from main memory. For TOY1, the IR is 16-bits since instructions are 16-bit wide.

The **Instruction Decoder** is a CPU component that decodes and interprets the contents of the Instruction Register, i.e. its splits whole instruction into fields for the Control Unit to interpret. The Instruction decoder is often considered to be a part of the Control Unit.

The **Control Unit** is the CPU component that co-ordinates all activity within the CPU. It has connections to all parts of the CPU, and includes a sophisticated timing circuit.

The **Arithmetic & Logic Unit (ALU)** is the CPU component that carries out arithmetic and logical operations e.g. addition, comparison, boolean AND/OR/NOT.

The **ALU Input Registers 1 & 2** are special registers that hold the input operands for the ALU.

The **ALU Output Register** is a special register that holds the result of an ALU operation. On completion of an ALU operation, the result is copied from the ALU Output register to its final destination, e.g. to a CPU register, or main-memory, or to an I/O device.

The **General Registers R0, R1, R2, R3** are available for the programmer to use in his/her programs. Typically the programmer tries to maximise the use of these registers in order to speed program execution. For TOY1, the general registers are the same size as memory locations, i.e. 16-bits.

²⁶ By the appropriate number of memory words.

The **Buses** serve as communication highways for passing information within the CPU (CPU internal bus) and between the CPU and the main memory (the **address bus**, the **data bus**, and the **control bus**). The address bus is used to send addresses from the CPU to the main memory; these addresses indicate the memory location the CPU wishes to read or write. Unlike the address bus, the data bus is bi-directional; for writing, the data bus is used to send a word from the CPU to main-memory; for reading, the data bus is used to send a word from main-memory to the CPU. For TOY1, the Control bus²⁷ is used to indicate whether the CPU wishes to read from a memory location or write to a memory location. For simplicity we've omitted two special registers, the **Memory Address Register (MAR)** and the **Memory Data Register (MDR)**. These registers lie at the boundary of the CPU and Address bus and Data bus respectively and serve to buffer data to/from the buses.

Buses can normally transfer more than 1-bit at a time. For the TOY1, the address bus is 10-bits (the size of an address), the data bus is 16-bits (size of a memory location), and the control bus is 1-bit (to indicate a memory read operation or a memory write operation).

Interlude: the Von Neumann Machine Model

Most computers conform to the von Neumann's machine model, named after the Hungarian-American mathematician John von Neumann (1903-57).

In von Neumann's model, a computer has **3 subsystems** (i) a CPU, (ii) a main memory, and (iii) an I/O system. The main memory holds the program as well as data and the computer is allowed to manipulate its own program²⁸. In the von-Neumann model, instructions are executed **sequentially** (one at a time).

In the von-Neumann model a **single path** exists between the control unit and main-memory, this leads to the so-called "**von Neumann bottleneck**" since memory fetches are the slowest part of an instruction they become the bottleneck in any computation.

Instruction Execution (Fetch-Execute-Cycle Micro-steps)

In order to execute our 3-instruction program, the control unit has to issue and coordinate a series of micro-instructions. These micro-instructions form the fetch-execute cycle. For our example we will assume that the Program Counter register (PC) already holds the address of the first instruction, namely 080H.

²⁷ Most control-buses are wider than a single bit, these extras bits are used to provide more sophisticated memory operations and I/O operations.

²⁸ This type of manipulation is not regarded as a good technique for general assembly programming.

LOAD R2, [201H]

0000	1000	0000	<table border="1" style="display: inline-table; vertical-align: middle;"> <tr> <td>0001</td><td>10</td><td>10</td><td>0000</td><td>0001</td></tr> <tr> <td>1</td><td>A</td><td>0</td><td>0</td><td>1</td></tr> </table>	0001	10	10	0000	0001	1	A	0	0	1	Copy the value in memory word 201H into Register 2
0001	10	10	0000	0001										
1	A	0	0	1										
0	8	0												

Control Unit Action FETCH INSTRUCTION²⁹

Control Unit Action	Data flows		
PC to Address Bus ³⁰	080H	→	080H Address Bus
0 to Control Bus ³¹	0	→	0 Control Bus
Address Bus to Memory	080H	→	080H Memory
Control Bus to Memory	0	READ →	0 Memory
Increment PC ³²	080	INC →	081H PC becomes PC+1 ³³
Memory [080H] to Data Bus	1A01H	→	1A01H Data Bus
Data Bus to Instruction Register	1A01H	→	1A01H Instruction Register

DECODE INSTRUCTION

IR to Instruction Decoder	1A01H	→	1A01H	Instruction Decoder
Instruction Decoder to Control Unit ³⁴	1, 2, 201H	→	1, 2, 201H	Control Unit

EXECUTE INSTRUCTION³⁵

Control Unit to Address Bus	201H	→	201H	Address Bus
0 to Control Bus	0	→	0	Control Bus
Address Bus to Memory	201H	→	201H	Memory
Control Bus to Memory	0	READ →	0	Memory
Memory [201H] to Data bus	0009H	→	0009H	Data Bus
Data Bus to Register 2	0009H	→	0009H	Register 2

²⁹ The micro-steps in the Fetch and Decode phases are common for all instructions.

³⁰ This and the next 4 micro-steps initiate a fetch of the next instruction to be executed, which is to be found at memory address 80H. In practice a Memory Address Register (MAR) acts as an intermediate buffer for the Address, similarly a Memory Data Register (MDR) buffers data to/from the data bus.

³¹ We will use 0 for a memory READ request, and 1 for a memory WRITE request.

³² For simplicity, we will assume that the PC is capable of performing the increment internally. If not, the Control Unit would have to transfer the contents of the PC to the ALU, get the ALU to perform the increment and send the results back to the PC. All this while we are waiting for the main-memory to return the word at address 80H.

³³ Since TOY1's main-memory is word-addressed, and all instructions are 1 word. If main-memory was byte-addressed we would need to add 2.

³⁴ The Instruction decoder splits the instruction into the individual instruction fields OPCODE, REG and ADDRESS for interpretation by the Control Unit.

³⁵ The micro-steps for the execute phase actually perform the operation.

ADD R2, [202H]

0000	1000	0001	<table border="1" style="display: inline-table; vertical-align: middle;"> <tr> <td>0011</td><td>10</td><td>1000000002</td></tr> <tr> <td>3</td><td>A</td><td>0 2</td></tr> </table>	0011	10	1000000002	3	A	0 2
0011	10	1000000002							
3	A	0 2							
0	8	1							

Add³⁶ the value in memory word 202H to Register 2

Control Unit Action

FETCH INSTRUCTION

		Data flows		
PC to Address Bus	081H	→	081H	Address Bus
0 to Control Bus	0	→	0	Control Bus
Address Bus to Memory	081H	→	081H	Memory
Control Bus to Memory	0	READ	0	Memory
Increment PC	081H	INC	082H	PC becomes PC+1
Memory [081H] to Data Bus	3A02H	→	3A02H	Data Bus
Data Bus to Instruction Register	3A02H	→	3A02H	Instruction Register

DECODE INSTRUCTION

IR to Instruction Decoder	3A02H	→	3A02H	Instruction Decoder
Instruction Decoder to Control Unit	3, 2, 202H	→	3, 2, 202H	Control Unit

EXECUTE INSTRUCTION

Register 2 to ALU Input Reg 1	0009	→	0009	ALU Input Reg 1
Control Unit to Address Bus	202H	→	202H	Address Bus
0 to Control Bus	0	→	0	Control Bus
Address Bus to Memory	202H	→	202H	Memory
Control Bus to Memory	0	READ	0	Memory
Memory [202H] to Data bus	0006H	→	0006H	Data Bus
Data Bus to ALU Input Reg 2	0006H	→	0006H	ALU Input Reg 2
Control Unit to ALU		ADD	000FH	Output Register
ALU Output Reg to Register 2	000F	→	000FH	Register 2

³⁶ Using two's complement arithmetic.

STORE R2, [200H]

0000	1000	0001		[0010]	[10]	1000000000
0	8	2		2	A	0 0

Copy the value in Register 2 into memory word 202H

Control Unit Action

FETCH INSTRUCTION

Control Unit Action	Data flows		
PC to Address Bus	082H	→	082H Address Bus
0 to Control Bus	0	→	0 Control Bus
Address Bus to Memory	082H	→	082H Memory
Control Bus to Memory	0	READ →	0 Memory
Increment PC	082H	INC →	083H PC becomes PC+1
Memory [082] to Data Bus	2A00H	→	2A00H Data Bus
Data Bus to Instruction Register	2A00H	→	2A00H Instruction Register

DECODE INSTRUCTION

IR to Instruction Decoder	2A00	→	2A00	Instruction Decoder
Instruction Decoder to Control Unit	2, 2, 200H	→	2, 2, 200H	Control Unit

EXECUTE INSTRUCTION

Register 2 to Data Bus	000FH	→	000FH	Data Bus
Control Unit to Address Bus	200H	→	200H	Address Bus
1 to Control Bus	1	→	1	Control Bus
Data Bus to Memory	000FH	→	000FH	Memory
Address Bus to Memory	200H	→	200H	Memory
Control Bus to Memory	1	WRITE →	1	Memory

MEMORY ORGANIZATION

Topics for discussion

- Memory organization
- Memory hierarchy
- Types of memory
- Memory management hardware

MEMORY ORGANIZATION

- Memory hierarchy
- Main memory
- Auxiliary memory
- Associative memory
- Cache memory
 - Storage technologies and trends
 - Locality of reference
 - Caching in the memory hierarchy
- Virtual memory
- Memory management hardware.

RANDOM-ACCESS MEMORY (RAM)

- Key features
 - RAM is packaged as a chip.
 - Basic storage unit is a **cell** (one bit per cell).
 - Multiple RAM chips form a memory.
- Static RAM (**SRAM**)
 - Each cell stores bit with a six-transistor circuit.
 - Retains value indefinitely, as long as it is kept powered.
 - Relatively insensitive to disturbances such as electrical noise.
 - Faster and more expensive than DRAM.

Cont...

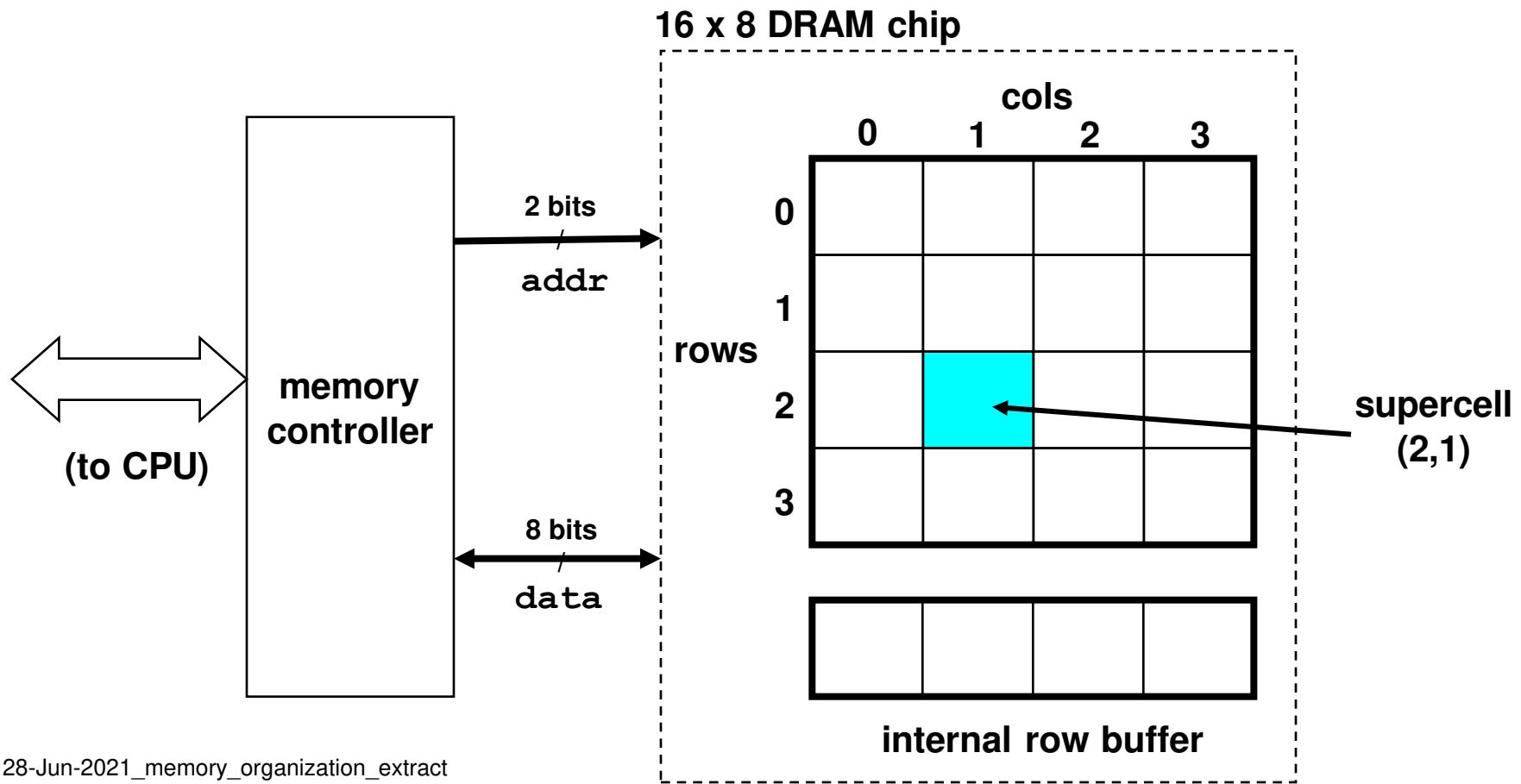
- Dynamic RAM (**DRAM**)
 - Each cell stores bit with a capacitor and transistor.
 - Value must be refreshed every 10-100 ms.
 - Sensitive to disturbances.
 - Slower and cheaper than SRAM.

SRAM VS DRAM SUMMARY

	Tran. per bit	Access time	Persist?	Sensitive?	Cost	Applications
SRAM	6	1X	Yes	No	100x	cache memories
DRAM	1	10X	No	Yes	1X	Main memories, frame buffers

CONVENTIONAL DRAM ORGANIZATION

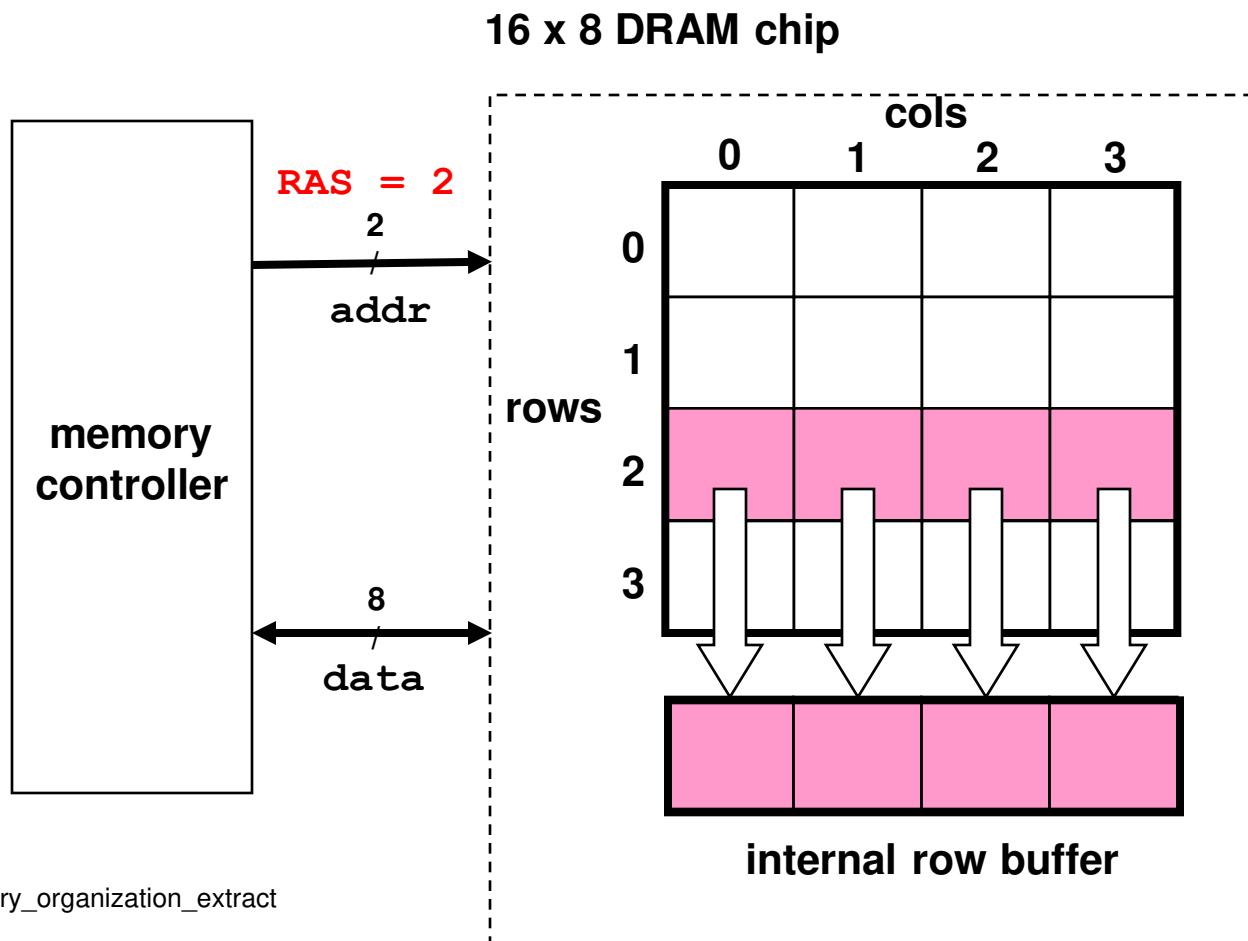
- $d \times w$ DRAM:
 - dw total bits organized as d **supercells** of size w bits



READING DRAM SUPERCELL (2,1)

Step 1(a): Row access strobe (**RAS**) selects row 2.

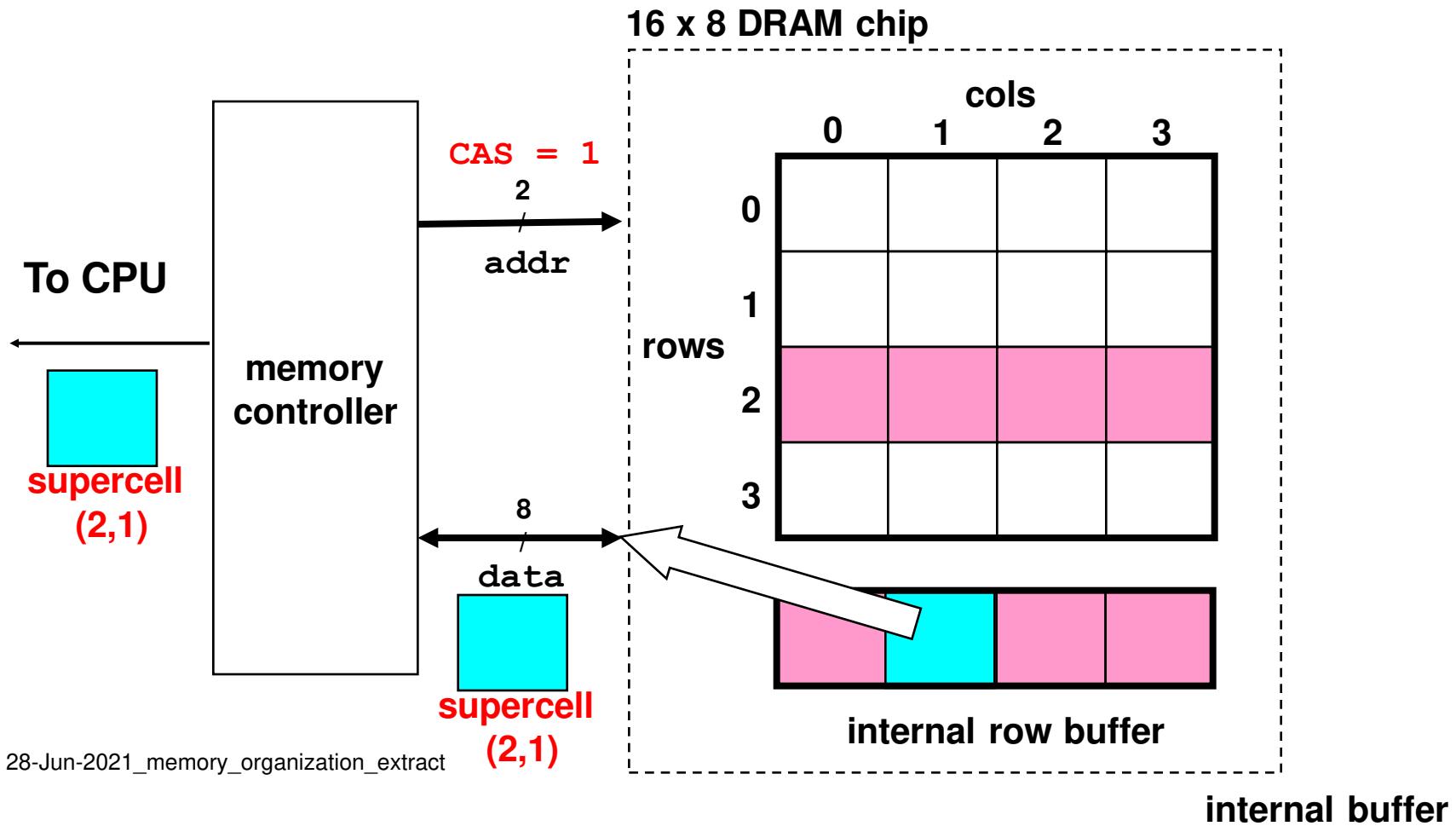
Step 1(b): Row 2 copied from DRAM array to row buffer.



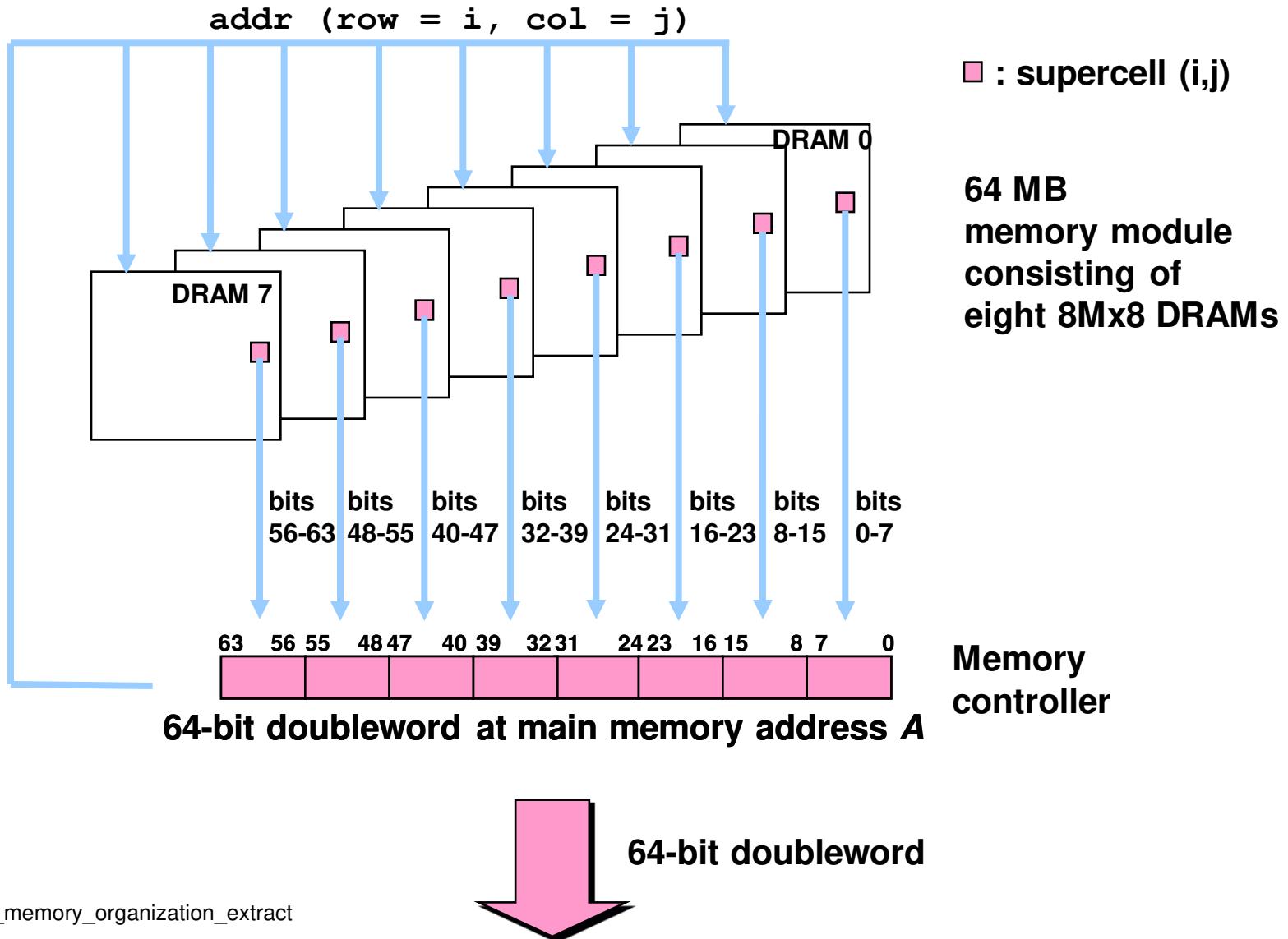
READING DRAM SUPERCELL (2,1)

Step 2(a): Column access strobe (**CAS**) selects column 1.

Step 2(b): Supercell (2,1) copied from buffer to data lines, and eventually back to the CPU.



MEMORY MODULES



ENHANCED DRAMS

- All enhanced DRAMs are built around the conventional DRAM core.
 - Fast page mode DRAM (**FPM DRAM**)
 - Access contents of row with [RAS, CAS, CAS, CAS, CAS] instead of [(RAS,CAS), (RAS,CAS), (RAS,CAS), (RAS,CAS)].
 - Extended data out DRAM (**EDO DRAM**)
 - Enhanced FPM DRAM with more closely spaced CAS signals.
 - Synchronous DRAM (**SDRAM**)
 - Driven with rising clock edge instead of asynchronous control signals.

Cont...

- Double data-rate synchronous DRAM (**DDR SDRAM**)
 - Enhancement of SDRAM that uses both clock edges as control signals.
- Video RAM (**VRAM**)
 - Like FPM DRAM, but output is produced by shifting row buffer
 - Dual ported (allows concurrent reads and writes)

NONVOLATILE MEMORIES

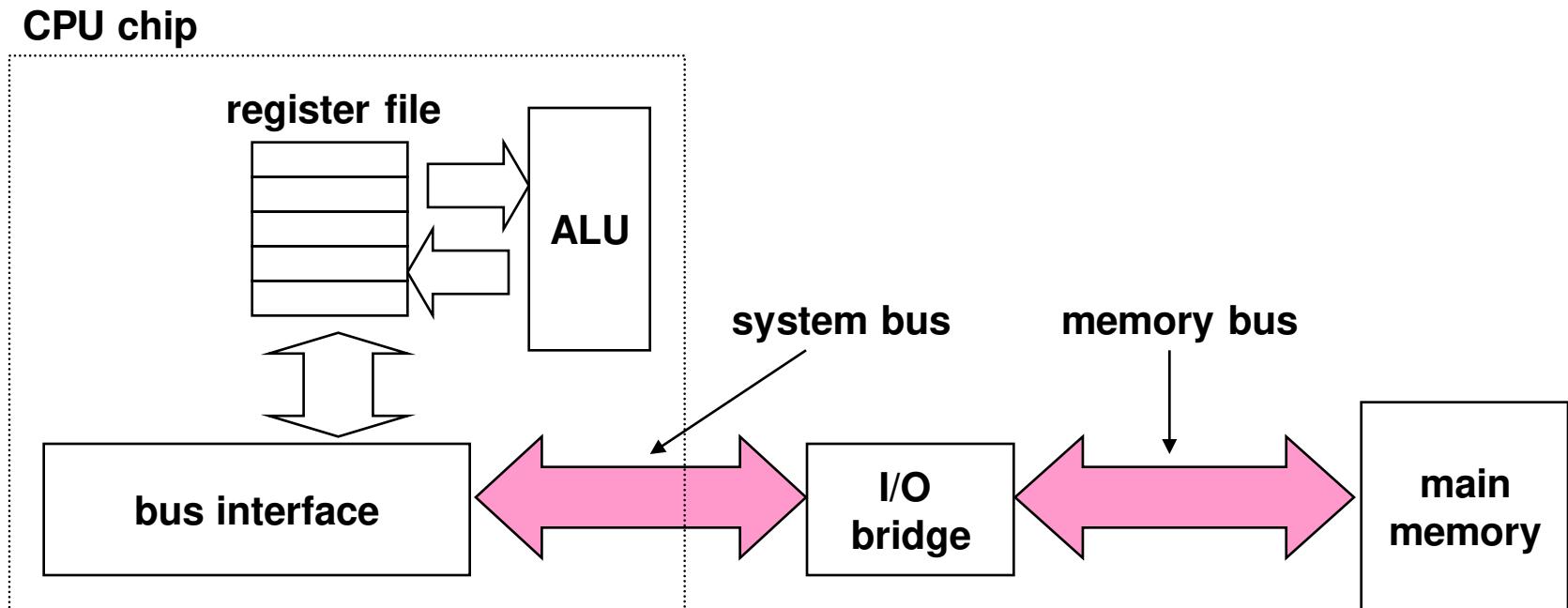
- DRAM and SRAM are volatile memories
 - Lose information if powered off.
- Nonvolatile memories retain value even if powered off.
 - Generic name is read-only memory (**ROM**).
 - Misleading because some ROMs can be read and modified.
- Types of ROMs
 - Programmable ROM (**PROM**)
 - Eraseable programmable ROM (**EPROM**)
 - Electrically erasable PROM (**EEPROM**)
 - Flash memory

Cont...

- **Firmware**
 - Program stored in a ROM
 - Boot time code, BIOS (basic input/output system)
 - graphics cards, disk controllers.

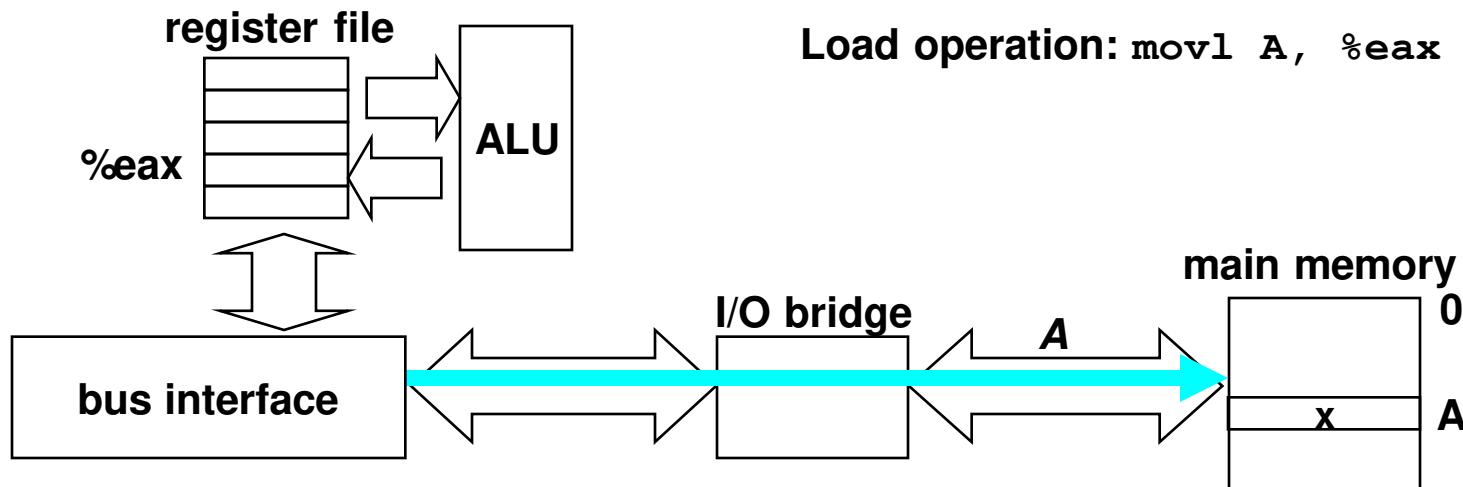
TYPICAL BUS STRUCTURE CONNECTING CPU AND MEMORY

- A **bus** is a collection of parallel wires that carry address, data, and control signals.
- Buses are typically shared by multiple devices.



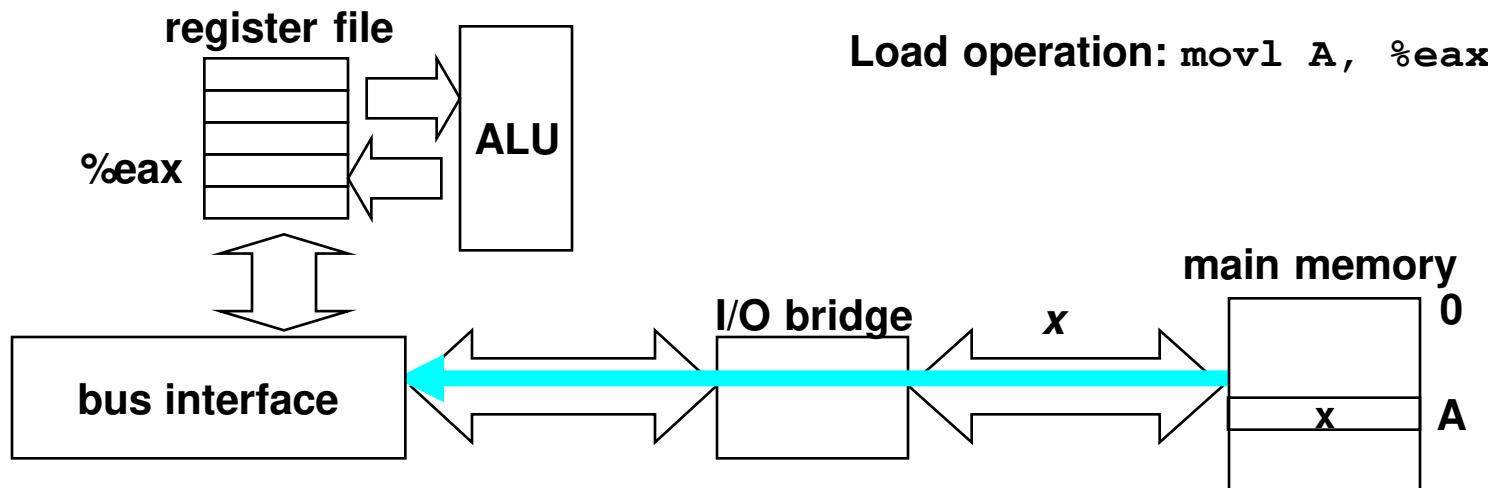
MEMORY READ TRANSACTION (1)

- CPU places address A on the memory bus.



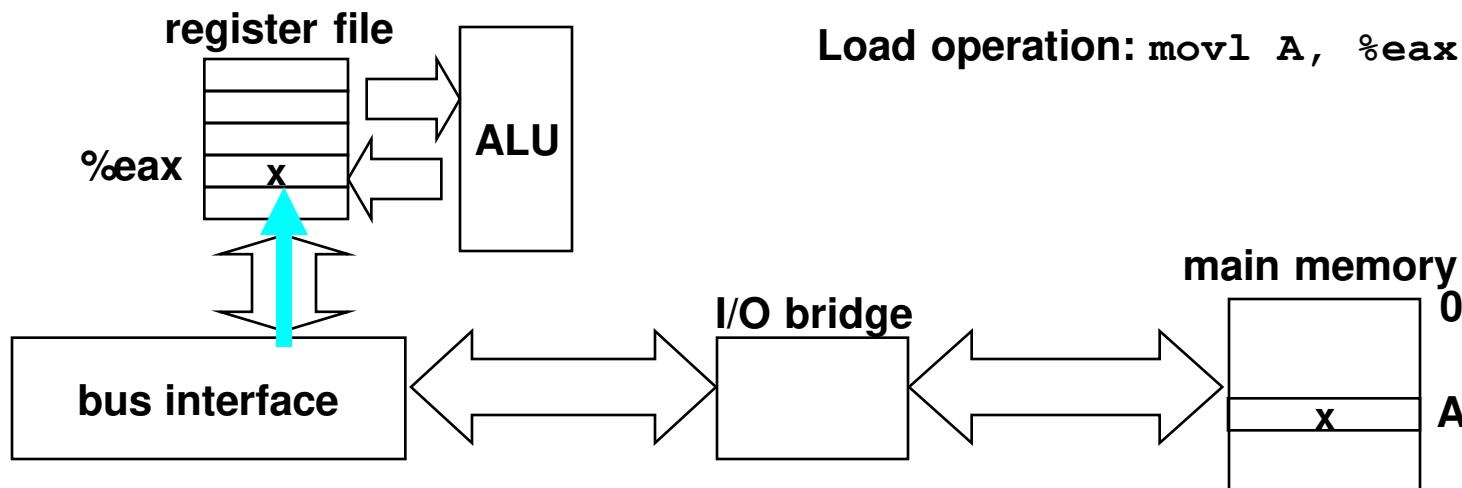
MEMORY READ TRANSACTION (2)

- Main memory reads A from the memory bus, retrieves word x, and places it on the bus.



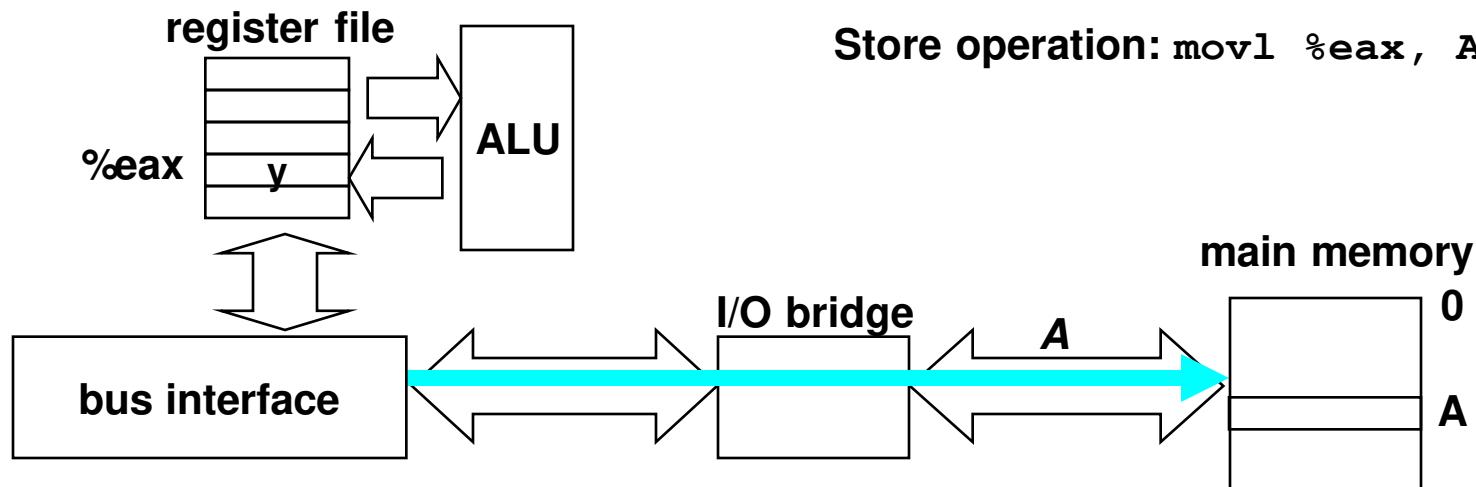
MEMORY READ TRANSACTION (3)

- CPU read word x from the bus and copies it into register $\%eax$.



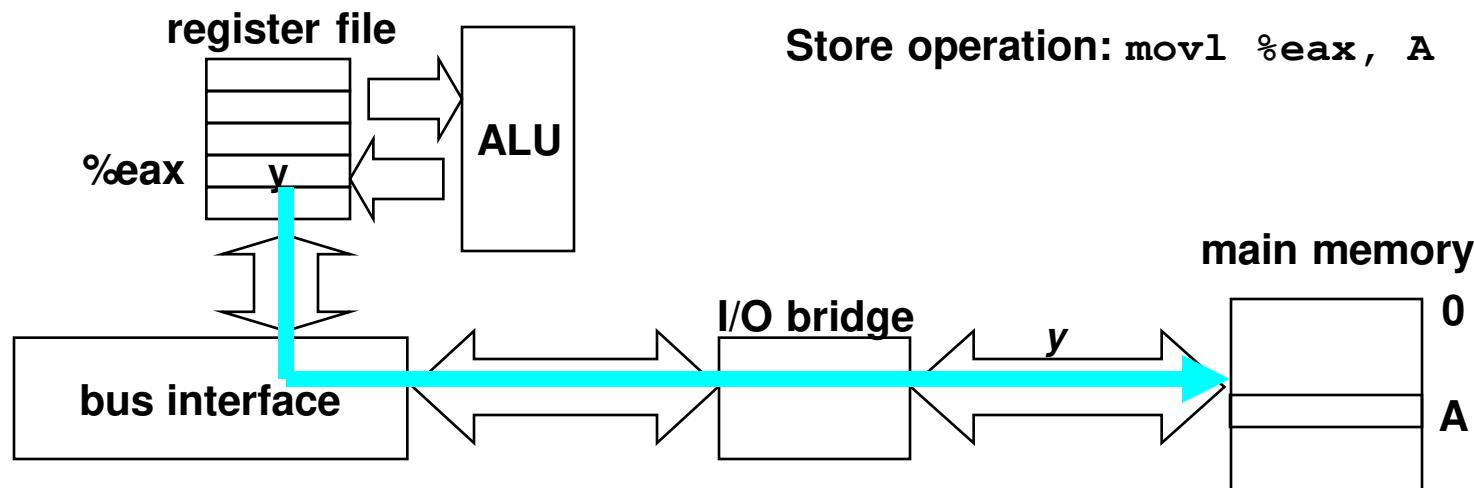
MEMORY WRITE TRANSACTION (1)

- CPU places address A on bus. Main memory reads it and waits for the corresponding data word to arrive.



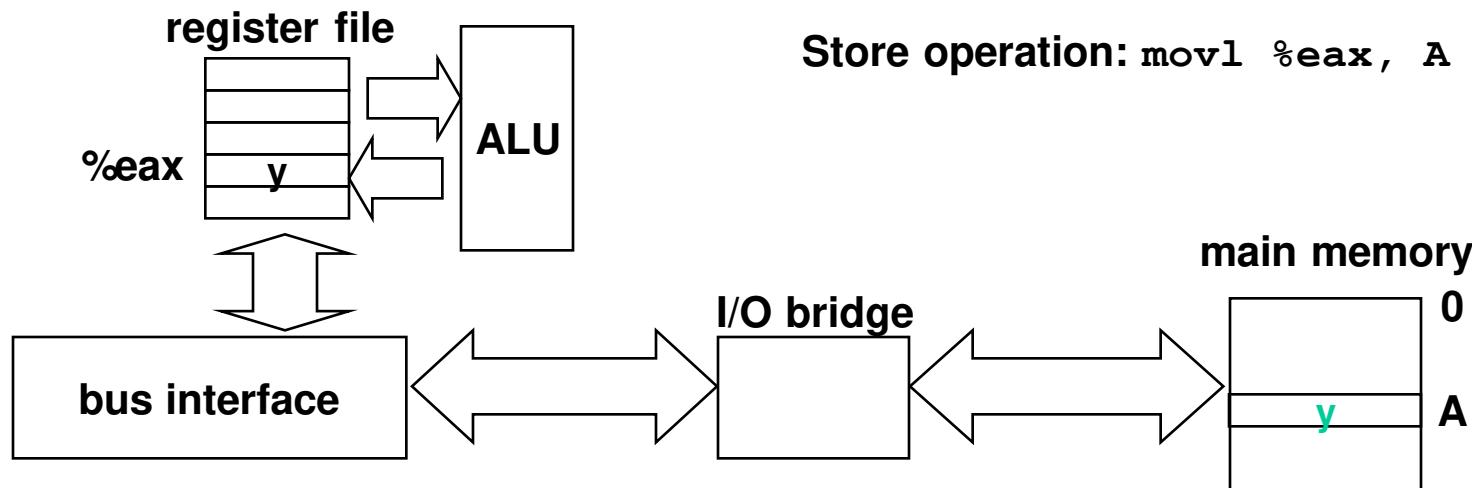
MEMORY WRITE TRANSACTION (2)

- CPU places data word y on the bus.



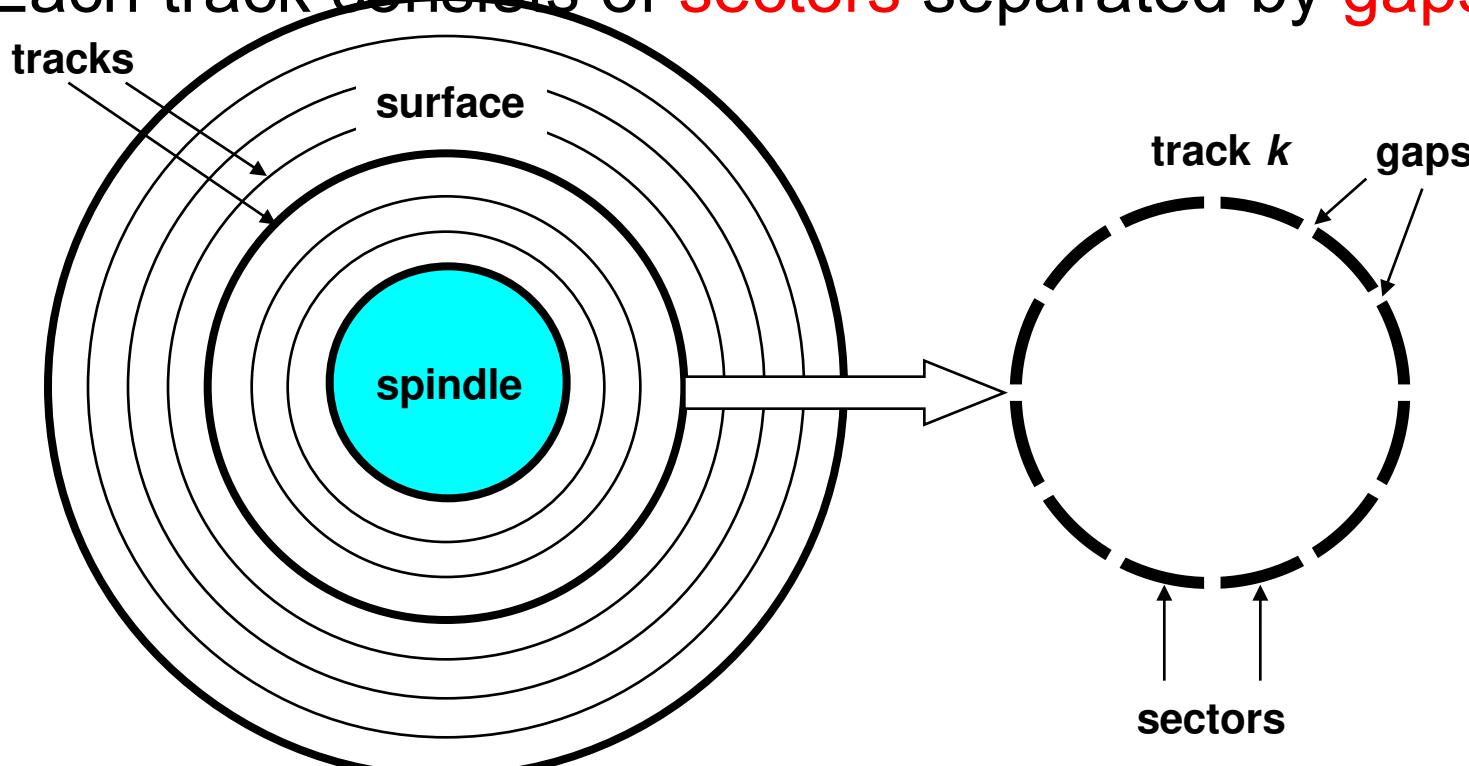
MEMORY WRITE TRANSACTION (3)

- Main memory read data word y from the bus and stores it at address A.



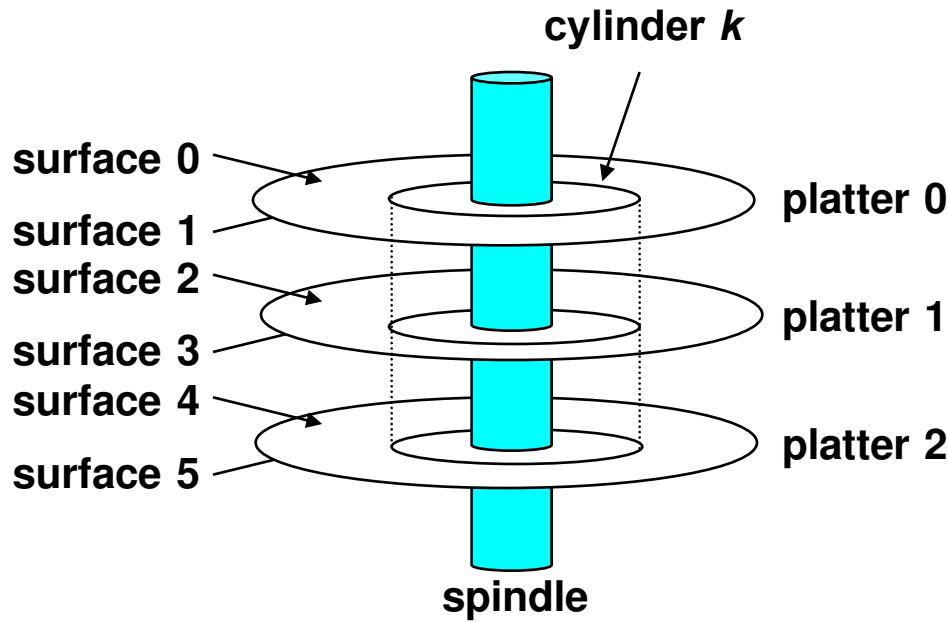
DISK GEOMETRY

- Disks consist of **platters**, each with two **surfaces**.
- Each surface consists of concentric rings called **tracks**.
- Each track consists of **sectors** separated by **gaps**.



DISK GEOMETRY (MULTIPLE-PLATTER VIEW)

- Aligned tracks form a cylinder.



DISK CAPACITY

- **Capacity:** maximum number of bits that can be stored.
 - Vendors express capacity in units of gigabytes (GB), where $1\text{ GB} = 10^9$.
- Capacity is determined by these technology factors:
 - **Recording density** (bits/in): number of bits that can be squeezed into a 1 inch segment of a track.
 - **Track density** (tracks/in): number of tracks that can be squeezed into a 1 inch radial segment.
 - **Areal density** (bits/in²): product of recording and track density.

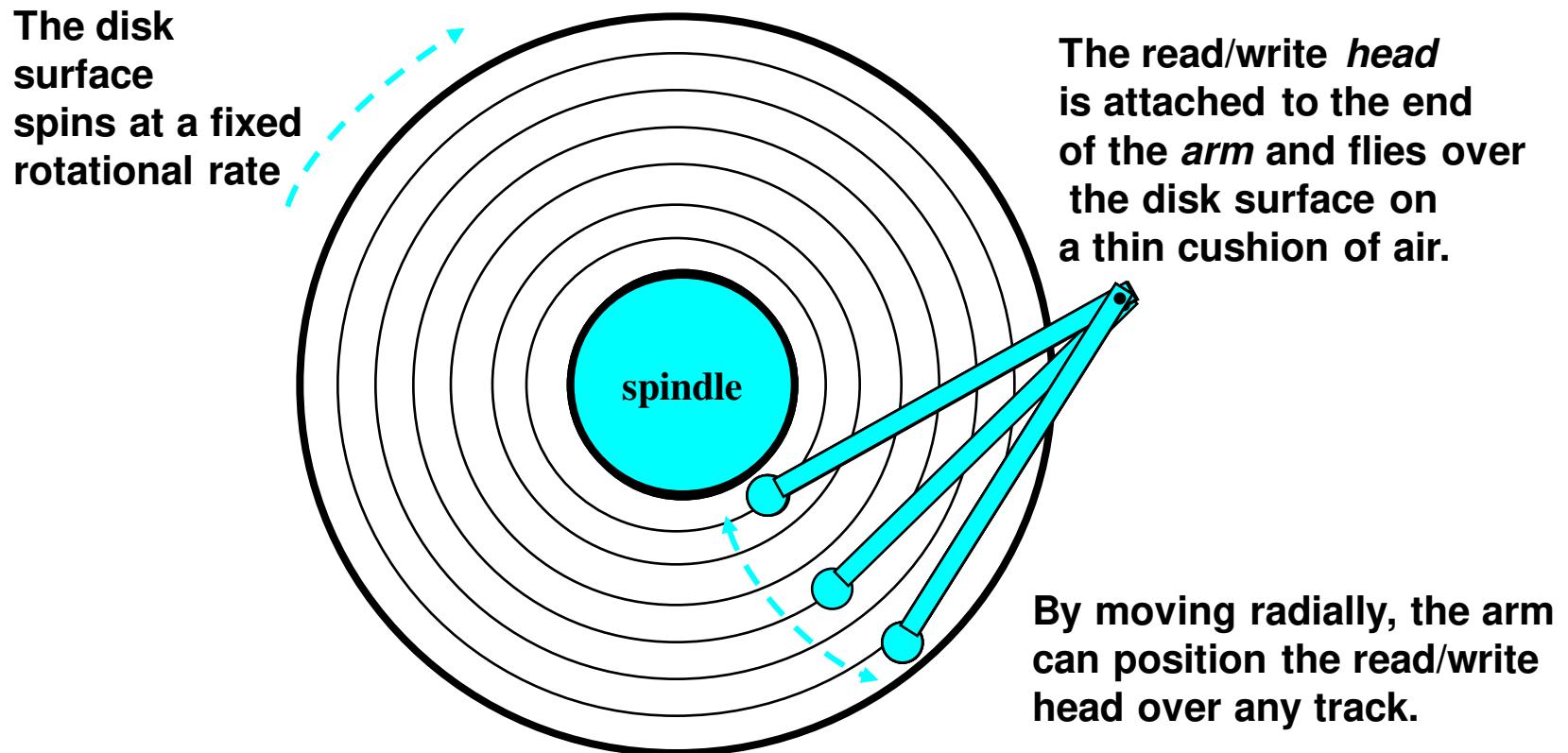
Cont...

- Modern disks partition tracks into disjoint subsets called **recording zones**
 - Each track in a zone has the same number of sectors, determined by the circumference of innermost track.
 - Each zone has a different number of sectors/track

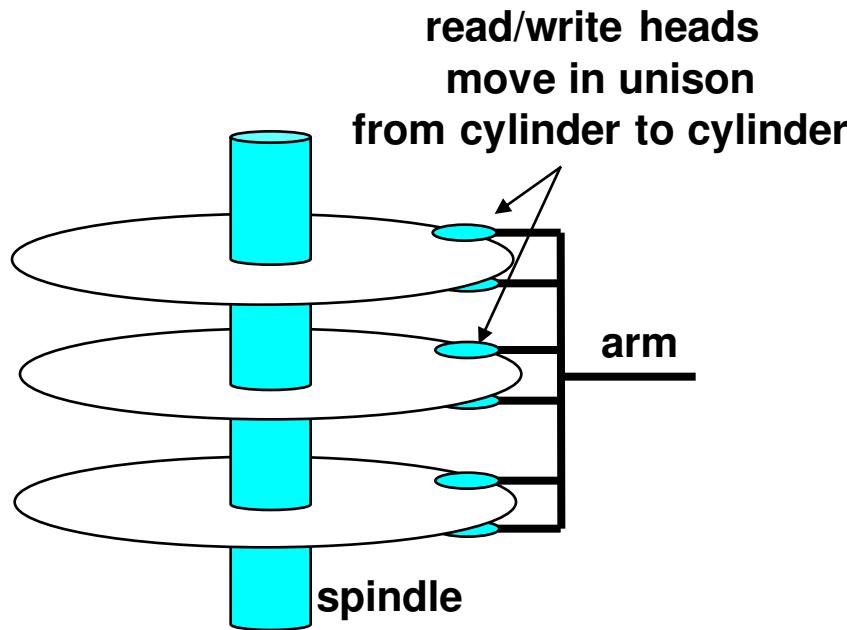
COMPUTING DISK CAPACITY

- Capacity = (# bytes/sector) x (avg. # sectors/track) x (# tracks/surface) x (# surfaces/platter) x (# platters/disk)
- Example:
 - 512 bytes/sector
 - 300 sectors/track (on average)
 - 20,000 tracks/surface
 - 2 surfaces/platter
 - 5 platters/disk
- Capacity = $512 \times 300 \times 20000 \times 2 \times 5$
 $= 30,720,000,000 = 30.72 \text{ GB}$

DISK OPERATION (SINGLE-PLATTER VIEW)



DISK OPERATION (MULTI-PLATTER VIEW)



DISK ACCESS TIME

- Average time to access some target sector approximated by :
 - $T_{\text{access}} = T_{\text{avg seek}} + T_{\text{avg rotation}} + T_{\text{avg transfer}}$
- **Seek time ($T_{\text{avg seek}}$)**
 - Time to position heads over cylinder containing target sector.
 - Typical $T_{\text{avg seek}} = 9 \text{ ms}$
- **Rotational latency ($T_{\text{avg rotation}}$)**
 - Time waiting for first bit of target sector to pass under r/w head.
 - $T_{\text{avg rotation}} = 1/2 \times 1/\text{RPMs} \times 60 \text{ sec/1 min}$

DISK ACCESS TIME

- Transfer time ($T_{avg\ transfer}$)
 - Time to read the bits in the target sector.
 - $T_{avg\ transfer} = 1/RPM \times 1/(avg\ # \ sectors/track) \times 60\ secs/1\ min.$

DISK ACCESS TIME EXAMPLE

- Given:
 - Rotational rate = 7,200 RPM
 - Average seek time = 9 ms.
 - Avg # sectors/track = 400.
- Derived:
 - $T_{\text{avg rotation}} = 1/2 \times (60 \text{ secs}/7200 \text{ RPM}) \times 1000 \text{ ms/sec} = 4 \text{ ms.}$
 - $T_{\text{avg transfer}} = 60/7200 \text{ RPM} \times 1/400 \text{ secs/track} \times 1000 \text{ ms/sec} = 0.02 \text{ ms}$
 - $T_{\text{access}} = 9 \text{ ms} + 4 \text{ ms} + 0.02 \text{ ms}$

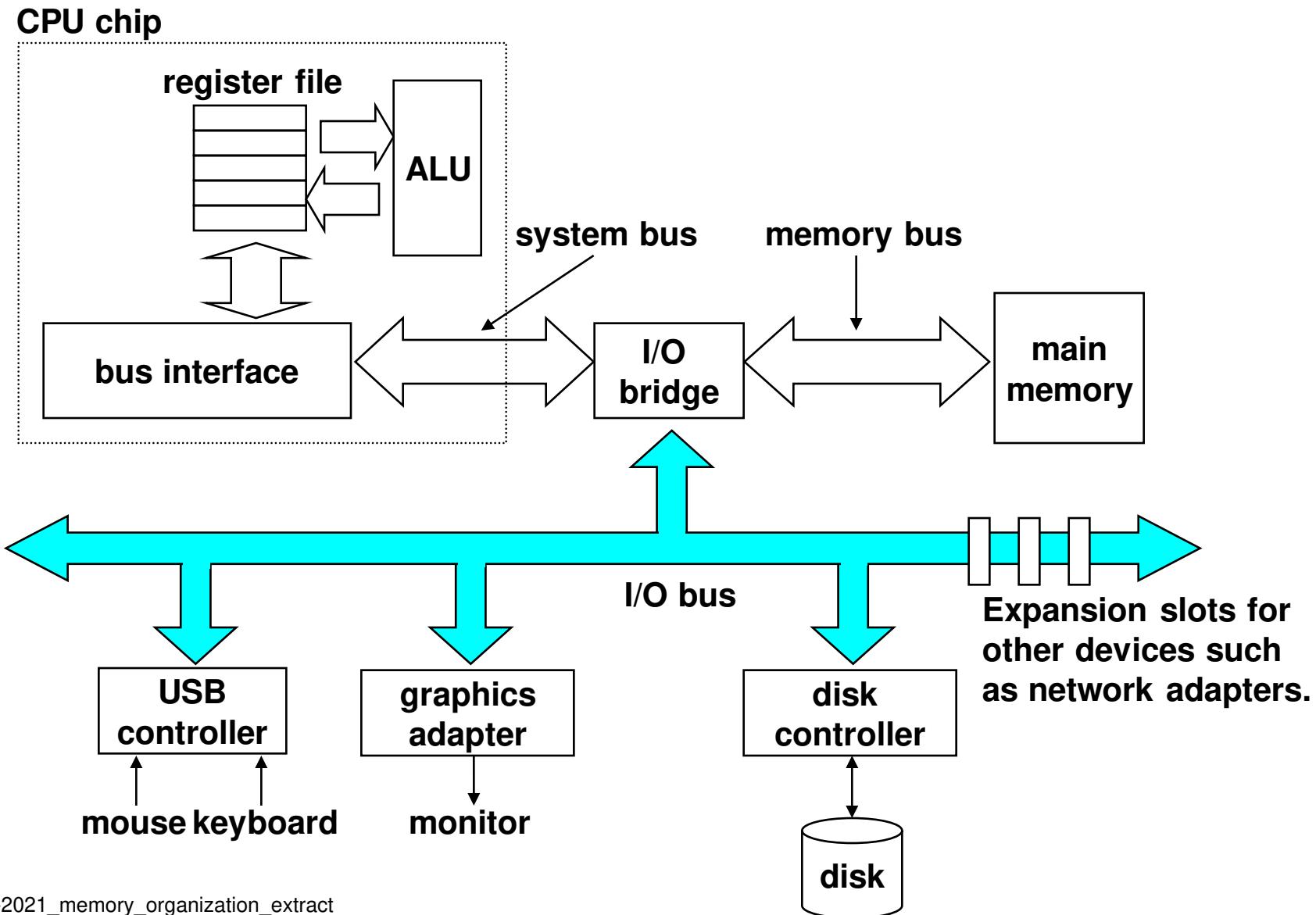
DISK ACCESS TIME EXAMPLE

- Important points:
 - Access time dominated by seek time and rotational latency.
 - First bit in a sector is the most expensive, the rest are free.
 - SRAM access time is about 4 ns/double word, DRAM about 60 ns
 - Disk is about 40,000 times slower than SRAM,
 - 2,500 times slower than DRAM.

LOGICAL DISK BLOCKS

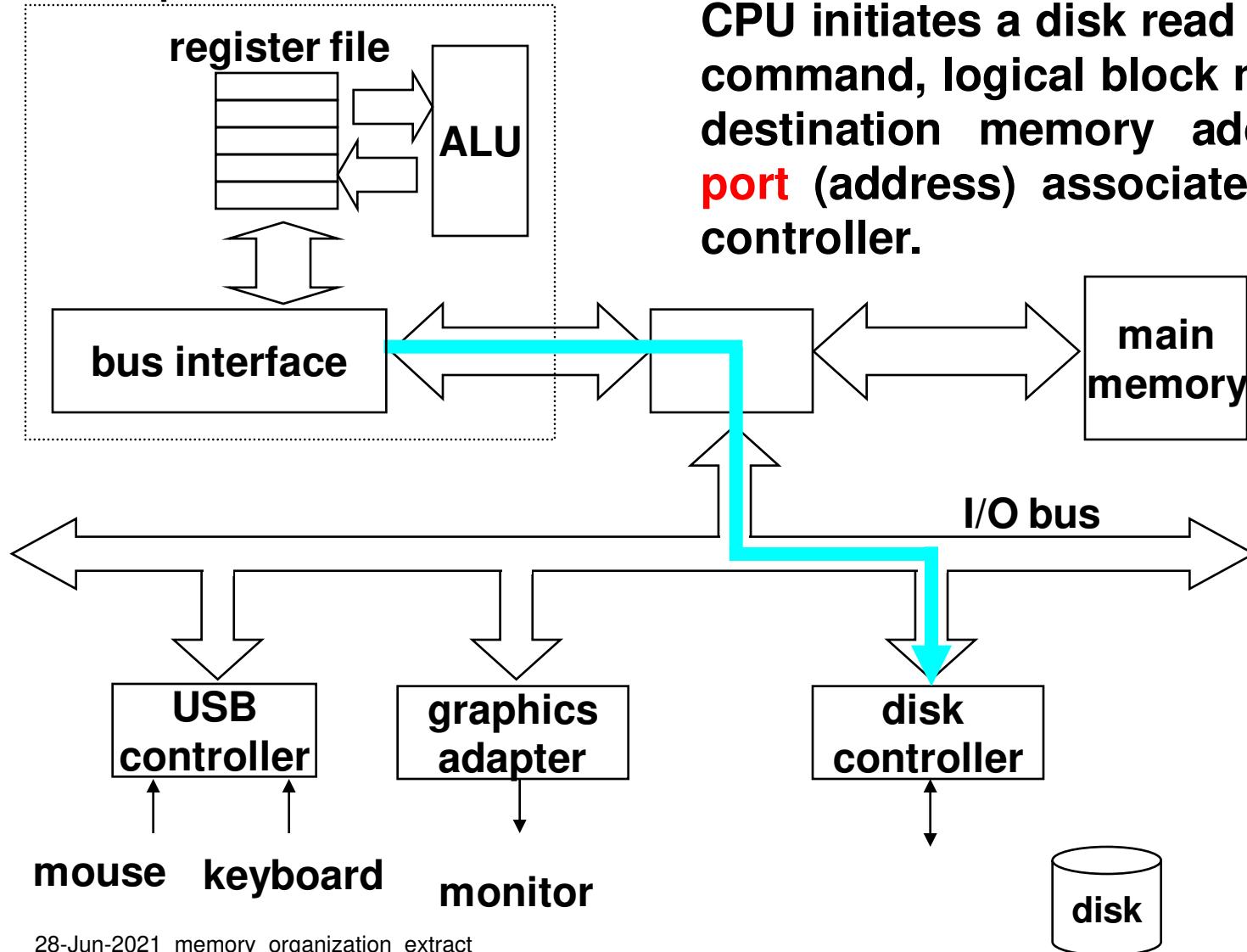
- Modern disks present a simpler abstract view of the complex sector geometry:
 - The set of available sectors is modeled as a sequence of b-sized **logical blocks** (0, 1, 2, ...)
- Mapping between logical blocks and actual (physical) sectors
 - Maintained by hardware/firmware device called disk controller.
 - Converts requests for logical blocks into (surface,track,sector) triples.
- Allows controller to set aside spare cylinders for each zone.
 - Accounts for the difference in “**formatted capacity**” and “**maximum capacity**”.

I/O BUS



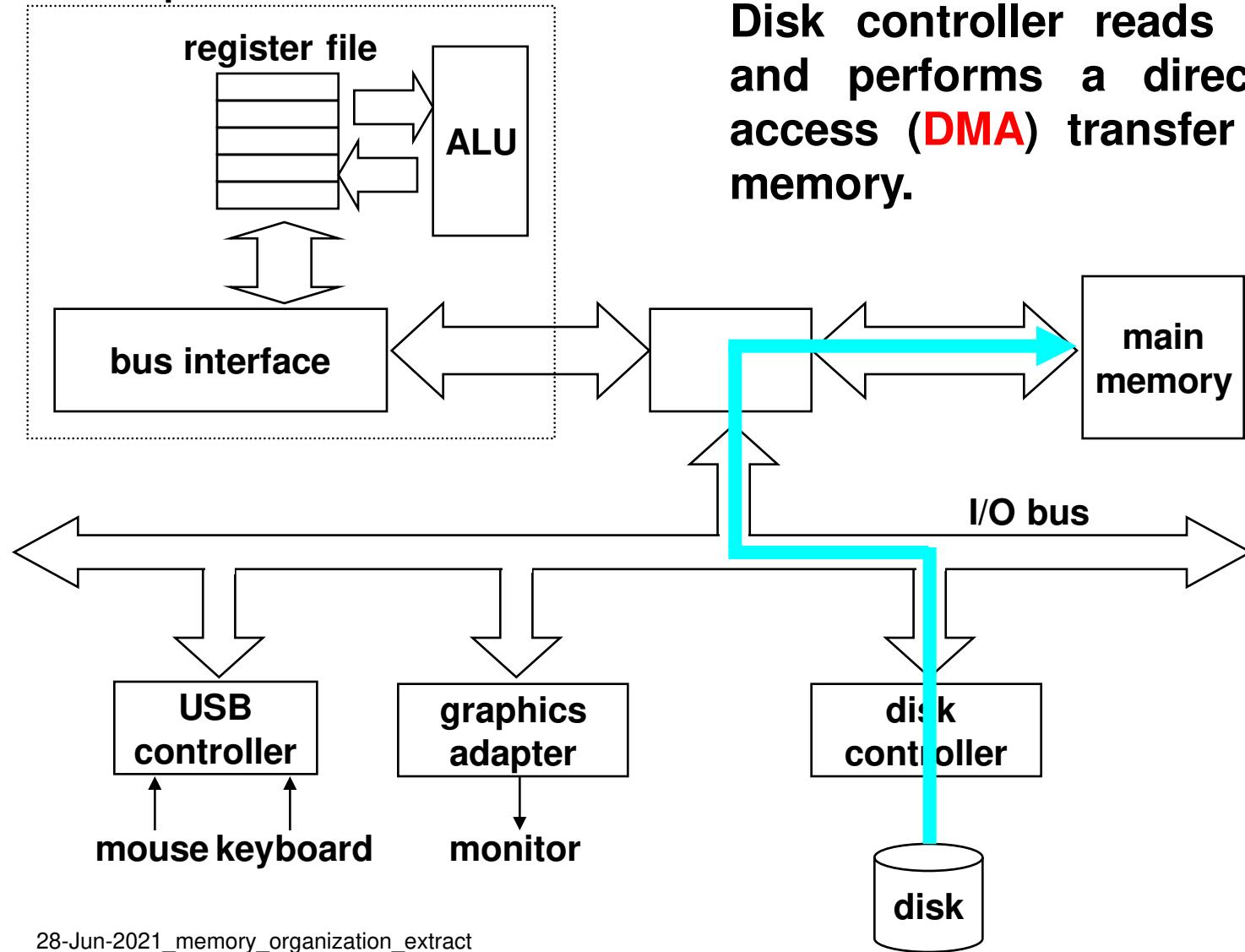
READING A DISK SECTOR (1)

CPU chip



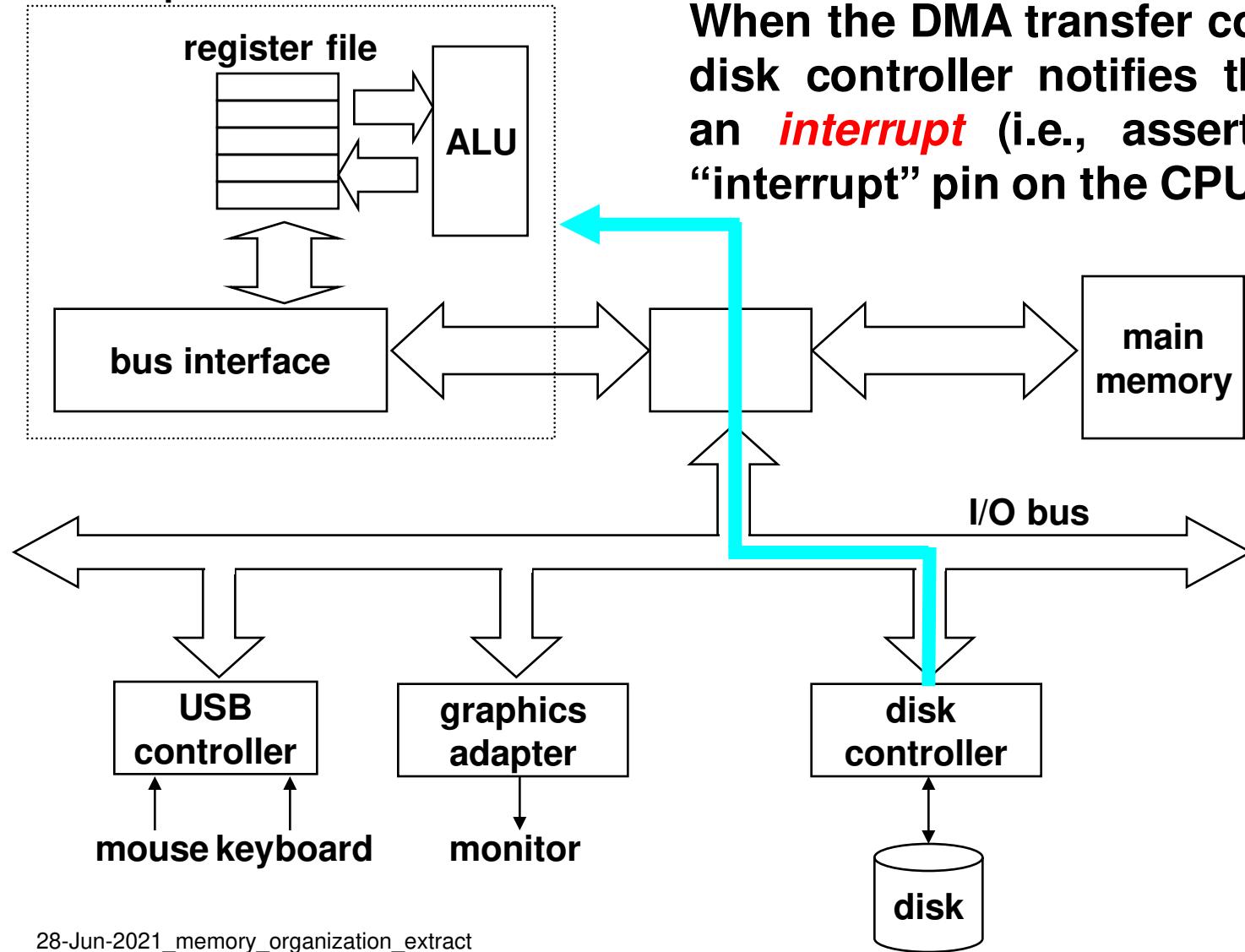
READING A DISK SECTOR (2)

CPU chip



READING A DISK SECTOR (3)

CPU chip



LOCALITY EXAMPLE

Claim: Being able to look at code and get a qualitative sense of its locality is a key skill for a professional programmer.

Question: Does this function have good locality?

```
int sumarrayrows(int a[M] [N])
{
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum
}
```

LOCALITY EXAMPLE

Question: Does this function have good locality?

```
int sumarraycols(int a[M] [N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum
}
```

LOCALITY EXAMPLE

Question: Can you permute the loops so that the function scans the 3-d array a[] with a stride-1 reference pattern (and thus has good spatial locality)?

```
int sumarray3d(int a[M] [N] [N])
{
    int i, j, k, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            for (k = 0; k < N; k++)
                sum += a[k] [i] [j];
    return sum
}
```

MEMORY HIERARCHIES

- Some fundamental and enduring properties of hardware and software:
 - Fast storage technologies cost more per byte and have less capacity.
 - The gap between CPU and main memory speed is widening.
 - Well-written programs tend to exhibit good locality.
- These fundamental properties complement each other beautifully.
- They suggest an approach for organizing memory and storage systems known as a **memory hierarchy**.

AUXILIARY MEMORY

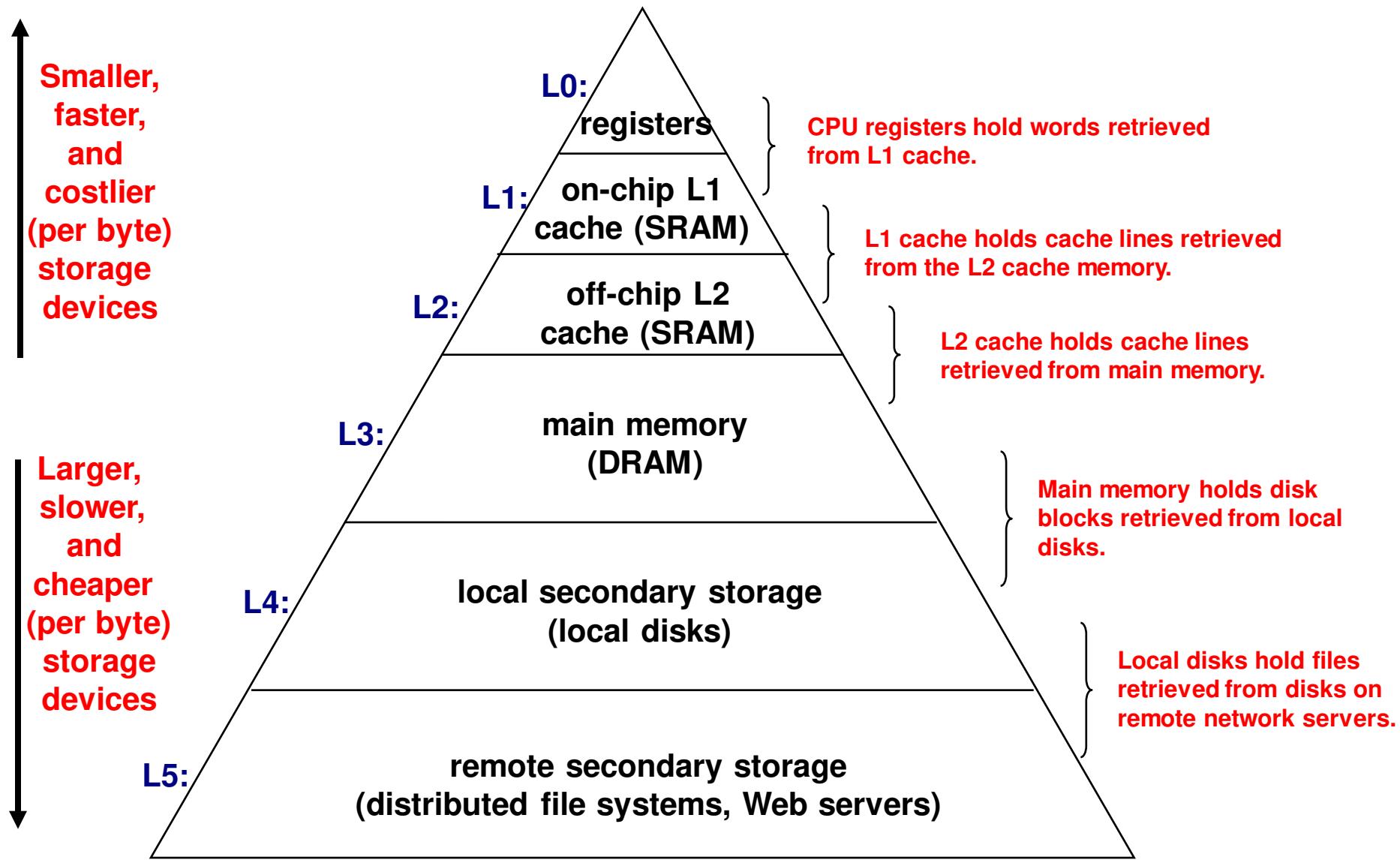
Physical Mechanism

- Magnetic
- Electronic
- Electromechanical

Characteristic of any device

- Access mode
- Access Time
- Transfer Rate
- Capacity
- Cost

AN EXAMPLE MEMORY HIERARCHY



ACCESS METHODS

- **Sequential**
 - Start at the beginning and read through in order
 - Access time depends on location of data and previous location – e.g. tape
- **Direct**
 - Individual blocks have unique address
 - Access is by jumping to vicinity plus sequential search
 - Access time depends on location and previous location – e.g. disk

Random

- Individual addresses identify locations exactly
- Access time is independent of location or previous access – e.g. RAM

• Associative

- Data is located by a comparison with contents of a portion of the store
- Access time is independent of location or previous access – e.g. cache

PERFORMANCE

- **Access time**
 - Time between presenting the address and getting the valid data
- **Memory Cycle time**
 - Time may be required for the memory to “recover” before next access
 - *Cycle time* is access + recovery
- **Transfer Rate**
 - Rate at which data can be moved

MAIN MEMORY

SRAM vs. DRAM

- Both volatile
 - Power needed to preserve data
- Dynamic cell
 - Simpler to build, smaller
 - More dense
 - Less expensive
 - Needs refresh
 - Larger memory units (DIMMs)
- Static
 - Faster
 - Cache

Cont...

1K x 8:

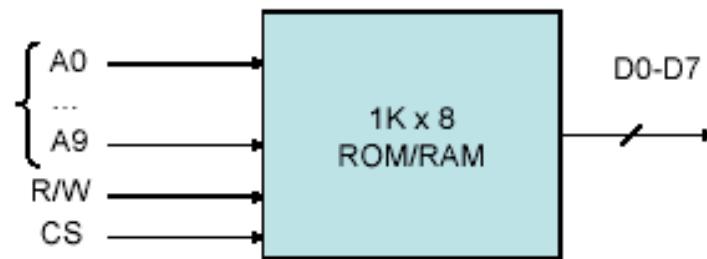
1K = 2ⁿ,

n: number of address lines

8: number of data lines

R/W: Read/Write Enable

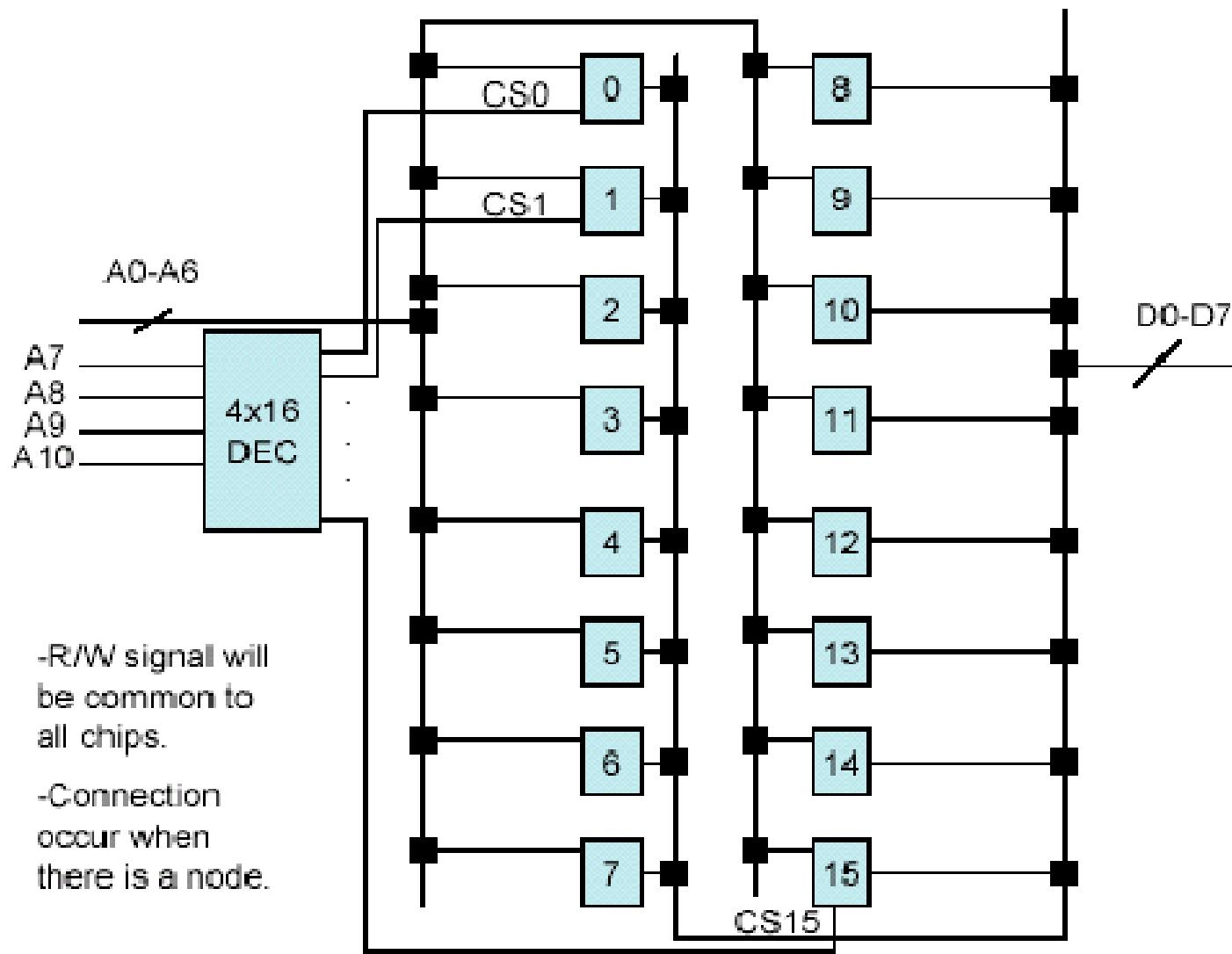
CS: Chip Select.



PROBLEMS

- a) For a memory capacity of 2048 bytes, using 128x8 chips, we need $2048/128=16$ chips.
- b) We need 11 address lines to access $2048 = 2^{11}$, the common lines are 7 (since each chip has 7 address lines; $128= 2^7$)
- c) We need a decoder to select which chip is to accessed. Draw a diagram to show the connections.

Cont...



Cont...

The address range for chip 0 will be:

0000 0000000 to 0000 1111111 , thus
000 to 07F (Hexadecimal)

The address range for chip 1 will be:

0001 0000000 to 0001 1111111 , thus
080 to 0FF (Hexadecimal)

And so on until we hit 7FF. (check this!)

MAGNETIC DISK AND DRUMS

- Magnetic Disk and Drums are similar in operation
- High Rotating surfaces with magnetic recording medium
- Rotating surface
 - Disk- a round flat plate
 - Drum – cylinder
- Rotating surface rotates at uniform speed and is not stopped or started during access operations
- Bits are recorded as magnetic spots on the surface as it passes a stationary mechanism-WRITE HEAD
- Stored bits are detected by a change in a magnetic field produced by a recorded spot on a surface as it passes thru the READ HEAD
 - HEAD -(conducting coil)

MAGNETIC DISK

- Bits are stored in magnetized surface in spots along the concentric circle called tracks
- Track divided into sections –sectors
 - Single read/write head for each disk surface-the track address bits are used by a mechanical assembly to move the head into the specified track position be for reading and writing.
 - Separate read/write head for each track in each surface .The address bits can then select a particular track electronically through a decoder circuit.
- More expensive found in large computer

Cont...

- Permanent timing tracks are used in disks to synchronize the bits and recognize the sectors
- A disk system is addressed by address bits that specify the disk no. The disk surface, sector no., and the track within the sector
- After the read/write heads are positioned in the specified track. The system has to wait until the rotating disk reaches the specified sector under the read/write head.
- Information transfer is very fast once the beginning of a sector has been reached
- Disk with multiple heads and simultaneous transfer of bits from several tracks at the same time

Cont...

- A track in a given sector near the circumference is longer than a track near the center of the disk.
- If bits are recorded with equal density, some tracks will contain more recorded bits than other
- To make all records in a sector of equal length, some disks uses variable recording density with higher density on tracks near the center than on tracks near the circumference. This equalizes the number of bits on all tracks of a given sector
- Disks
 - Hard disk
 - Floppy Disk

MAGNETIC TAPES

- A magnetic tape transport system consist of the electrical, mechanical ,electronic component to provide the parts and control mechanism for a magnetic tape
- Tape is a strip of plastic coated with a magnetic recording medium
- Bits are recorded as magnetic spots on the tape along several tracks
- Read/Write heads are mounted on in each track so that data can be recorded and read as a sequence of characters
- Magnetic tape can't be stopped or started fast enough between individuals characters because of this info is recorded in blocks where the tape can be stopped.

Cont...

- The tape starts moving while in a gap and attains constant speed by the time it reaches the next record
- Each record on a tape has an identification bit pattern at the beginning and end.
- By reading the bit pattern at the end of the record the control recognizes the beginning of a gap.
- A tape is addressed by specifying the record number and the number of characters in a record.
- Records may be fixed or variable length

ASSOCIATIVE MEMORY

- It is a memory unit accessed by content (Content Addressable Memory CAM).
- Word read/written no address specified memory find the empty unused location to store the data similarly memory located all word which match the specified content and marks them for reading
- Uniquely suited for parallel searches by data association.
- More expensive than RAM because each cell must have storage and logic circuits for matching with an external argument.
- Each word in memory is compared with the argument register (A). If a word matches, then the corresponding bit in the match register will be set.
- (K) is the key register responsible for masking the data to select a field in the argument word.

Cont...

Fig.1:Block diagram of Associative memory

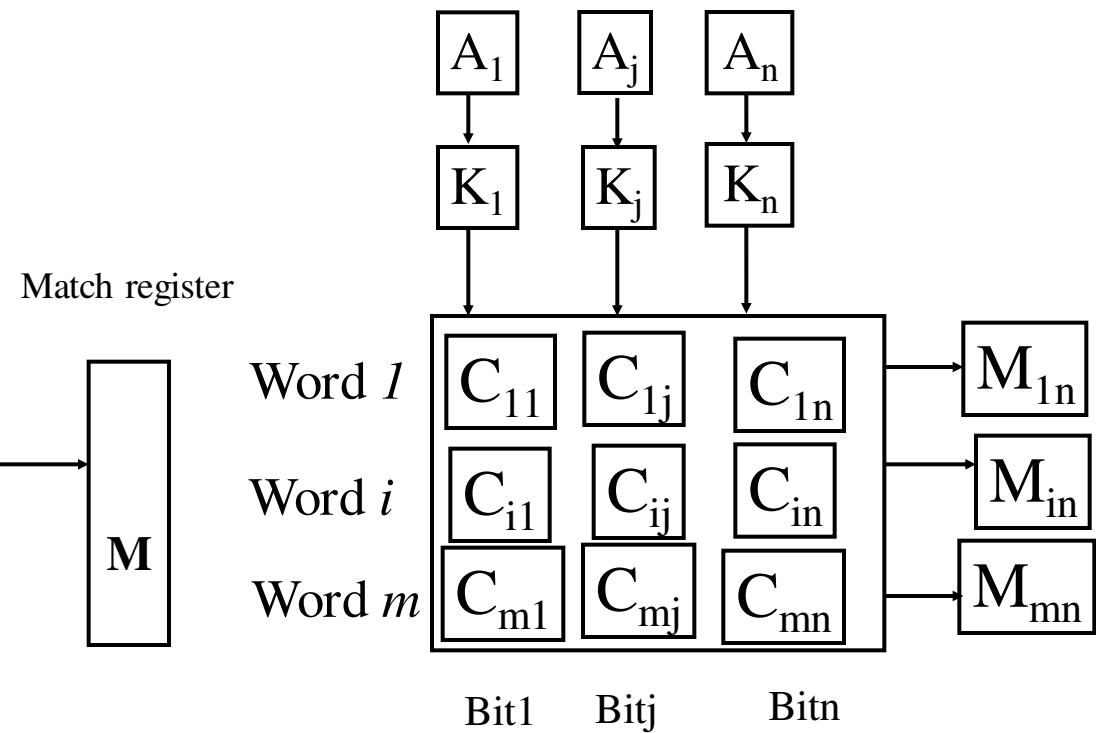
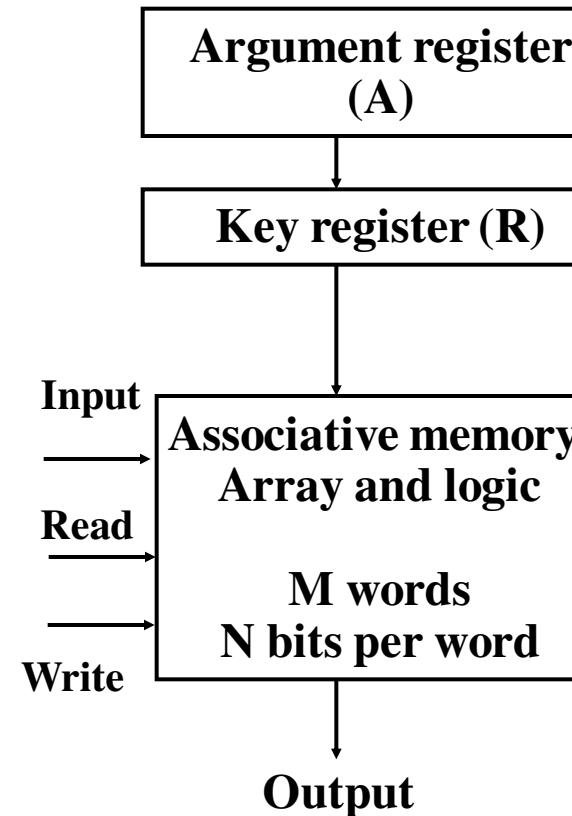
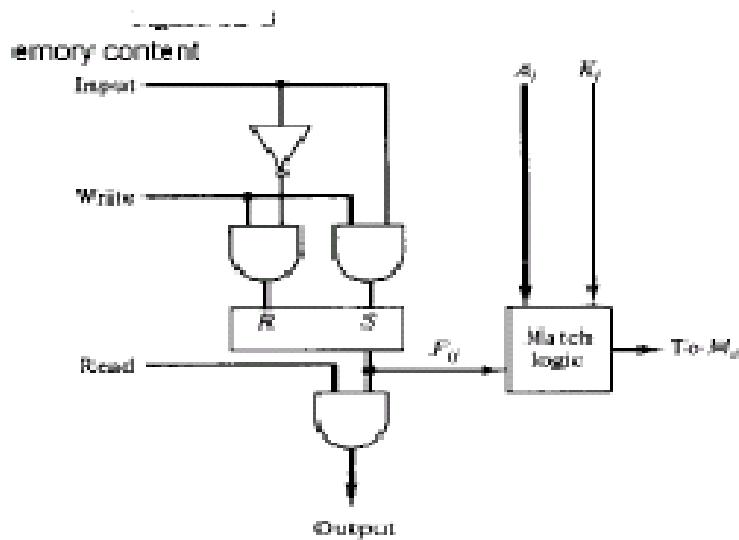


Fig.2:An Associative array of one word

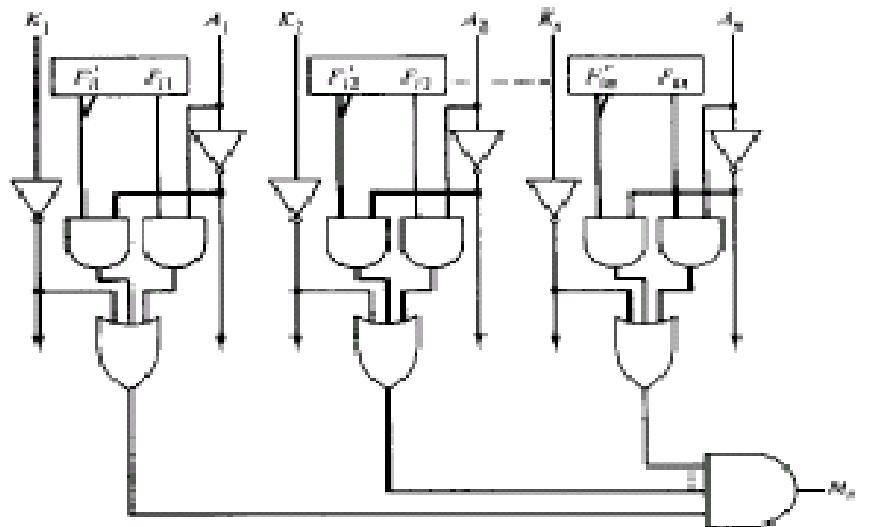
A	101 111100
K	111 000000
Word 1	100111100
Word 2	101 000001

Cont...

One cell for associative memory



Match logic for one word of associative memory



$$M_i = \prod_{j=1}^n (A_j F_{ij} + \overline{A}_j \overline{F}_{ij} + K_j) = \prod_{j=1}^n (x_j + K_j)$$

The requirement is that when $K=1$, then x_j is chosen, otherwise, $x_j + 1 = 1$, thus no comparison. $x_j = A_j F_{ij} + \overline{A}_j \overline{F}_{ij}$.

Cont...

- A read operation takes place for those locations where $M_i=1$.
- Usually one location, but if more than one, then locations will be read in sequence.
- A write can be done in a RAM like addressing, thus device will operate in a RAM writing CAM reading.
- A TAG register is available with a number of bits that is the same as the number of word, to keep track of which locations are empty (0) or full (1), after a read/write operation.

LOCALITY

Principle of Locality:

- Programs tend to reuse data and instructions near those they have used recently, or that were recently referenced themselves.
- **Temporal locality:** Recently referenced items are likely to be referenced in the near future.
- **Spatial locality:** Items with nearby addresses tend to be referenced close together in time.

Locality Example:

- Data
 - Reference array elements in succession (stride-1 reference pattern): **Spatial locality**
 - Reference sum each iteration: **Temporal locality**
- Instructions
 - Reference instructions in sequence: **Spatial locality**
 - Cycle through loop repeatedly: **Temporal locality**

```
sum = 0;  
for (i = 0; i < n; i++)  
    sum += a[i];  
return sum;
```

LOCALITY EXAMPLE

Locality Example:

- Data
 - Reference array elements in succession (stride-1 reference pattern): **Spatial locality**
 - Reference sum each iteration: **Temporal locality**
- Instructions
 - Reference instructions in sequence: **Spatial locality**
 - Cycle through loop repeatedly: **Temporal locality**

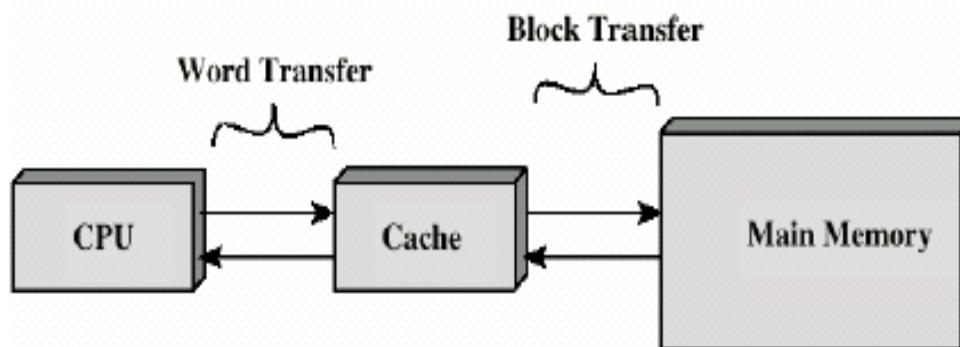
```
sum = 0;  
for (i = 0; i < n; i++)  
    sum += a[i];  
return sum;
```

CACHE MEMORY

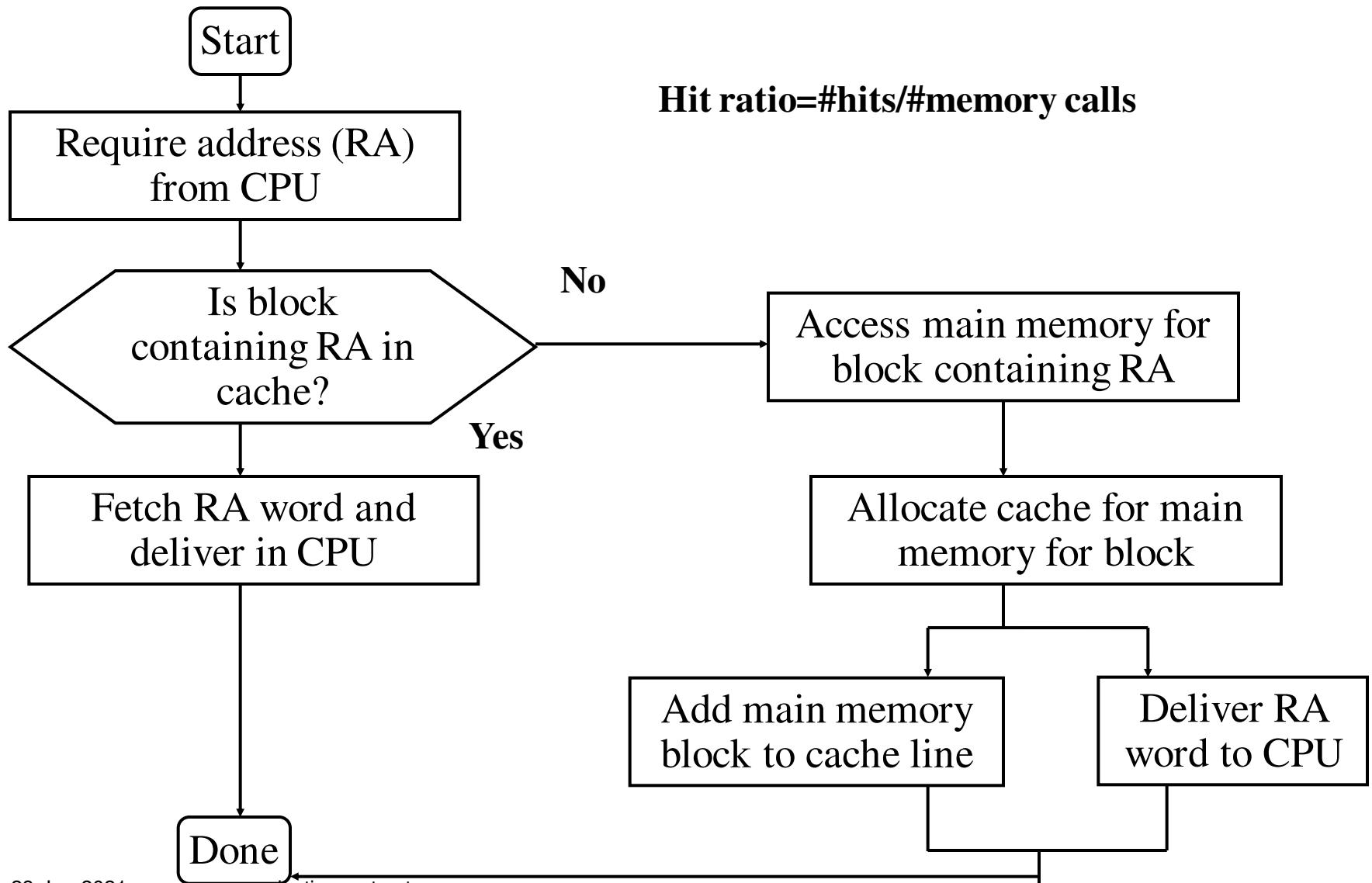
- References at any given time tend to be confined within a few localized area in memory - Locality of Reference
- To lesser memory reference –Cache

CACHE (\$)

- Small amount of fast memory
- Sits between normal main memory and CPU
- May be located on CPU chip or module



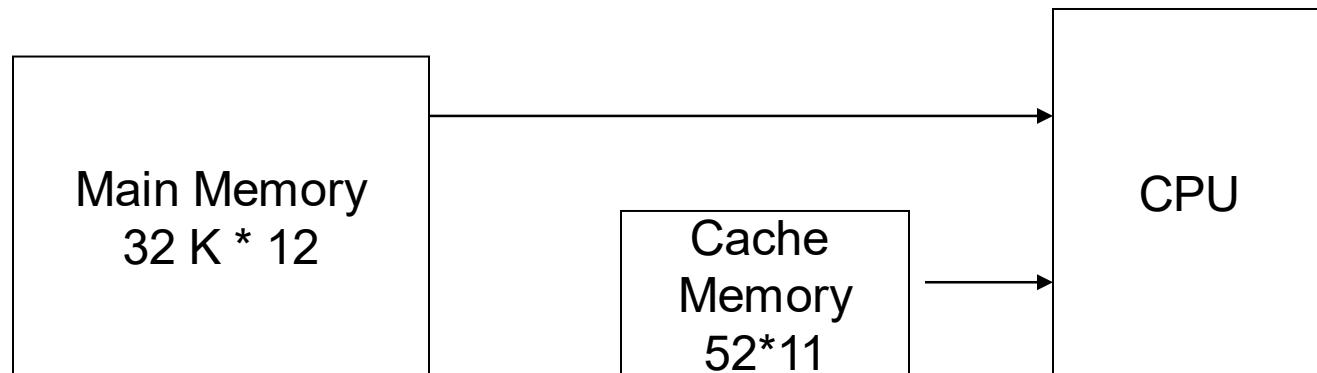
CACHE READ OPERATION



Cont...

- Transformation of data from Memory to \$ is referred to as *Mapping*.
- 3 types of mapping:
 - Associative Mapping (fastest, most flexible)
 - Direct mapping (HW efficient)
 - Set-associative mapping

Mem: 15-bit address
Same address is sent to \$

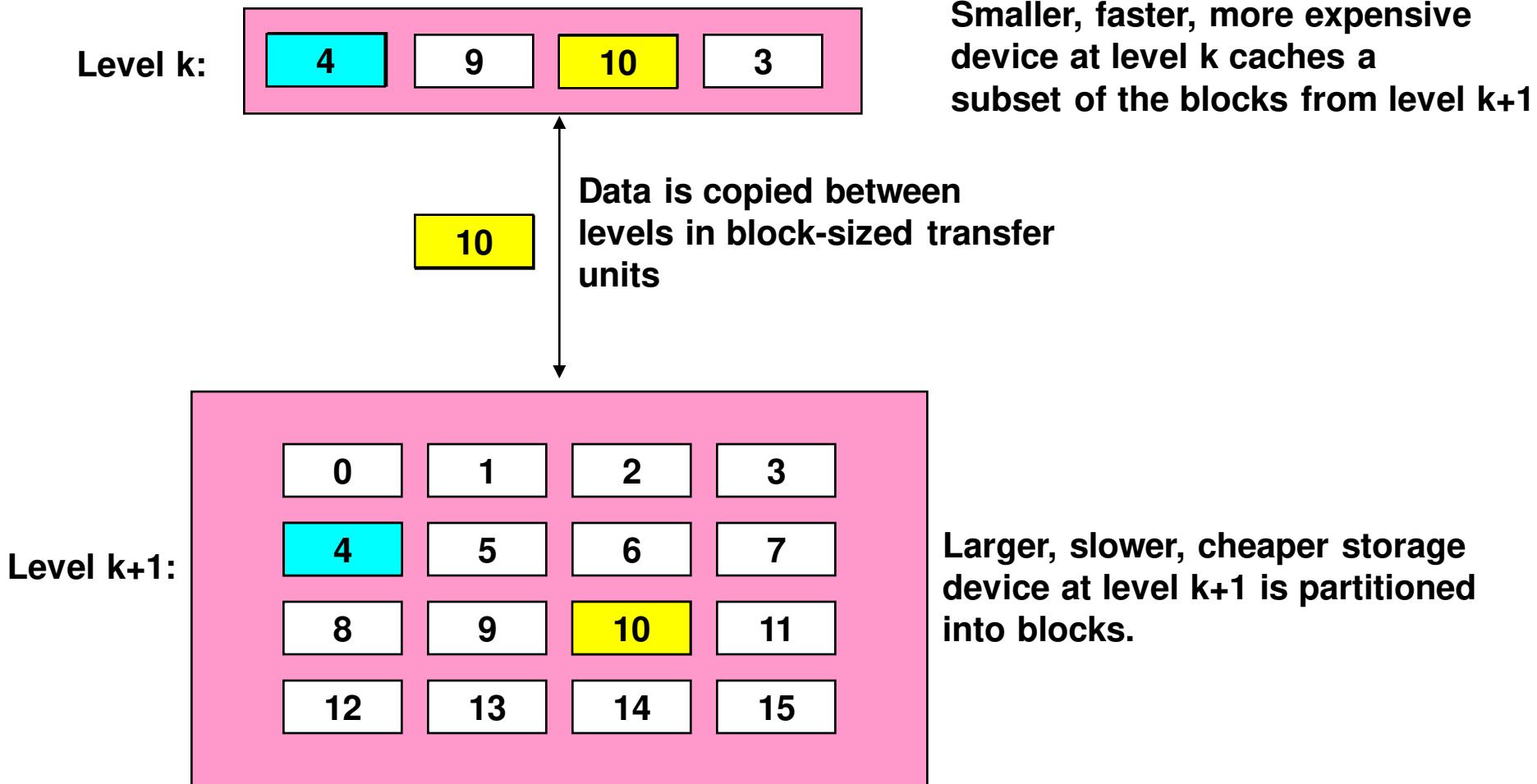


Example of Cache Memory

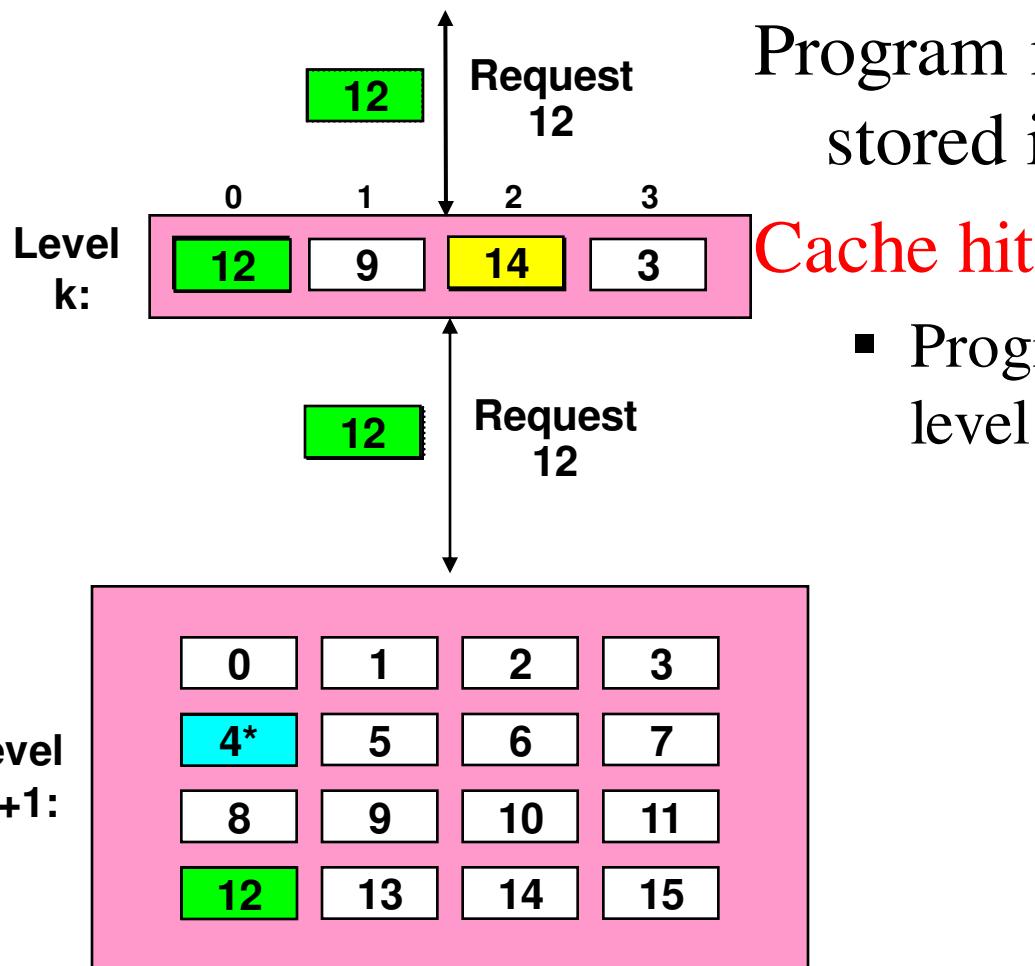
CACHES

- **Cache:** A smaller, faster storage device that acts as a staging area for a subset of the data in a larger, slower device.
- Fundamental idea of a memory hierarchy:
 - For each k , the faster, smaller device at level k serves as a cache for the larger, slower device at level $k+1$.
- Why do memory hierarchies work?
 - Programs tend to access the data at level k more often than they access the data at level $k+1$.
 - Thus, the storage at level $k+1$ can be slower, and thus larger and cheaper per bit.
 - Net effect: A large pool of memory that costs as much as the cheap storage near the bottom, but that serves data to programs at the rate of the fast storage near the top.

CACHING IN A MEMORY HIERARCHY



GENERAL CACHING CONCEPTS



Program needs object d, which is stored in some block b.

Cache hit

- Program finds b in the cache at level k. E.g., block 14.

Cont...

Cache miss

b is not at level k, so level k cache must fetch it from level k+1. E.g., block 12.

If level k cache is full, then some current block must be replaced (evicted). Which one is the “victim”?

Placement policy: where can the new block go?
E.g., $b \bmod 4$

Replacement policy: which block should be evicted? E.g., LRU

Types of cache misses:

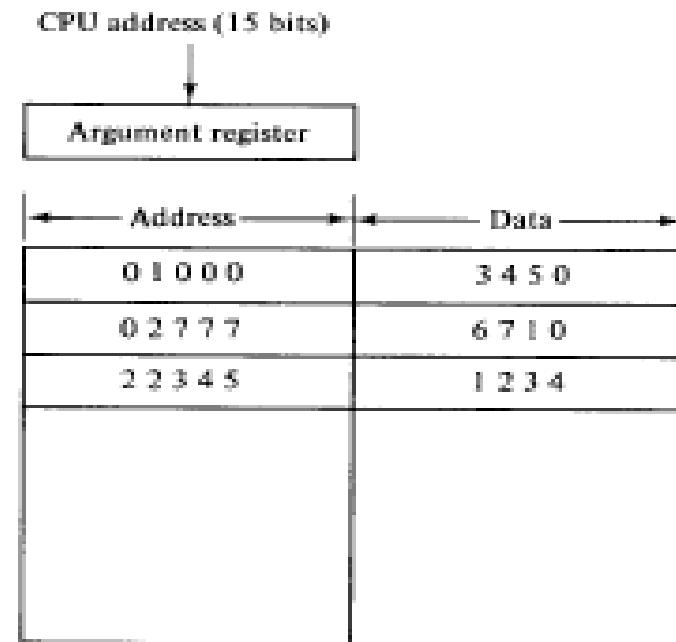
- Cold (compulsory) miss
 - ✓ Cold misses occur because the cache is empty.
- Conflict miss
 - ✓ Most caches limit blocks at level $k+1$ to a small subset (sometimes a singleton) of the block positions at level k .
 - ✓ E.g. Block i at level $k+1$ must be placed in block $(i \bmod 4)$ at level $k+1$.
 - ✓ Conflict misses occur when the level k cache is large enough, but multiple data objects all map to the same level k block.
 - ✓ E.g. Referencing blocks 0, 8, 0, 8, 0, 8, ... would miss every time.
- Capacity miss
 - ✓ Occurs when the set of active cache blocks (working set) is larger than the cache.

EXAMPLES OF CACHING IN THE HIERARCHY

Cache Type	What Cached	Where Cached	Latency (cycles)	Managed By
Registers	4-byte word	CPU registers	0	Compiler
TLB	Address translations	On-Chip TLB	0	Hardware
L1 cache	32-byte block	On-Chip L1	1	Hardware
L2 cache	32-byte block	Off-Chip L2	10	Hardware
Virtual Memory	4-KB page	Main memory	100	Hardware+OS
Buffer cache	Parts of files	Main memory	100	OS
Network buffer cache	Parts of files	Local disk	10,000,000	AFS/NFS client
Browser cache	Web pages	Local disk	10,000,000	Web browser
Web cache	Web pages	Remote server disks	1,000,000,000	Web proxy server

ASSOCIATIVE MAPPING:

- The 15-bit address as well as its corresponding data word are stored in \$.
- If a match in address is found (address from CPU is placed in (A) register), data word is sent to CPU.



**Associative Mapping of Cache
(all no. in octal)**

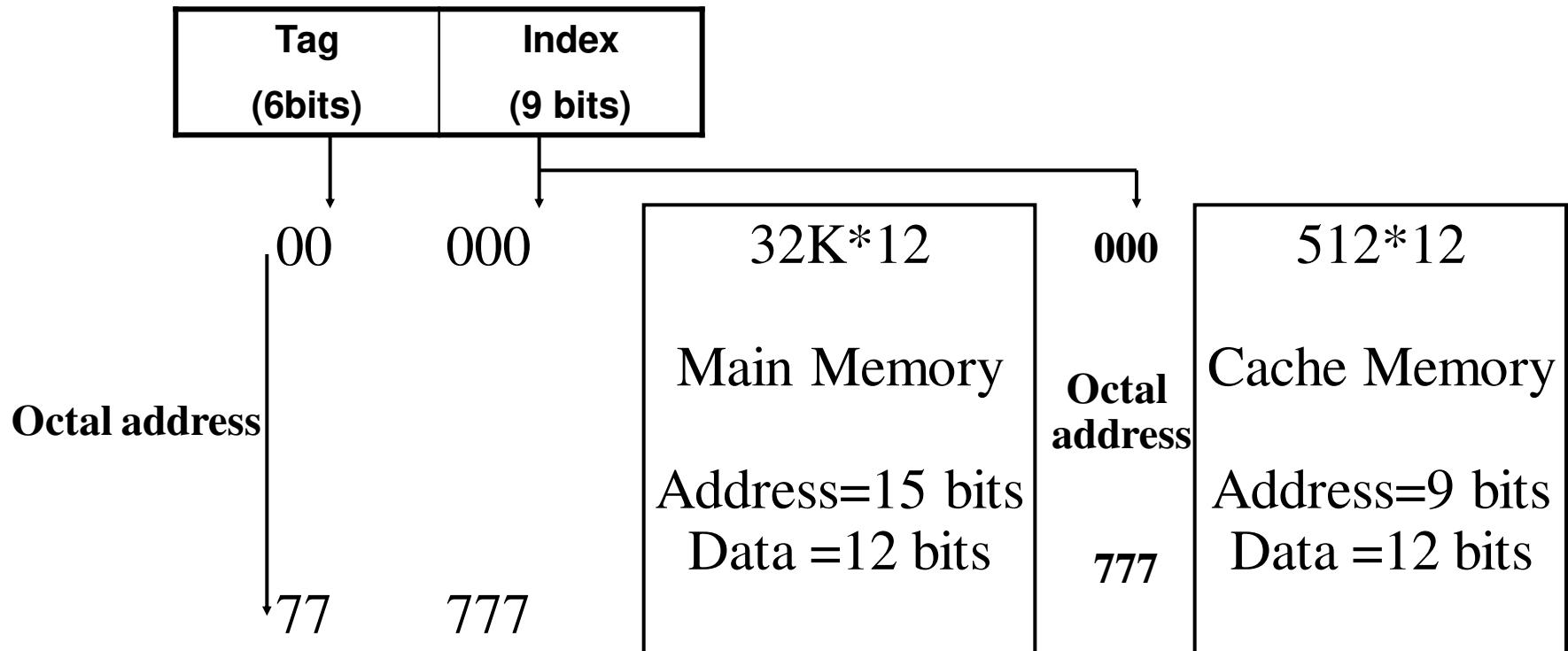
Cont...

- If no match, then data word is accessed from Memory, and the address data pair are transferred to \$.
- If \$ is full, a replacement algorithm is used to free some space.

DIRECT MAPPING

- A RAM is used for Cache (\$).
- The 15-bit address is divided into
Index=k, and TAG=n-k.
 $n=15$ (address for Memory), $k=9$ (address for \$).
- Each word in \$ consists of the data word along with its associated TAG.
- When CPU issues a read, the index part is used to locate the address in \$, and then the remaining portion is compared to TAG, if there is a match, then that is a HIT.
IF there is no match, then this is a MISS.
- If MISS, then read from Memory and store word + TAG in \$ again.

ADDRESSING RELATIONSHIP BETWEEN CACHE AND MAIN



DIRECT MAPPING CACHE ORGANISATION

Memory address	Memory data
00000	1 2 2 0
00777	2 3 4 0
01000	3 4 5 0
01777	4 5 6 0
02000	5 6 7 0
02777	6 7 1 0

(a) Main memory

Index address	Tag	Data
000	0 0	1 2 2 0
777	0 2	6 7 1 0

(b) Cache memory

Block size of 1.

Cont...

- Disadvantage
 - what if two or more words whose addresses have the same index but different TAG?
Increase MISS ratio!
 - Usually, this will happen when words are far away in the address range

Far from \$ size, i.e. after 512 location in this example.

$64 \times 8 = 512$

64 blocks

8 words/block

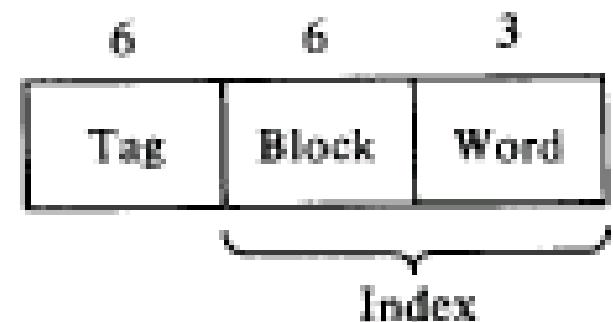
Block (6 bits) Word (3 bits)

Index=007 Block 0, word 8

Index=103 Block 8, word 4

DIRECT MAPPING

	Index	Tag	Data
Block 0	000	0 1	3 4 5 0
	007	0 1	6 5 7 8
Block 1	010		
	017		
Block 63	770	0 2	
	777	0 2	6 7 1 0



Block size of 8.

$64 \times 8 = 512$
64 blocks
8 words/block

Cont...

index

Block (6 bits)	Word (3 bits)
----------------	---------------

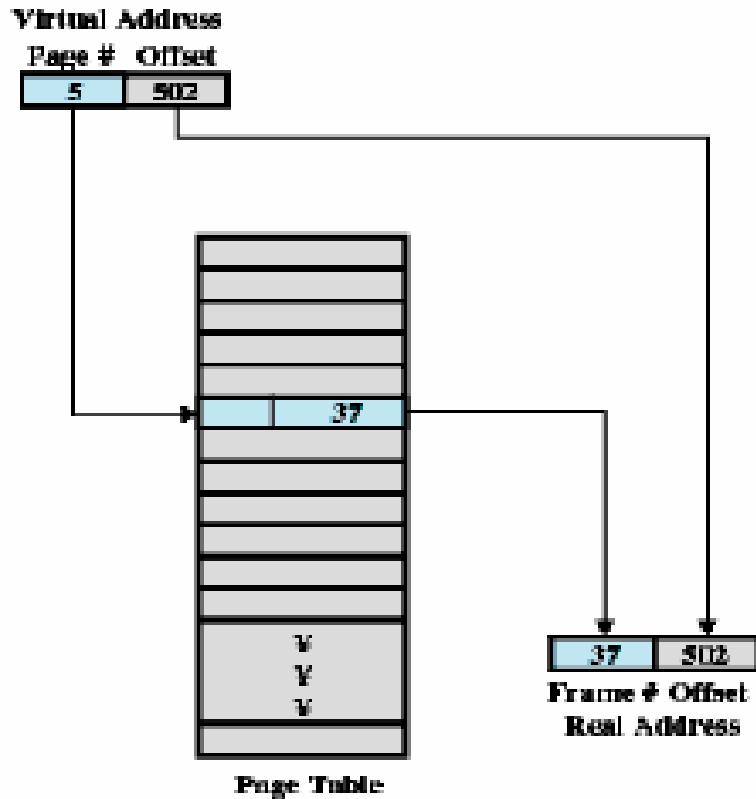
Index=007 → Block 0, word 8

Index=103 → Block 8, word 4

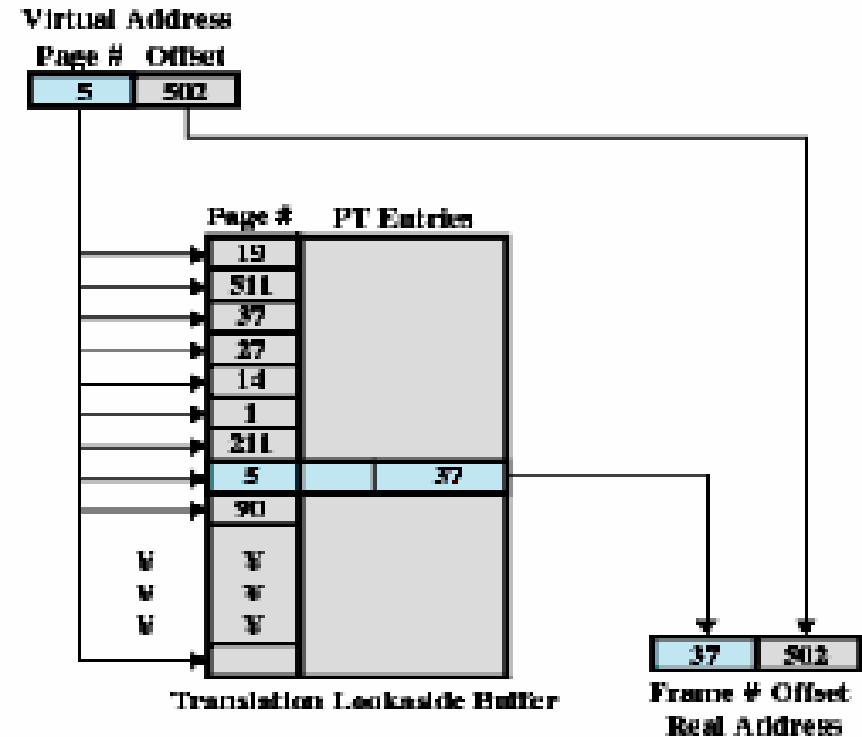
SET ASSOCIATIVE

Improvement over direct mapping

Cont...



(a) Direct mapping



(b) Associative mapping

Direct Versus Associative Lookup for Page Table Entries

WRITING TO \$

Two methods:

- Write through
 - update main memory with every memory write operation with cache being updated in parallel if it contain the word at the specified address
- Write back
 - only cache location is updated during write operation. This location is then marked by a flag so that later when the word is removed from the it is copied into main memory

VIRTUAL MEMORY

- Virtual memory (VM) is used to give programmers the illusion that they have a very large memory at their command.
- A computer has a limited memory size.
- VM provides a mechanism for translating program oriented addresses into correct memory addresses.
- Address mapping can be performed using an extra memory chip, using main memory itself (portion of it) or using associative memory using page tables.

PROBLEMS

- a) Memory is 64Kx16, and \$ is 1K words, with block size of 4.
- b) Each \$ location will have the 16-bits of data, added to them the number of TAG bits, as well as the valid bit, thus 23-bits.
 - Index = 10 bits TAG = 6 bits
 - Block = 8 bits, word = 2 bits

HARDWARE AND CONTROL STRUCTURES

- Memory references are dynamically translated into physical addresses at run time
 - A process may be swapped in and out of main memory such that it occupies different regions
- A process may be broken up into pieces that do not need to located contiguously in main memory
- All pieces of a process do not need to be loaded in main memory during execution

EXECUTION OF A PROGRAM

- Operating system brings into main memory a few pieces of the program
- Resident set - portion of process that is in main memory
- An interrupt is generated when an address is needed that is not in main memory
- Operating system places the process in a blocking state

EXECUTION OF A PROGRAM

- Piece of process that contains the logical address is brought into main memory
 - Operating system issues a disk I/O Read request
 - Another process is dispatched to run while the disk I/O takes place
 - An interrupt is issued when disk I/O complete which causes the operating system to place the affected process in the Ready state

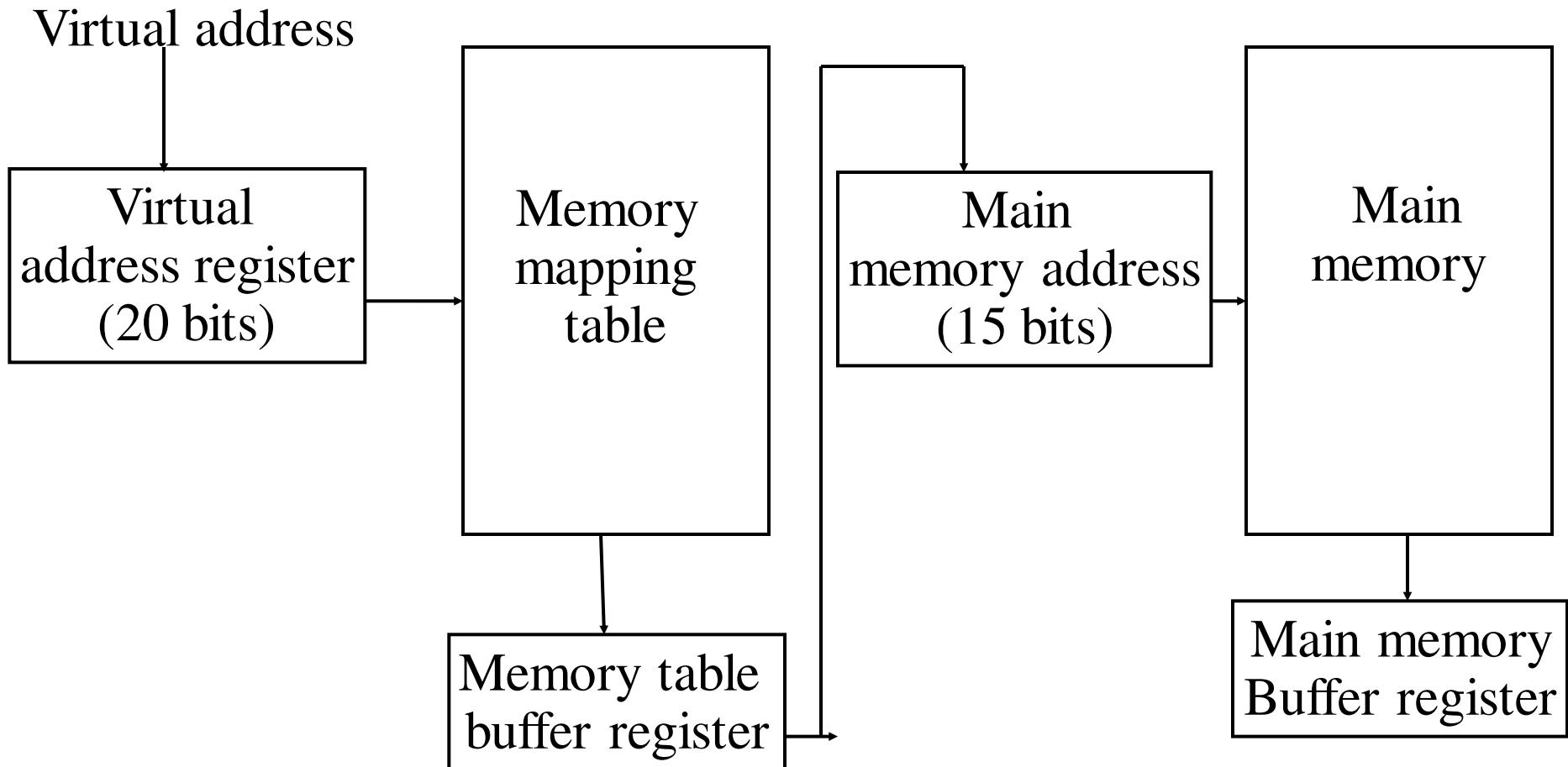
ADVANTAGES OF BREAKING A PROCESS

- More processes may be maintained in main memory
 - Only load in some of the pieces of each process
 - With so many processes in main memory, it is very likely a process will be in the Ready state at any particular time
- A process may be larger than all of main memory

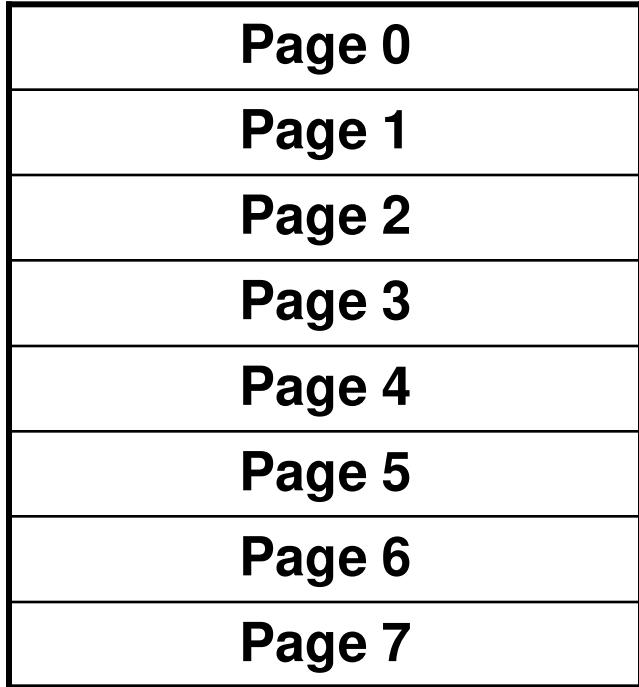
TYPES OF MEMORY

- Real memory
 - Main memory
- Virtual memory
 - Memory on disk
 - Allows for effective multiprogramming and relieves the user of tight constraints of main memory

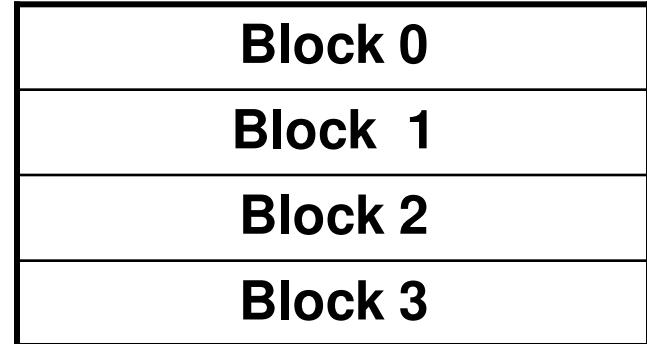
MEMORY TABLE FOR MAPPING A VIRTUAL ADDRESS



ADDRESS AND MEMORY SPACE SPLIT INTO GROUPS OF 1K WORDS

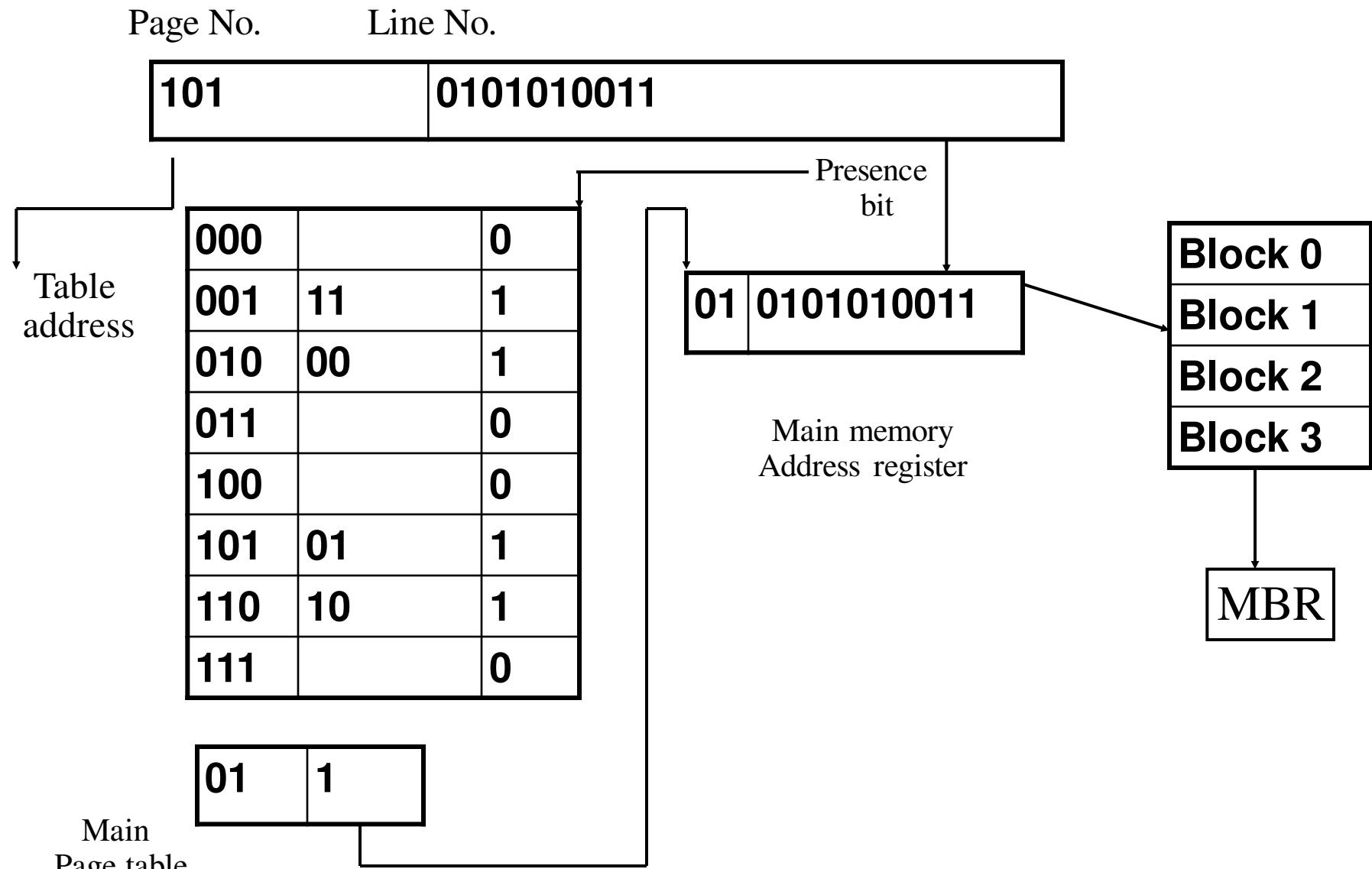


Address space
 $N=8$ $K=2^{13}$



Memory space
 $N=4$ $K=2^{12}$

MEMORY TABLE IN A PAGED SYSTEM



ASSOCIATIVE MEMORY PAGE TABLE

Page No.

Virtual register.

101	Line Number
-----	-------------

Argument register.

111	00
-----	----

Key register

000	11
001	00
010	01
011	10

Associative memory

Page No. Block No

THRASHING

- Swapping out a piece of a process just before that piece is needed
- The processor spends most of its time swapping pieces rather than executing user instructions

PRINCIPLE OF LOCALITY

- Program and data references within a process tend to cluster
- Only a few pieces of a process will be needed over a short period of time
- Possible to make intelligent guesses about which pieces will be needed in the future
- This suggests that virtual memory may work efficiently

SUPPORT NEEDED FOR VIRTUAL MEMORY

- Hardware must support paging and segmentation
- Operating system must be able to management the movement of pages and/or segments between secondary memory and main memory

PAGING

- Each process has its own page table
- Each page table entry contains the frame number of the corresponding page in main memory
- A bit is needed to indicate whether the page is in main memory or not

PAGING

Virtual Address



Page Table Entry



(a) Paging only

MODIFY BIT IN PAGE TABLE

- Modify bit is needed to indicate if the page has been altered since it was last loaded into main memory
- If no change has been made, the page does not have to be written to the disk when it needs to be swapped out

PAGE TABLES

- The entire page table may take up too much main memory
- Page tables are also stored in virtual memory
- When a process is running, part of its page table is in main memory

TRANSLATION LOOKASIDE BUFFER

- Each virtual memory reference can cause two physical memory accesses
 - One to fetch the page table
 - One to fetch the data
- To overcome this problem a high-speed cache is set up for page table entries
 - Called a Translation Lookaside Buffer (TLB)

TRANSLATION LOOKASIDE BUFFER

Contains page table entries that have been most recently used

- Given a virtual address, processor examines the TLB
- If page table entry is present (TLB hit), the frame number is retrieved and the real address is formed
- If page table entry is not found in the TLB (TLB miss), the page number is used to index the process page table

First checks if page is already in main memory

- If not in main memory a page fault is issued

The TLB is updated to include the new page entry

PAGE SIZE

- Smaller page size, less amount of internal fragmentation
- Smaller page size, more pages required per process
- More pages per process means larger page tables
- Larger page tables means large portion of page tables in virtual memory
- Secondary memory is designed to efficiently transfer large blocks of data so a large page size is better

PAGE SIZE

- Small page size, large number of pages will be found in main memory
- As time goes on during execution, the pages in memory will all contain portions of the process near recent references. Page faults low.
- Increased page size causes pages to contain locations further from any recent reference. Page faults rise.

SEGMENTATION

- May be unequal, dynamic size
- Simplifies handling of growing data structures
- Allows programs to be altered and recompiled independently
- Lends itself to sharing data among processes
- Lends itself to protection

SEGMENT TABLES

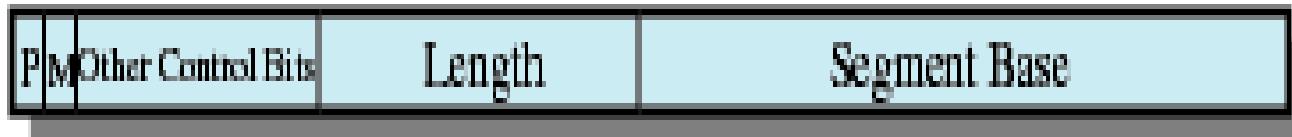
- Corresponding segment in main memory
- Each entry contains the length of the segment
- A bit is needed to determine if segment is already in main memory
- Another bit is needed to determine if the segment has been modified since it was loaded in main memory

SEGMENT TABLE ENTRIES

Virtual Address



Segment Table Entry



(b) Segmentation only

COMBINED PAGING AND SEGMENTATION

- Paging is transparent to the programmer
- Segmentation is visible to the programmer
- Each segment is broken into fixed-size pages

COMBINED SEGMENTATION AND PAGING

Virtual Address



Segment Table Entry



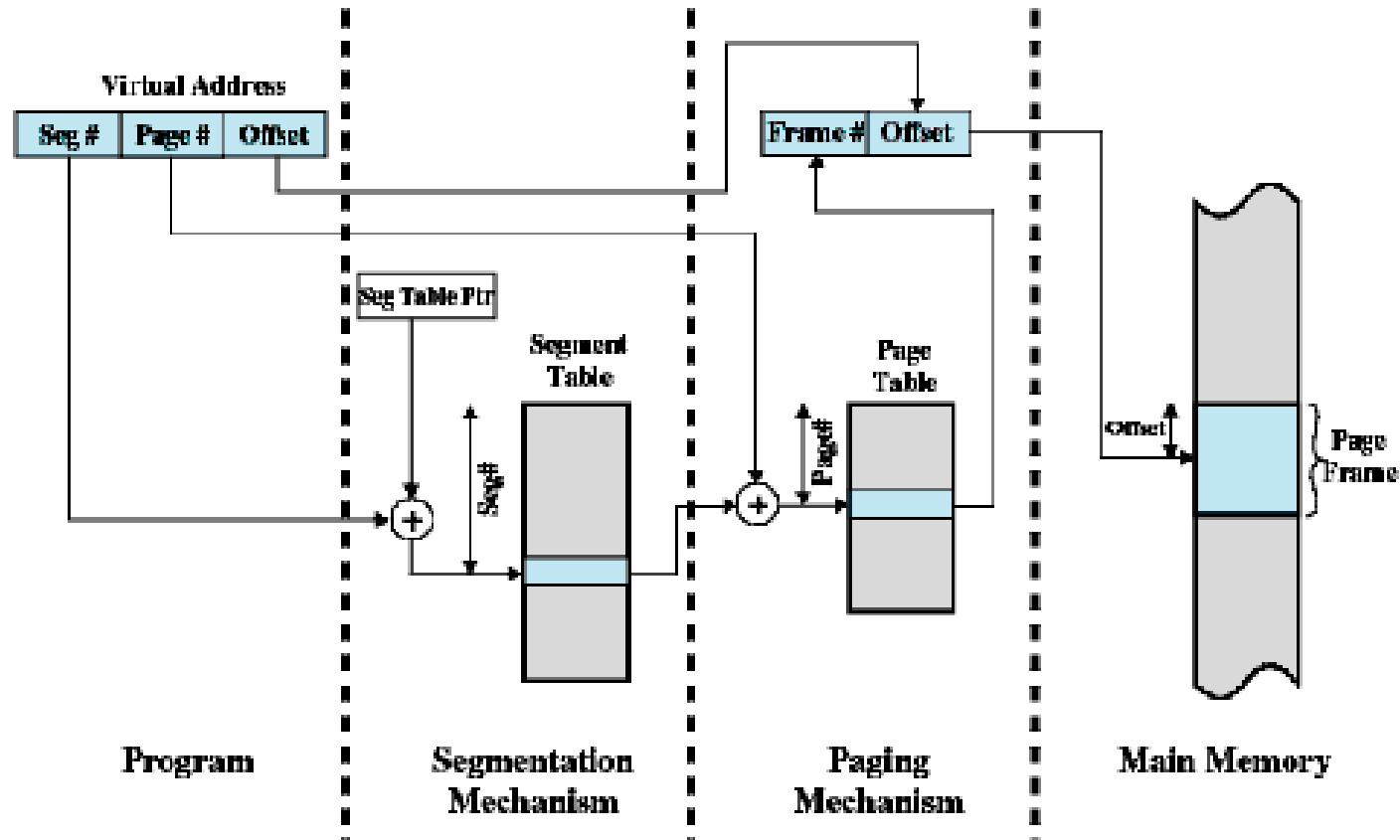
Page Table Entry



P = present bit
M = Modified bit

(c) Combined segmentation and paging

Cont...



FETCH POLICY

- Fetch Policy
 - Determines when a page should be brought into memory
 - Demand paging only brings pages into main memory when a reference is made to a location on the page
 - Many page faults when process first started
 - Prepaging brings in more pages than needed
 - More efficient to bring in pages that reside contiguously on the disk

PLACEMENT POLICY

- Determines where in real memory a process piece is to reside
- Important in a segmentation system
- Paging or combined paging with segmentation hardware performs address translation

REPLACEMENT POLICY

- Placement Policy
 - Which page is replaced?
 - Page removed should be the page least likely to be referenced in the near future
 - Most policies predict the future behavior on the basis of past behavior

Cont...

- Frame Locking
 - If frame is locked, it may not be replaced
 - Kernel of the operating system
 - Control structures
 - I/O buffers
 - Associate a lock bit with each frame

BASIC REPLACEMENT ALGORITHMS

- Optimal policy
 - Selects for replacement that page for which the time to the next reference is the longest
 - Impossible to have perfect knowledge of future events

BASIC REPLACEMENT ALGORITHMS

- Least Recently Used (LRU)
 - Replaces the page that has not been referenced for the longest time
 - By the principle of locality, this should be the page least likely to be referenced in the near future
 - Each page could be tagged with the time of last reference. This would require a great deal of overhead.

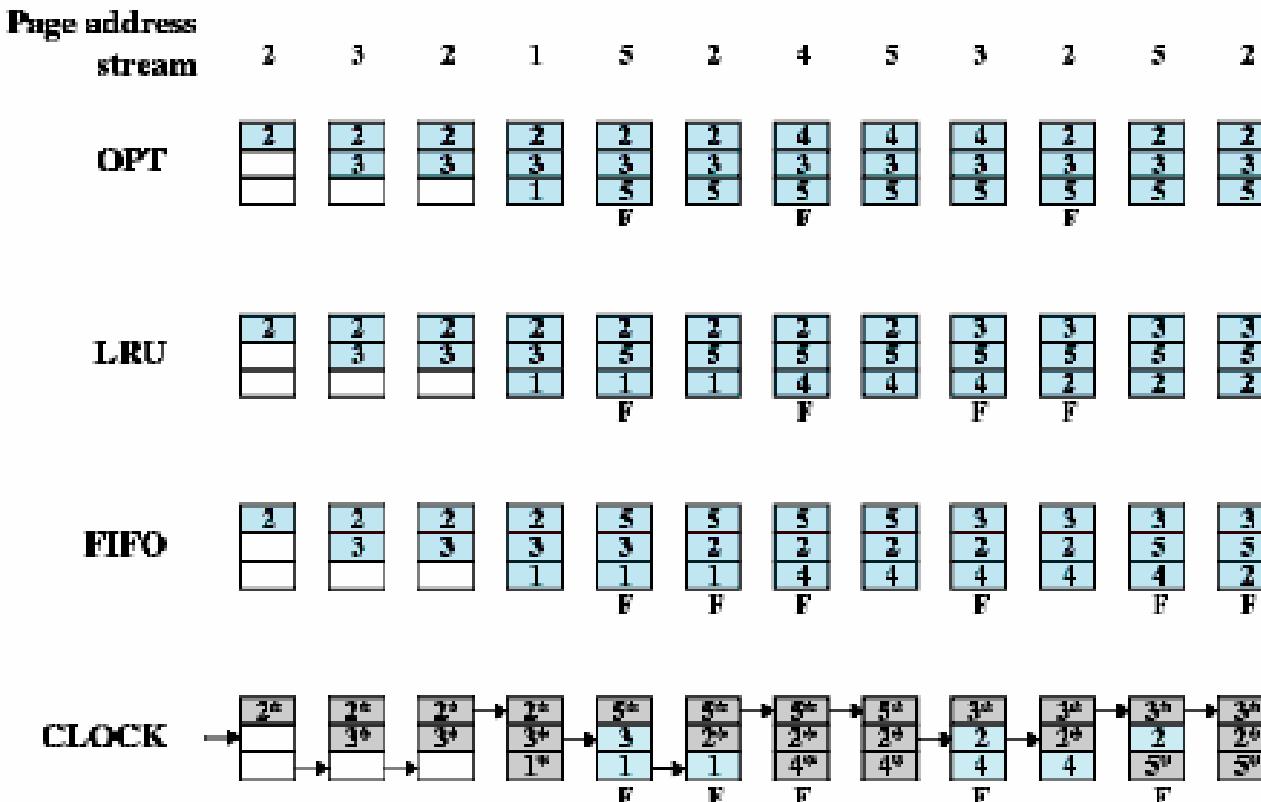
Cont...

- First-in, first-out (FIFO)
 - Treats page frames allocated to a process as a circular buffer
 - Pages are removed in round-robin style
 - Simplest replacement policy to implement
 - Page that has been in memory the longest is replaced
 - These pages may be needed again very soon

Cont...

- Clock Policy
 - Additional bit called a use bit
 - When a page is first loaded in memory, the use bit is set to 1
 - When the page is referenced, the use bit is set to 1
 - When it is time to replace a page, the first frame encountered with the use bit set to 0 is replaced.
 - During the search for replacement, each use bit set to 1 is changed to 0

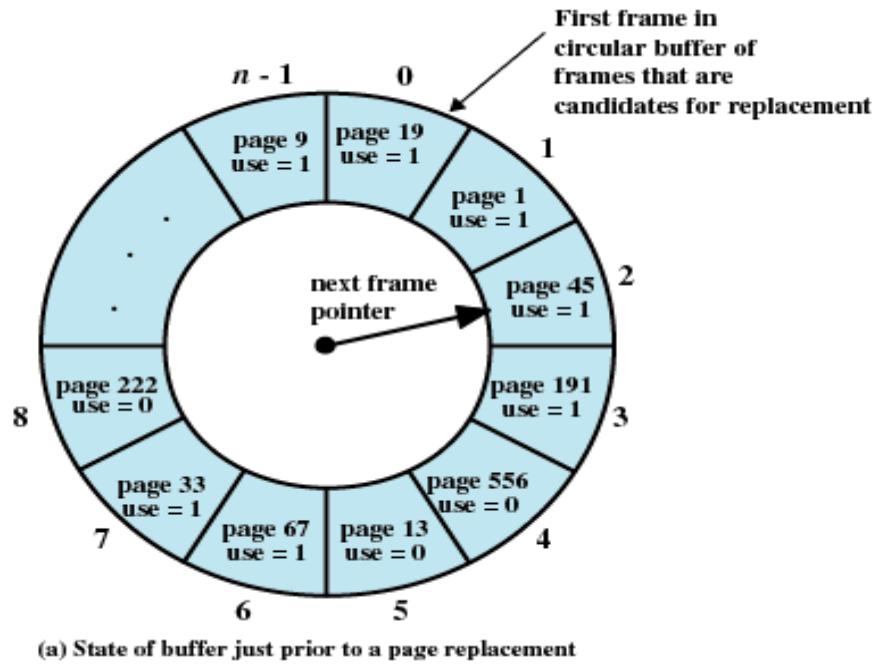
Cont...



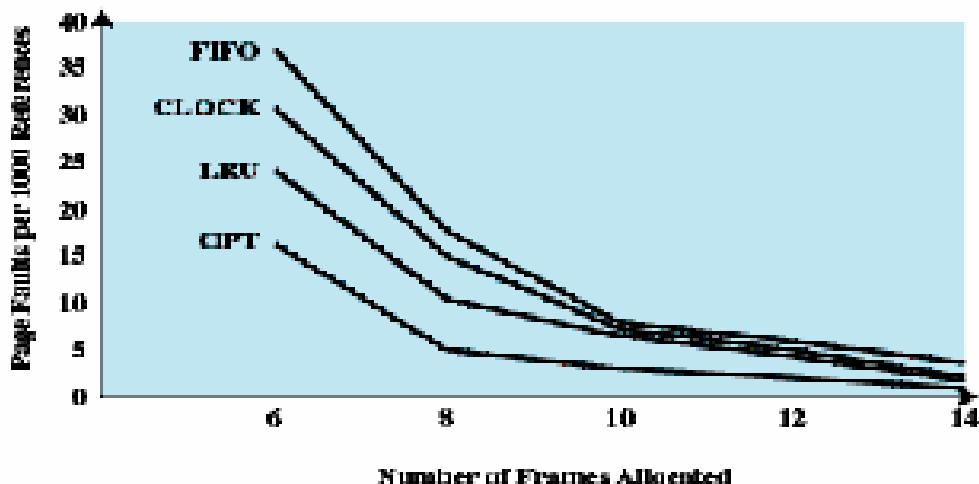
F - page fault occurring after the frame allocation is initially filled

Behavior of Four Page-Replacement Algorithms

Cont...



COMPARISON OF PLACEMENT ALGORITHMS



Comparison of Fixed-Allocation, Local Page Replacement Algorithms

BASIC REPLACEMENT ALGORITHMS

- Page Buffering
 - Replaced page is added to one of two lists
 - Free page list if page has not been modified
 - Modified page list

RESIDENT SET SIZE

- Fixed-allocation
 - Gives a process a fixed number of pages within which to execute
 - When a page fault occurs, one of the pages of that process must be replaced
- Variable-allocation
 - Number of pages allocated to a process varies over the lifetime of the process

FIXED ALLOCATION, LOCAL SCOPE

- Decide ahead of time the amount of allocation to give a process
- If allocation is too small, there will be a high page fault rate
- If allocation is too large there will be too few programs in main memory

VARIABLE ALLOCATION GLOBAL SCOPE

- Easiest to implement
- Adopted by many operating systems
- Operating system keeps list of free frames
- Free frame is added to resident set of process when a page fault occurs
- If no free frame, replaces one from another process

Cont...

- When new process added, allocate number of page frames based on application type, program request, or other criteria
- When page fault occurs, select page from among the resident set of the process that suffers the fault
- Reevaluate allocation from time to time

CLEANING POLICY

- Demand cleaning
 - A page is written out only when it has been selected for replacement
- Precleaning
 - Pages are written out in batches

CLEANING POLICY

- Best approach uses page buffering
 - Replaced pages are placed in two lists
 - **Modified and unmodified**
 - Pages in the modified list are periodically written out in batches
 - Pages in the unmodified list are either reclaimed if referenced again or lost when its frame is assigned to another page

LOAD CONTROL

- Determines the number of processes that will be resident in main memory
- Too few processes, many occasions when all processes will be blocked and much time will be spent in swapping
- Too many processes will lead to thrashing

PROCESS SUSPENSION

- Lowest priority process
- Faulting process
 - This process does not have its working set in main memory so it will be blocked anyway
- Last process activated
 - This process is least likely to have its working set resident

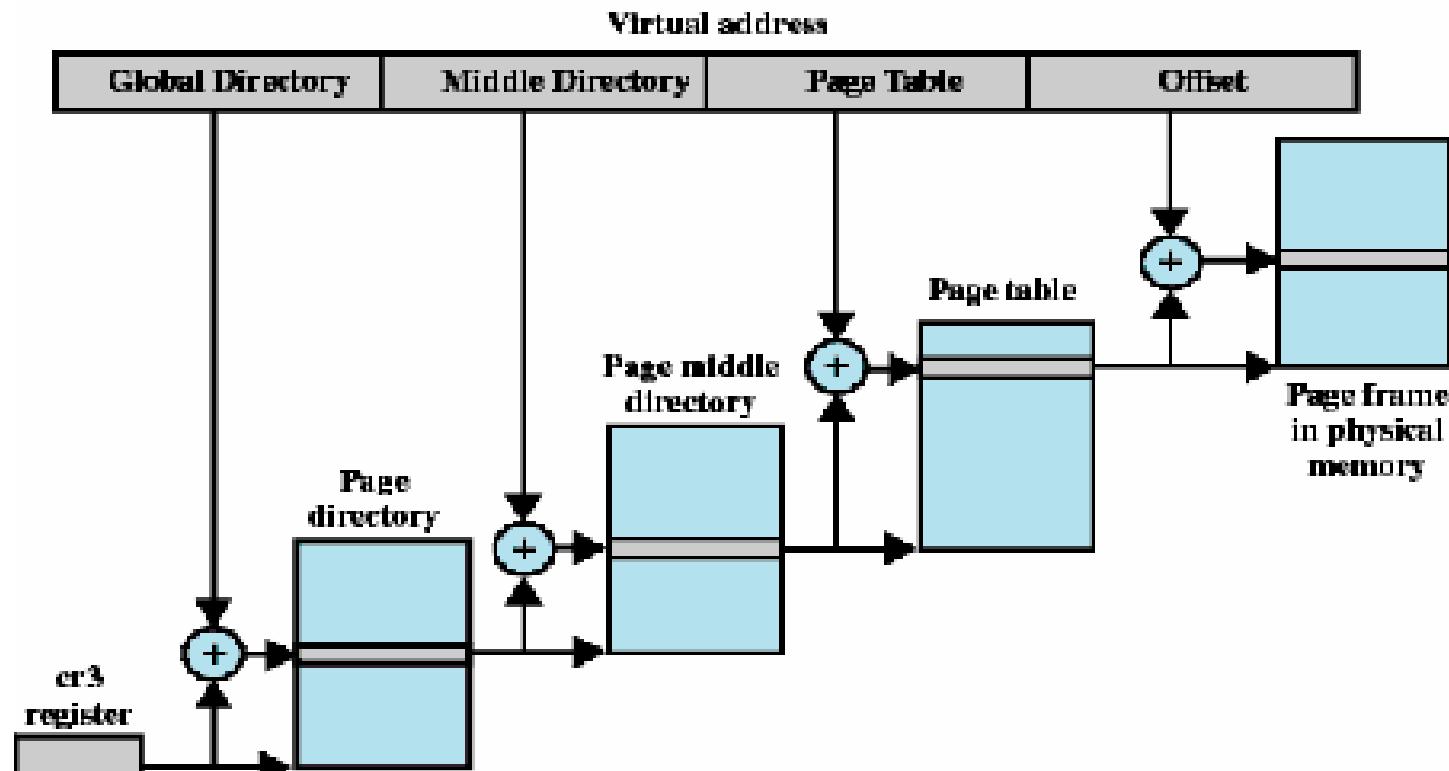
Cont...

- Process with smallest resident set
 - This process requires the least future effort to reload
- Largest process
 - Obtains the most free frames
- Process with the largest remaining execution window

LINUX MEMORY MANAGEMENT

- Page directory
- Page middle directory
- Page table

Cont...



Address Translation in Linux Virtual Memory Scheme

CONCLUSIONS

- Memory hierarchy
- Types of memory
- Mapping schemes
- Paging
- Segmentation
- Replacement Algorithm

William Stallings

Computer Organization

and Architecture

8th Edition

Chapter 4

Cache Memory

Characteristics

- Location
- Capacity
- Unit of transfer
- Access method
- Performance
- Physical type
- Physical characteristics
- Organisation

Location

- CPU
- Internal
- External

Capacity

- Word size
 - The natural unit of organisation
- Number of words
 - or Bytes

Unit of Transfer

- Internal
 - Usually governed by data bus width
- External
 - Usually a block which is much larger than a word
- Addressable unit
 - Smallest location which can be uniquely addressed
 - Word internally
 - Cluster on M\$ disks

Access Methods (1)

- Sequential
 - Start at the beginning and read through in order
 - Access time depends on location of data and previous location
 - e.g. tape
- Direct
 - Individual blocks have unique address
 - Access is by jumping to vicinity plus sequential search
 - Access time depends on location and previous location
 - e.g. disk

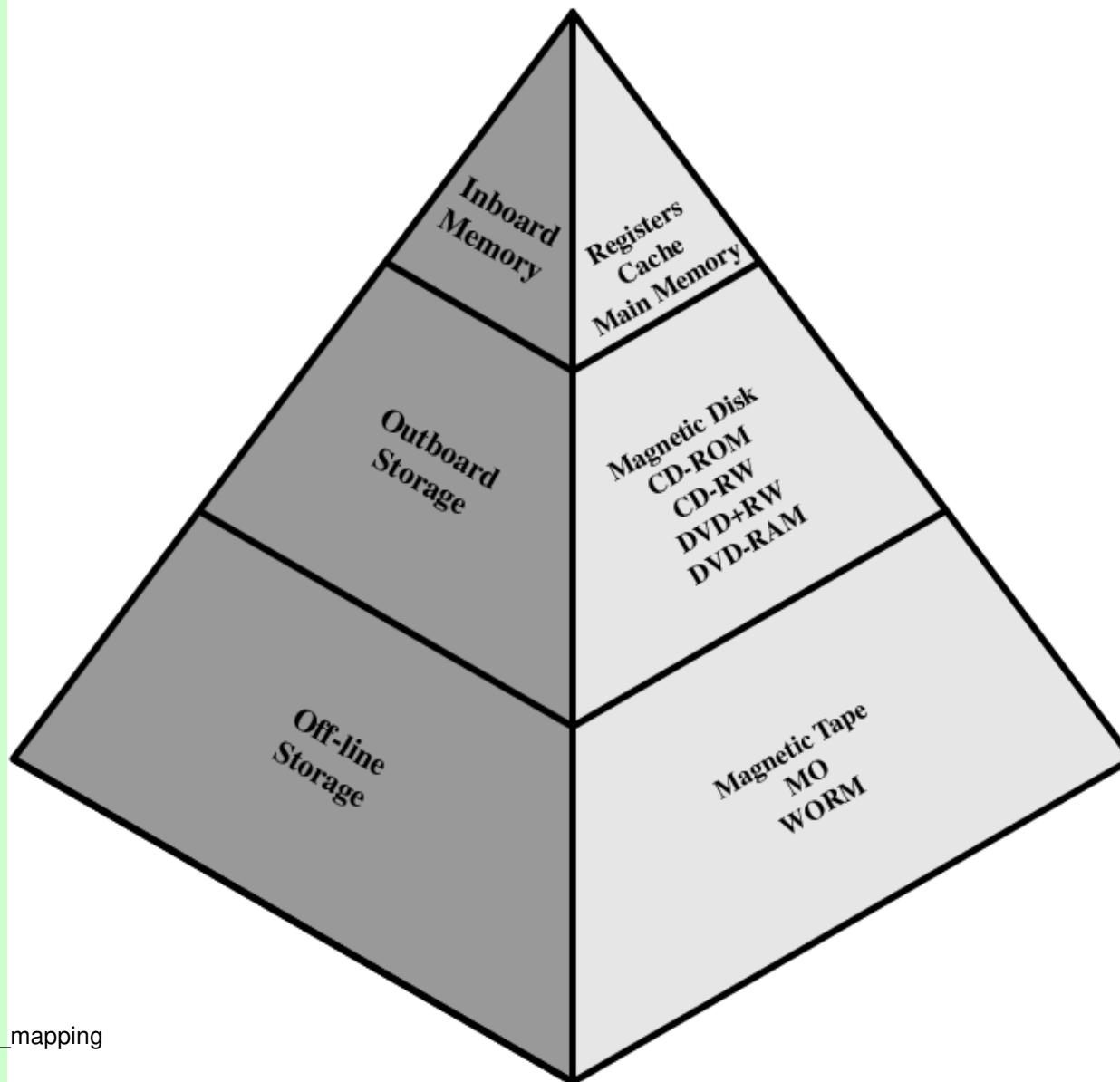
Access Methods (2)

- Random
 - Individual addresses identify locations exactly
 - Access time is independent of location or previous access
 - e.g. RAM
- Associative
 - Data is located by a comparison with contents of a portion of the store
 - Access time is independent of location or previous access
 - e.g. cache

Memory Hierarchy

- Registers
 - In CPU
- Internal or Main memory
 - May include one or more levels of cache
 - “RAM”
- External memory
 - Backing store

Memory Hierarchy - Diagram



Performance

- Access time
 - Time between presenting the address and getting the valid data
- Memory Cycle time
 - Time may be required for the memory to “recover” before next access
 - Cycle time is access + recovery
- Transfer Rate
 - Rate at which data can be moved

Physical Types

- Semiconductor
 - RAM
- Magnetic
 - Disk & Tape
- Optical
 - CD & DVD
- Others
 - Bubble
 - Hologram

Physical Characteristics

- Decay
- Volatility
- Erasable
- Power consumption

Organisation

- Physical arrangement of bits into words
- Not always obvious
- e.g. interleaved

The Bottom Line

- How much?
 - Capacity
- How fast?
 - Time is money
- How expensive?

Hierarchy List

- Registers
- L1 Cache
- L2 Cache
- Main memory
- Disk cache
- Disk
- Optical
- Tape

So you want fast?

- It is possible to build a computer which uses only static RAM (see later)
- This would be very fast
- This would need no cache
 - How can you cache cache?
- This would cost a very large amount

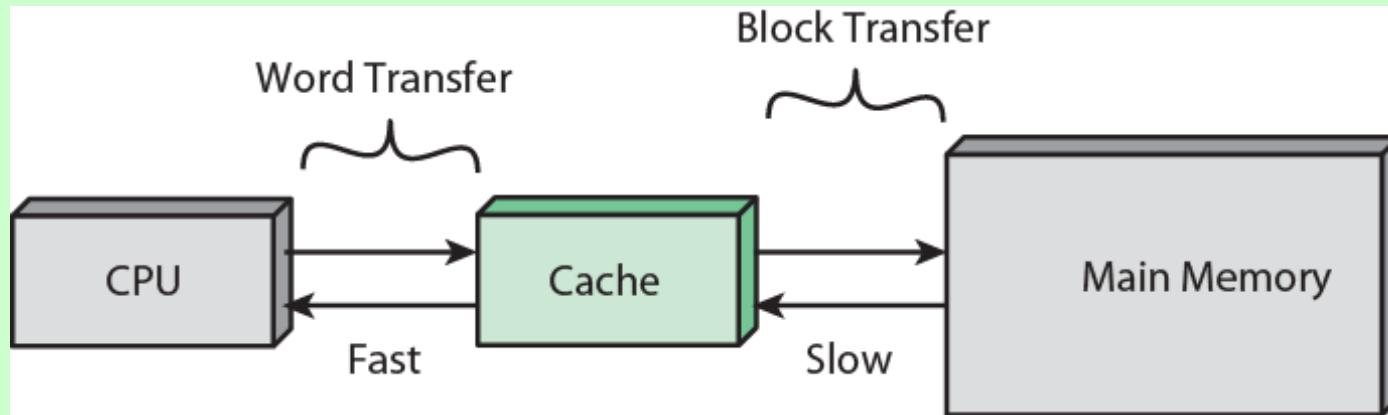
Locality of Reference

- During the course of the execution of a program, memory references tend to cluster
- e.g. loops

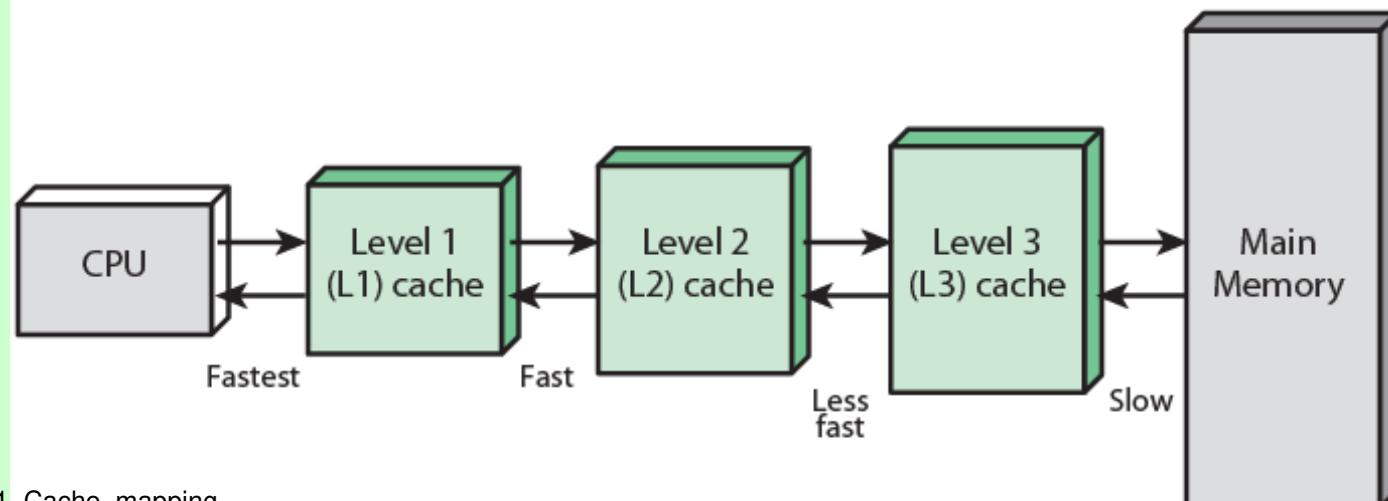
Cache

- Small amount of fast memory
- Sits between normal main memory and CPU
- May be located on CPU chip or module

Cache and Main Memory

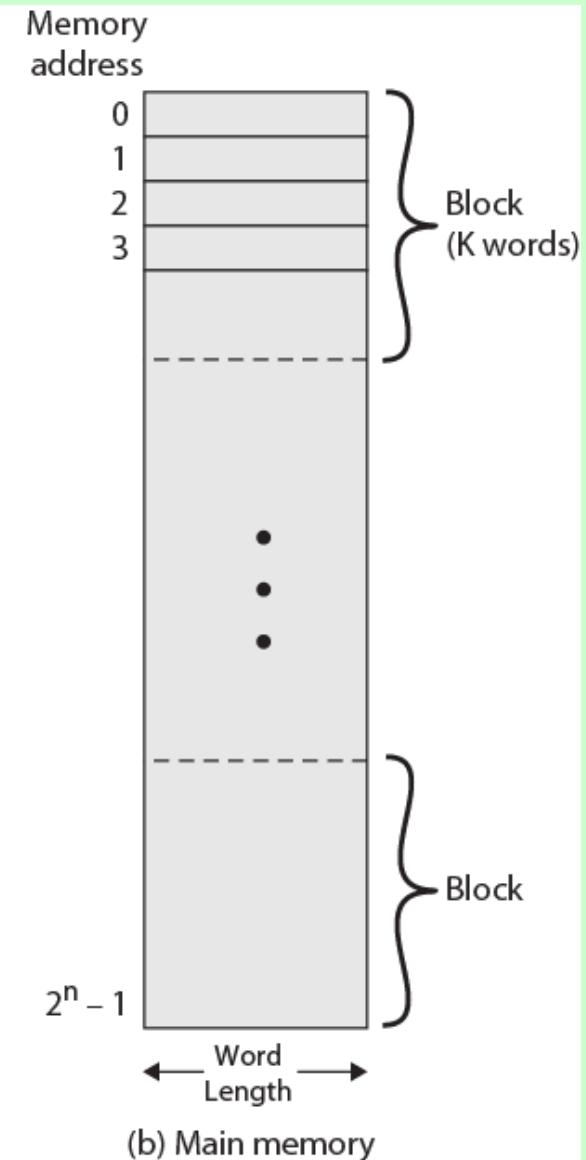
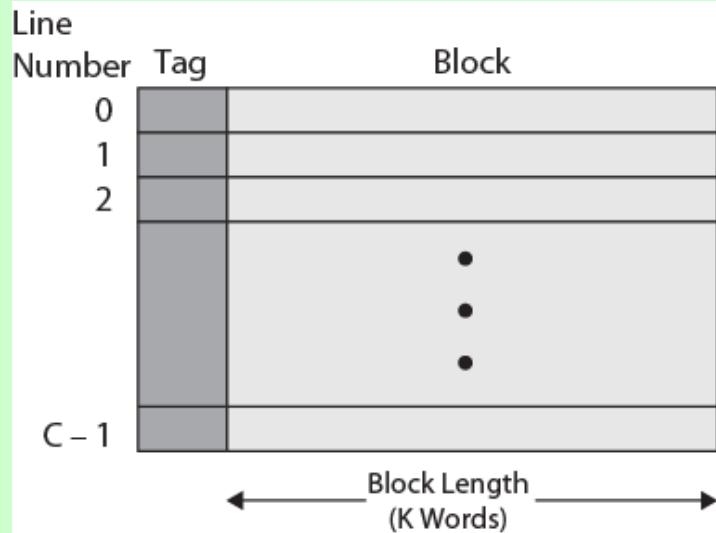


(a) Single cache



(b) Three-level cache organization

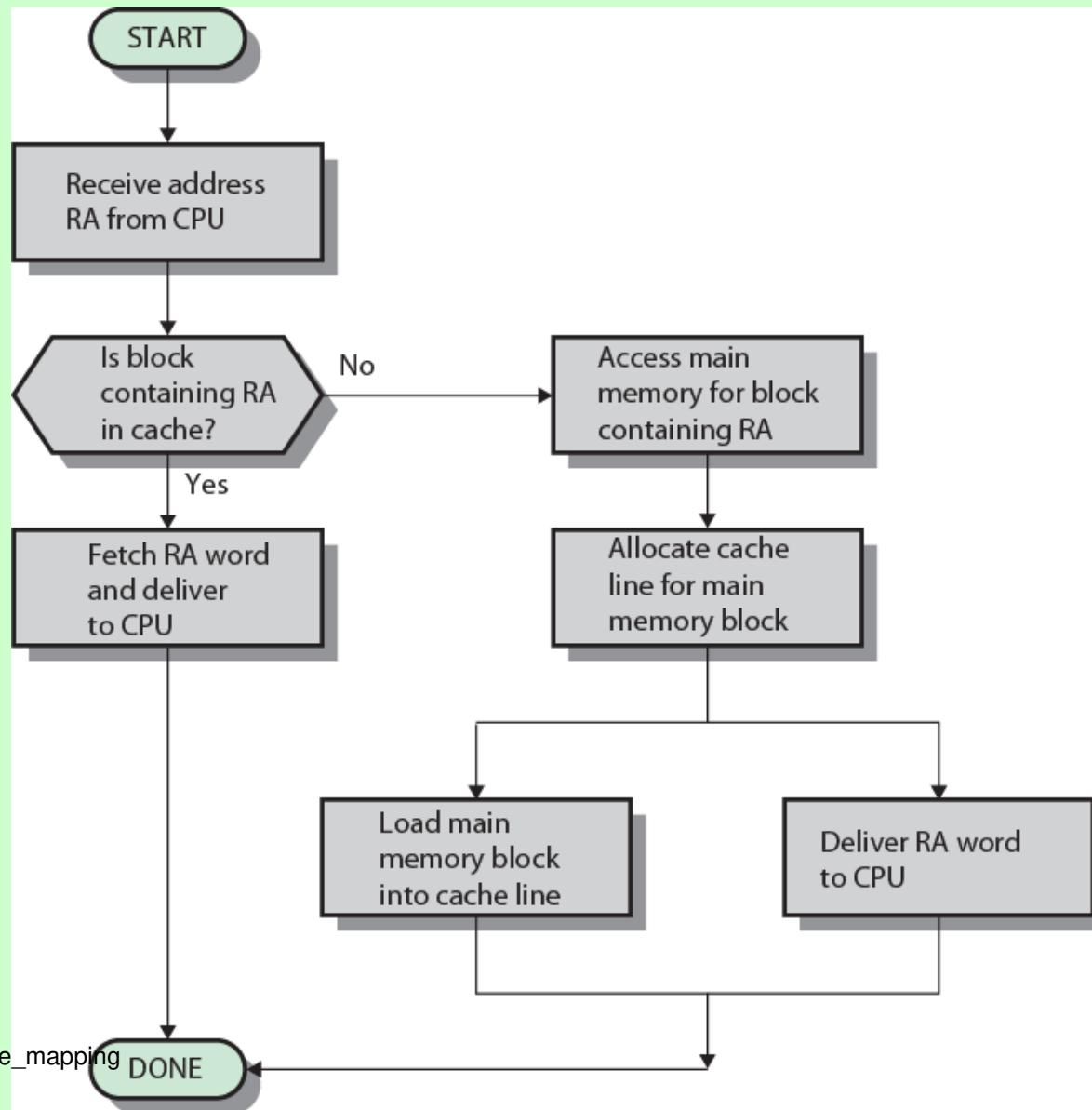
Cache/Main Memory Structure



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

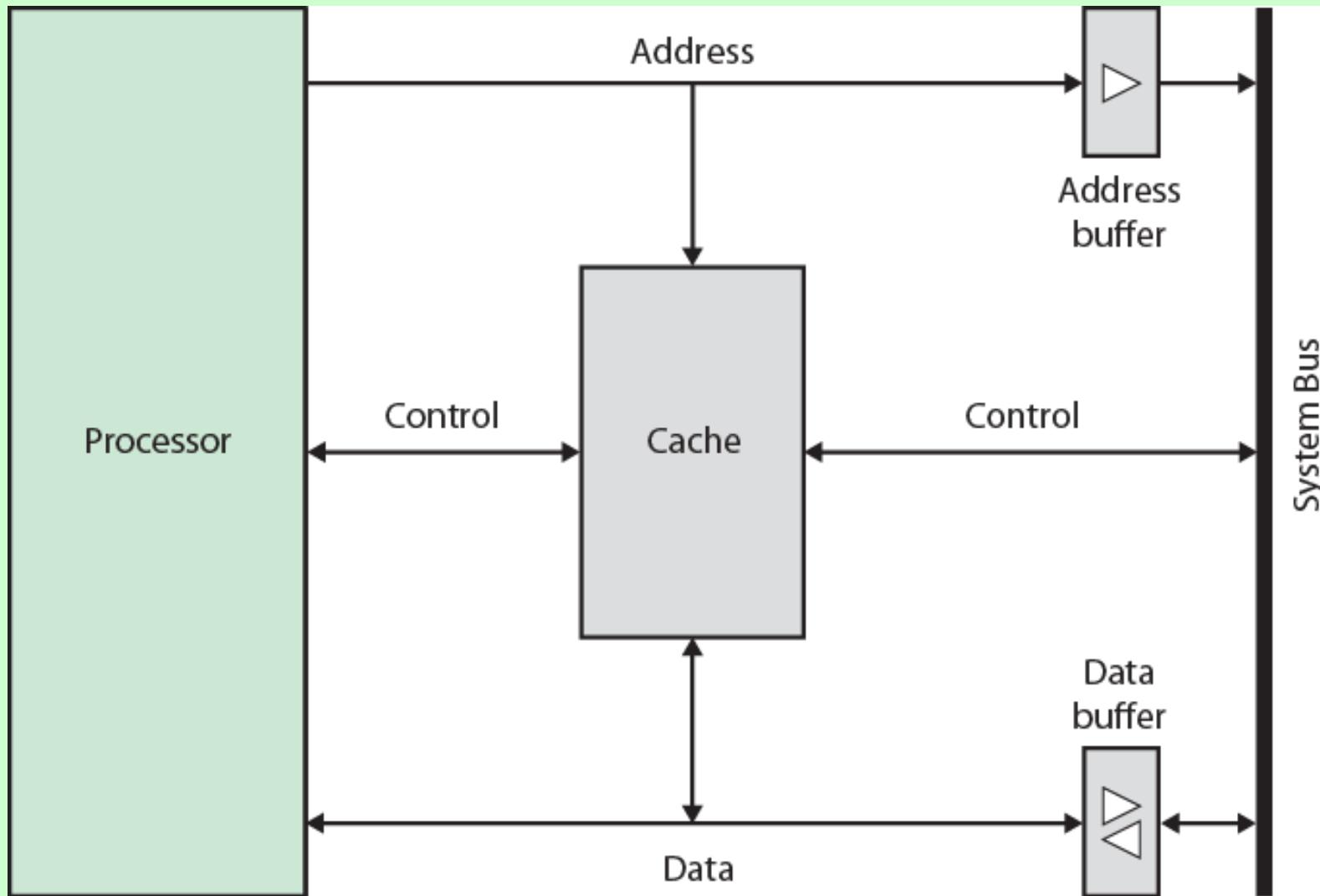
Cache Addressing

- 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 through 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



Comparison of Cache Sizes

Processor	Type	Year of Introduction	L1 cache	L2 cache	L3 cache
IBM 360/85	Mainframe	1968	16 to 32 KB	—	—
PDP-11/70	Minicomputer	1975	1 KB	—	—
VAX 11/780	Minicomputer	1978	16 KB	—	—
IBM 3033	Mainframe	1978	64 KB	—	—
IBM 3090	Mainframe	1985	128 to 256 KB	—	—
Intel 80486	PC	1989	8 KB	—	—
Pentium	PC	1993	8 KB/8 KB	256 to 512 KB	—
PowerPC 601	PC	1993	32 KB	—	—
PowerPC 620	PC	1996	32 KB/32 KB	—	—
PowerPC G4	PC/server	1999	32 KB/32 KB	256 KB to 1 MB	2 MB
IBM S/390 G4	Mainframe	1997	32 KB	256 KB	2 MB
IBM S/390 G6	Mainframe	1999	256 KB	8 MB	—
Pentium 4	PC/server	2000	8 KB/8 KB	256 KB	—
IBM SP	High-end server/ supercomputer	2000	64 KB/32 KB	8 MB	—
CRAY MTA ^b	Supercomputer	2000	8 KB	2 MB	—
Itanium	PC/server	2001	16 KB/16 KB	96 KB	4 MB
SGI Origin 2001	High-end server	2001	32 KB/32 KB	4 MB	—
Itanium 2	PC/server	2002	32 KB	256 KB	6 MB
IBM POWER5 29-Jun-2021 Cache mapping	High-end server	2003	64 KB	1.9 MB	36 MB
CRAY XD-1	Supercomputer	2004	64 KB/64 KB	1MB	—

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)

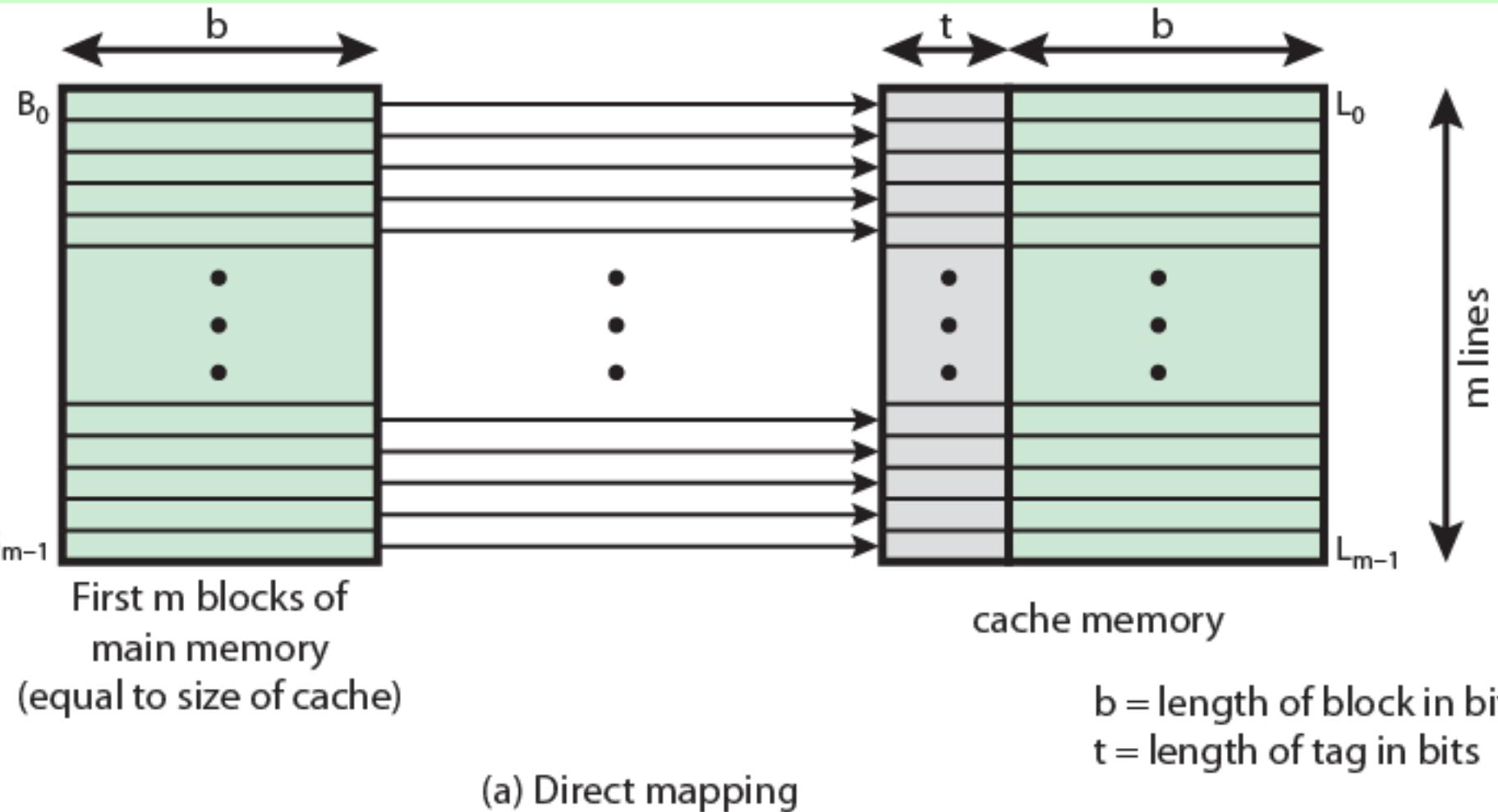
Direct Mapping

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

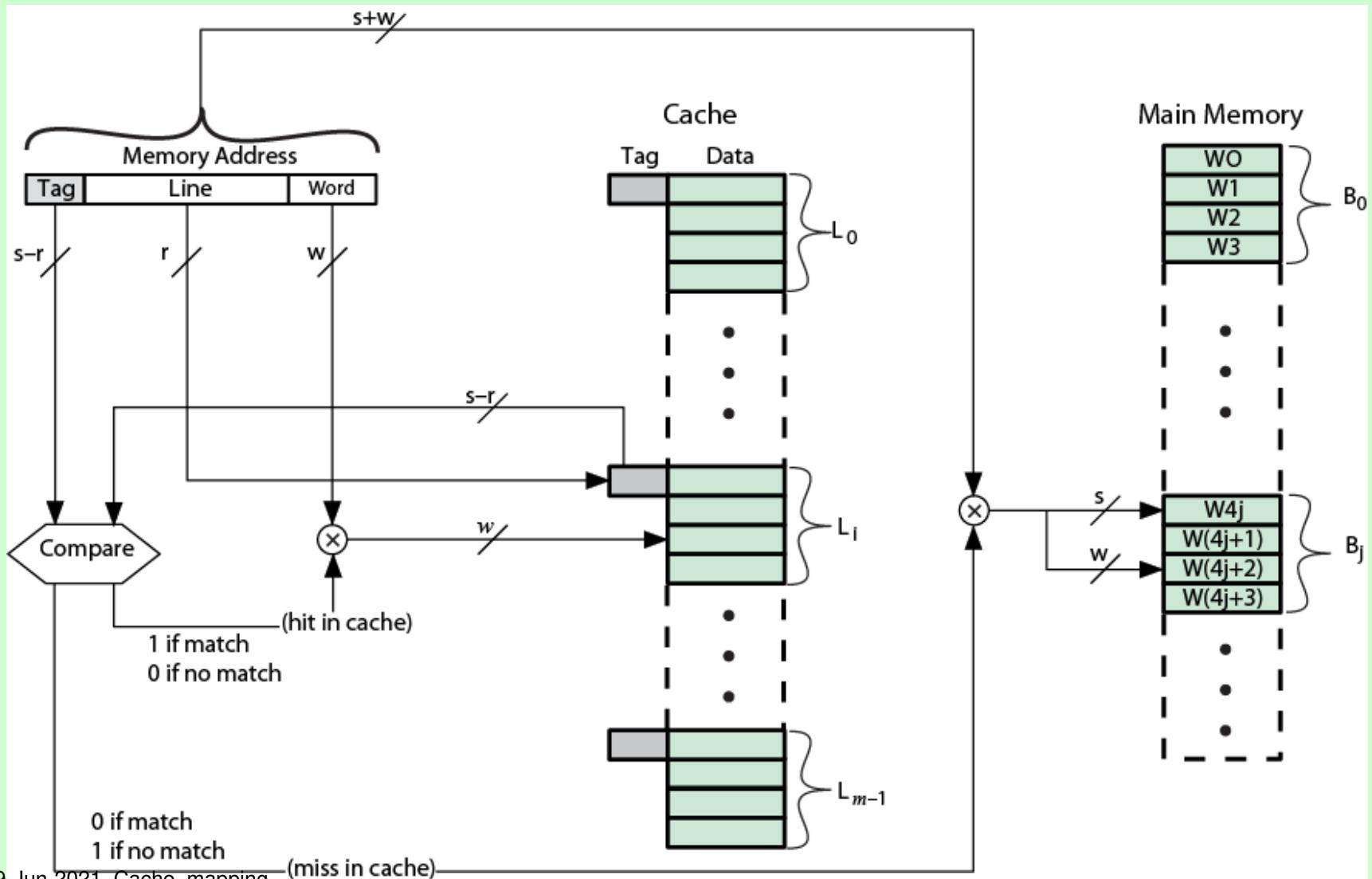


Direct Mapping

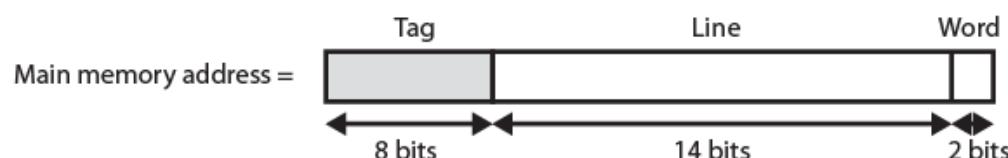
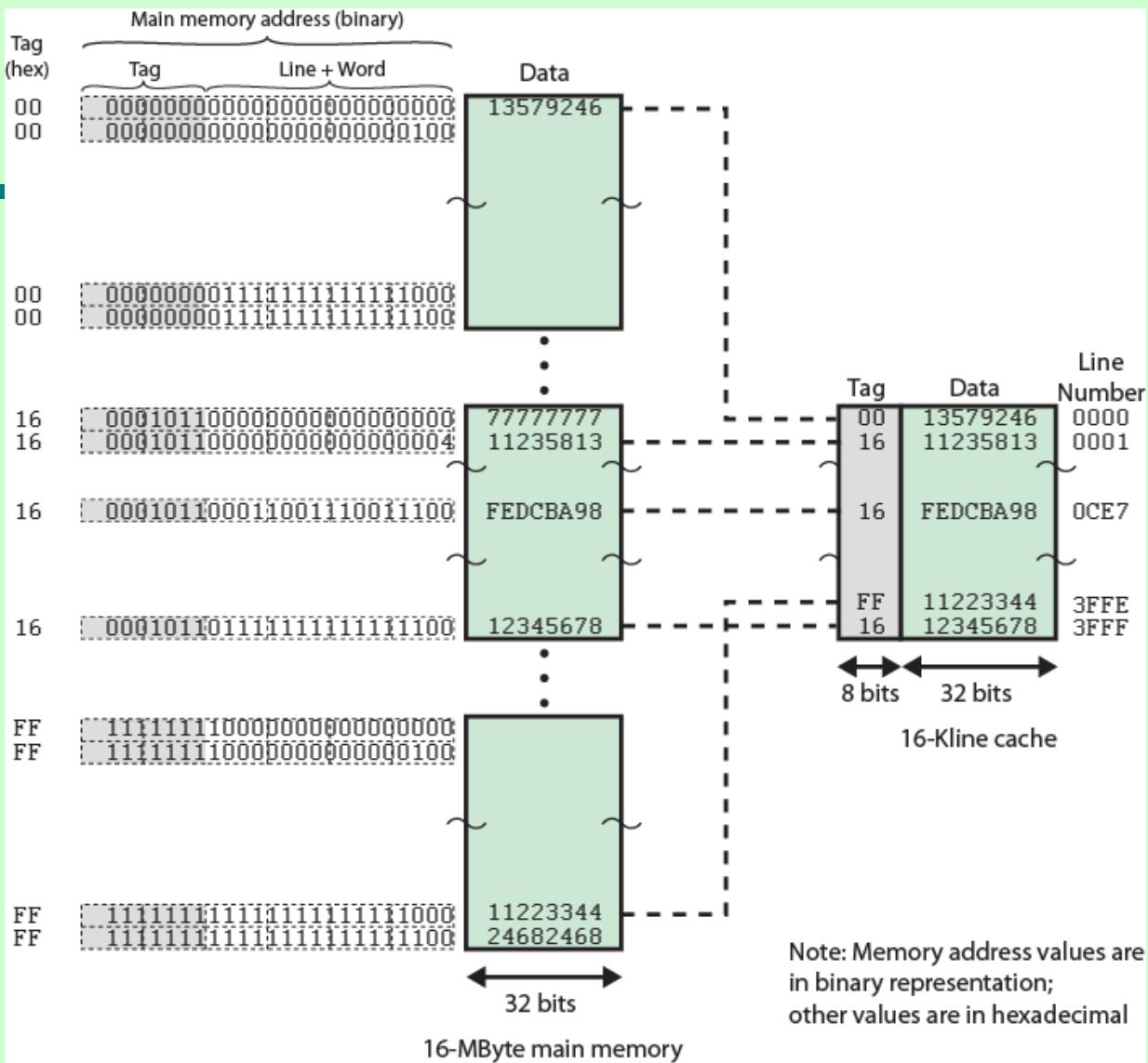
Cache Line Table

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



Direct Mapping Example



Direct Mapping Summary

- Address length = $(s + w)$ bits
- Number of addressable units = 2^{s+w} words or bytes
- Block size = line size = 2^w words or bytes
- Number of blocks in main memory = $2^{s+w}/2^w = 2^s$
- Number of lines in cache = $m = 2^r$
- Size of tag = $(s - r)$ bits

Direct Mapping pros & cons

- Simple
- Inexpensive
- Fixed location for given block
 - If a program accesses 2 blocks that map to the same line repeatedly, cache misses are very high

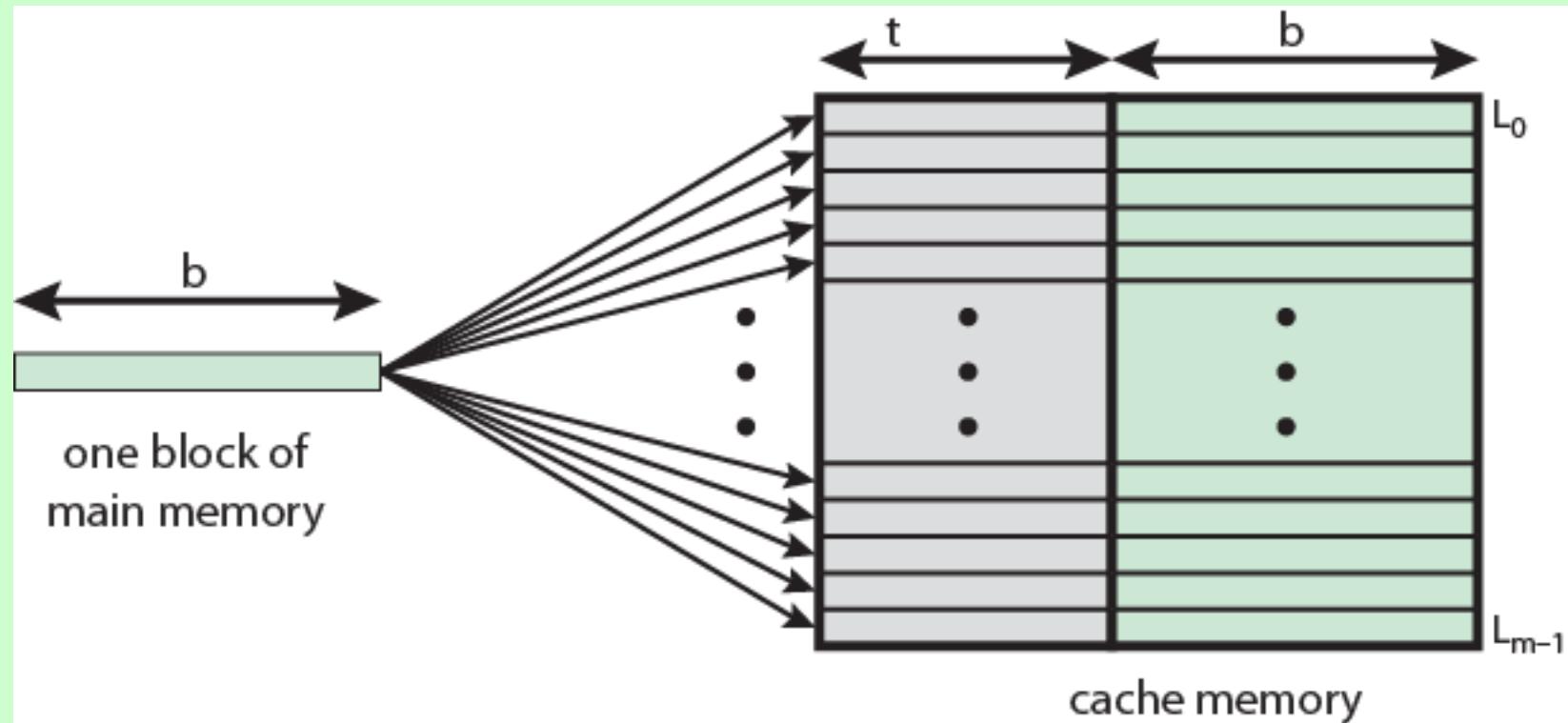
Victim Cache

- Lower miss penalty
- Remember what was discarded
 - Already fetched
 - Use again with little penalty
- Fully associative
- 4 to 16 cache lines
- Between direct mapped L1 cache and next memory level

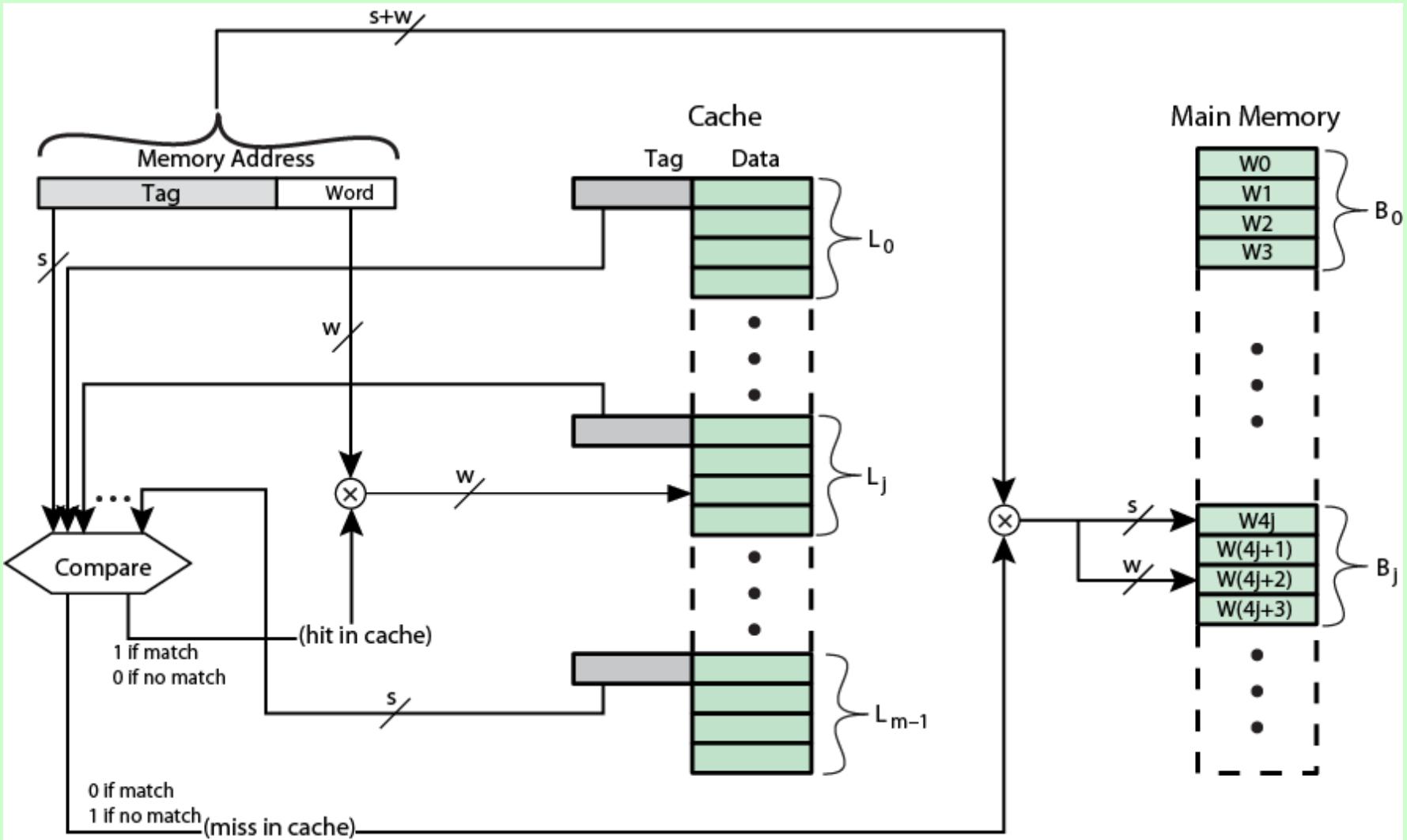
Associative Mapping

- A main memory block can load into any line of cache
- Memory address is interpreted as tag and word
- Tag uniquely identifies block of memory
- Every line's tag is examined for a match
- Cache searching gets expensive

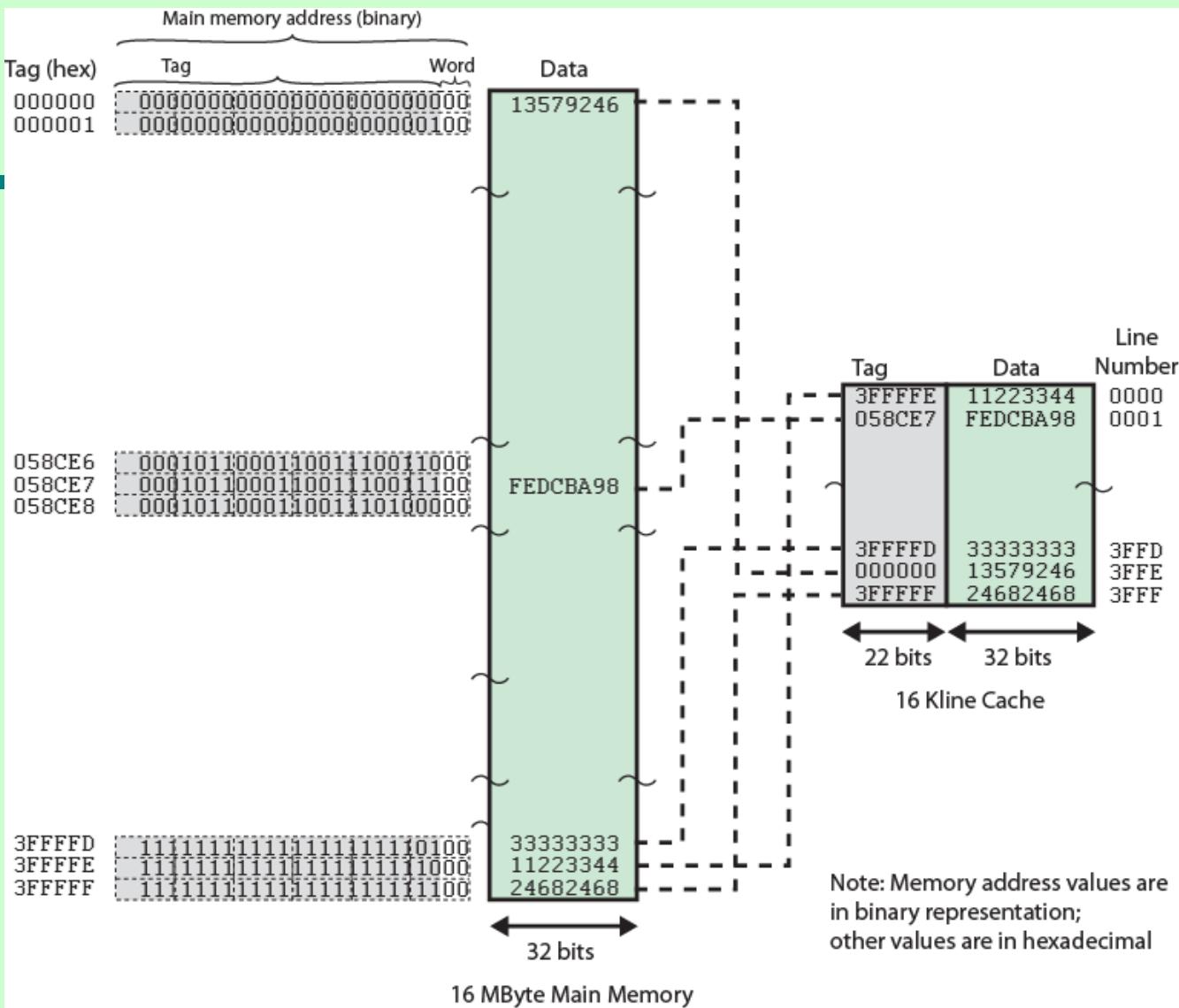
Associative Mapping from Cache to Main Memory



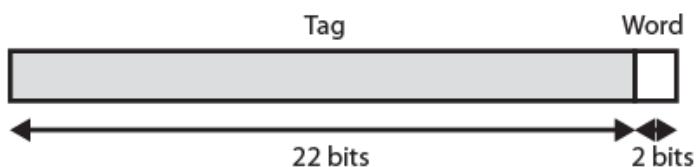
Fully Associative Cache Organization



Associative Mapping Example



Main Memory Address =



Associative Mapping

Address Structure

Tag 22 bit	Word 2 bit
------------	------------

- 22 bit tag stored with each 32 bit block of data
- Compare tag field with tag entry in cache to check for hit
- Least significant 2 bits of address identify which 16 bit word is required from 32 bit data block
- e.g.

Address	Tag	Data	Cache line
FFFFFC	FFFFC24682468	3FFF	

Associative Mapping Summary

- Address length = $(s + w)$ bits
- Number of addressable units = 2^{s+w} words or bytes
- Block size = line size = 2^w words or bytes
- Number of blocks in main memory = $2^{s+w}/2^w = 2^s$
- Number of lines in cache = undetermined
- Size of tag = s bits

Set Associative Mapping

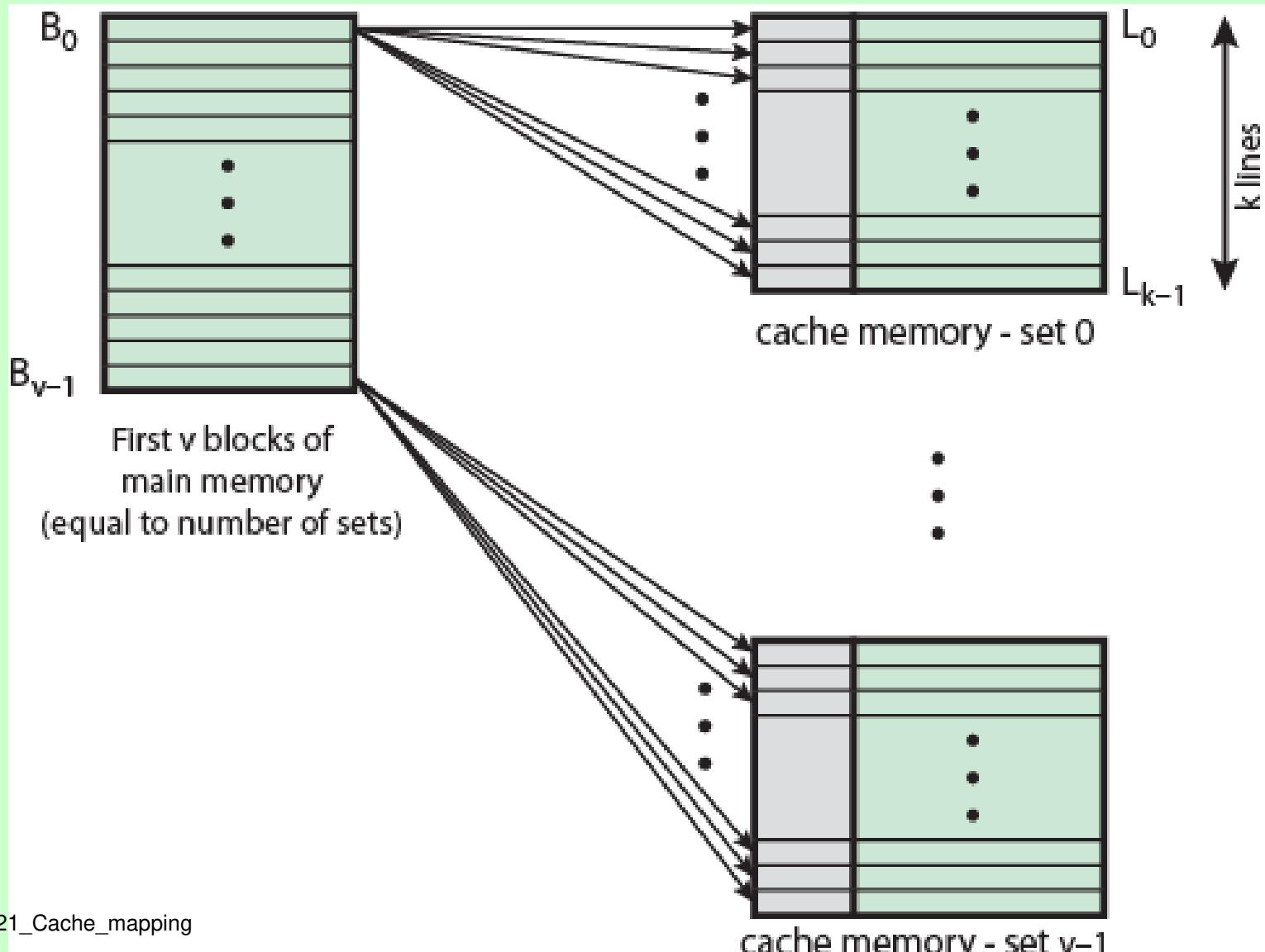
- Cache is divided into a number of sets
- Each set contains a number of lines
- A given block maps to any line in a given set
 - e.g. Block B can be in any line of set i
- e.g. 2 lines per set
 - 2 way associative mapping
 - A given block can be in one of 2 lines in only one set

Set Associative Mapping

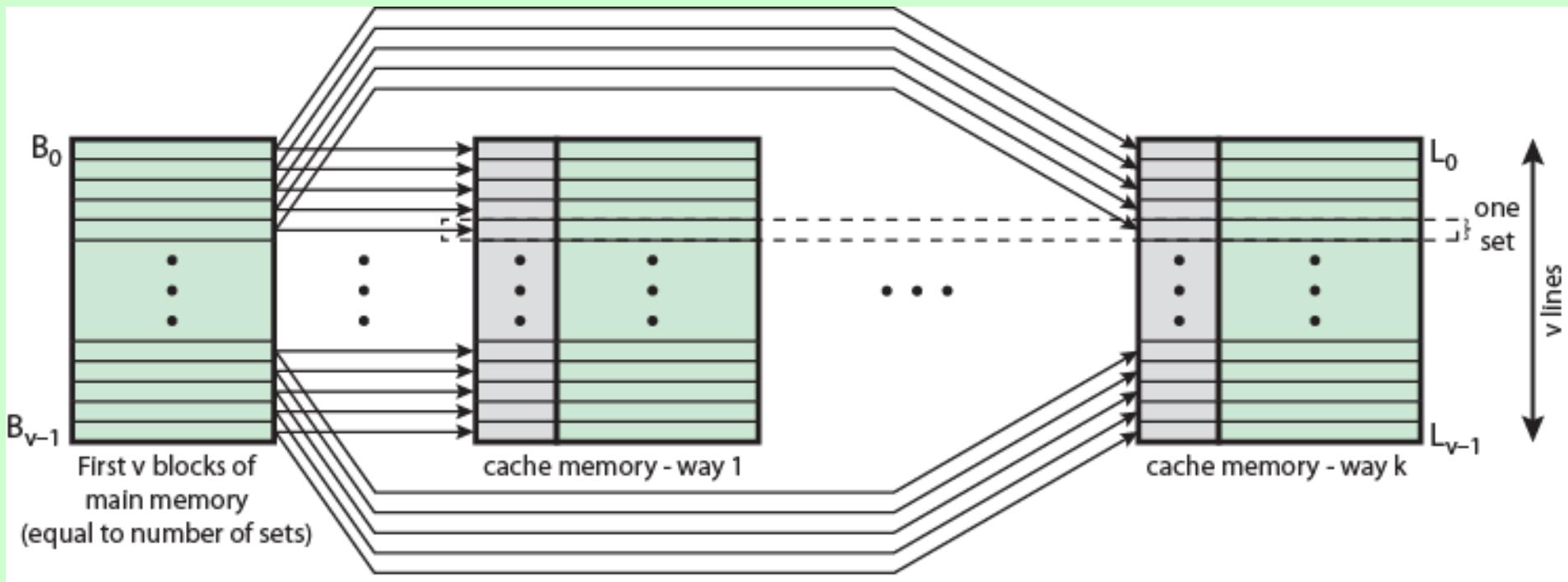
Example

- 13 bit set number
- Block number in main memory is modulo 2^{13}
- 000000, 00A000, 00B000, 00C000 ... map to same set

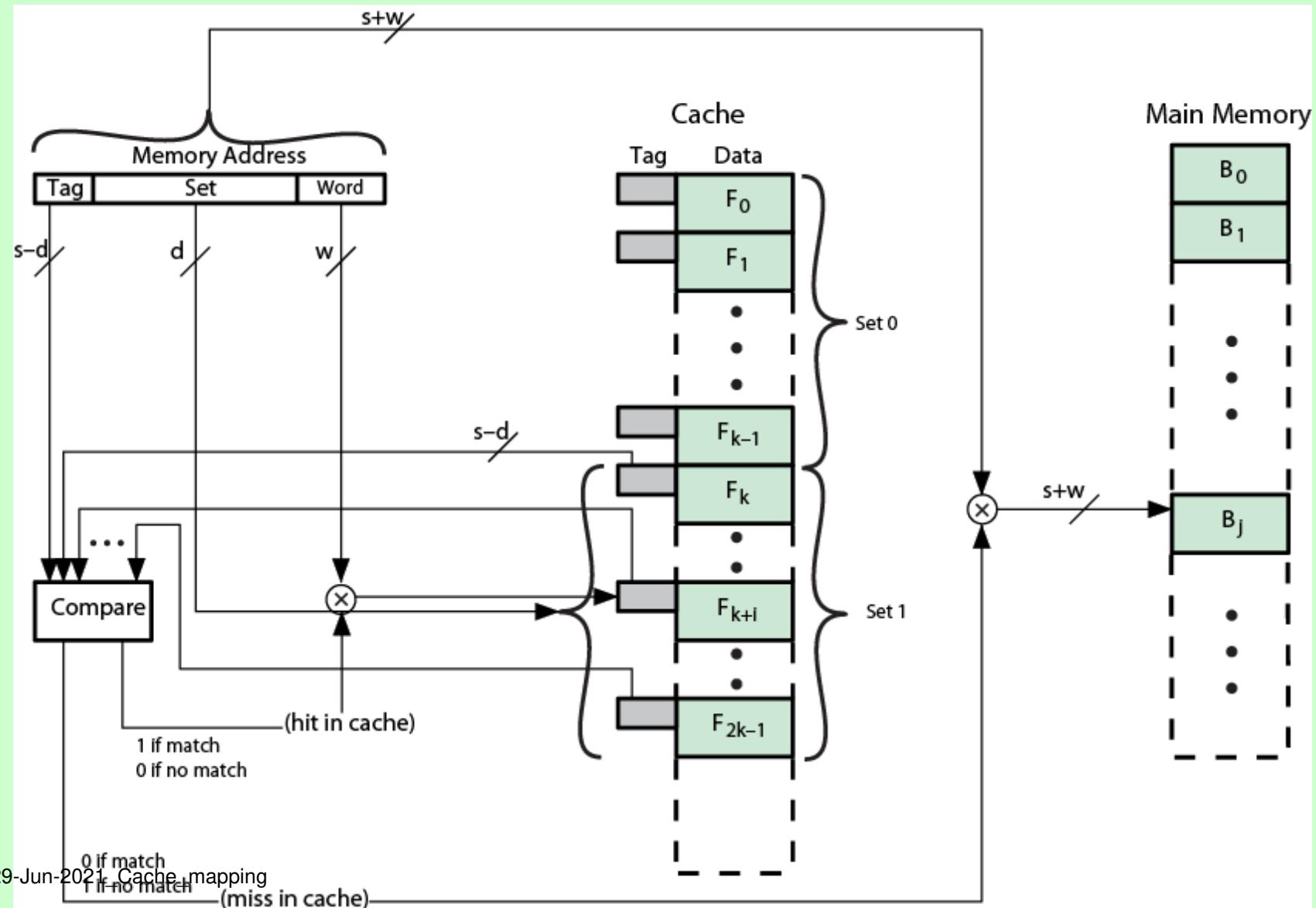
Mapping From Main Memory to Cache: v Associative



Mapping From Main Memory to Cache: k-way Associative



K-Way Set Associative Cache Organization



Set Associative Mapping

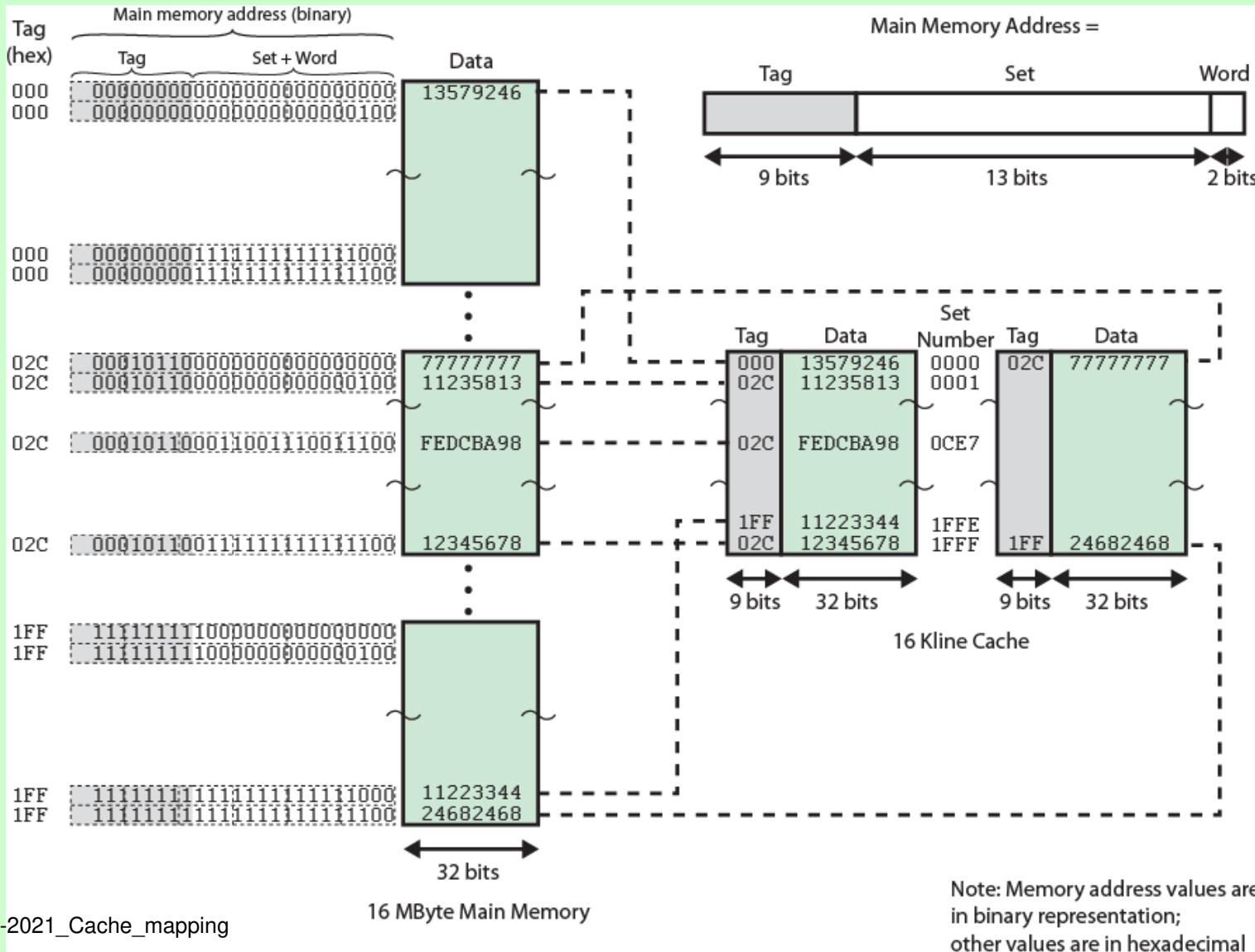
Address Structure

Tag 9 bit	Set 13 bit	Word 2 bit
-----------	------------	------------

- Use set field to determine cache set to look in
- Compare tag field to see if we have a hit
- e.g

—Address number	Tag	Data	Set
—1FF 7FFC	1FF	12345678	1FFF
—001 7FFC	001	11223344	1FFF

Two Way Set Associative Mapping Example



Set Associative Mapping Summary

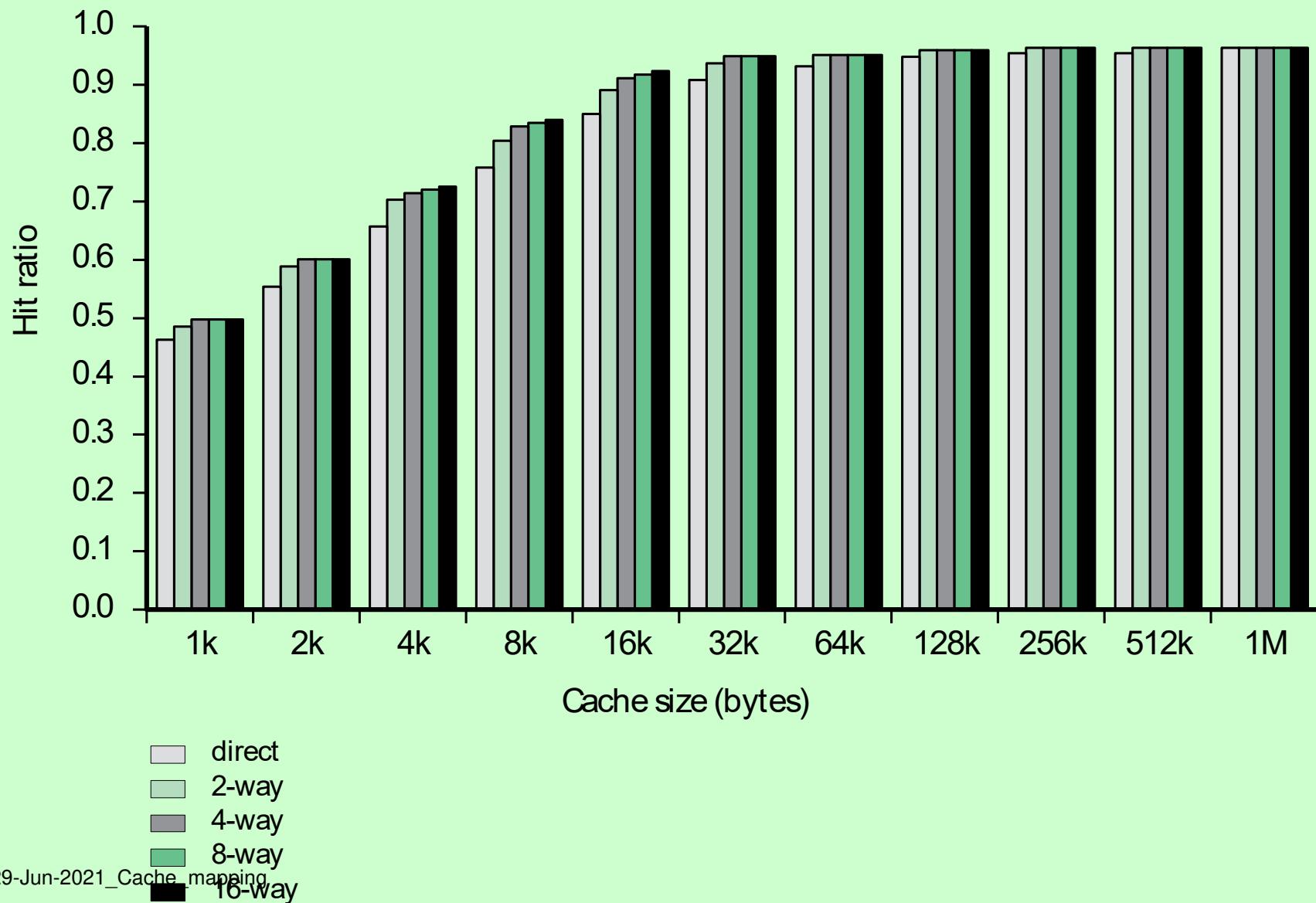
- Address length = $(s + w)$ bits
- Number of addressable units = 2^{s+w} words or bytes
- Block size = line size = 2^w words or bytes
- Number of blocks in main memory = 2^d
- Number of lines in set = k
- Number of sets = $v = 2^d$
- Number of lines in cache = $kv = k * 2^d$
- Size of tag = $(s - d)$ bits

Direct and Set Associative Cache Performance Differences

- Significant up to at least 64kB for 2-way
- Difference between 2-way and 4-way at 4kB much less than 4kB to 8kB
- Cache complexity increases with associativity
- Not justified against increasing cache to 8kB or 16kB
- Above 32kB gives no improvement
- (simulation results)

Figure 4.16

Varying Associativity over Cache Size



Replacement Algorithms (1)

Direct mapping

- No choice
- Each block only maps to one line
- Replace that line

Replacement Algorithms (2)

Associative & Set Associative

- Hardware implemented algorithm (speed)
- Least Recently used (LRU)
 - Which of the 2 block is lru?
- First in first out (FIFO)
 - replace block that has been in cache longest
- Least frequently used
 - replace block which has had fewest hits
- Random

Write Policy

- Must not overwrite a cache block unless main memory is up to date
- Multiple CPUs may have individual caches
- I/O may address main memory directly

Write through

- All writes go to main memory as well as cache
- Multiple CPUs can monitor main memory traffic to keep local (to CPU) cache up to date
- Lots of traffic
- Slows down writes
- Remember bogus write through caches!

Write back

- Updates initially made in cache only
- Update bit for cache slot is set when update occurs
- If block is to be replaced, write to main memory only if update bit is set
- Other caches get out of sync
- I/O must access main memory through cache
- N.B. 15% of memory references are writes

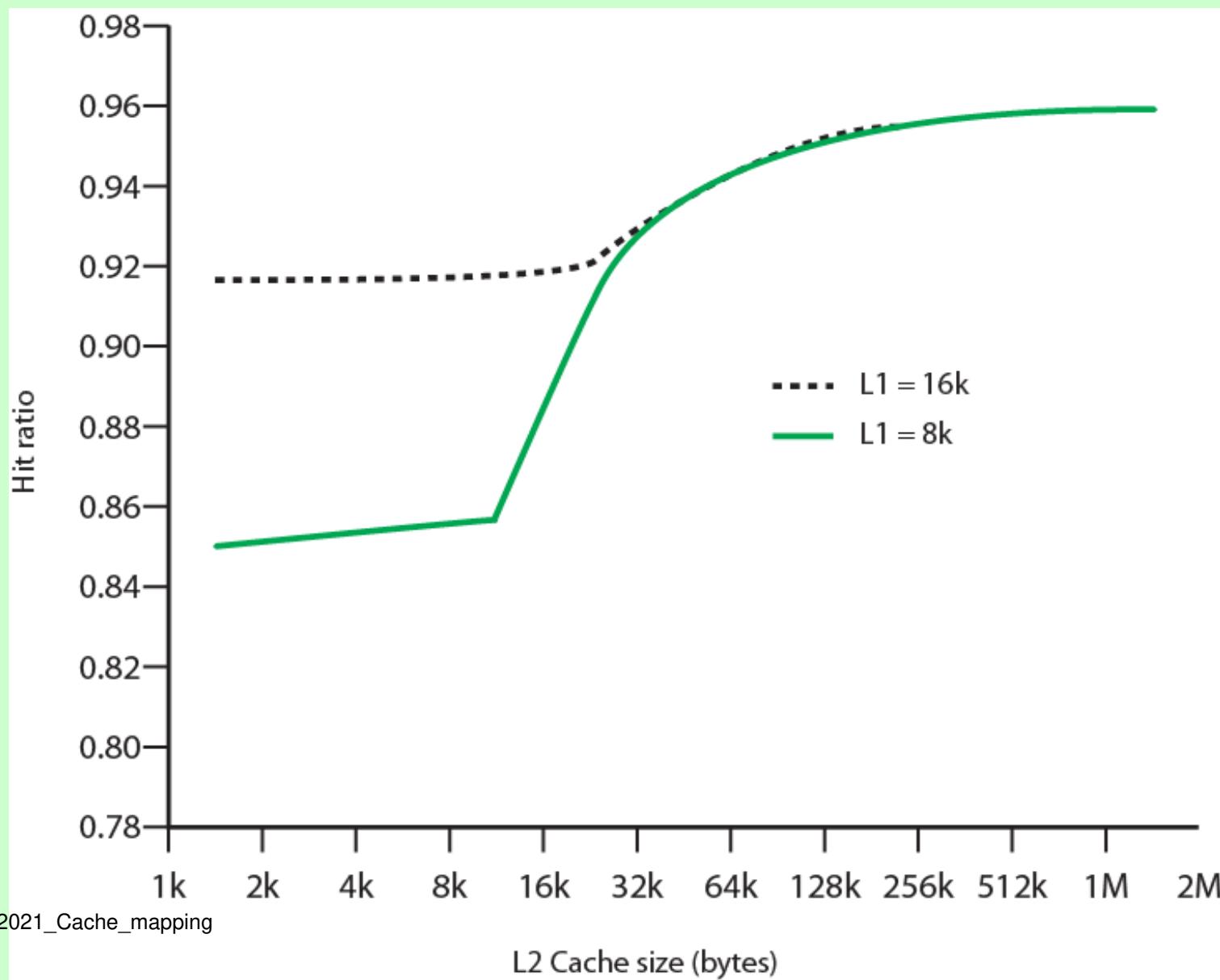
Line Size

- Retrieve not only desired word but a number of adjacent words as well
- Increased block size will increase hit ratio at first
 - the principle of locality
- Hit ratio will decreases as block becomes even bigger
 - Probability of using newly fetched information becomes less than probability of reusing replaced
- Larger blocks
 - Reduce number of blocks that fit in cache
 - Data overwritten shortly after being fetched
 - Each additional word is less local so less likely to be needed
- No definitive optimum value has been found
- 8 to 64 bytes seems reasonable
- For HPC systems, 64- and 128-byte most common

Multilevel Caches

- High logic density enables caches on chip
 - Faster than bus access
 - Frees bus for other transfers
- Common to use both on and off chip cache
 - L1 on chip, L2 off chip in static RAM
 - L2 access much faster than DRAM or ROM
 - L2 often uses separate data path
 - L2 may now be on chip
 - Resulting in L3 cache
 - Bus access or now on chip...

Hit Ratio (L1 & L2) For 8 kbytes and 16 kbyte L1



Unified v Split Caches

- One cache for data and instructions or two, one for data and one for instructions
- Advantages of unified cache
 - Higher hit rate
 - Balances load of instruction and data fetch
 - Only one cache to design & implement
- Advantages of split cache
 - Eliminates cache contention between instruction fetch/decode unit and execution unit
 - Important in pipelining

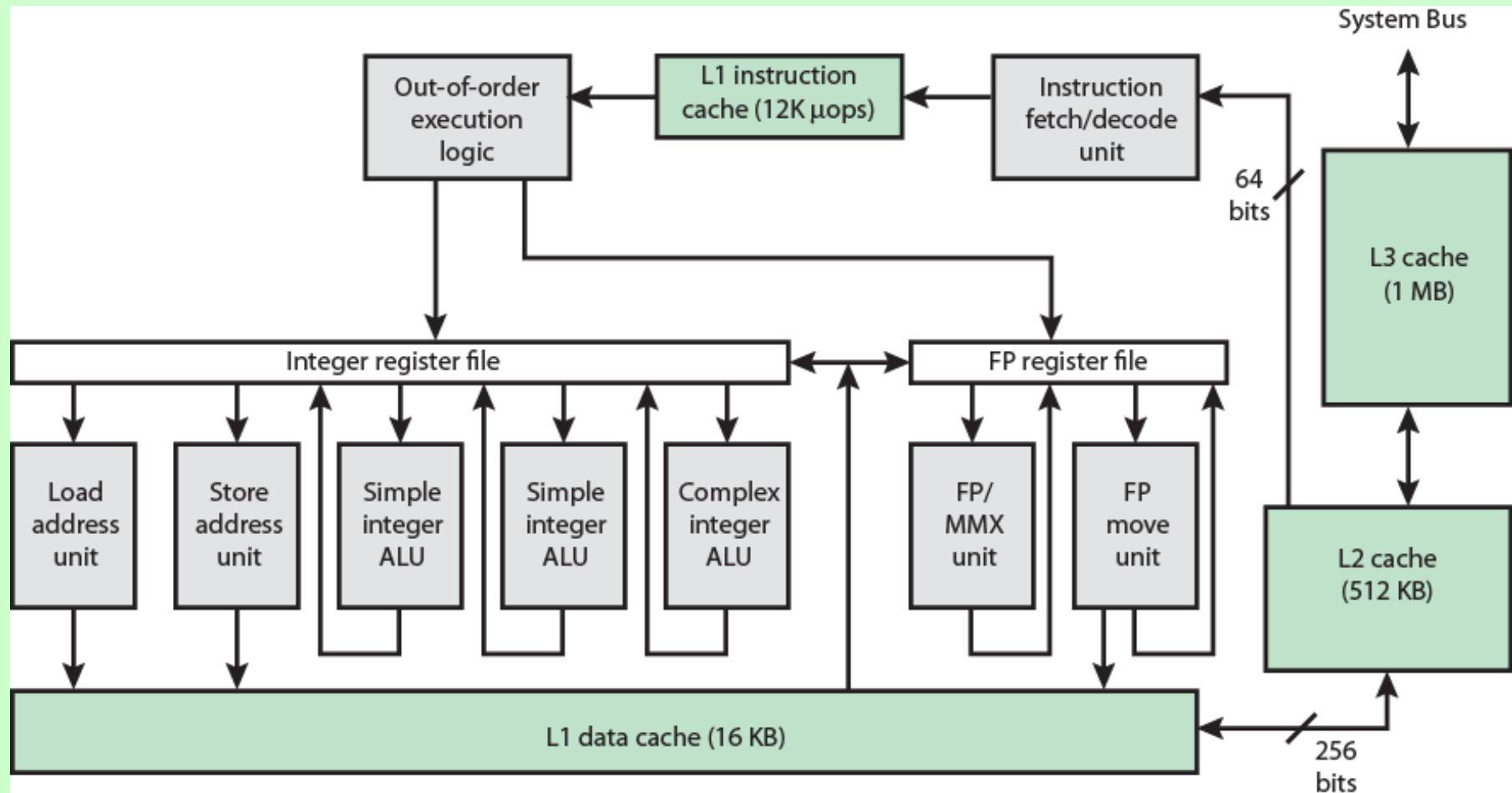
Pentium 4 Cache

- 80386 – no on chip cache
- 80486 – 8k using 16 byte lines and four way set associative organization
- Pentium (all versions) – two on chip L1 caches
 - Data & instructions
- Pentium III – L3 cache added off chip
- Pentium 4
 - L1 caches
 - 8k bytes
 - 64 byte lines
 - four way set associative
 - L2 cache
 - Feeding both L1 caches
 - 256k
 - 128 byte lines
 - 8 way set associative
 - L3 cache on chip

Intel Cache Evolution

Problem	Solution	Processor on which feature first appears
External memory slower than the system bus.	Add external cache using faster memory technology.	386
Increased processor speed results in external bus becoming a bottleneck for cache access.	Move external cache on-chip, operating at the same speed as the processor.	486
Internal cache is rather small, due to limited space on chip	Add external L2 cache using faster technology than main memory	486
Contention occurs when both the Instruction Prefetcher and the Execution Unit simultaneously require access to the cache. In that case, the Prefetcher is stalled while the Execution Unit's data access takes place.	Create separate data and instruction caches.	Pentium
Increased processor speed results in external bus becoming a bottleneck for L2 cache access.	Create separate back-side bus that runs at higher speed than the main (front-side) external bus. The BSB is dedicated to the L2 cache.	Pentium Pro
	Move L2 cache on to the processor chip.	Pentium II
Some applications deal with massive databases and must have rapid access to large amounts of data. The on-chip caches are too small. <small>29-Jun-2021 Cache_mapping</small>	Add external L3 cache.	Pentium III
	Move L3 cache on-chip.	Pentium 4

Pentium 4 Block Diagram



Pentium 4 Core Processor

- Fetch/Decode Unit
 - Fetches instructions from L2 cache
 - Decode into micro-ops
 - Store micro-ops in L1 cache
- Out of order execution logic
 - Schedules micro-ops
 - Based on data dependence and resources
 - May speculatively execute
- Execution units
 - Execute micro-ops
 - Data from L1 cache
 - Results in registers
- Memory subsystem
 - L2 cache and systems bus

Pentium 4 Design Reasoning

- Decodes instructions into RISC like micro-ops before L1 cache
- Micro-ops fixed length
 - Superscalar pipelining and scheduling
- Pentium instructions long & complex
- Performance improved by separating decoding from scheduling & pipelining
 - (More later – ch14)
- Data cache is write back
 - Can be configured to write through
- L1 cache controlled by 2 bits in register
 - CD = cache disable
 - NW = not write through
 - 2 instructions to invalidate (flush) cache and write back then invalidate
- L2 and L3 8-way set-associative
 - Line size 128 bytes

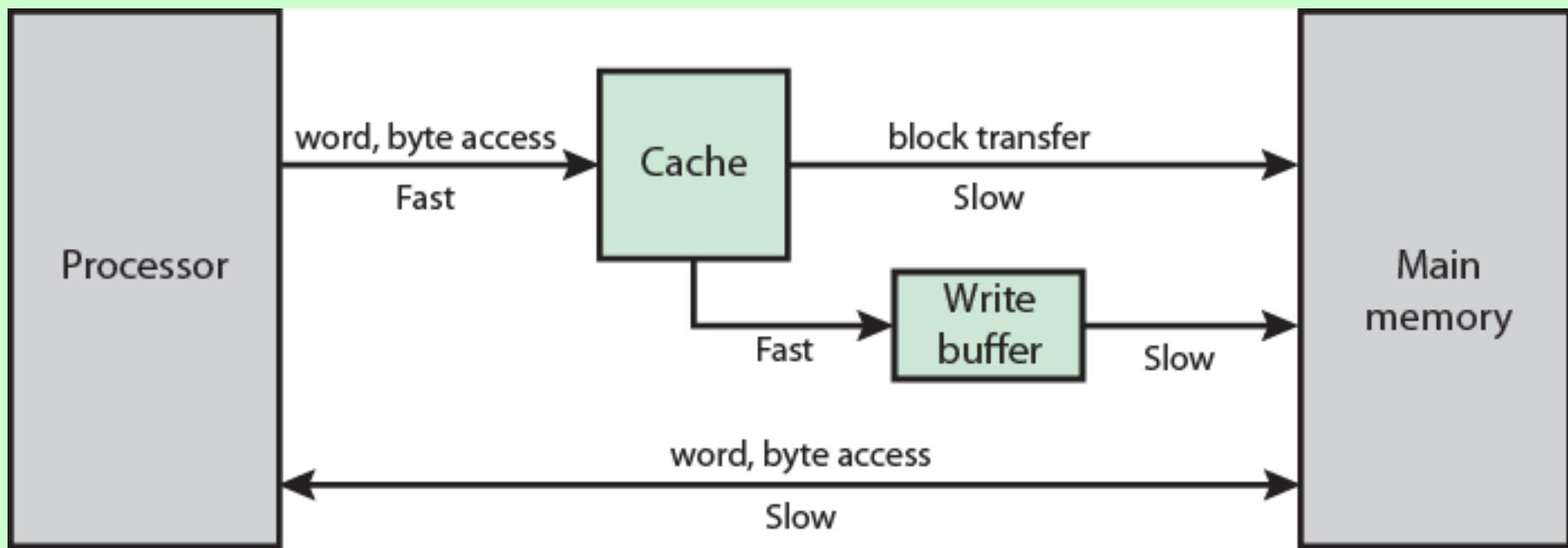
ARM Cache Features

Core	Cache Type	Cache Size (kB)	Cache Line Size (words)	Associativity	Location	Write Buffer Size (words)
ARM720T	Unified	8	4	4-way	Logical	8
ARM920T	Split	16/16 D/I	8	64-way	Logical	16
ARM926EJ-S	Split	4-128/4-128 D/I	8	4-way	Logical	16
ARM1022E	Split	16/16 D/I	8	64-way	Logical	16
ARM1026EJ-S	Split	4-128/4-128 D/I	8	4-way	Logical	8
Intel StrongARM	Split	16/16 D/I	4	32-way	Logical	32
Intel Xscale	Split	32/32 D/I	8	32-way	Logical	32
ARM1136-JF-S	Split	4-64/4-64 D/I	8	4-way	Physical	32

ARM Cache Organization

- Small FIFO write buffer
 - Enhances memory write performance
 - Between cache and main memory
 - Small c.f. cache
 - Data put in write buffer at processor clock speed
 - Processor continues execution
 - External write in parallel until empty
 - If buffer full, processor stalls
 - Data in write buffer not available until written
 - So keep buffer small

ARM Cache and Write Buffer Organization



Internet Sources

- Manufacturer sites
 - Intel
 - ARM
- Search on cache

Disks and RAID

50 Years Old!



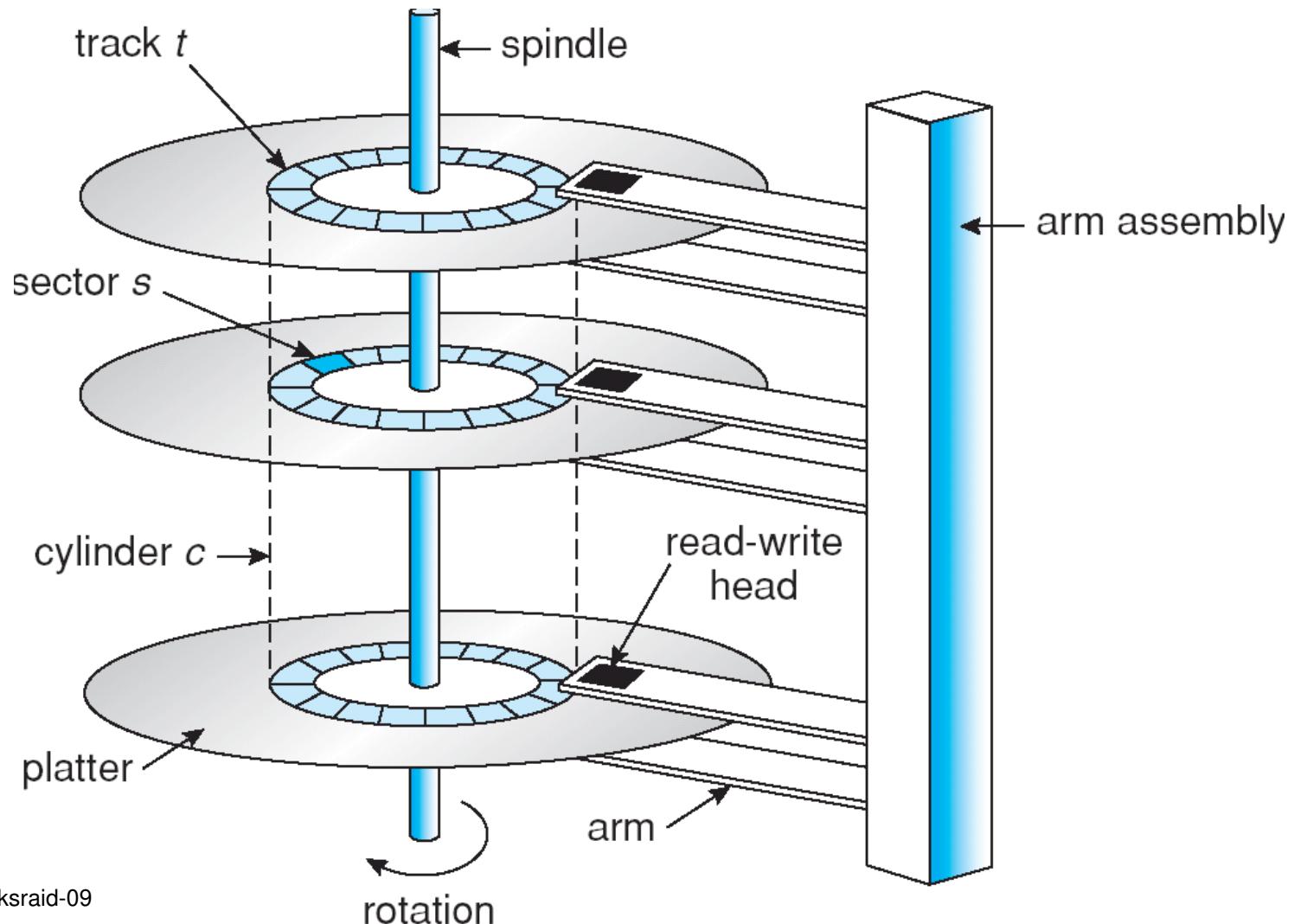
© CNET Networks

- 13th September 1956
- The IBM RAMAC 350



- 80000 times more data on the 8GB 1-inch drive in his right hand than on the 24-inch RAMAC one in his left...

What does the disk look like?

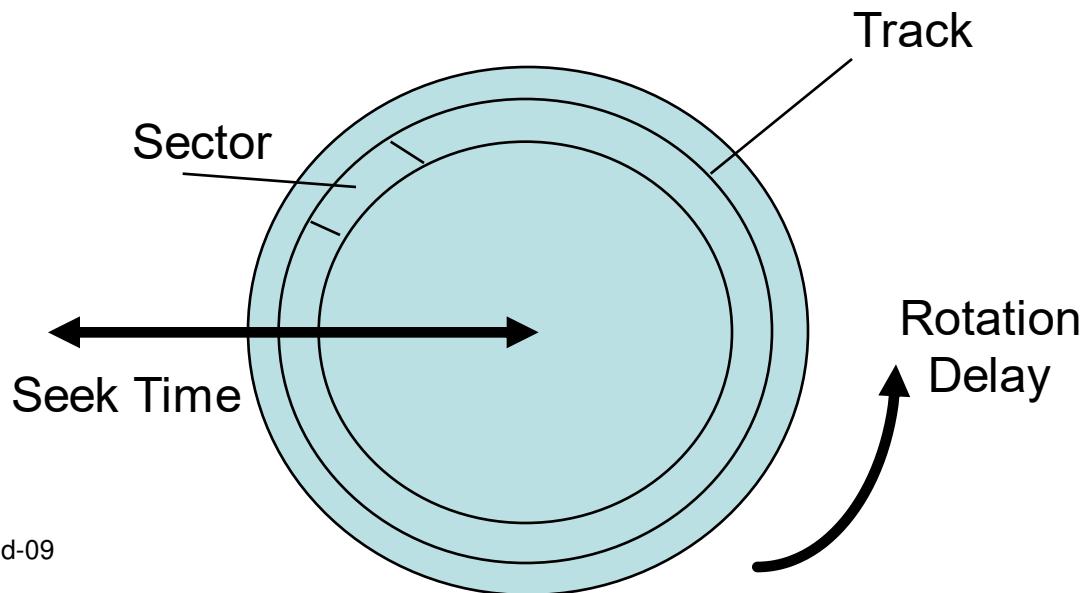


Some parameters

- 2-30 heads (platters * 2)
 - diameter 14" to 2.5"
- 700-20480 tracks per surface
- 16-1600 sectors per track
- sector size:
 - 64-8k bytes
 - 512 for most PCs
 - note: inter-sector gaps
- capacity: 20M-100G
- main adjectives: BIG, slow

Disk overheads

- To read from disk, we must specify:
 - cylinder #, surface #, sector #, transfer size, memory address
- Transfer time includes:
 - Seek time: to get to the track
 - Latency time: to get to the sector and
 - Transfer time: get bits off the disk



Modern disks

	Barracuda 180	Cheetah X15 36LP
Capacity	181GB	36.7GB
Disk/Heads	12/24	4/8
Cylinders	24,247	18,479
Sectors/track	~609	~485
Speed	7200RPM	15000RPM
Latency (ms)	4.17	2.0
Avg seek (ms)	7.4/8.2	3.6/4.2
Track-2- track(ms)	0.8/1.1	0.3/0.4

Disks vs. Memory

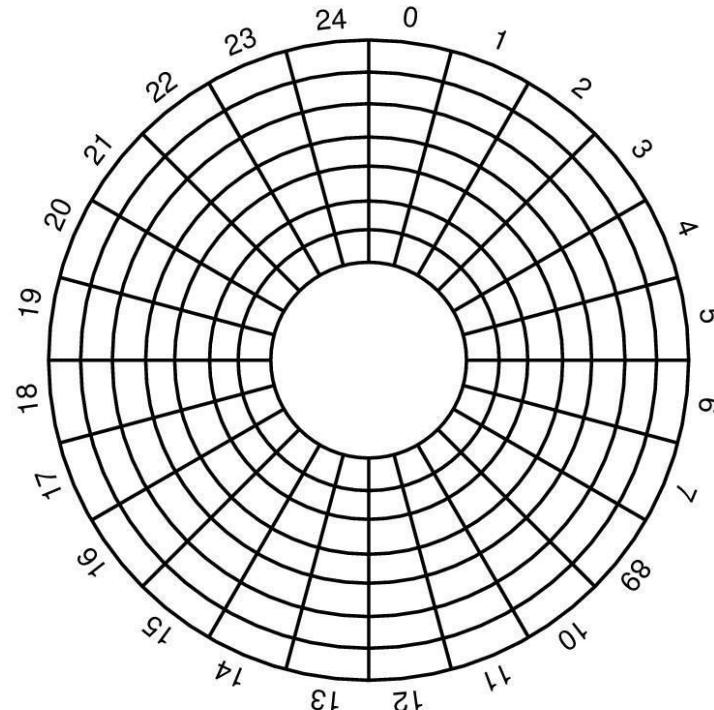
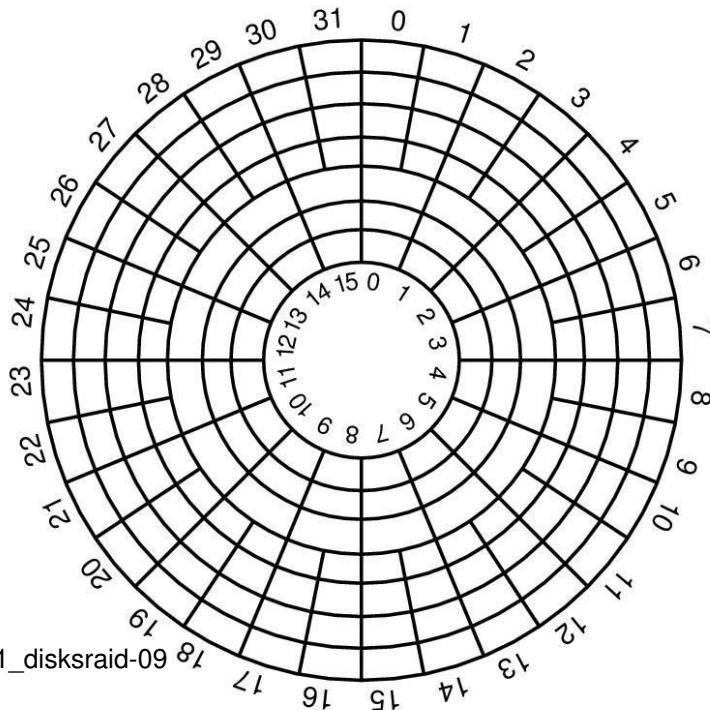
- Smallest write: sector
- Atomic write = sector
- Random access: 5ms
 - not on a good curve
- Sequential access: 200MB/s
- Cost \$.002MB
- Crash: doesn't matter ("non-volatile")
- (usually) bytes
- byte, word
- 50 ns
 - faster all the time
- 200-1000MB/s
- \$.10MB
- contents gone ("volatile")

Disk Structure

- Disk drives addressed as 1-dim arrays of *logical blocks*
 - the logical block is the smallest unit of transfer
- This array mapped sequentially onto disk sectors
 - Address 0 is 1st sector of 1st track of the outermost cylinder
 - Addresses incremented within track, then within tracks of the cylinder, then across cylinders, from innermost to outermost
- Translation is theoretically possible, but usually difficult
 - Some sectors might be defective
 - Number of sectors per track is not a constant

Non-uniform #sectors / track

- Reduce bit density per track for outer layers (Constant Linear Velocity, typically HDDs)
- Have more sectors per track on the outer layers, and increase rotational speed when reading from outer tracks (Constant Angular Velocity, typically CDs, DVDs)



Disk Scheduling

- The operating system tries to use hardware efficiently
 - for disk drives \Rightarrow having fast access time, disk bandwidth
- Access time has two major components
 - *Seek time* is time to move the heads to the cylinder containing the desired sector
 - *Rotational latency* is additional time waiting to rotate the desired sector to the disk head.
- Minimize seek time
- Seek time \approx seek distance
- Disk bandwidth is total number of bytes transferred, divided by the total time between the first request for service and the completion of the last transfer.

Disk Scheduling (Cont.)

- Several scheduling algos exist service disk I/O requests.
- We illustrate them with a request queue (0-199).

98, 183, 37, 122, 14, 124, 65, 67

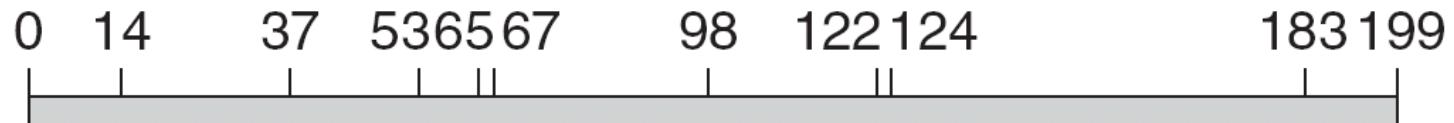
Head pointer 53

FCFS

Illustration shows total head movement of 640 cylinders.

queue = 98, 183, 37, 122, 14, 124, 65, 67

head starts at 53



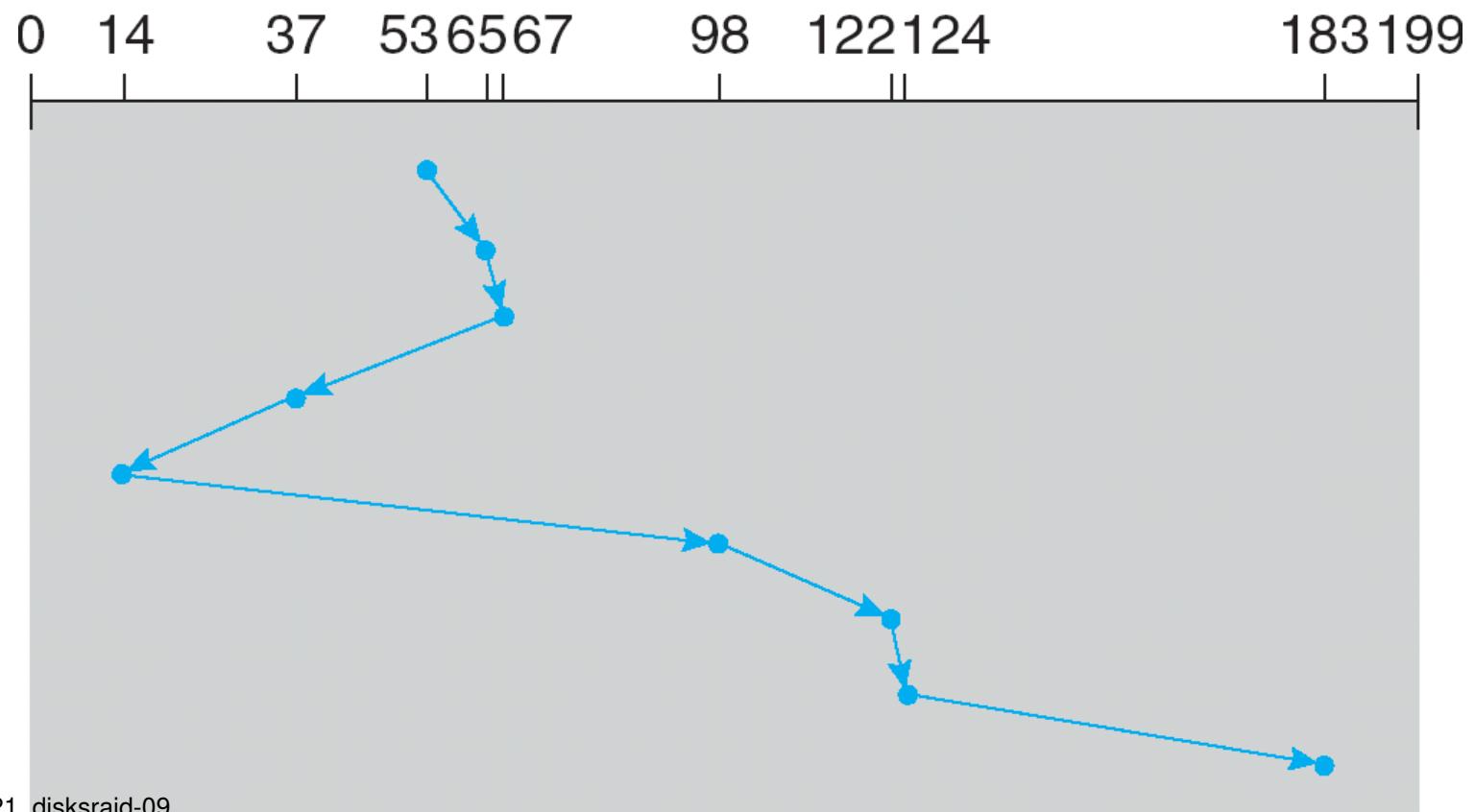
SSTF

- Selects request with minimum seek time from current head position
- SSTF scheduling is a form of SJF scheduling
 - may cause starvation of some requests.
- Illustration shows total head movement of 236 cylinders.

SSTF (Cont.)

queue = 98, 183, 37, 122, 14, 124, 65, 67

head starts at 53



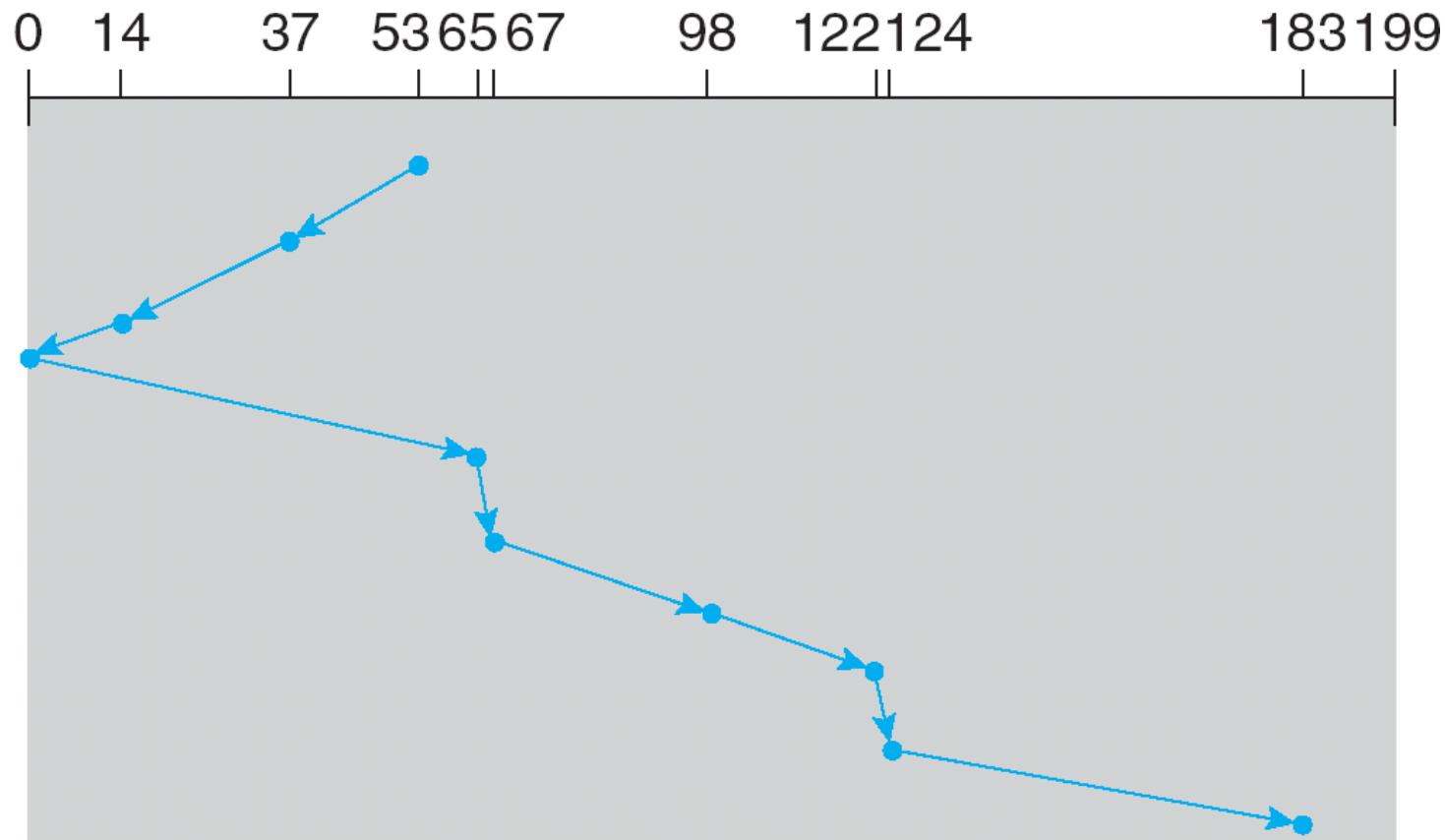
SCAN

- The disk arm starts at one end of the disk,
 - moves toward the other end, servicing requests
 - head movement is reversed when it gets to the other end of disk
 - servicing continues.
- Sometimes called the *elevator algorithm*.
- Illustration shows total head movement of 208 cylinders.

SCAN (Cont.)

queue = 98, 183, 37, 122, 14, 124, 65, 67

head starts at 53



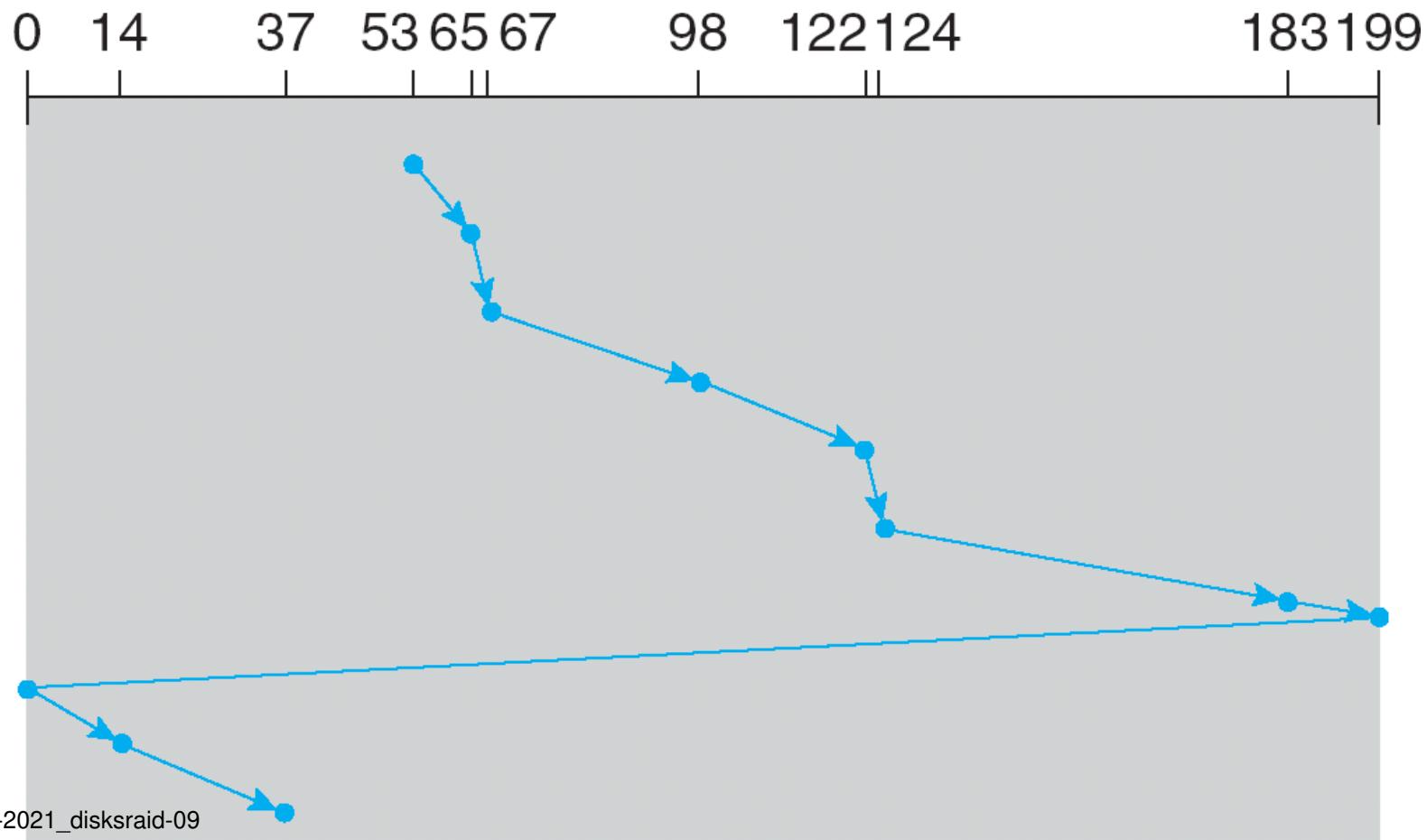
C-SCAN

- Provides a more uniform wait time than SCAN.
- The head moves from one end of the disk to the other.
 - servicing requests as it goes.
 - When it reaches the other end it immediately returns to beginning of the disk
 - No requests serviced on the return trip.
- Treats the cylinders as a circular list
 - that wraps around from the last cylinder to the first one.

C-SCAN (Cont.)

queue = 98, 183, 37, 122, 14, 124, 65, 67

head starts at 53



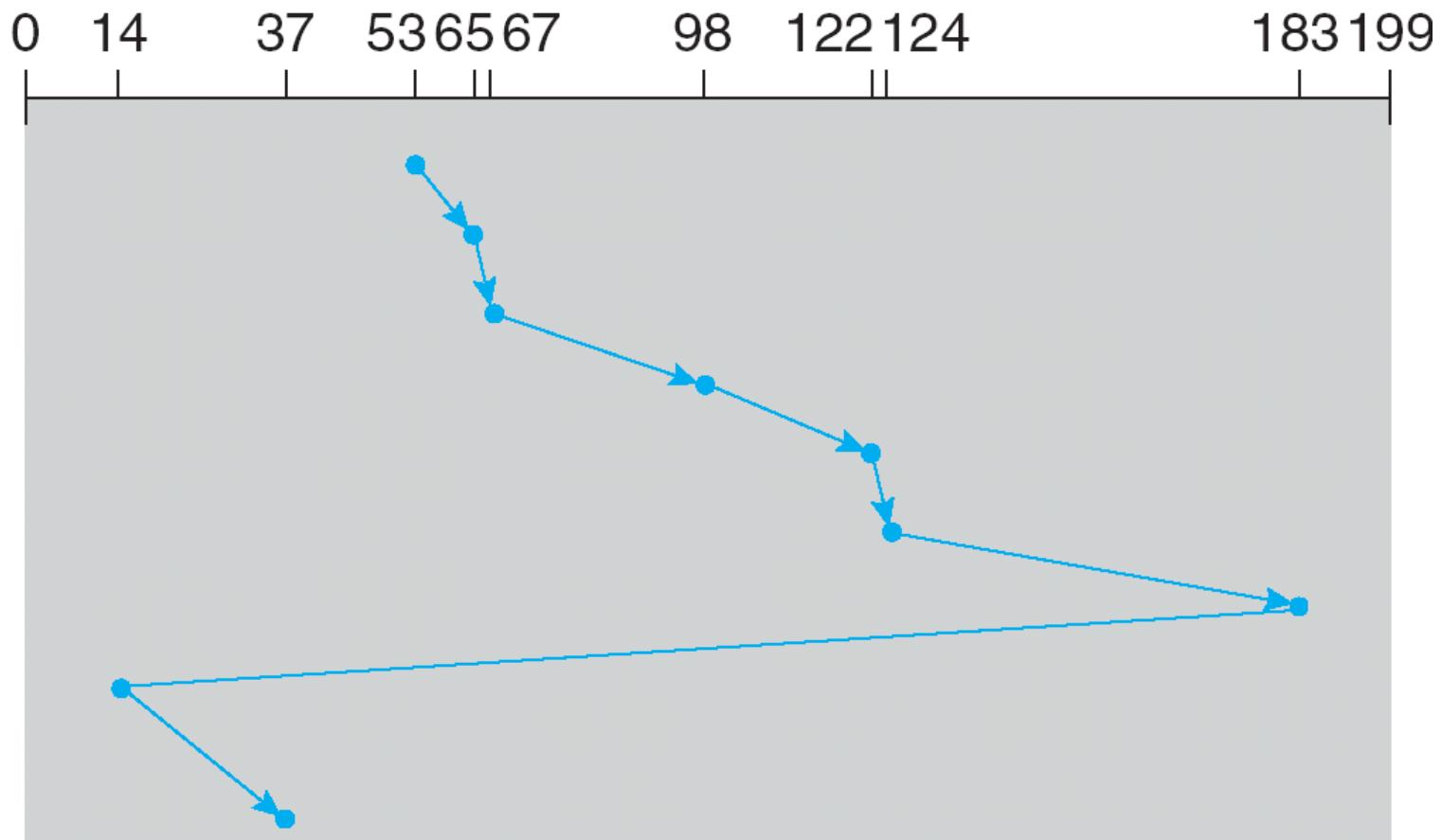
C-LOOK

- Version of C-SCAN
- Arm only goes as far as last request in each direction,
 - then reverses direction immediately,
 - without first going all the way to the end of the disk.

C-LOOK (Cont.)

queue 98, 183, 37, 122, 14, 124, 65, 67

head starts at 53



Selecting a Good Algorithm

- SSTF is common and has a natural appeal
- SCAN and C-SCAN perform better under heavy load
- Performance depends on number and types of requests
- Requests for disk service can be influenced by the file-allocation method.
- Disk-scheduling algorithm should be a separate OS module
 - allowing it to be replaced with a different algorithm if necessary.
- Either SSTF or LOOK is a reasonable default algorithm

Disk Formatting

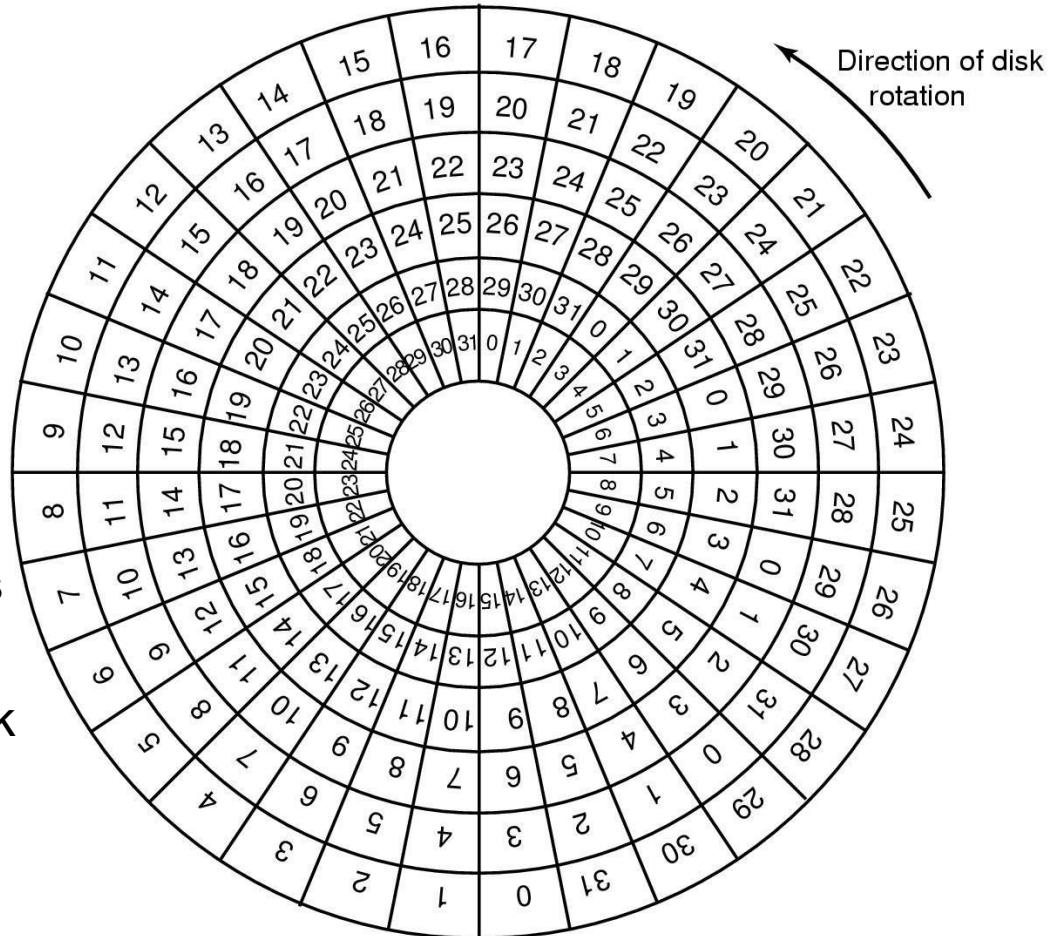
- After manufacturing disk has no information
 - Is stack of platters coated with magnetizable metal oxide
- Before use, each platter receives low-level format
 - Format has series of concentric tracks
 - Each track contains some sectors
 - There is a short gap between sectors

Preamble	Data	ECC
----------	------	-----

- Also contains cylinder and sector numbers
- Data is usually 512 bytes
- ECC field used to detect and recover from read errors

Cylinder Skew

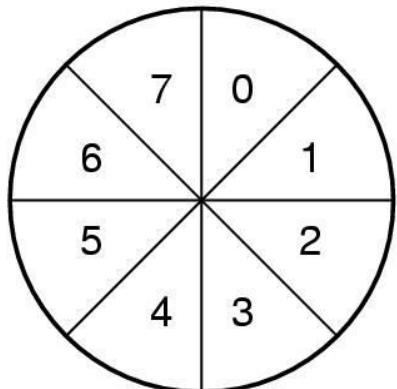
- Why cylinder skew?
- How much skew?
- Example, if
 - 10000 rpm
 - Drive rotates in 6 ms
 - Track has 300 sectors
 - New sector every 20 µs
 - If track seek time 800 µs
⇒ 40 sectors pass on seek



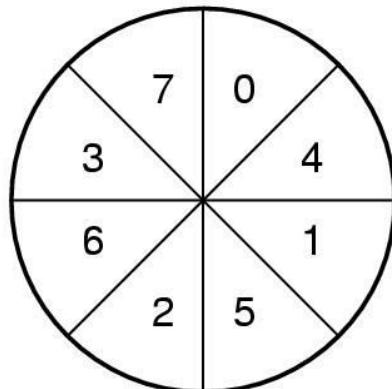
Cylinder skew: 40 sectors

Formatting and Performance

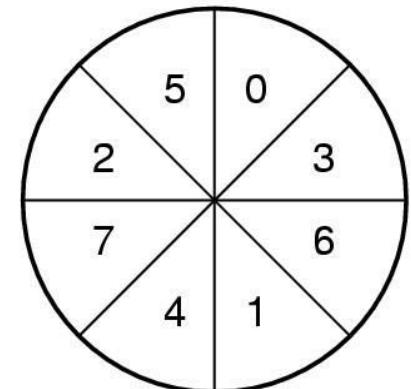
- If 10K rpm, 300 sectors of 512 bytes per track
 - 153600 bytes every 6 ms \Rightarrow 24.4 MB/sec transfer rate
- If disk controller buffer can store only one sector
 - For 2 consecutive reads, 2nd sector flies past during memory transfer of 1st track
 - Idea: Use single/double interleaving



06-Jul-2021_disksraids-09
(a)



(b)



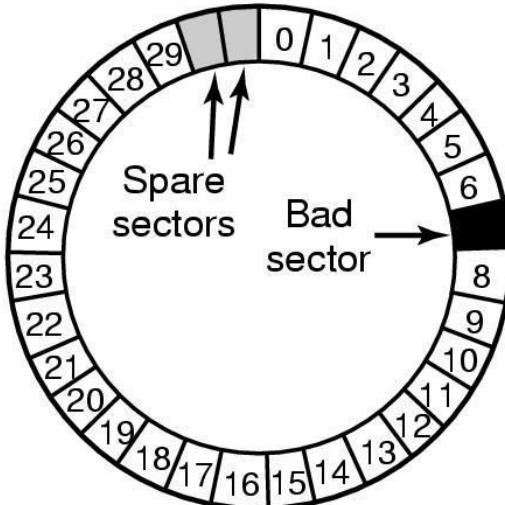
(c)

Disk Partitioning

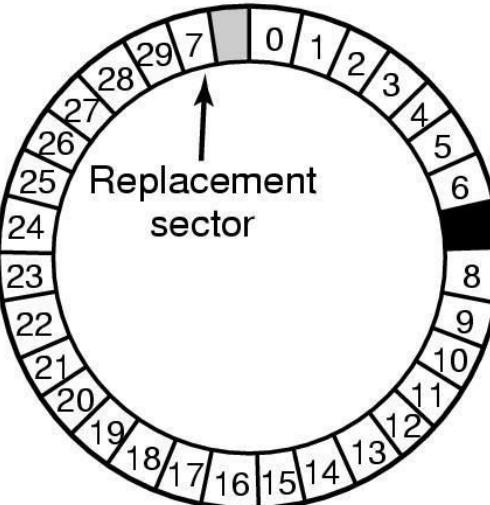
- Each partition is like a separate disk
- Sector 0 is MBR
 - Contains boot code + partition table
 - Partition table has starting sector and size of each partition
- High-level formatting
 - Done for each partition
 - Specifies boot block, free list, root directory, empty file system
- What happens on boot?
 - BIOS loads MBR, boot program checks to see active partition
 - Reads boot sector from that partition that then loads OS kernel, etc.

Handling Errors

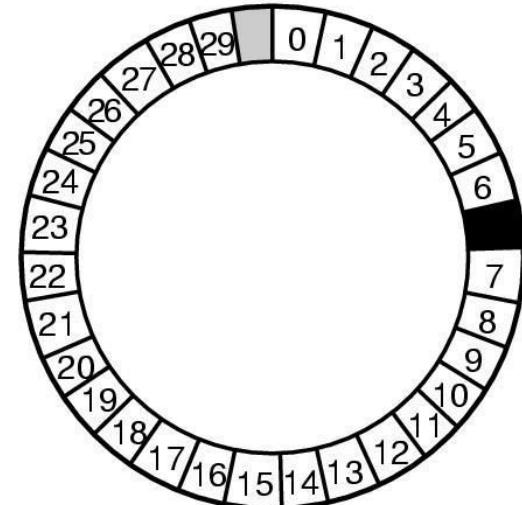
- A disk track with a bad sector
- Solutions:
 - Substitute a spare for the bad sector (sector sparing)
 - Shift all sectors to bypass bad one (sector forwarding)



(a)



(b)



(c)

RAID Motivation

- Disks are improving, but not as fast as CPUs
 - 1970s seek time: 50-100 ms.
 - 2000s seek time: <5 ms.
 - Factor of 20 improvement in 3 decades
- We can use multiple disks for improving performance
 - By Striping files across multiple disks (placing parts of each file on a different disk), parallel I/O can improve access time
- Striping reduces reliability
 - 100 disks have 1/100th mean time between failures of one disk
- So, we need Striping for performance, but we need something to help with reliability / availability
- To improve reliability, we can add redundant data to the disks, in addition to Striping

RAID

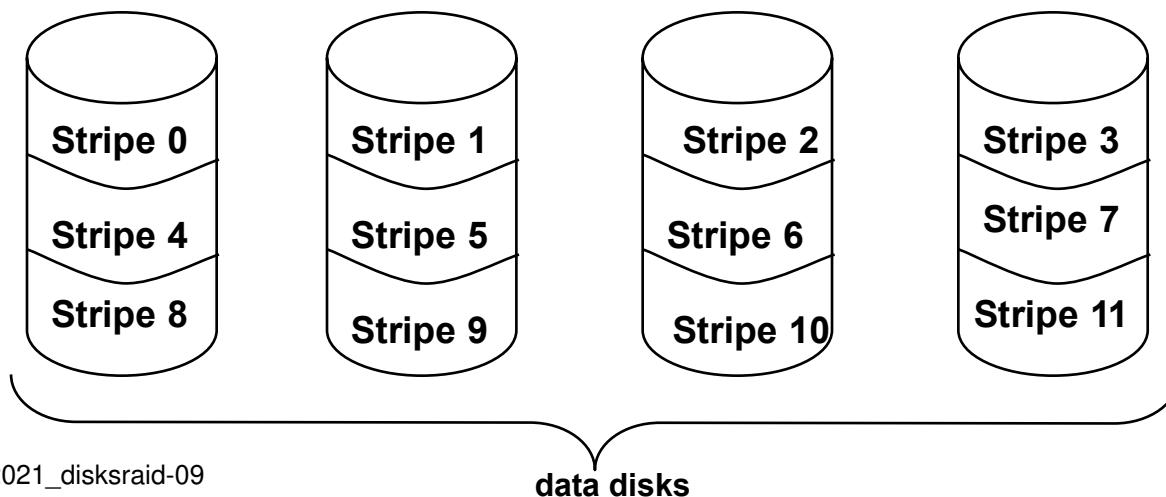
- A RAID is a Redundant Array of Inexpensive Disks
 - In industry, “I” is for “Independent”
 - The alternative is SLED, single large expensive disk
- Disks are small and cheap, so it’s easy to put lots of disks (10s to 100s) in one box for increased storage, performance, and availability
- The RAID box with a RAID controller looks just like a SLED to the computer
- Data plus some redundant information is Striped across the disks in some way
- How that Striping is done is key to performance and reliability.

Some Raid Issues

- **Granularity**
 - fine-grained: Stripe each file over all disks. This gives high throughput for the file, but limits to transfer of 1 file at a time
 - coarse-grained: Stripe each file over only a few disks. This limits throughput for 1 file but allows more parallel file access
- **Redundancy**
 - uniformly distribute redundancy info on disks: avoids load-balancing problems
 - concentrate redundancy info on a small number of disks: partition the set into data disks and redundant disks

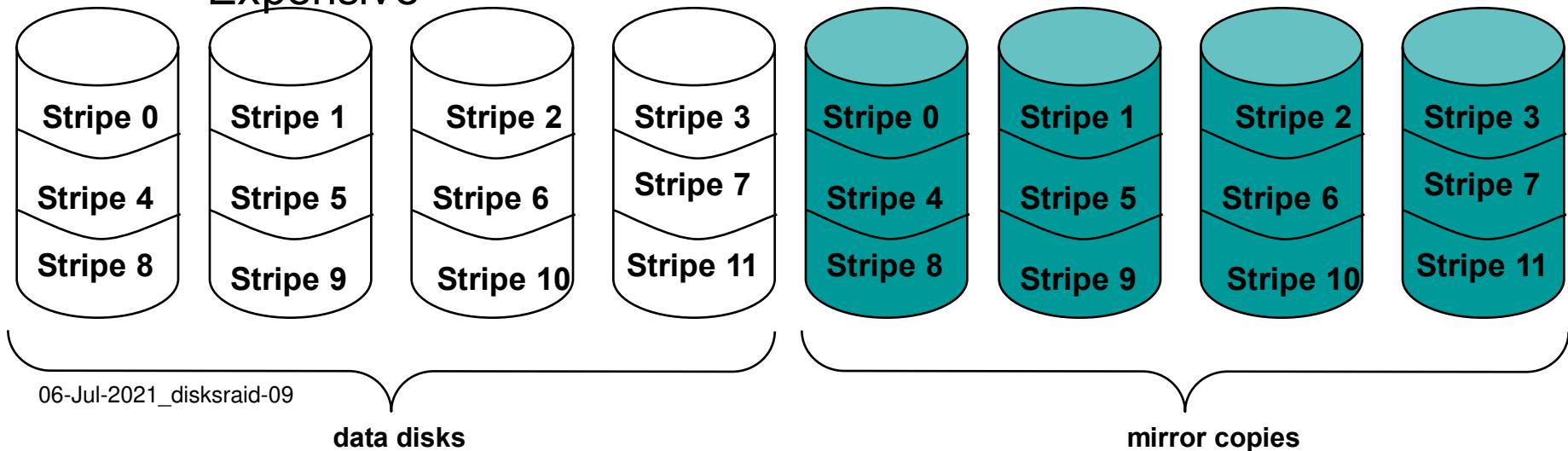
Raid Level 0

- Level 0 is nonredundant disk array
- Files are Striped across disks, no redundant info
- High read throughput
- Best write throughput (no redundant info to write)
- Any disk failure results in data loss
 - Reliability worse than SLED



Raid Level 1

- Mirrored Disks
- Data is written to two places
 - On failure, just use surviving disk
- On read, choose fastest to read
 - Write performance is same as single drive, read performance is 2x better
- Expensive



Parity and Hamming Codes

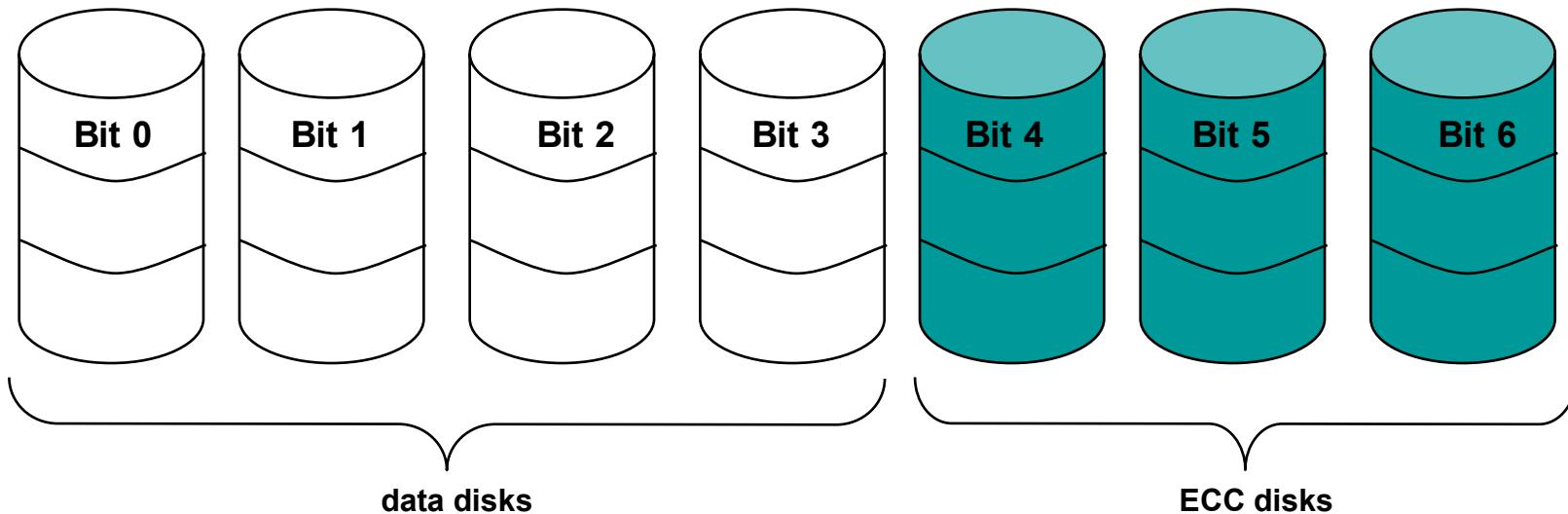
- What do you need to do in order to detect and correct a one-bit error ?
 - Suppose you have a binary number, represented as a collection of bits: $\langle b_3, b_2, b_1, b_0 \rangle$, e.g. 0110
- Detection is easy
- Parity:
 - Count the number of bits that are on, see if it's odd or even
 - EVEN parity is 0 if the number of 1 bits is even
 - $\text{Parity}(\langle b_3, b_2, b_1, b_0 \rangle) = P_0 = b_0 \otimes b_1 \otimes b_2 \otimes b_3$
 - $\text{Parity}(\langle b_3, b_2, b_1, b_0, p_0 \rangle) = 0$ if all bits are intact
 - $\text{Parity}(0110) = 0$, $\text{Parity}(01100) = 0$
 - $\text{Parity}(11100) = 1 \Rightarrow \text{ERROR!}$
 - Parity can detect a single error, but can't tell you which of the bits got flipped

Parity and Hamming Code

- Detection and correction require more work
- Hamming codes can detect double bit errors and detect & correct single bit errors
- 7/4 Hamming Code
 - $h_0 = b_0 \otimes b_1 \otimes b_3$
 - $h_1 = b_0 \otimes b_2 \otimes b_3$
 - $h_2 = b_1 \otimes b_2 \otimes b_3$
 - $H_0(<1101>) = 0$
 - $H_1(<1101>) = 1$
 - $H_2(<1101>) = 0$
 - Hamming($<1101>$) = $<b_3, b_2, b_1, h_2, b_0, h_1, h_0> = <110\textcolor{red}{0}110>$
 - If a bit is flipped, e.g. $<1110110>$
 - Hamming($<1111>$) = $<h_2, h_1, h_0> = <111>$ compared to $<010>, <101>$ are in error. Error occurred in bit 5.

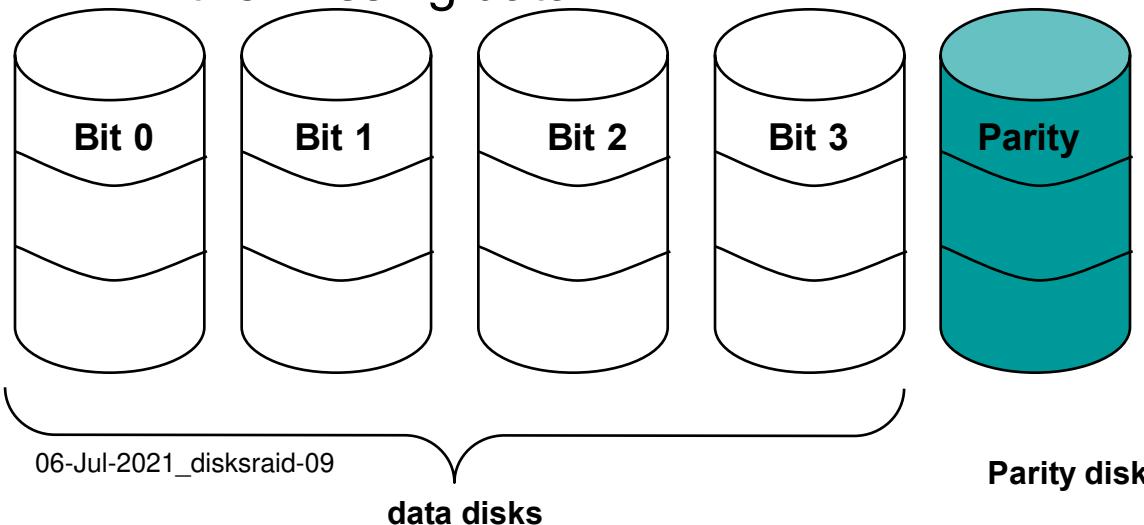
Raid Level 2

- Bit-level Striping with Hamming (ECC) codes for error correction
- All 7 disk arms are synchronized and move in unison
- Complicated controller
- Single access at a time
- Tolerates only one error, but with no performance degradation



Raid Level 3

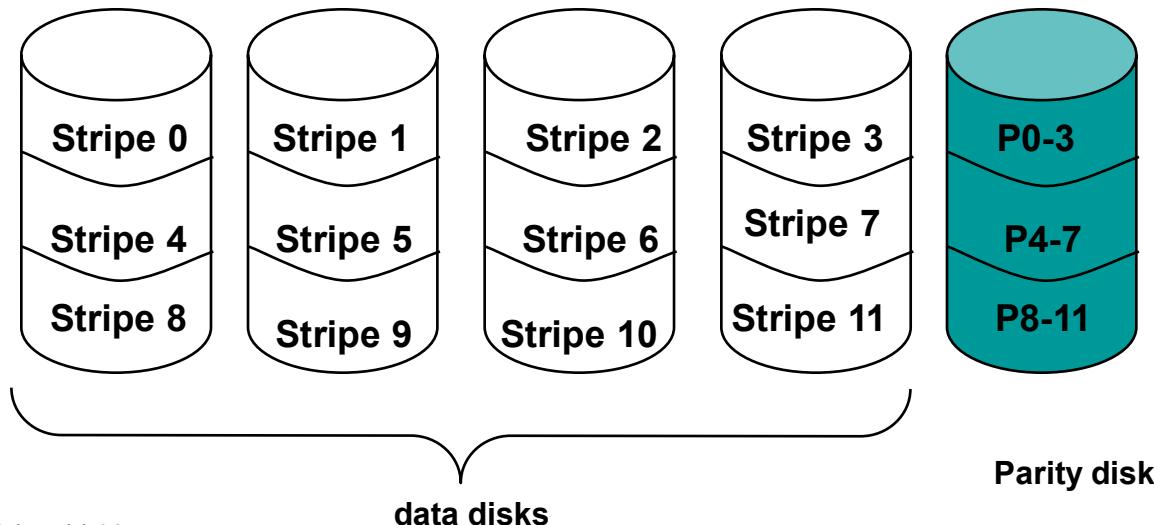
- Use a parity disk
 - Each bit on the parity disk is a parity function of the corresponding bits on all the other disks
- A read accesses all the data disks
- A write accesses all data disks plus the parity disk
- On disk failure, read remaining disks plus parity disk to compute the missing data



Single parity disk can be used to detect and correct errors

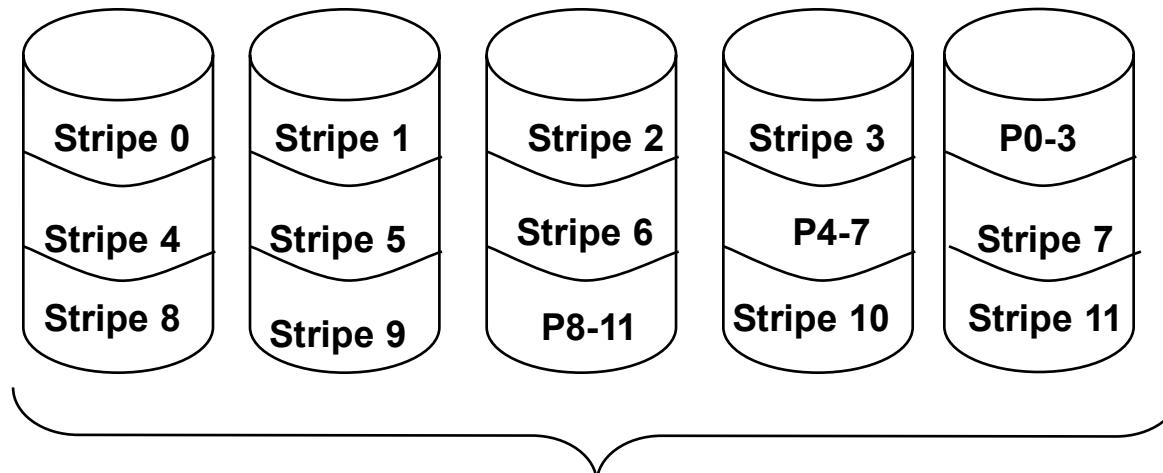
Raid Level 4

- Combines Level 0 and 3 – block-level parity with Stripes
- A read accesses all the data disks
- A write accesses all data disks plus the parity disk
- Heavy load on the parity disk



Raid Level 5

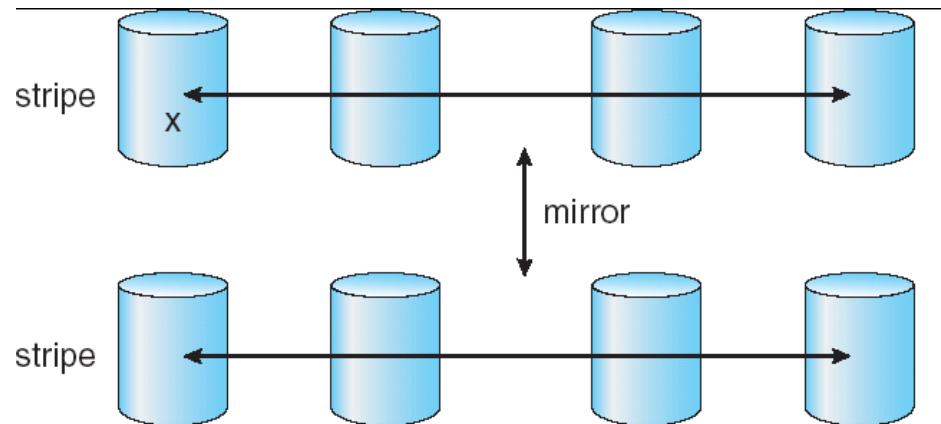
- Block Interleaved Distributed Parity
- Like parity scheme, but distribute the parity info over all disks (as well as data over all disks)
- Better read performance, large write performance
 - Reads can outperform SLEDs and RAID-0



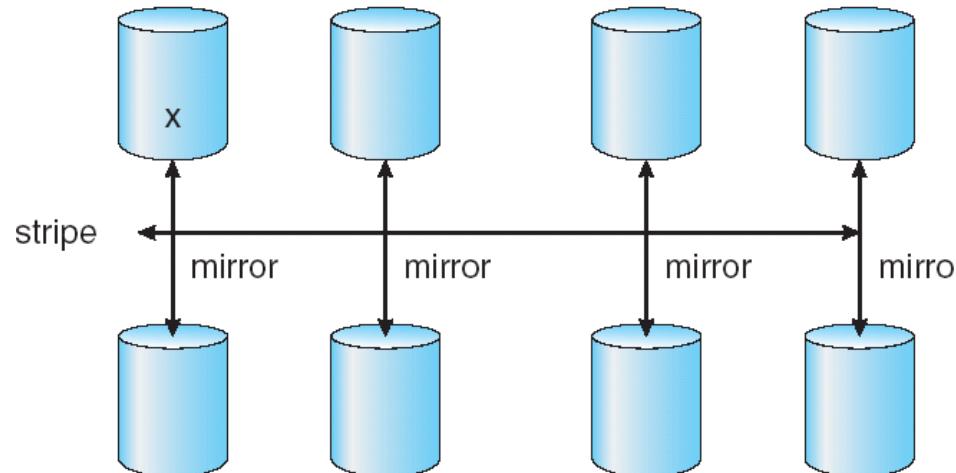
Raid Level 6

- Level 5 with an extra parity bit
- Can tolerate two failures
 - What are the odds of having two concurrent failures ?
- May outperform Level-5 on reads, slower on writes

RAID 0+1 and 1+0



a) RAID 0 + 1 with a single disk failure.



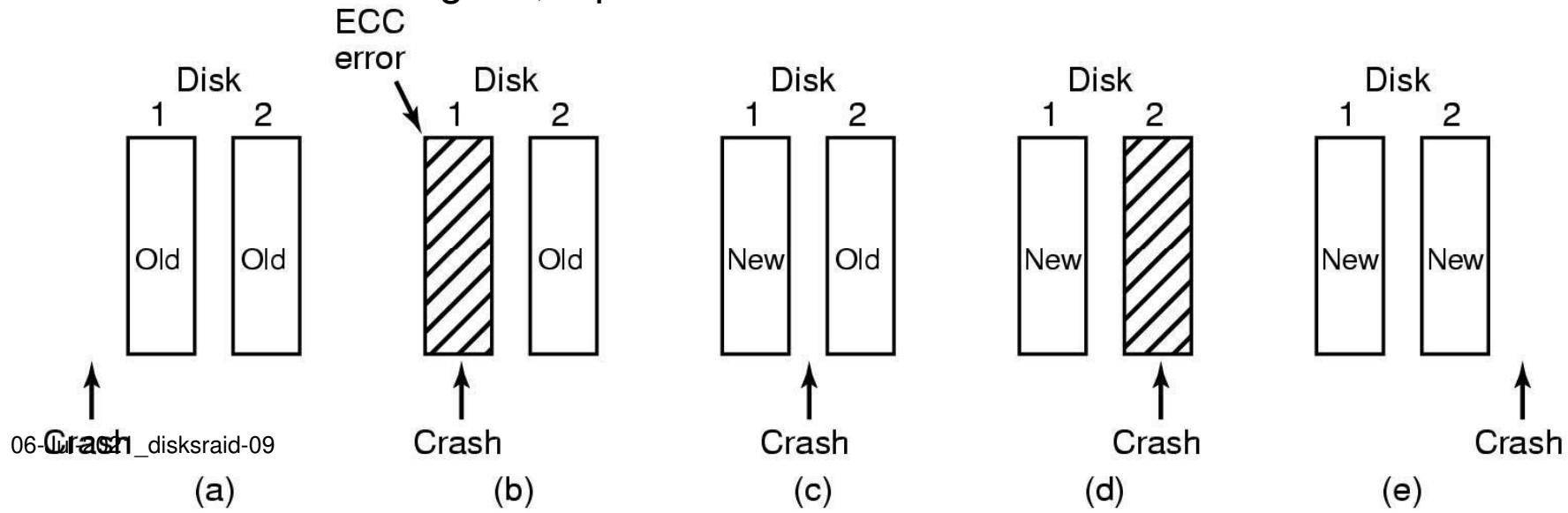
b) RAID 1 + 0 with a single disk failure.

Stable Storage

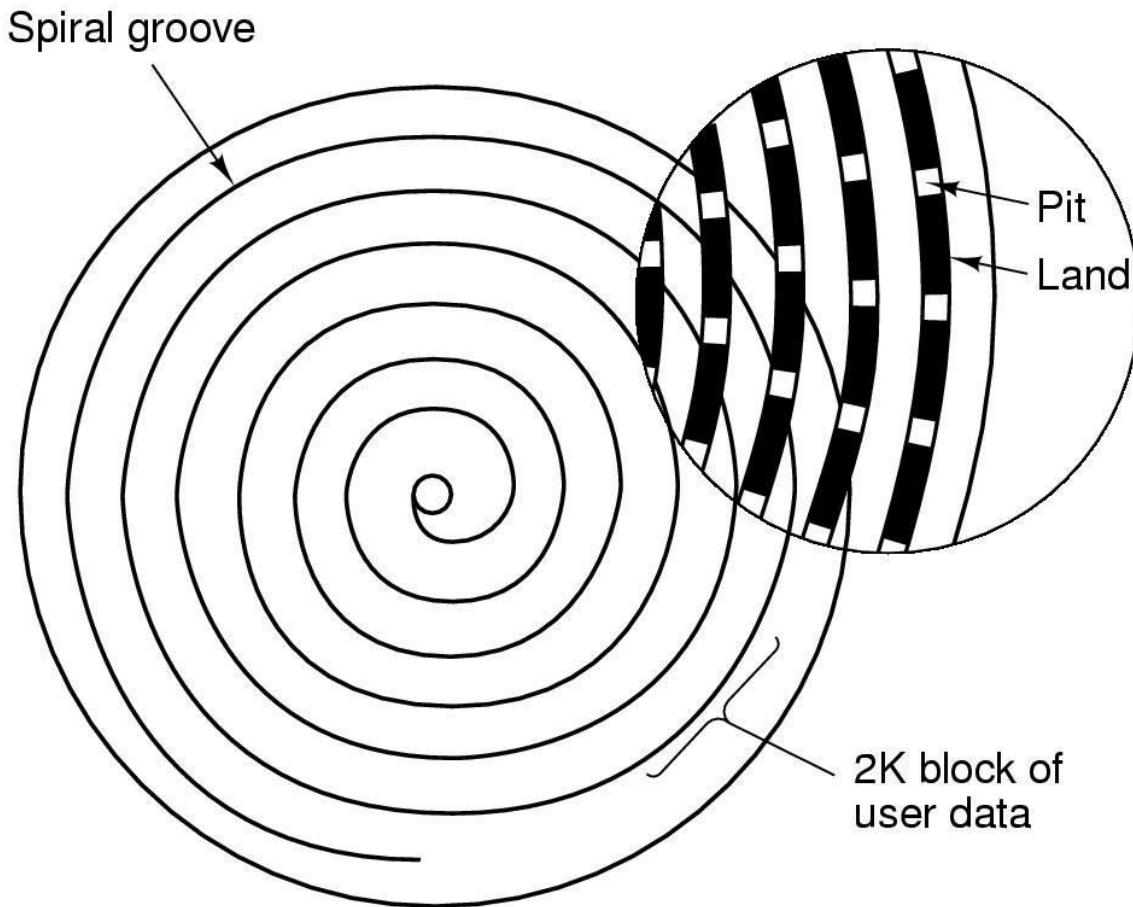
- Handling disk write errors:
 - Write lays down bad data
 - Crash during a write corrupts original data
- What we want to achieve? Stable Storage
 - When a write is issued, the disk either correctly writes data, or it does nothing, leaving existing data intact
- Model:
 - An incorrect disk write can be detected by looking at the ECC
 - It is very rare that same sector goes bad on multiple disks
 - CPU is fail-stop

Approach

- Use 2 identical disks
 - corresponding blocks on both drives are the same
- 3 operations:
 - Stable write: retry on 1st until successful, then try 2nd disk
 - Stable read: read from 1st. If ECC error, then try 2nd
 - Crash recovery: scan corresponding blocks on both disks
 - If one block is bad, replace with good one
 - If both are good, replace block in 2nd with the one in 1st



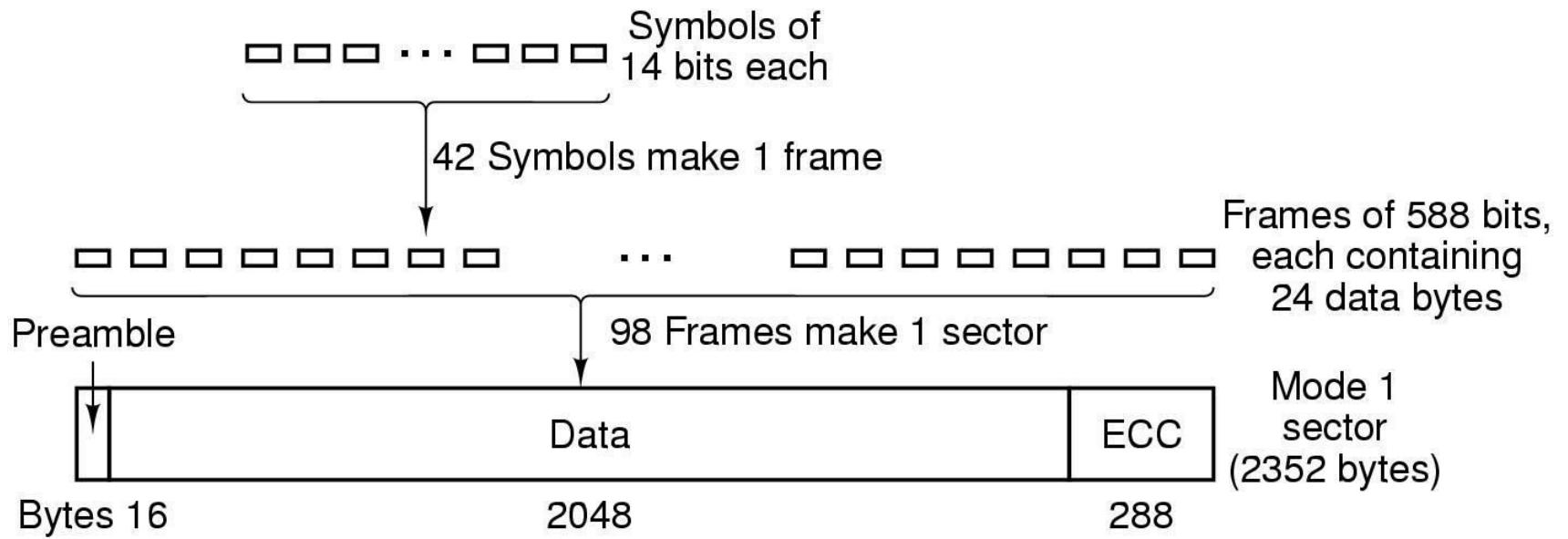
CD-ROMs



Spiral makes 22,188 revolutions around disk (approx 600/mm).

Will be 5.6 km long. Rotation rate: 530 rpm to 200 rpm

CD-ROMs



Logical data layout on a CD-ROM

Pipelining

- What is it?
- How does it work?
- What are the benefits?
- What could go wrong?

What is pipelining?

Pipelining is a design feature that allows individual common processor tasks to run simultaneously, such as:

Fetch
Decode
Execute

What is pipelining?

Or, more specifically in modern computers,

Instruction read

Decode

Operand read

Execute

Operand write

What is pipelining?

- No more “one instruction at a time” processing
- Processor works simultaneously on multiple instructions
- The cycle time of the processor is reduced, thus increasing instruction issue-rate in most cases.

How does pipelining work?

- First instruction is fetched from memory

How does pipelining work?

- First instruction is fetched from memory
- First instruction is decoded;
second instruction is fetched

How does pipelining work?

- First instruction is fetched from memory
- First instruction is decoded; second instruction is fetched
- First instruction's operands are fetched; second instruction is decoded; third instruction is fetched

How does pipelining work?

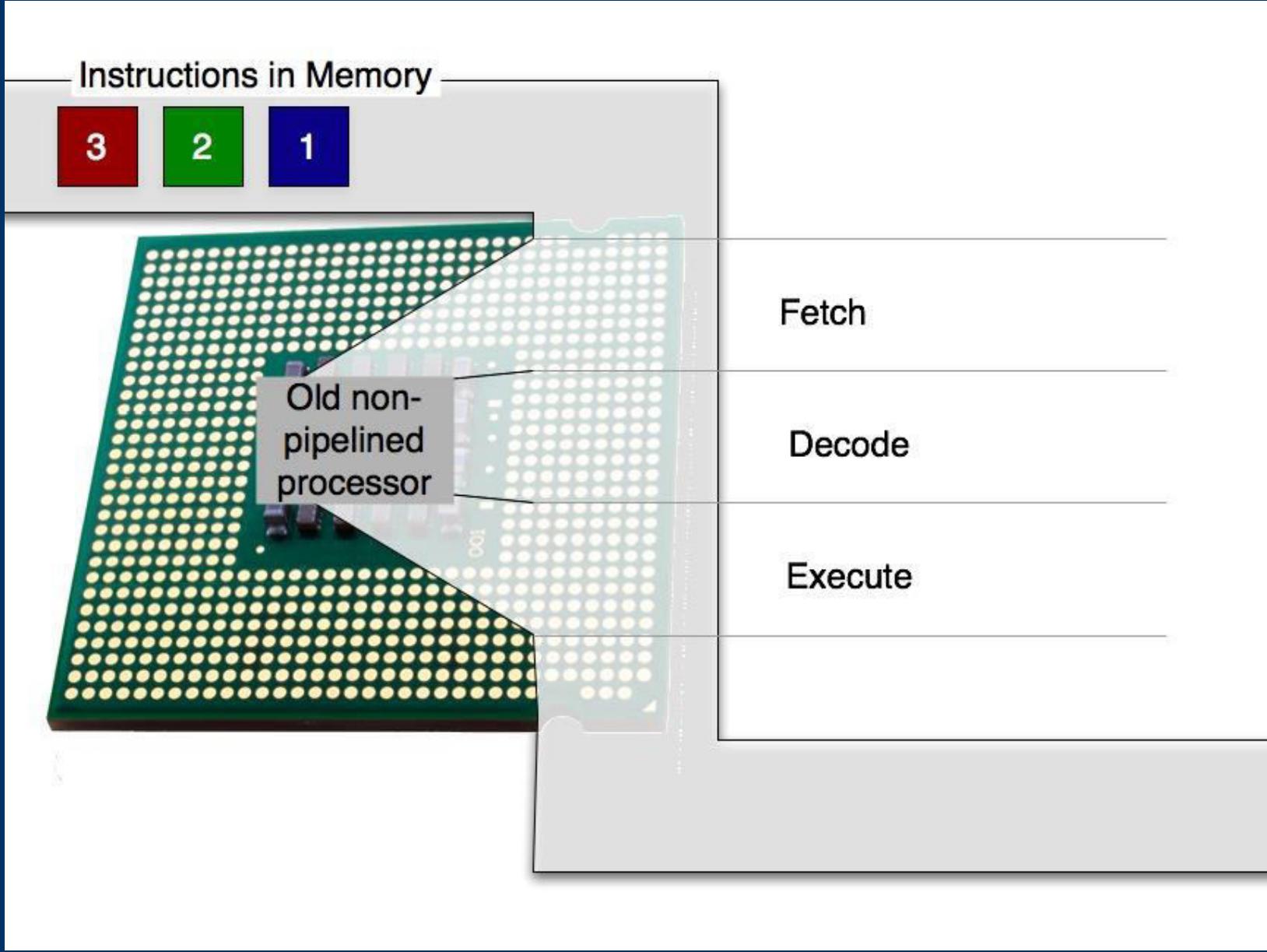
- First instruction is fetched from memory
- First instruction is decoded; second instruction is fetched
- First instruction's operands are fetched; second instruction is decoded; third instruction is fetched
- And on, and on, and on...

How does pipelining work?

- Because each instruction demands one stage of the processor, the maximum number of simultaneous instructions is the number of stages in the processor

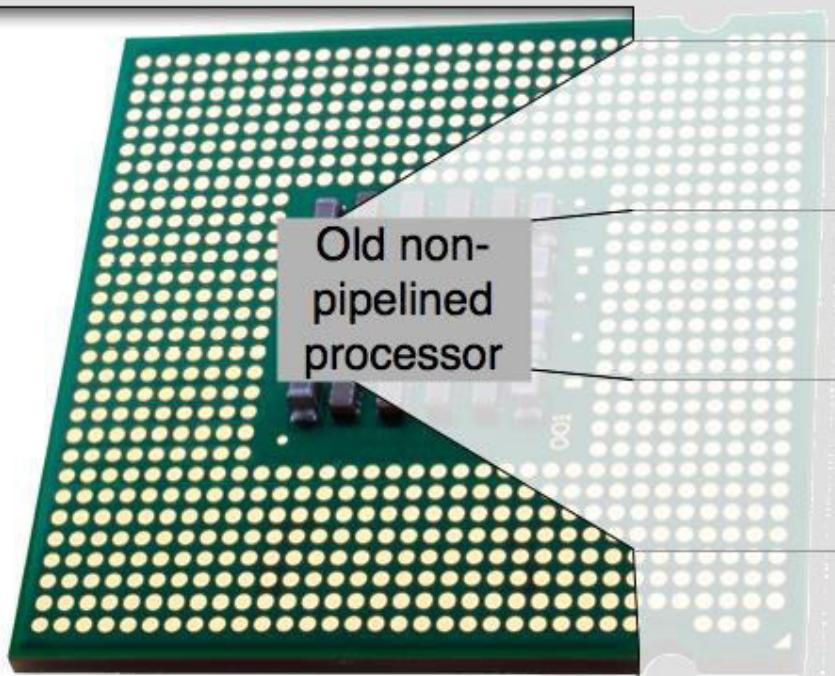
How does pipelining work?

A short animation
of simple processor routines



Instructions in Memory

3 2 1



Fetch

Decode

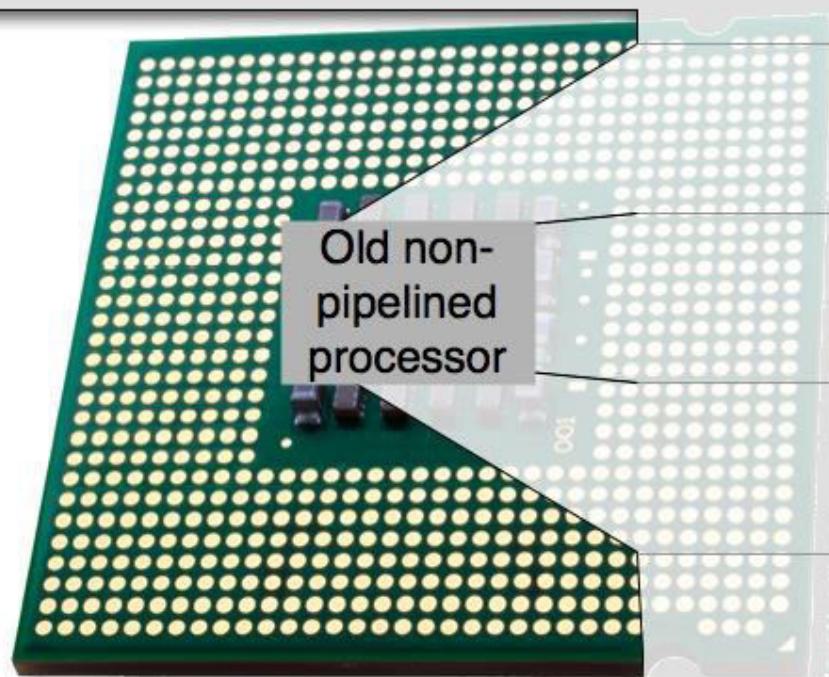
Execute

Instructions in Memory

3

2

1



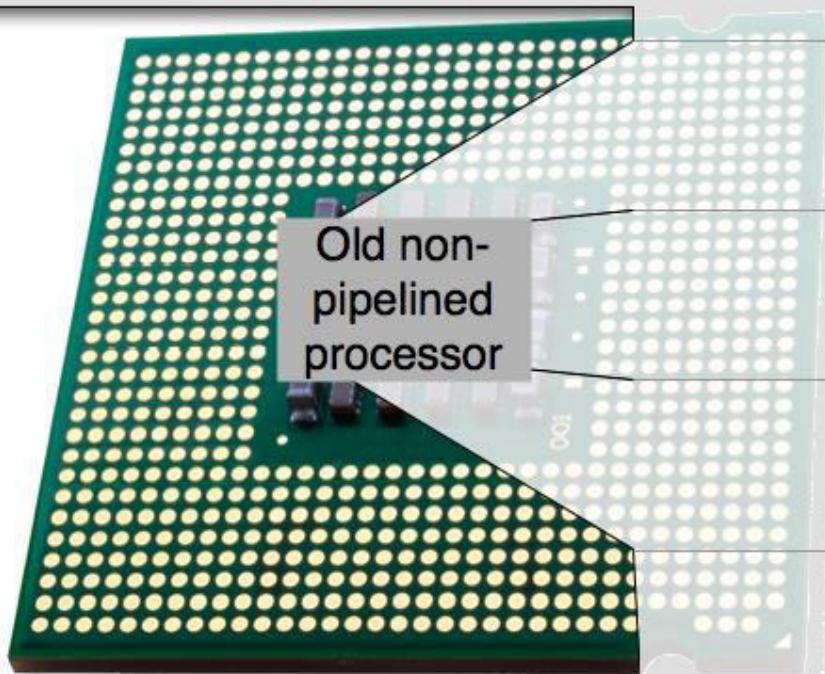
Fetch

Decode

Execute

Instructions in Memory

3 2 1

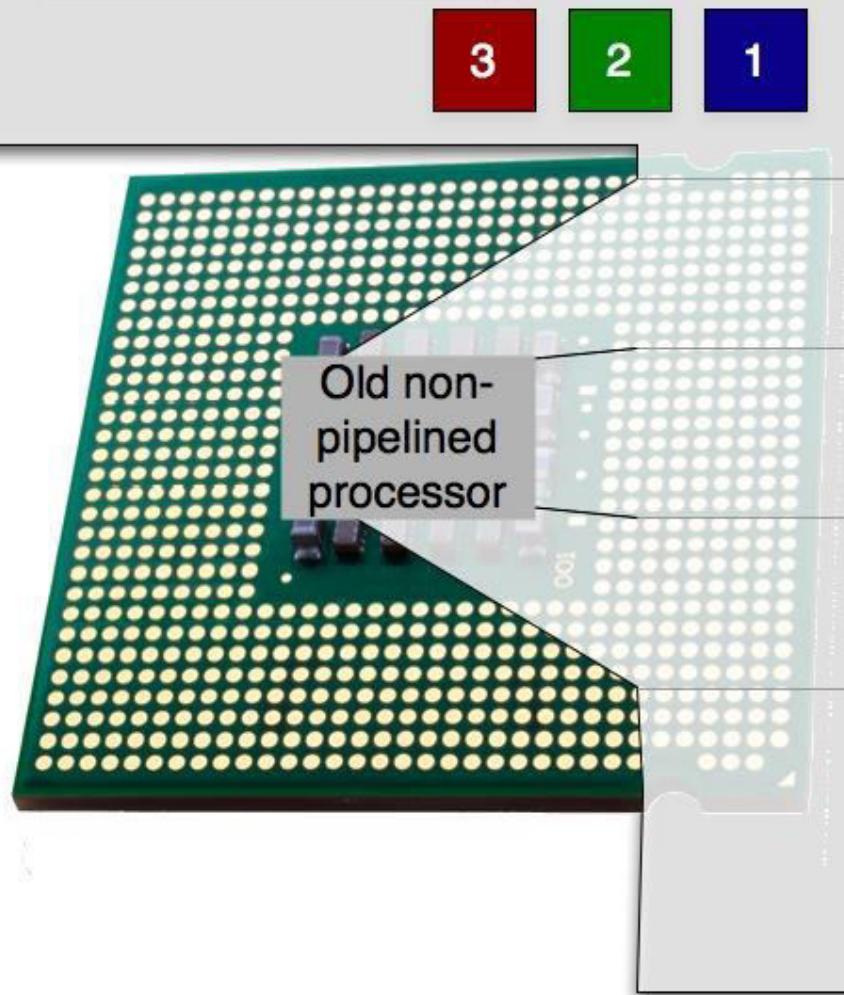


Fetch

Decode

Execute

Instructions in Memory

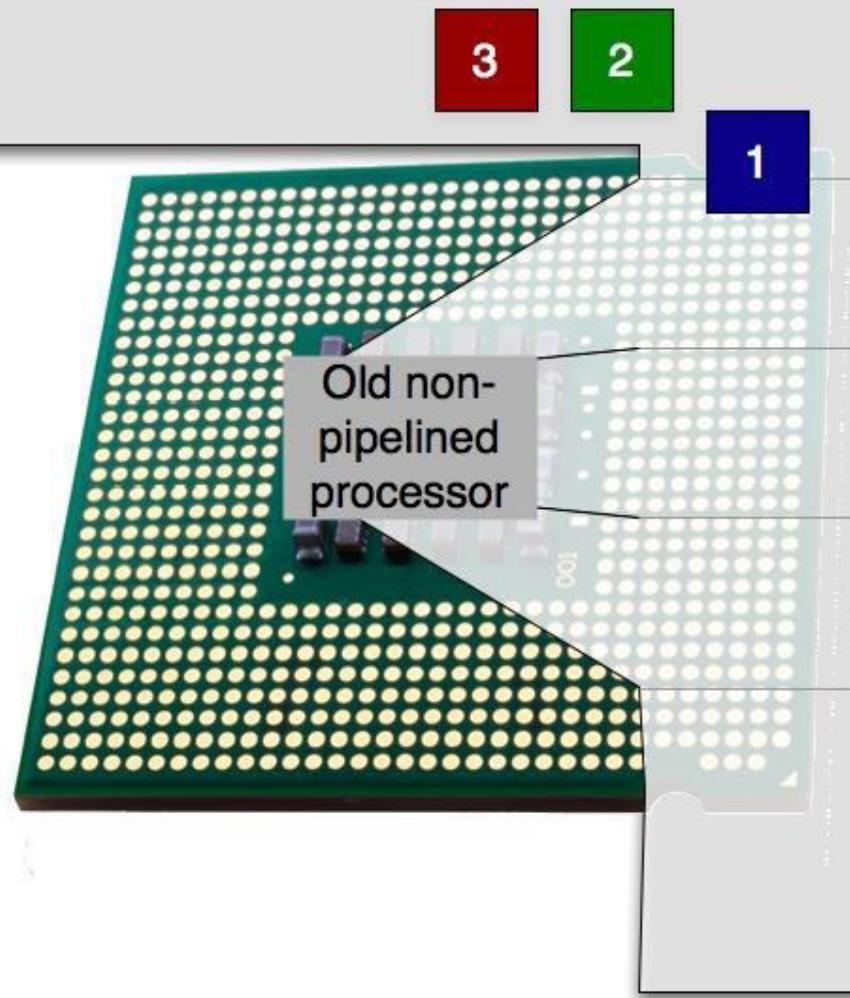


Fetch

Decode

Execute

Instructions in Memory

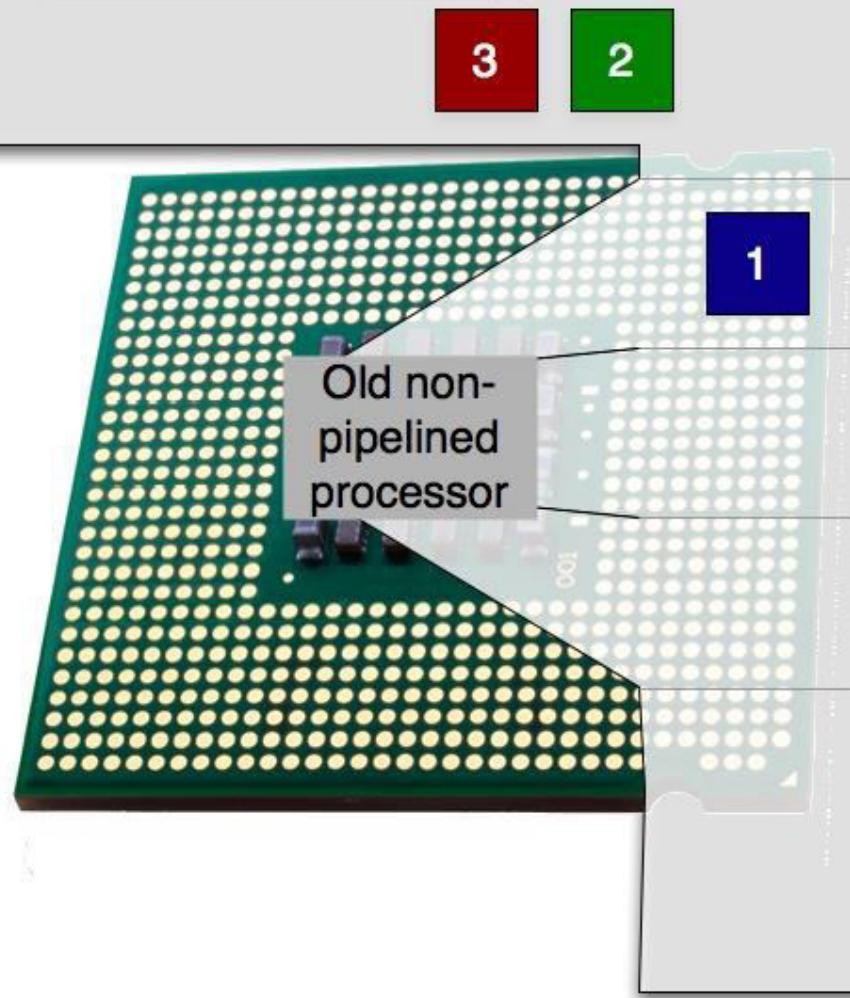


Fetch

Decode

Execute

Instructions in Memory

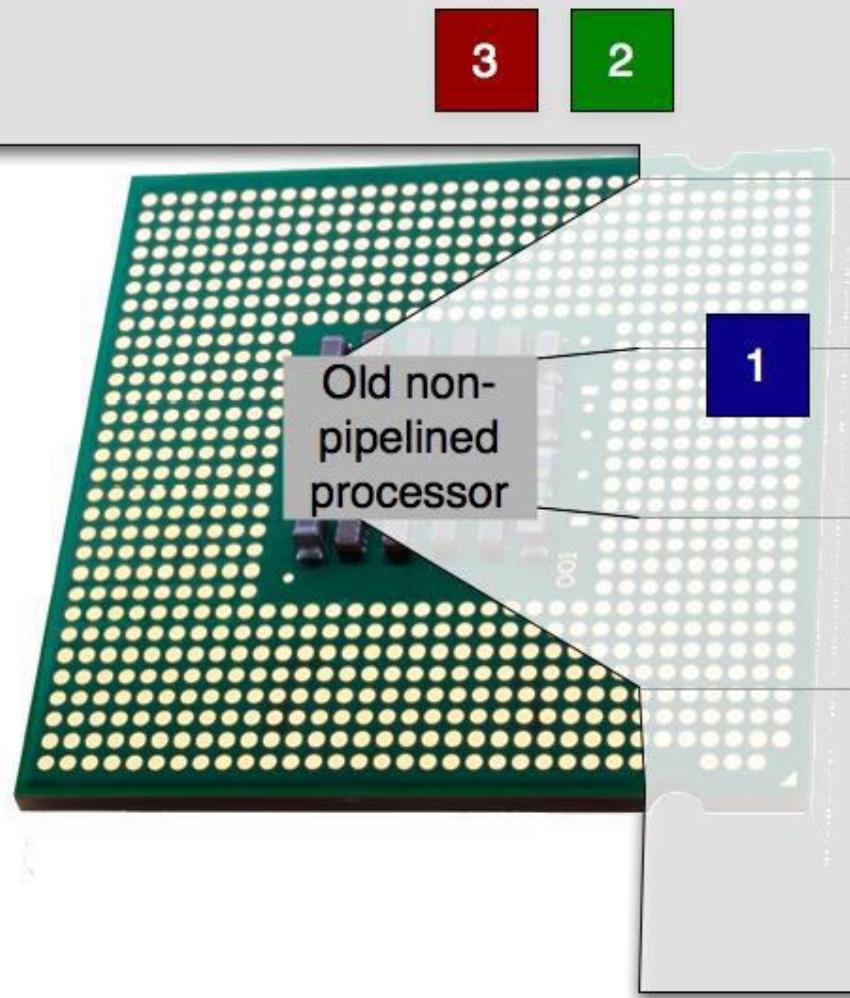


Fetch

Decode

Execute

Instructions in Memory

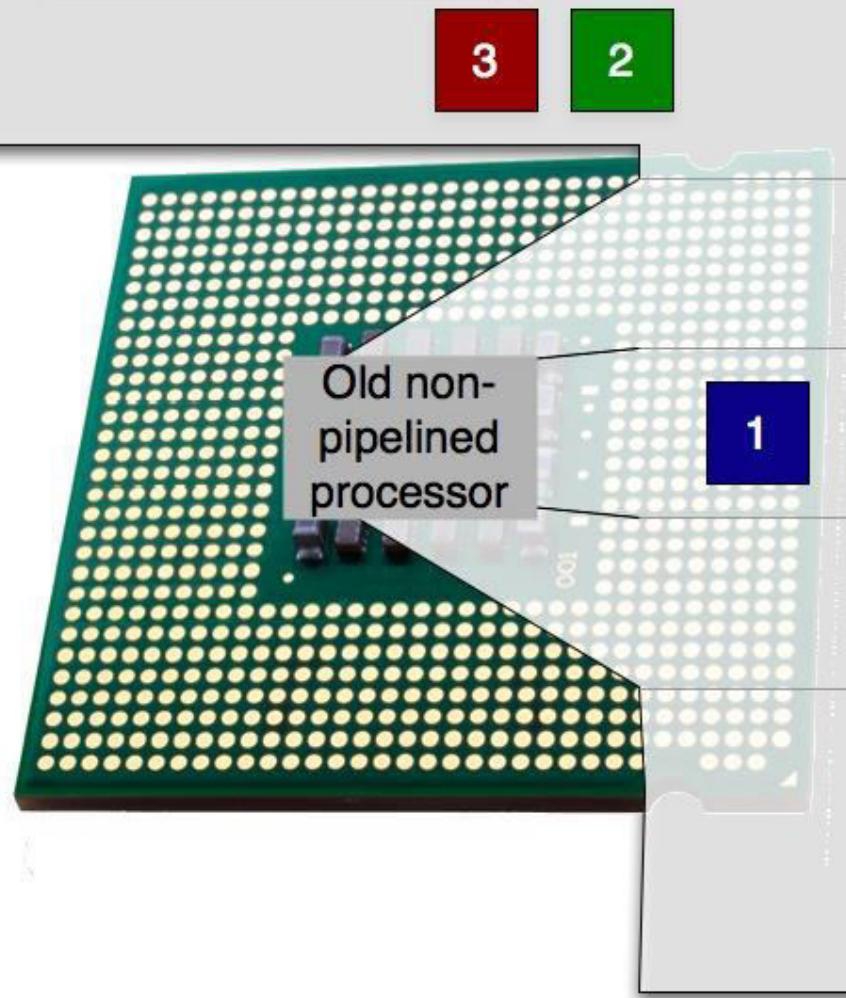


Fetch

Decode

Execute

Instructions in Memory

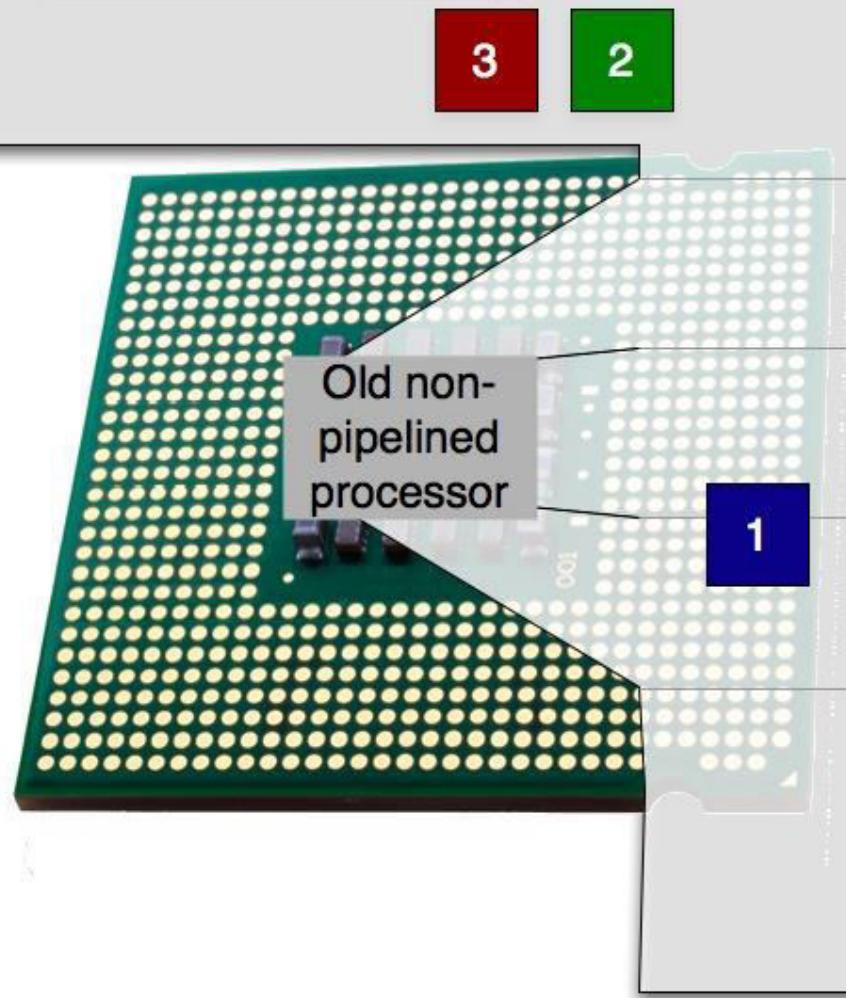


Fetch

Decode

Execute

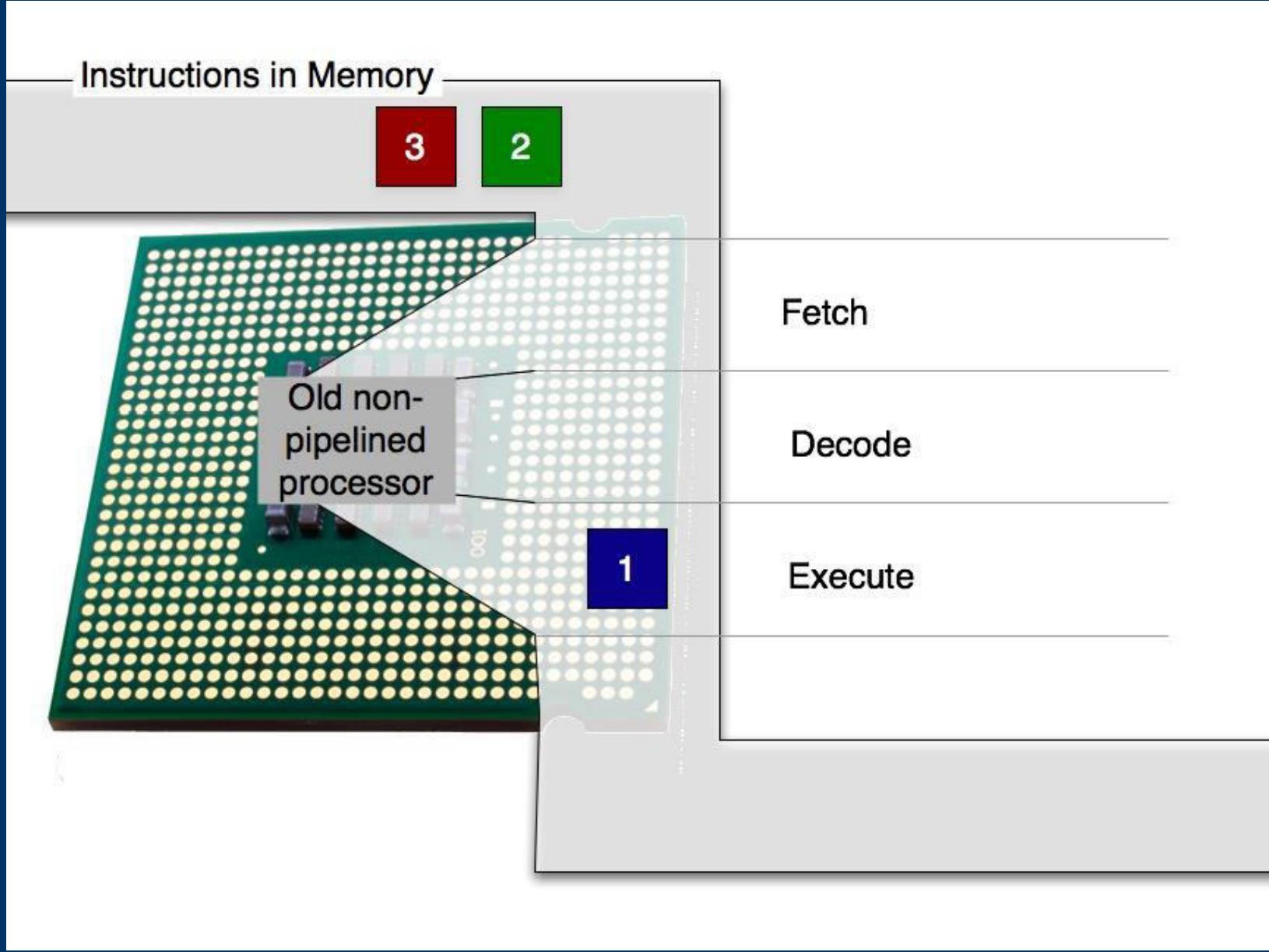
Instructions in Memory



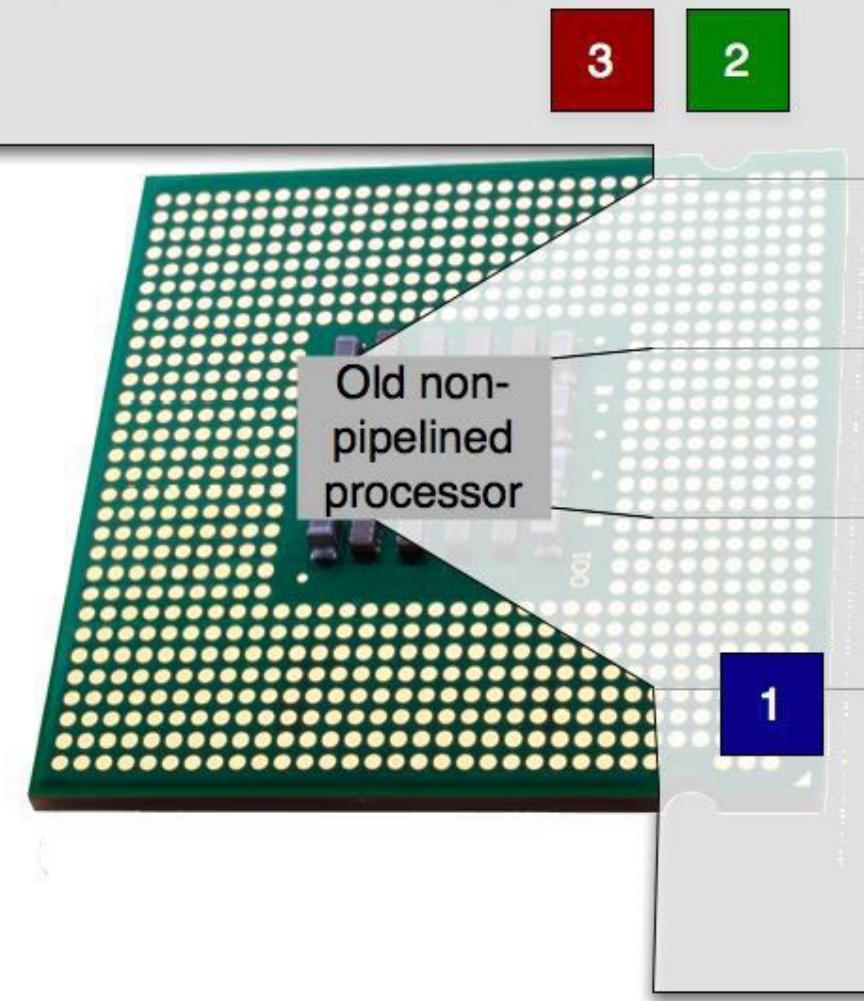
Fetch

Decode

Execute



Instructions in Memory

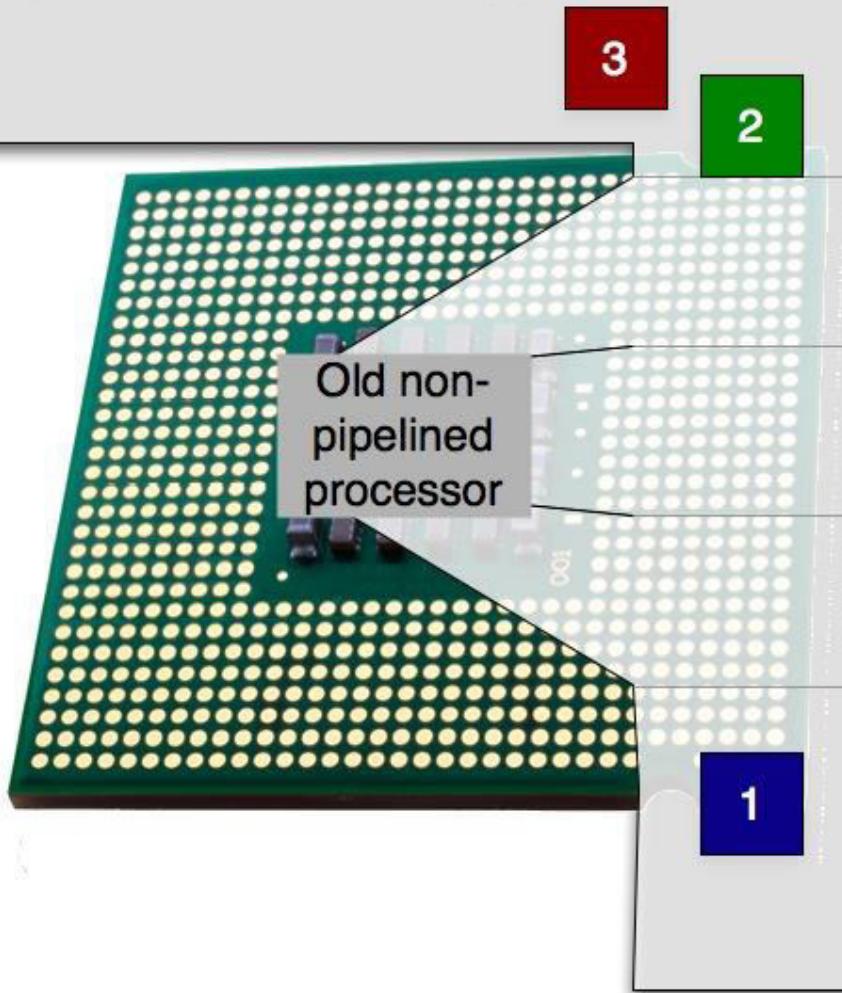


Fetch

Decode

Execute

Instructions in Memory

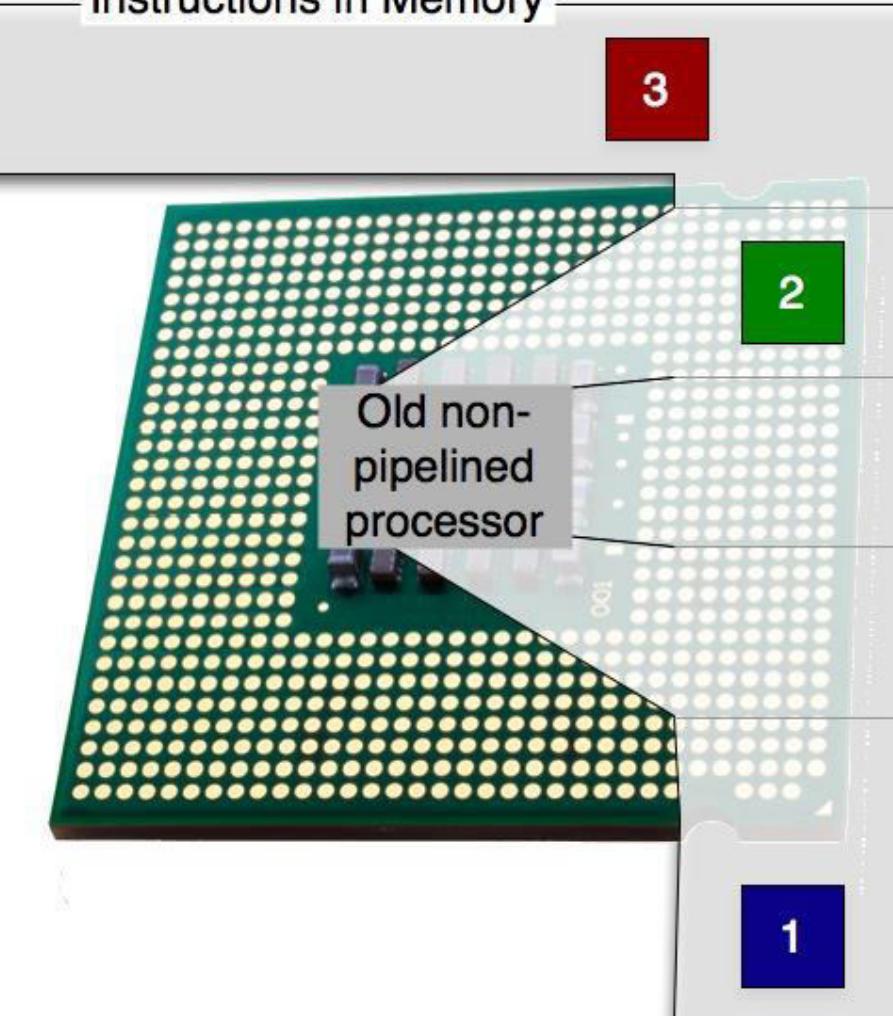


Fetch

Decode

Execute

Instructions in Memory

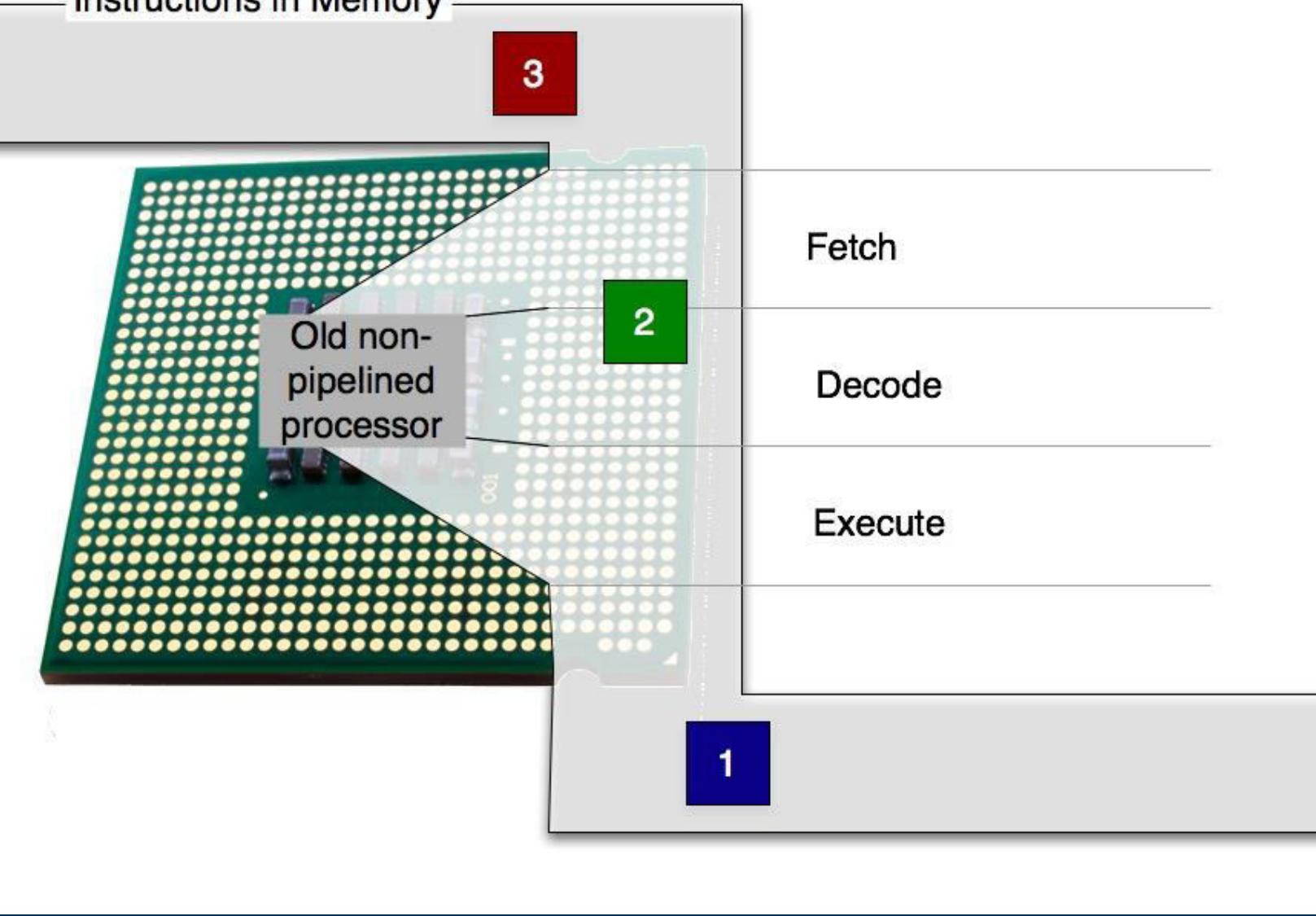


Fetch

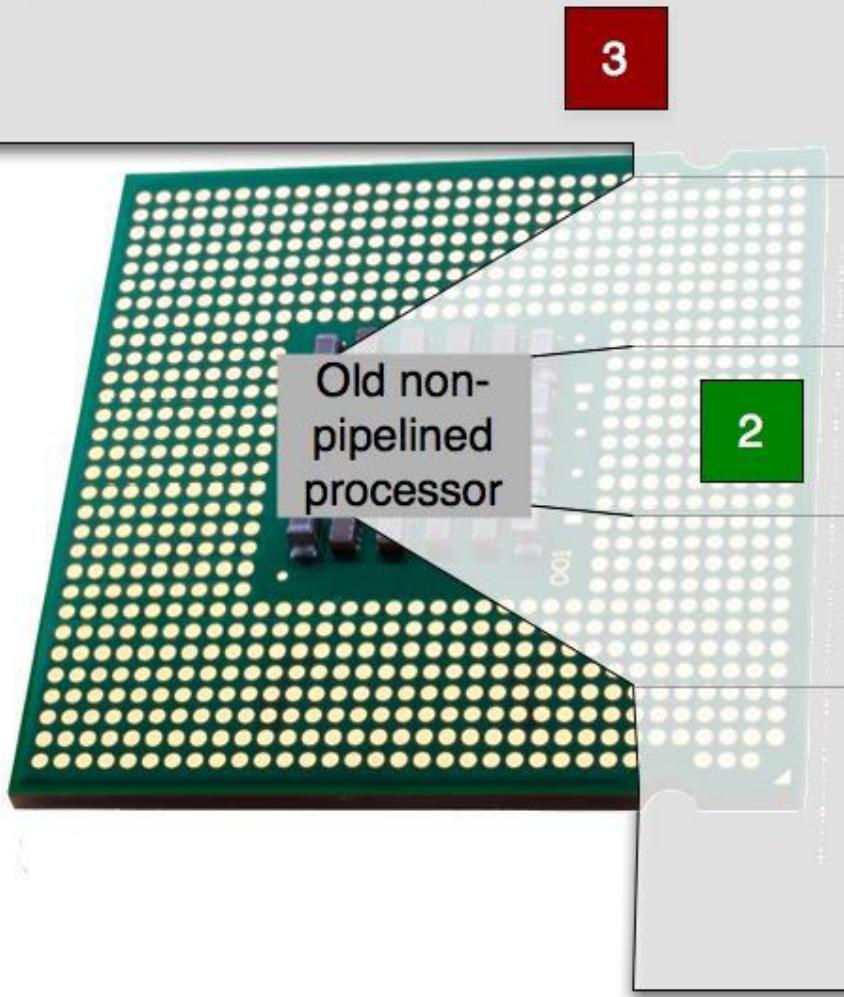
Decode

Execute

Instructions in Memory



Instructions in Memory

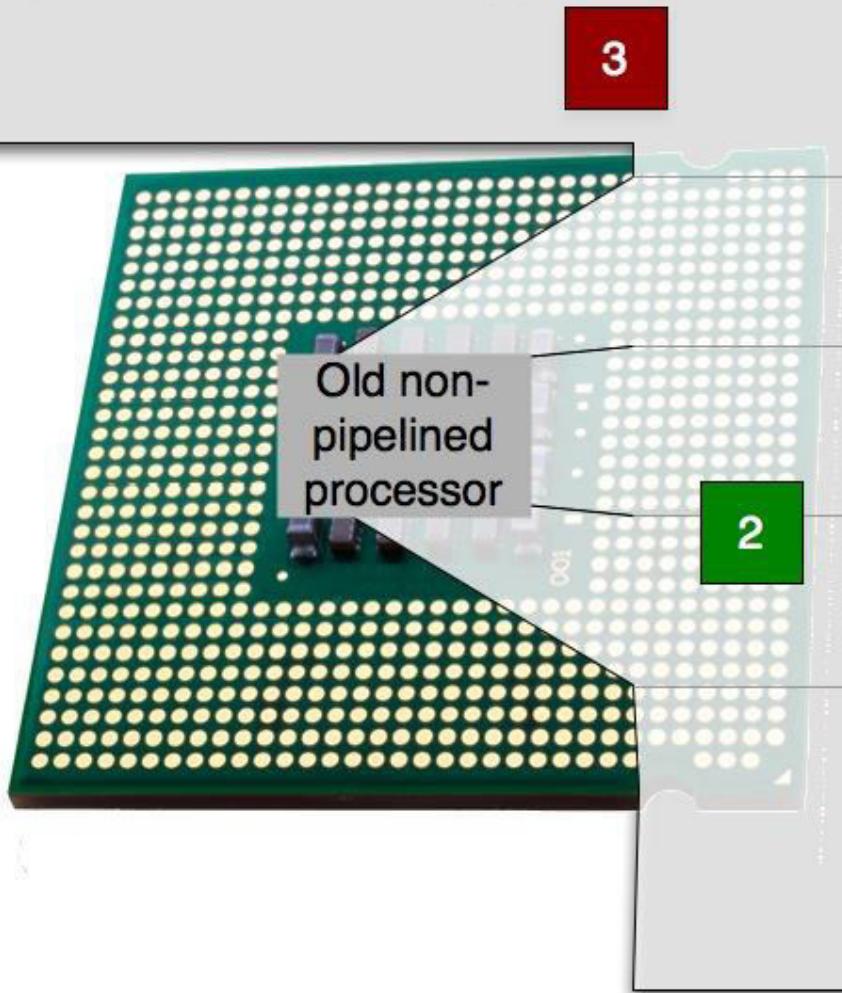


Fetch

Decode

Execute

Instructions in Memory

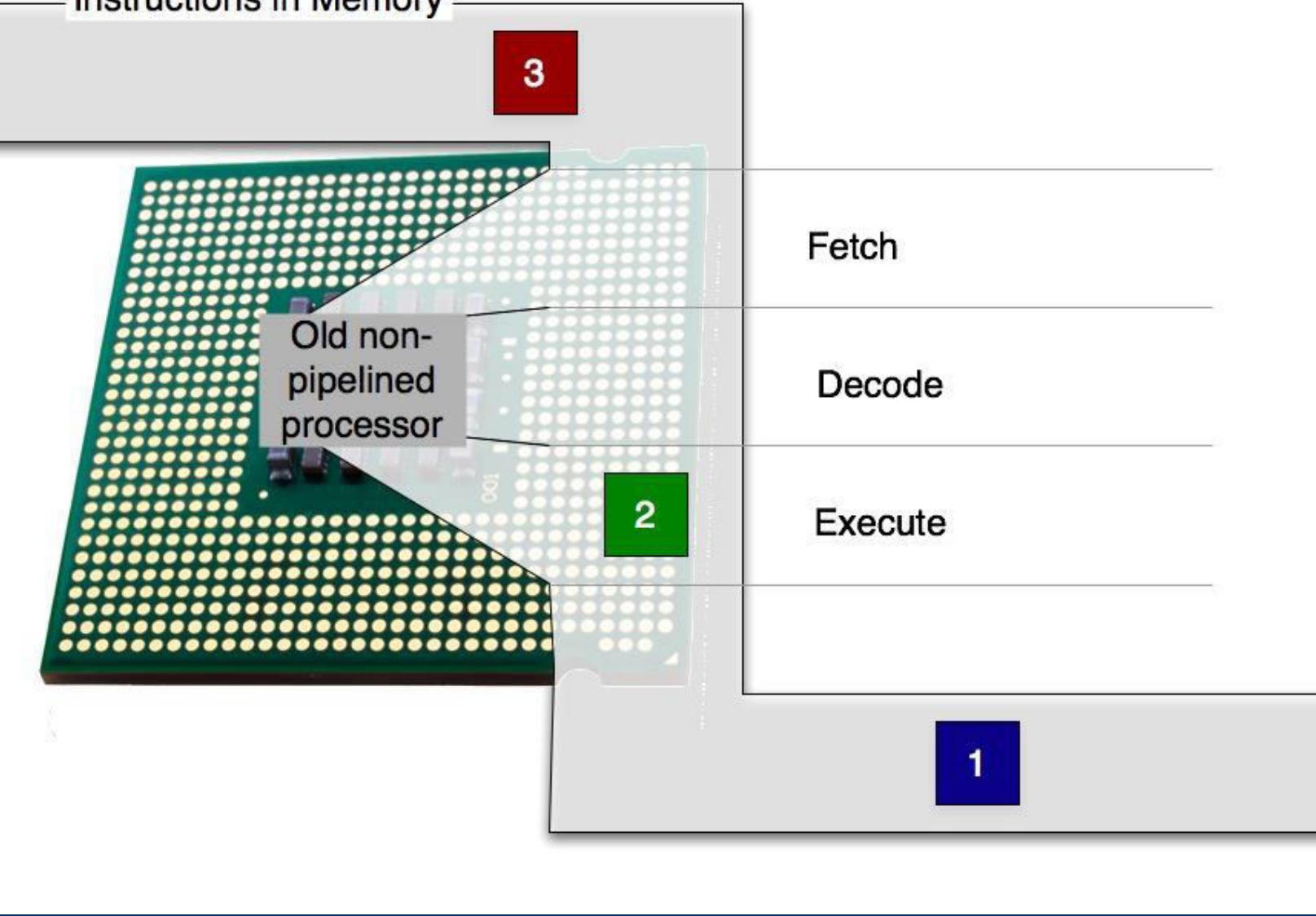


Fetch

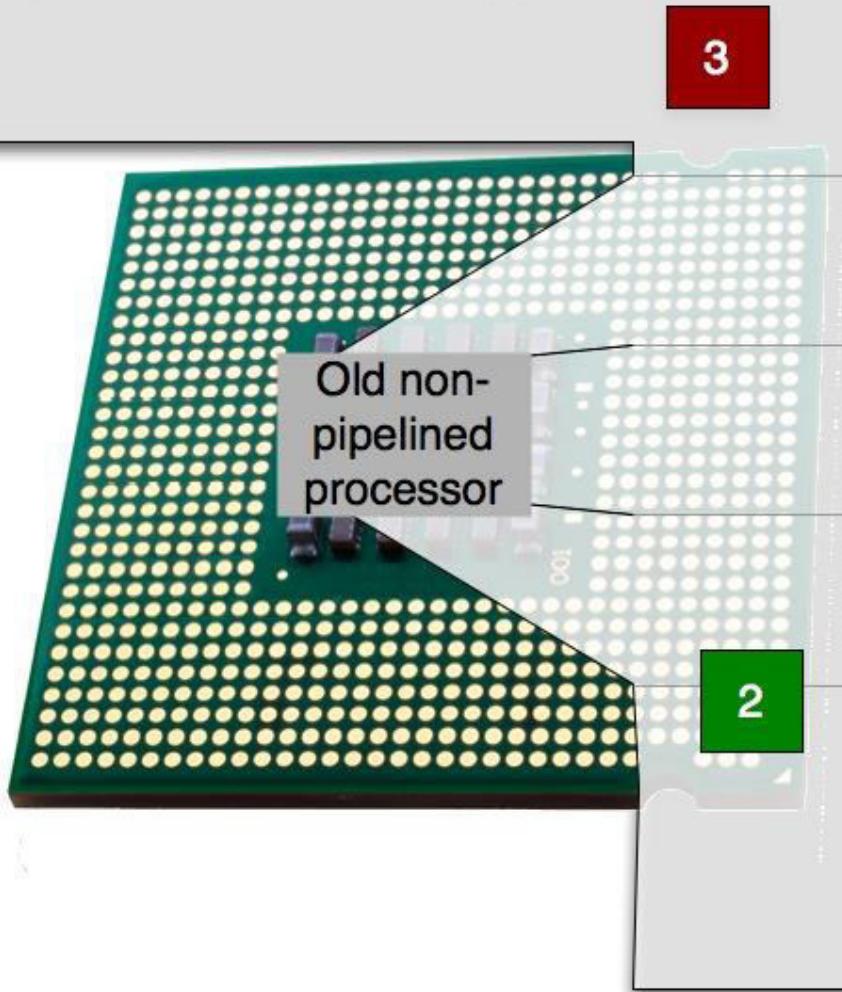
Decode

Execute

Instructions in Memory



Instructions in Memory

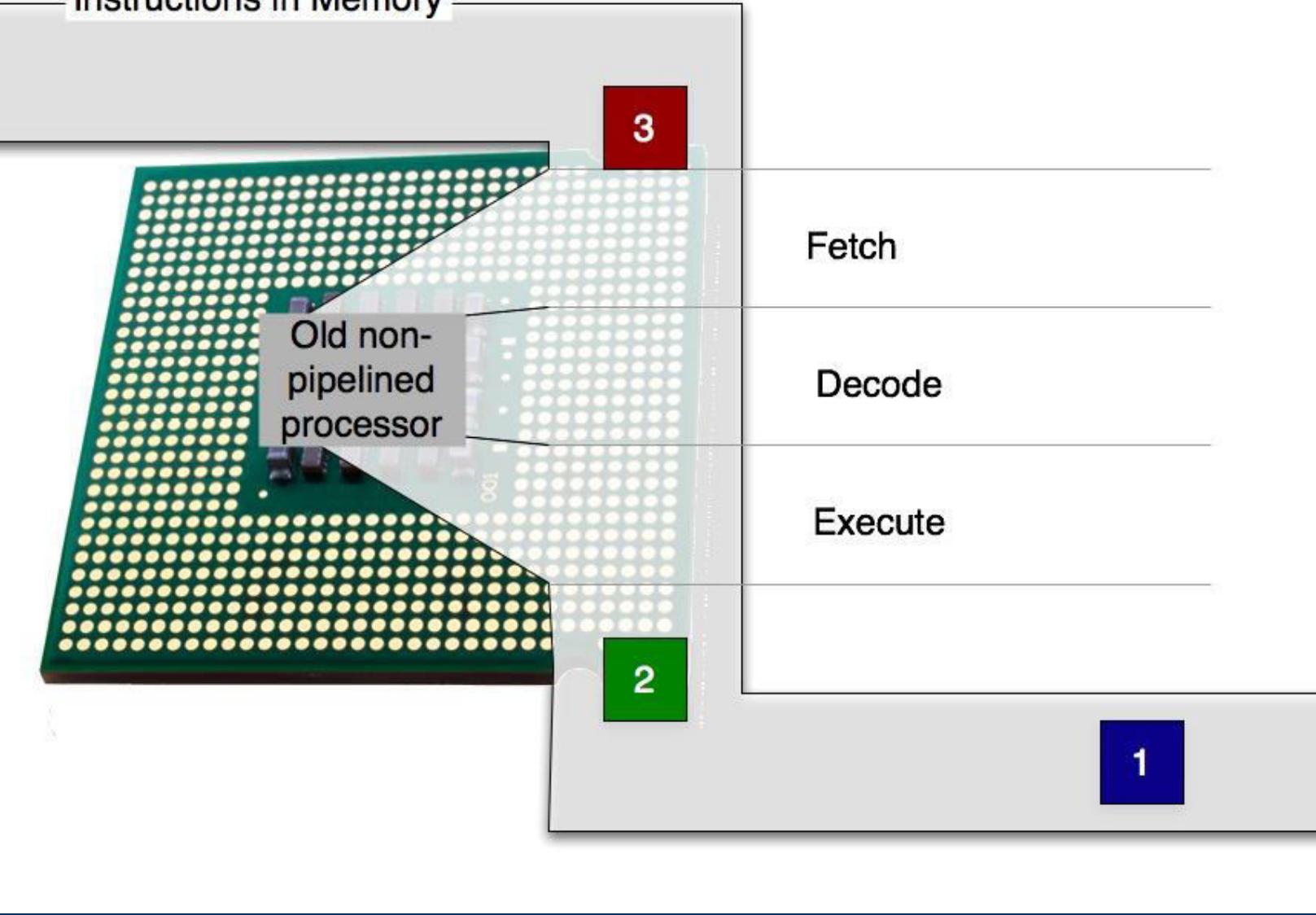


Fetch

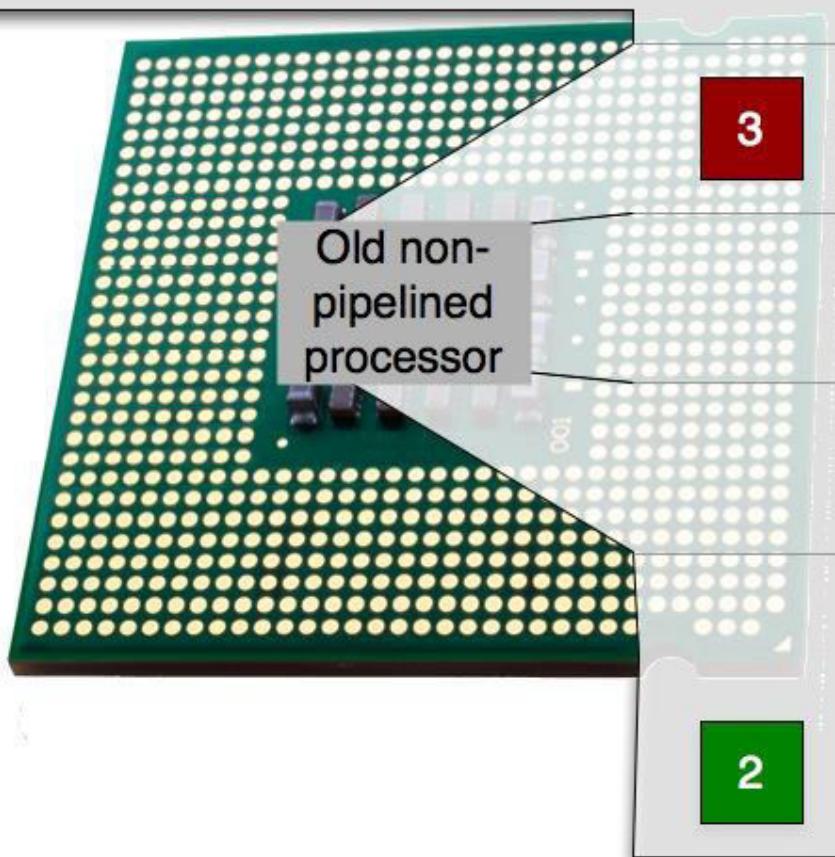
Decode

Execute

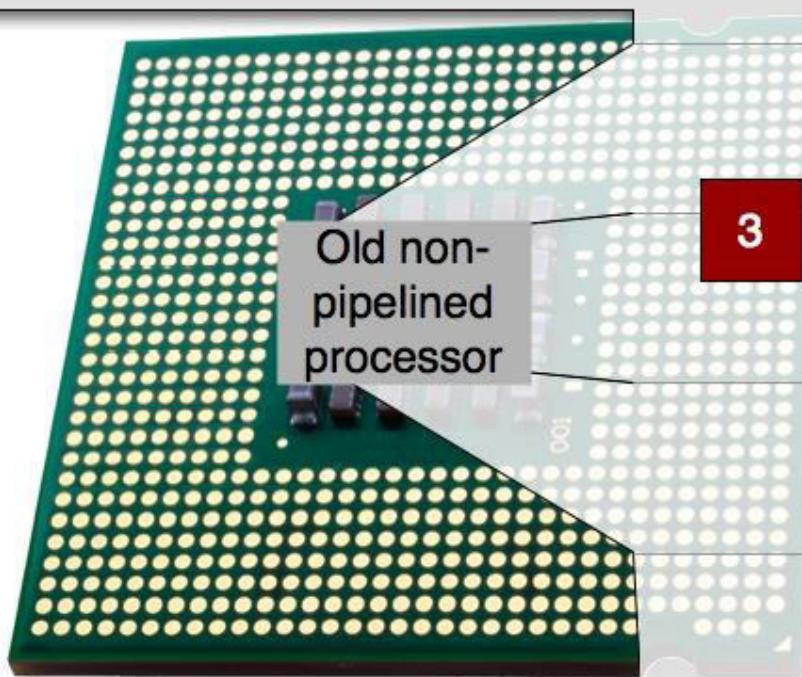
Instructions in Memory



Instructions in Memory



Instructions in Memory



Fetch

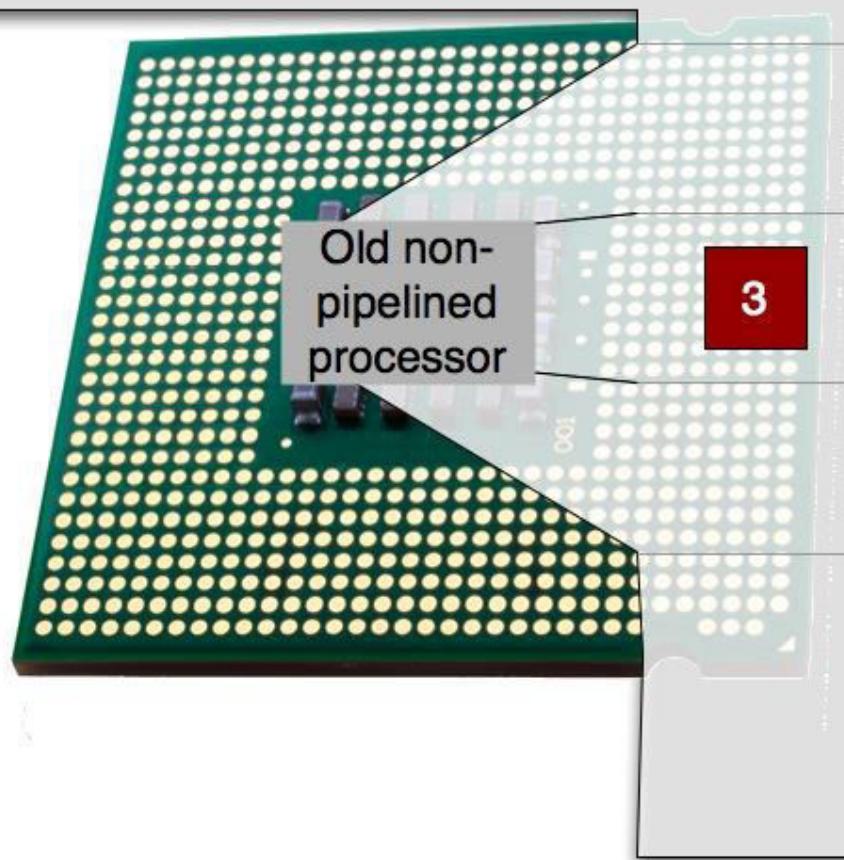
Decode

Execute

2

1

Instructions in Memory

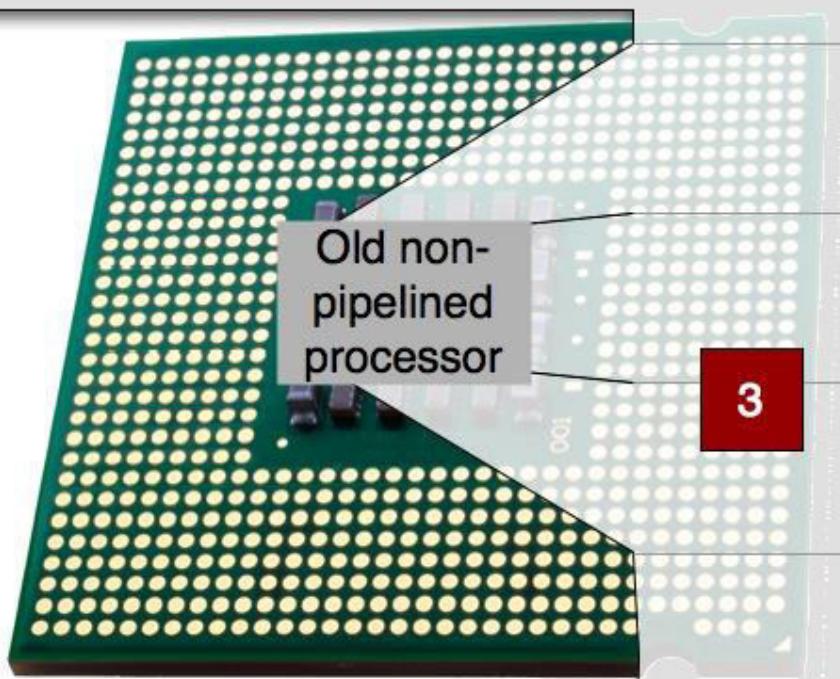


Fetch

Decode

Execute

Instructions in Memory



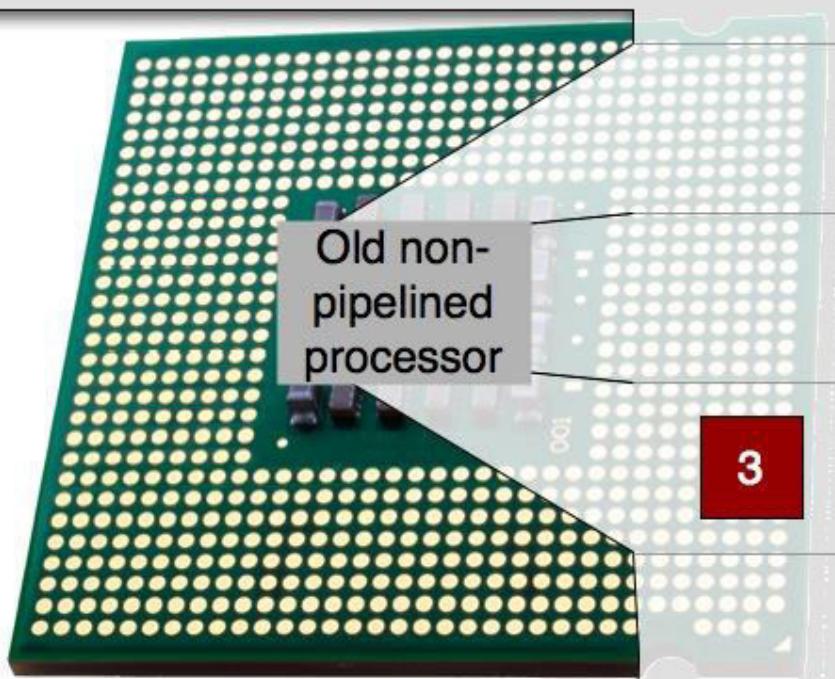
Fetch

Decode

Execute

2

Instructions in Memory



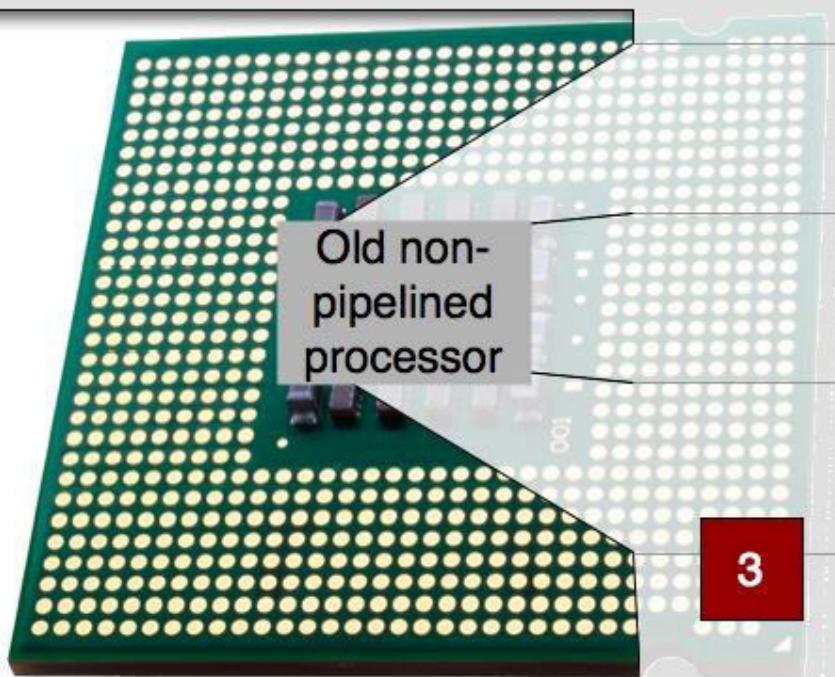
Fetch

Decode

Execute

2

Instructions in Memory



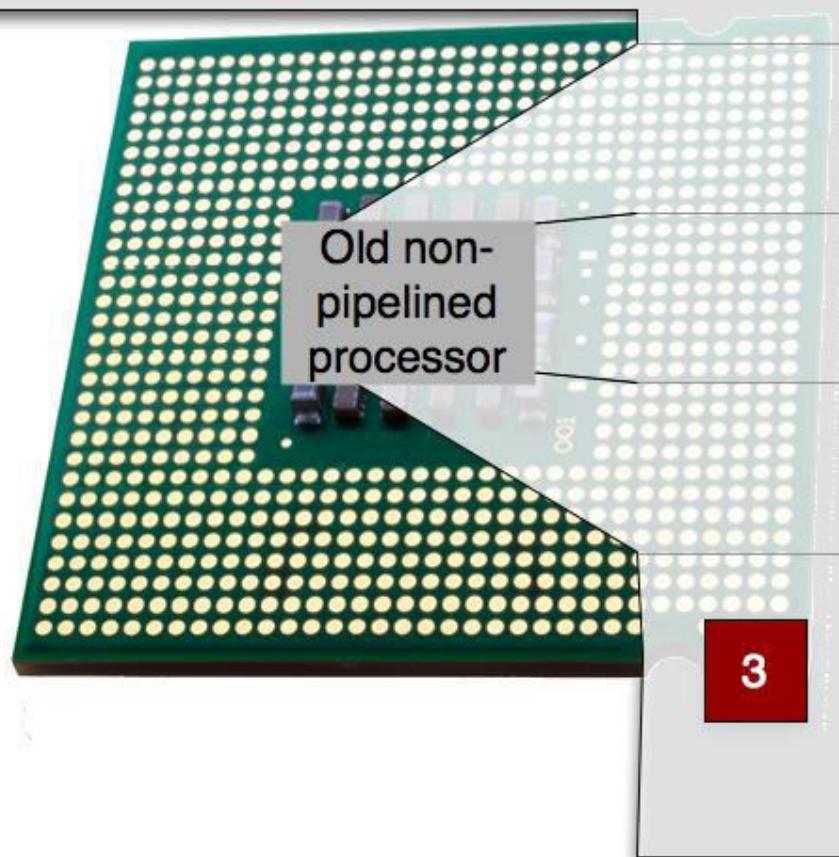
Fetch

Decode

Execute

2

Instructions in Memory



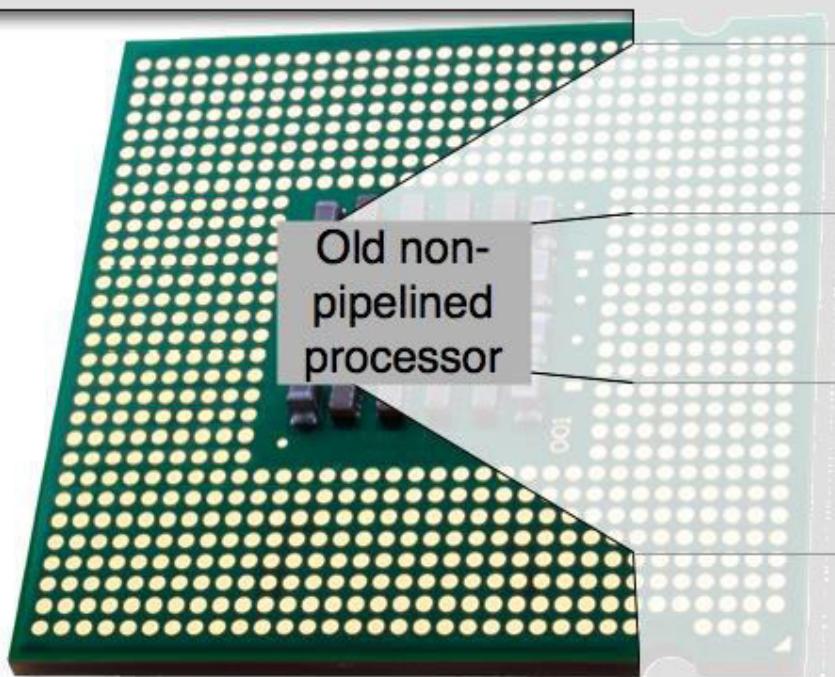
Fetch

Decode

Execute

2

Instructions in Memory



Fetch

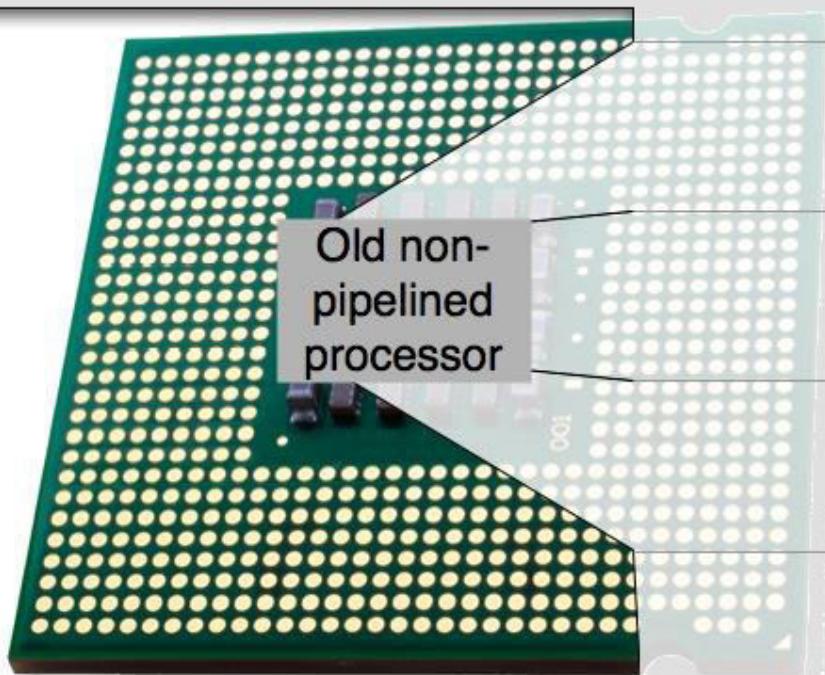
Decode

Execute

3

2

Instructions in Memory



Fetch

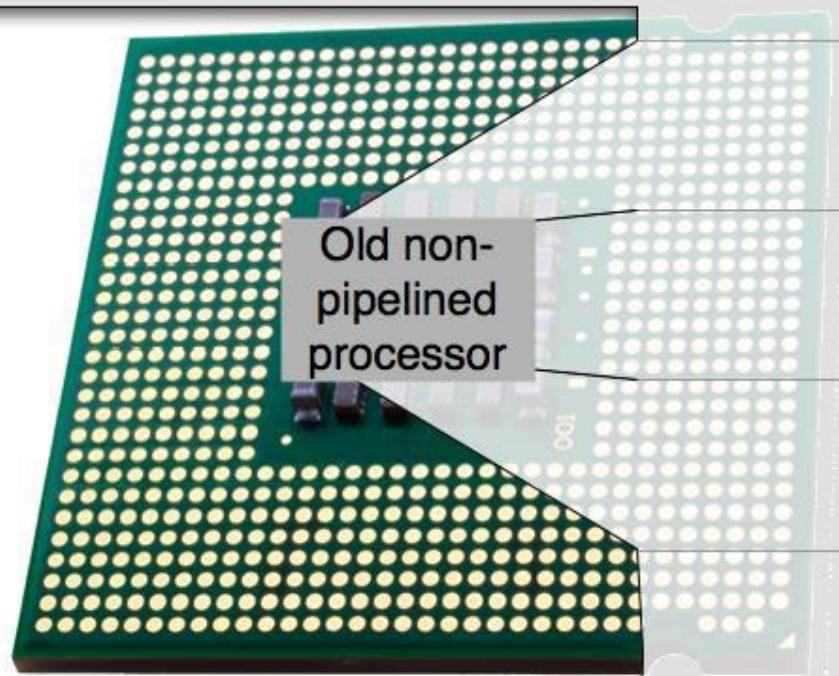
Decode

Execute

3

2

Instructions in Memory



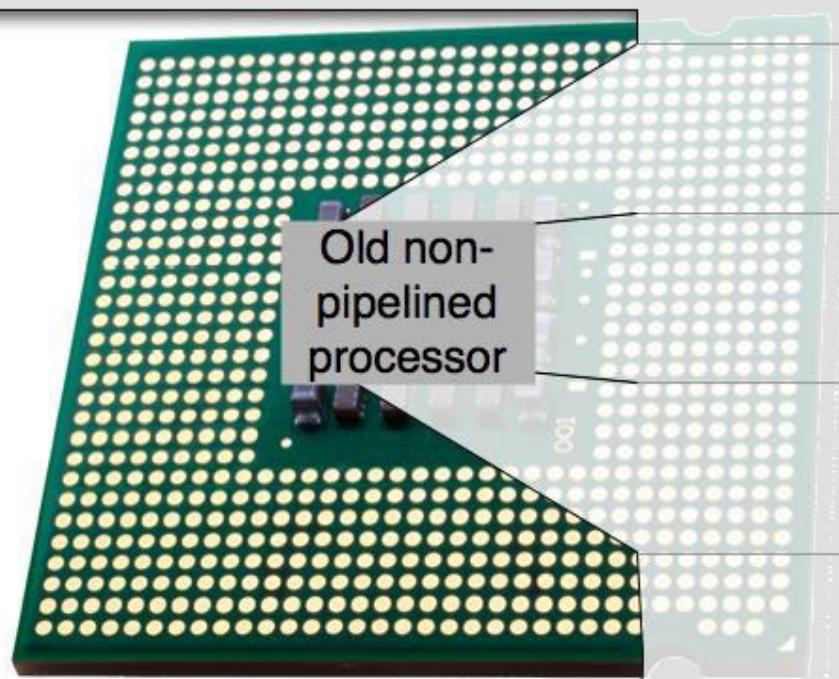
Fetch

Decode

Execute

3

Instructions in Memory



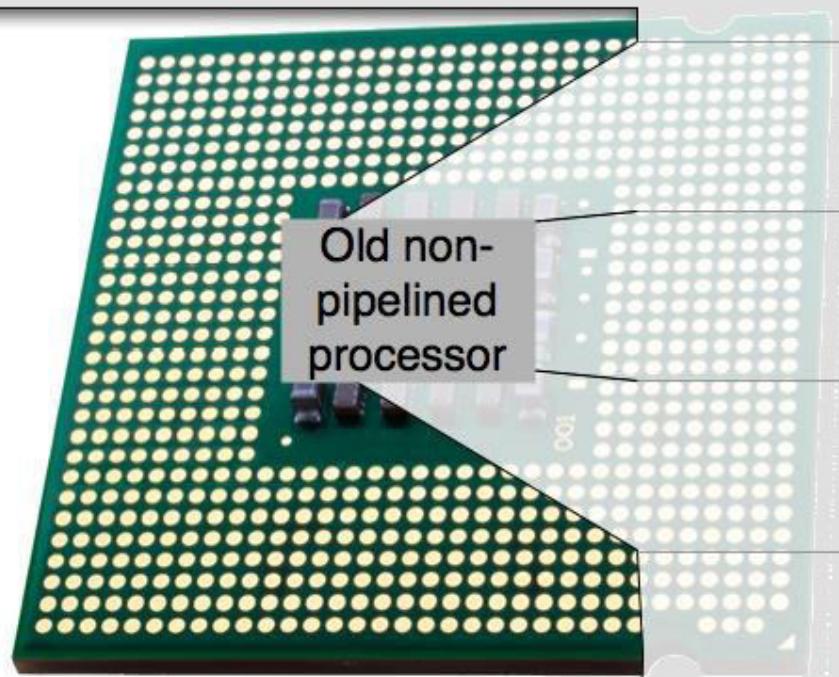
Fetch

Decode

Execute

3

Instructions in Memory



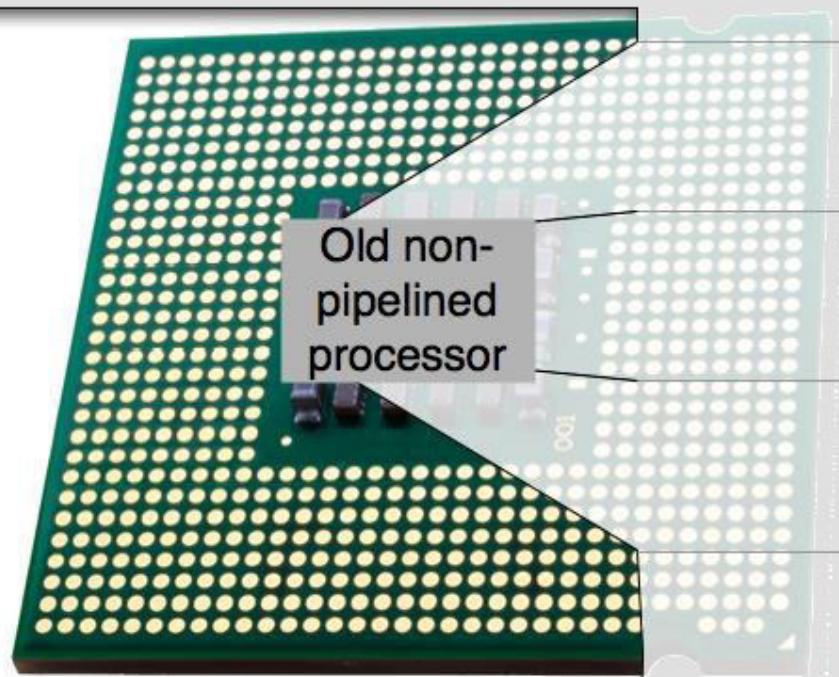
Fetch

Decode

Execute

3

Instructions in Memory



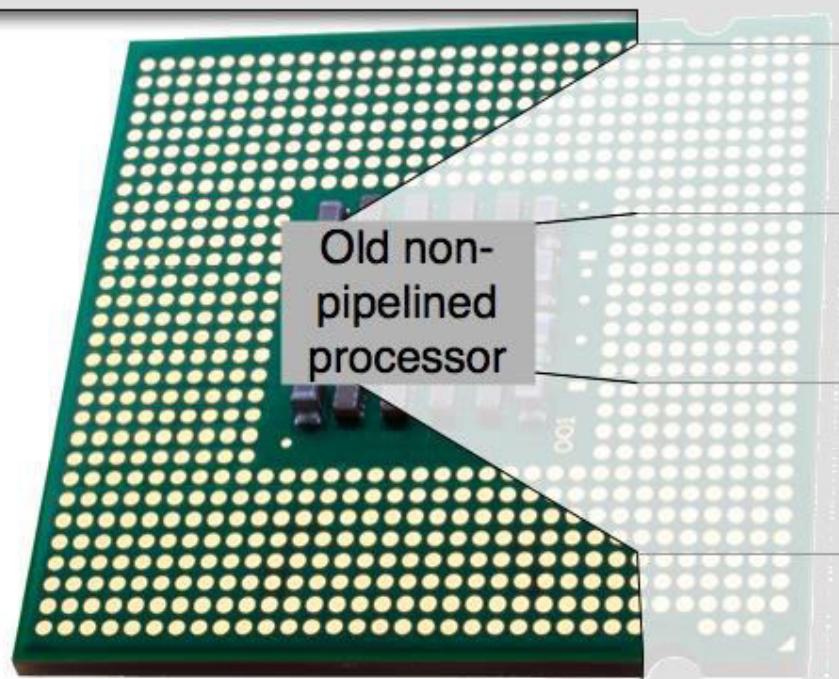
Fetch

Decode

Execute

3

Instructions in Memory



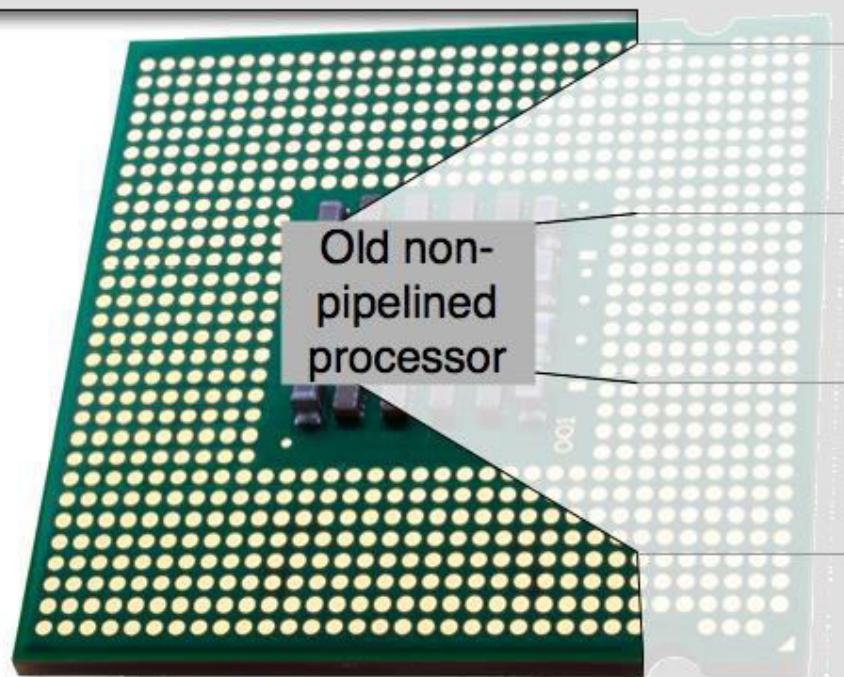
Fetch

Decode

Execute

3

Instructions in Memory



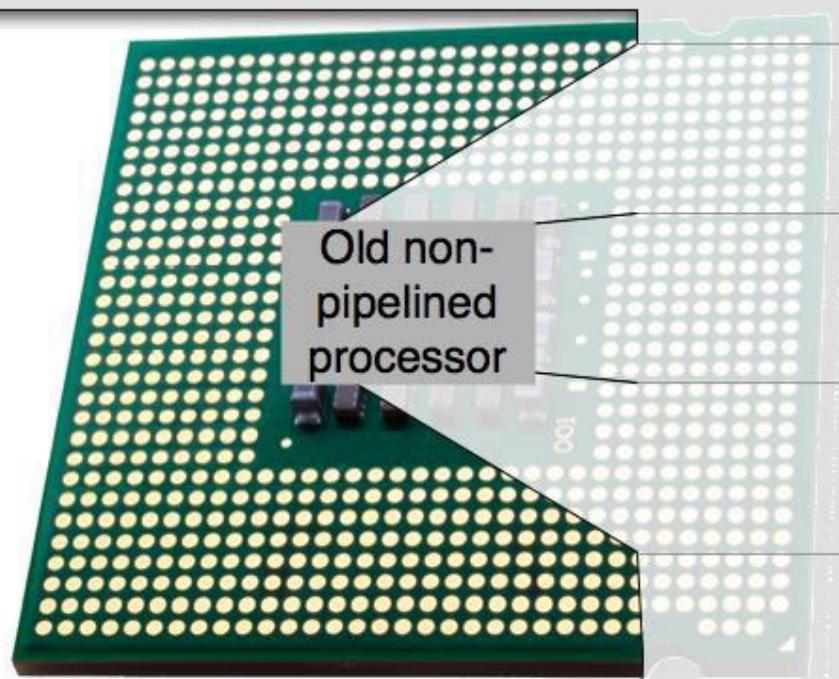
Fetch

Decode

Execute

3

Instructions in Memory

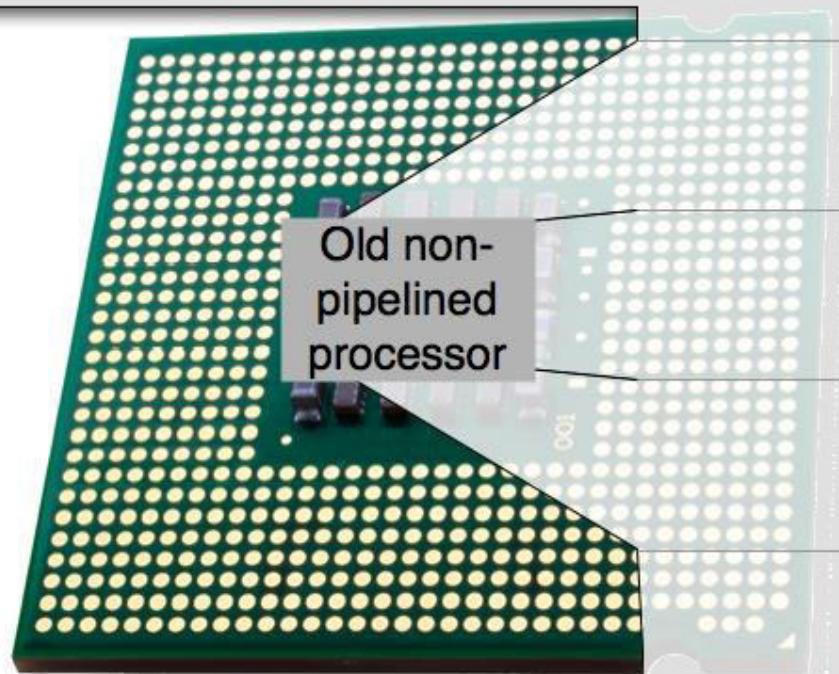


Fetch

Decode

Execute

Instructions in Memory



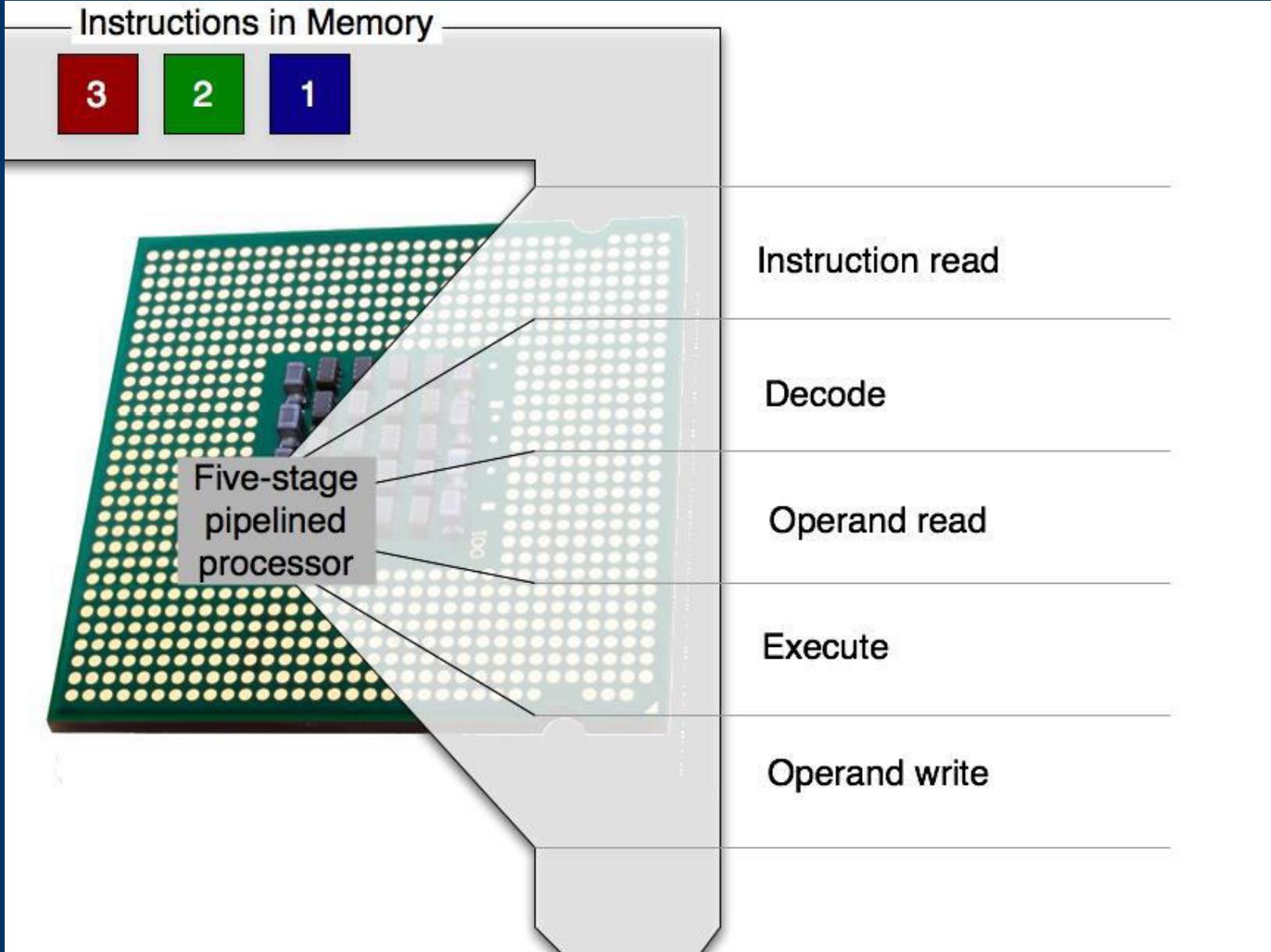
Fetch

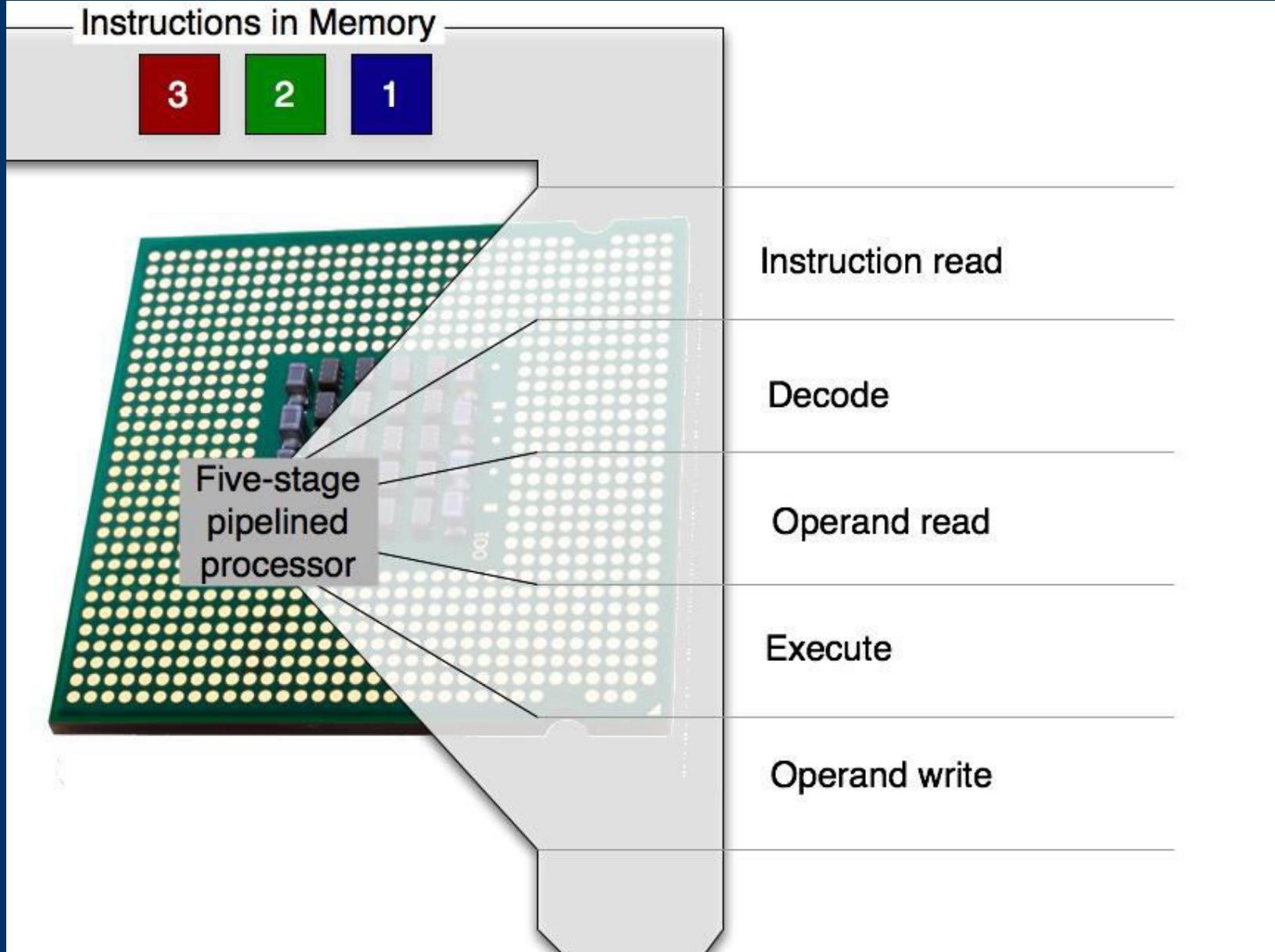
Decode

Execute

How does pipelining work?

A short(er) animation
of pipelined processor routines



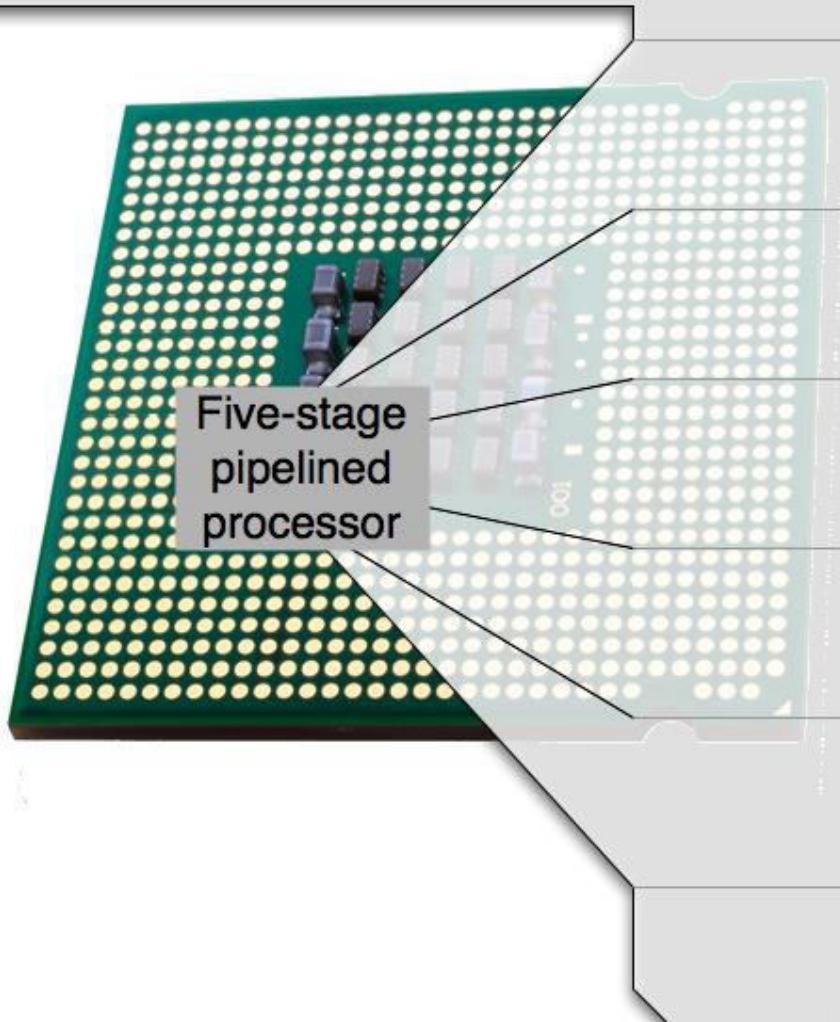


Instructions in Memory

3

2

1



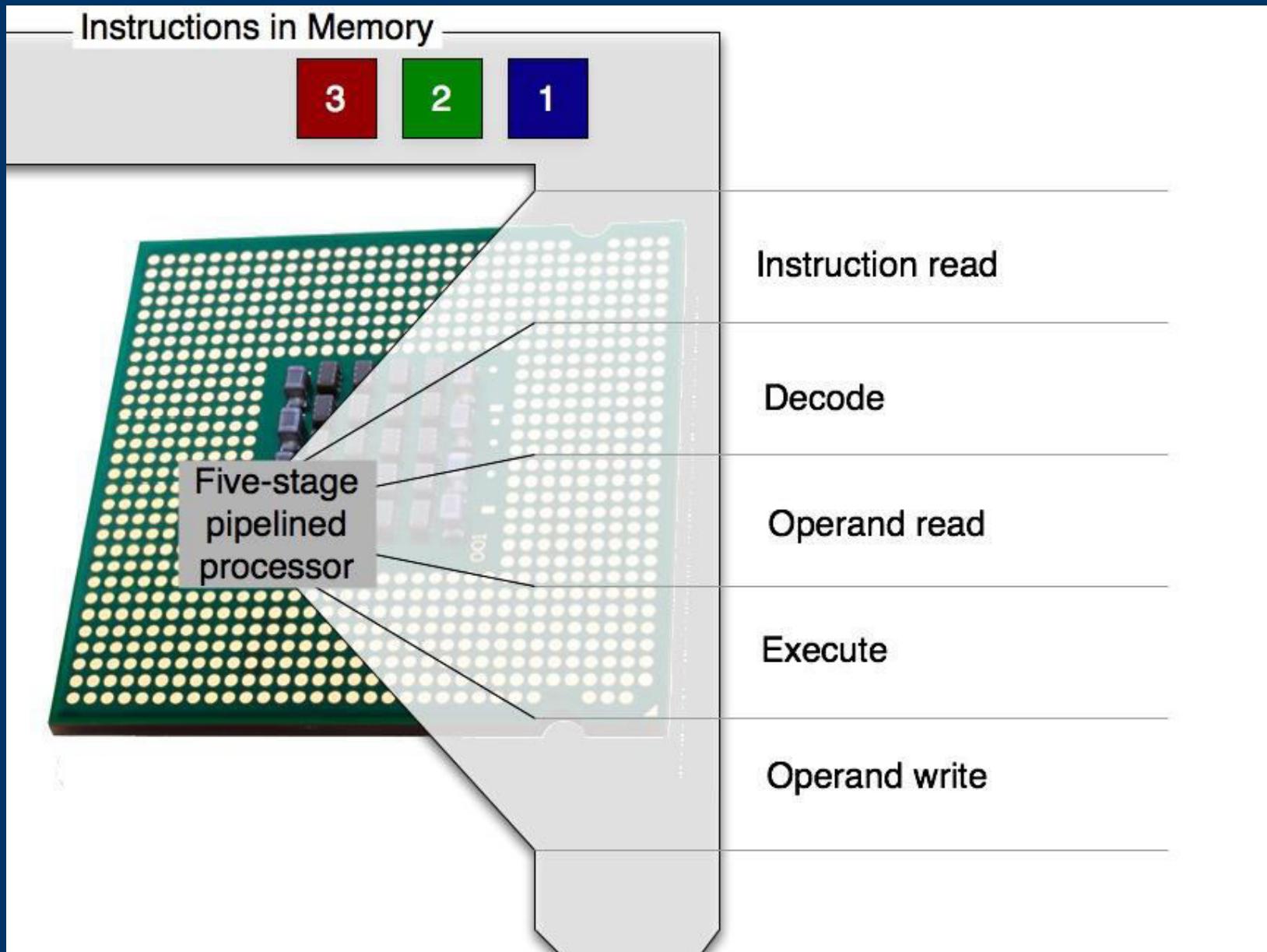
Instruction read

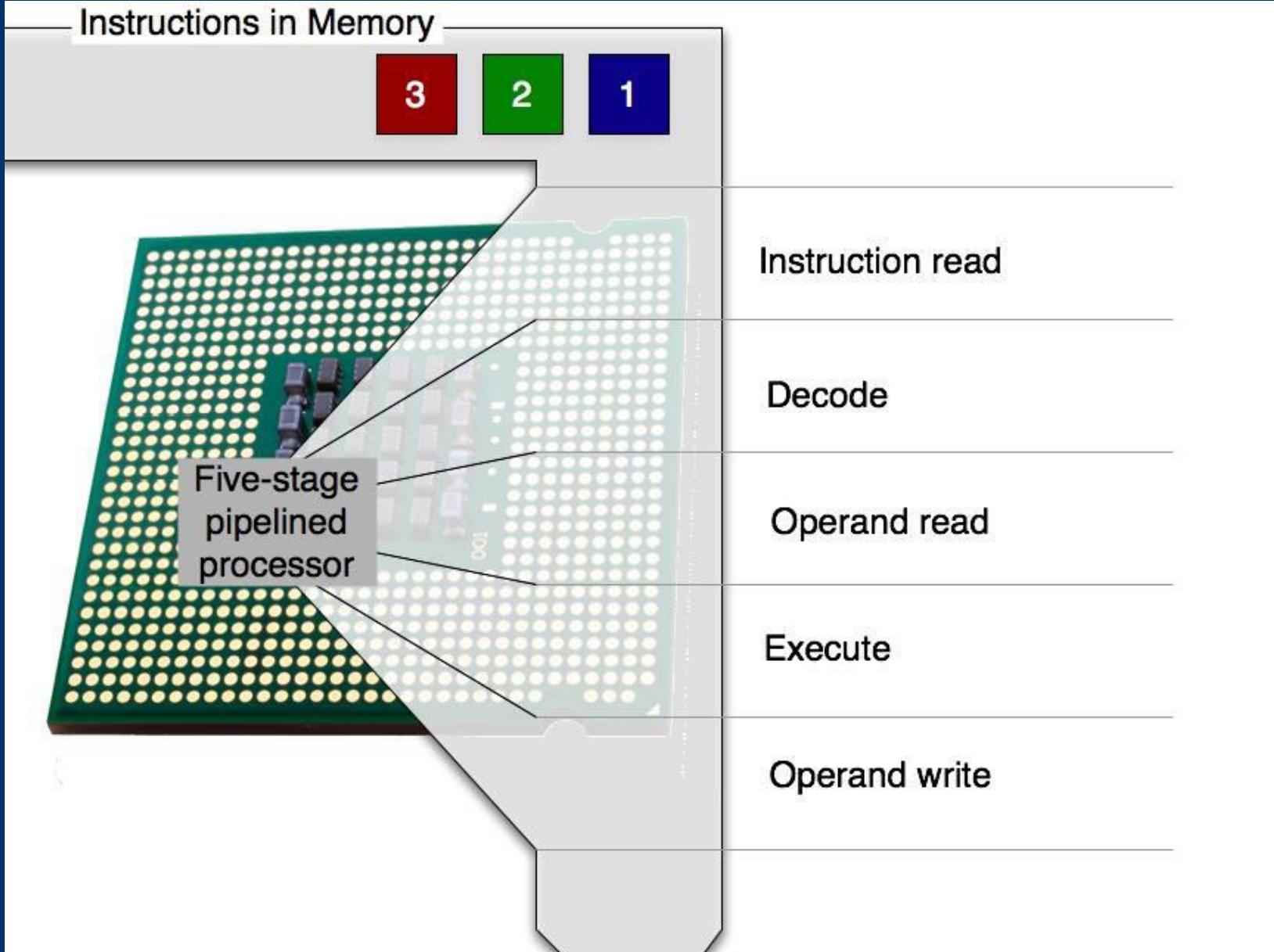
Decode

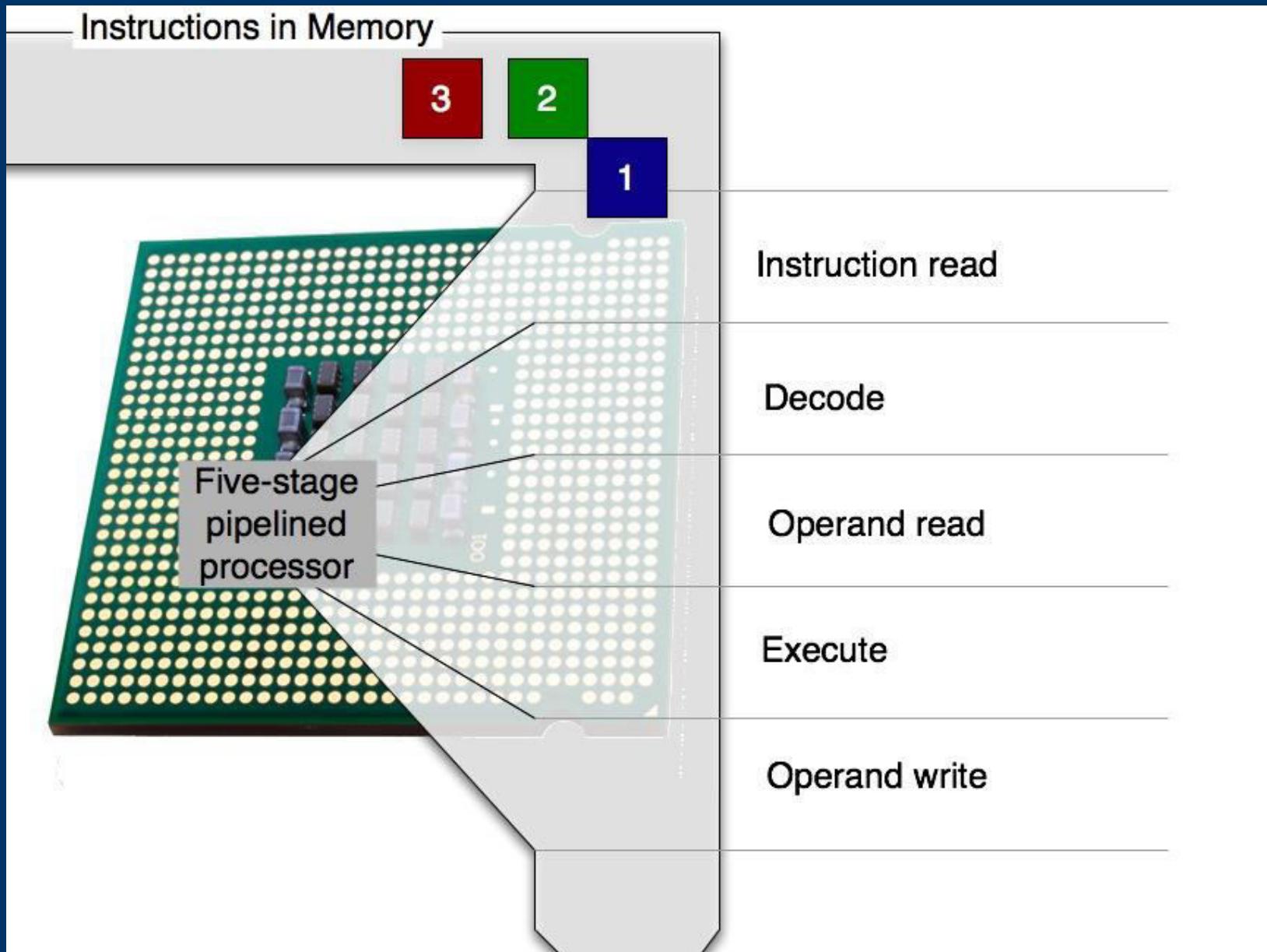
Operand read

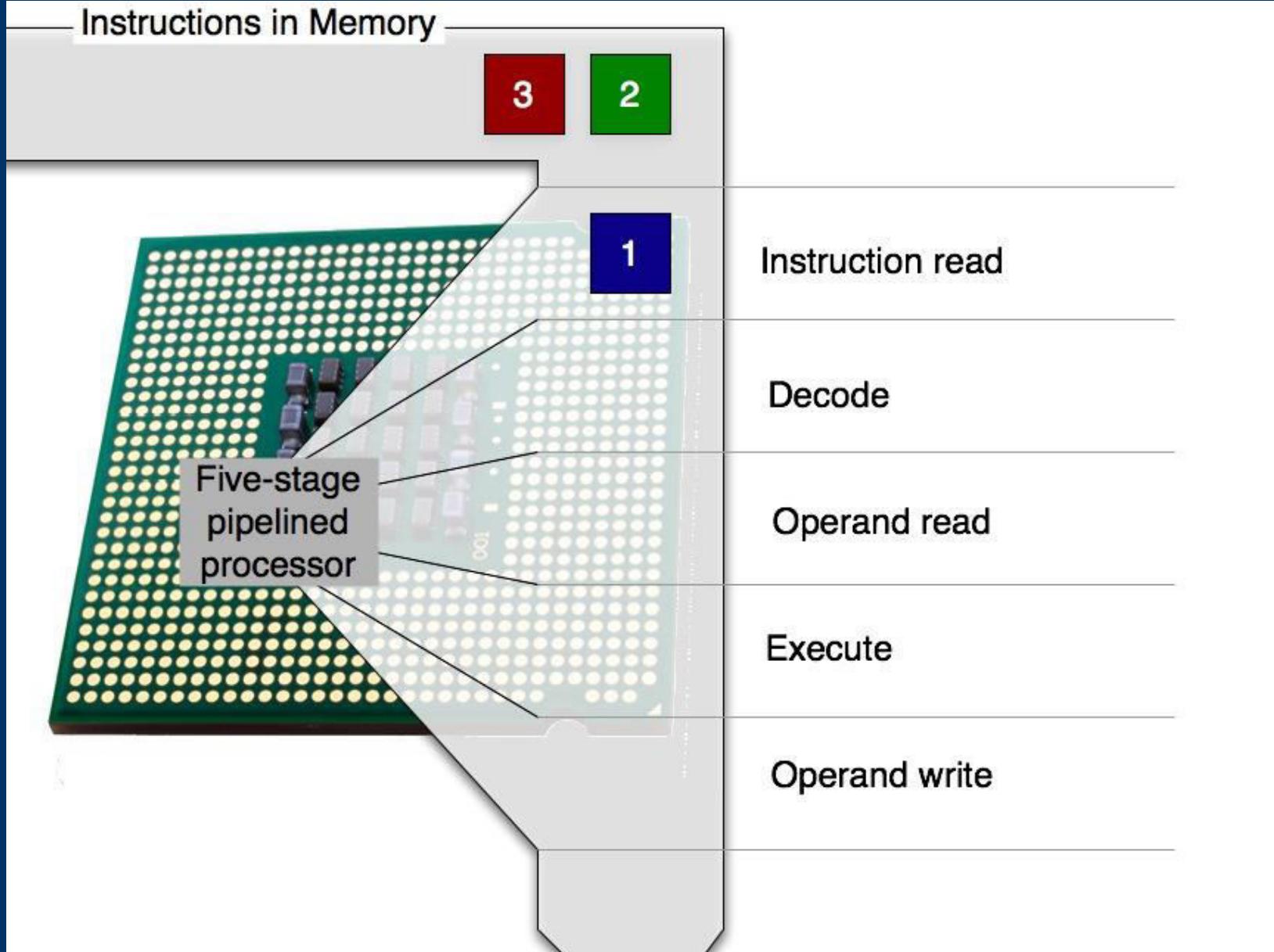
Execute

Operand write

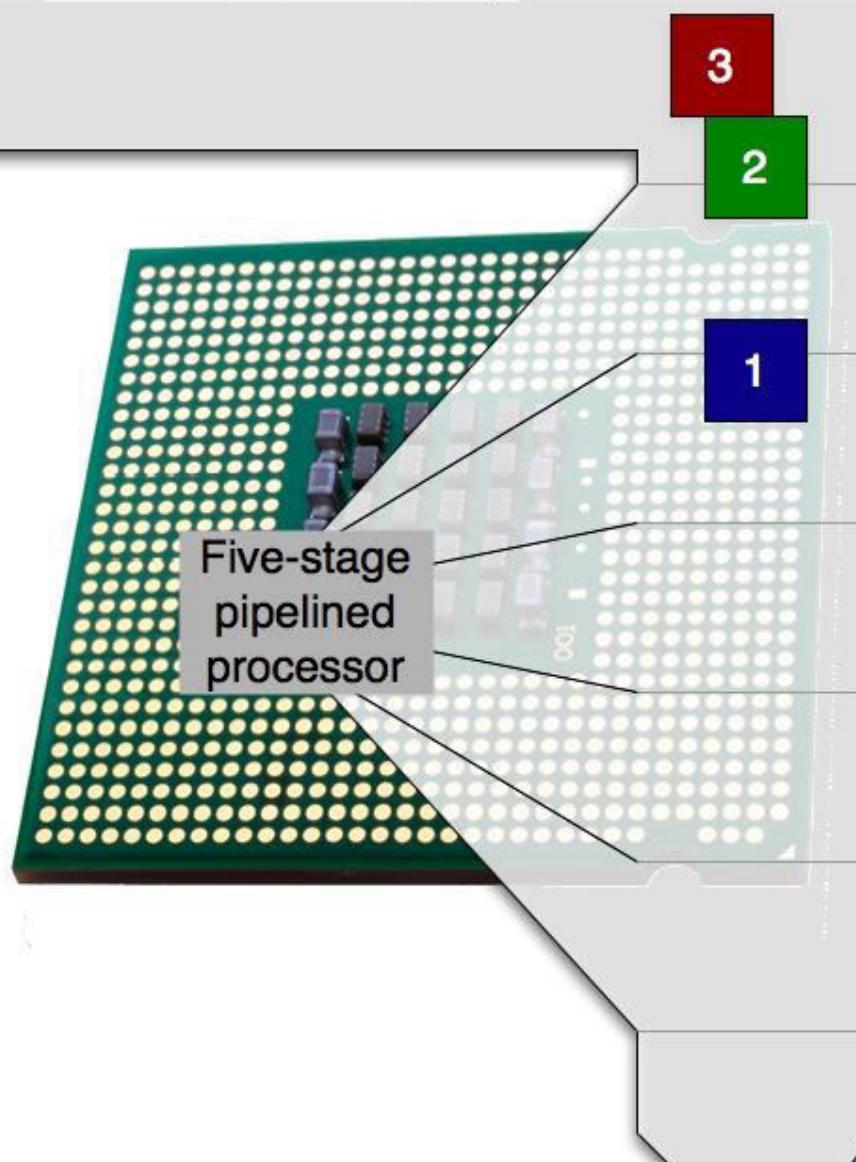








Instructions in Memory



Instruction read

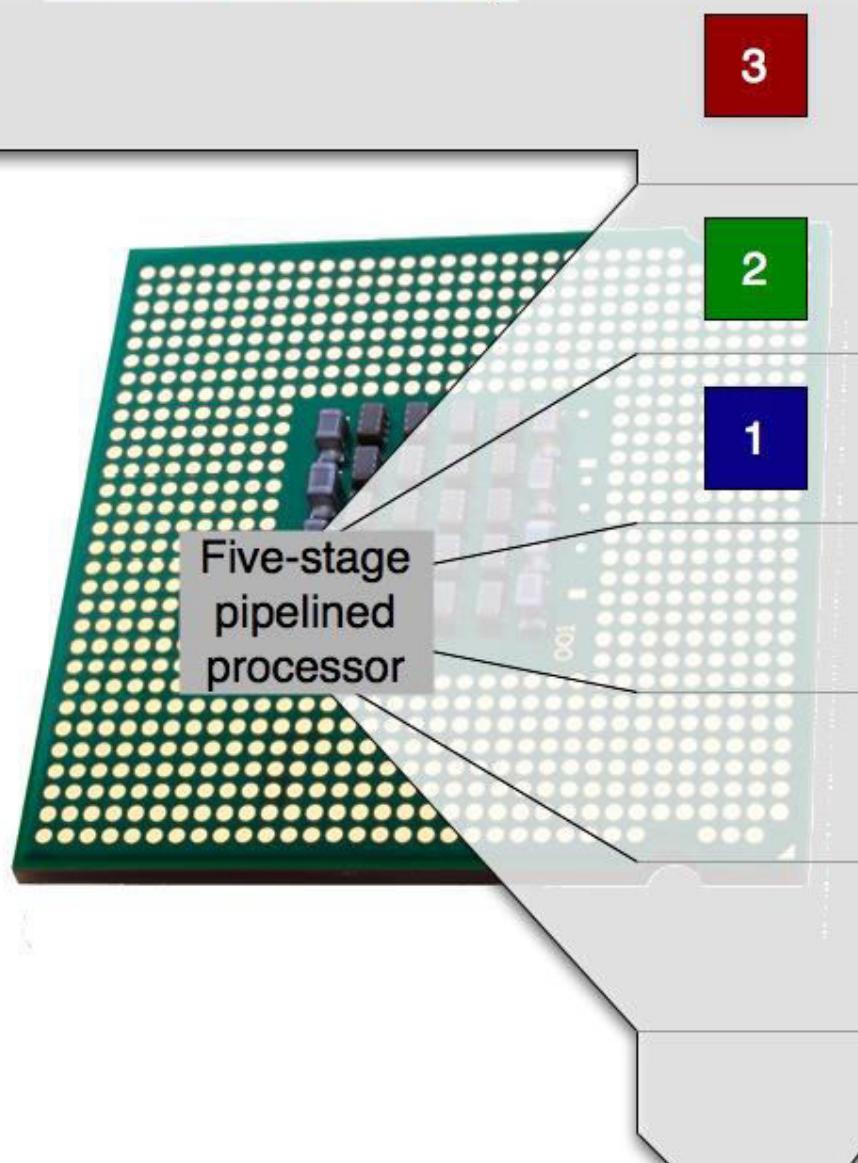
Decode

Operand read

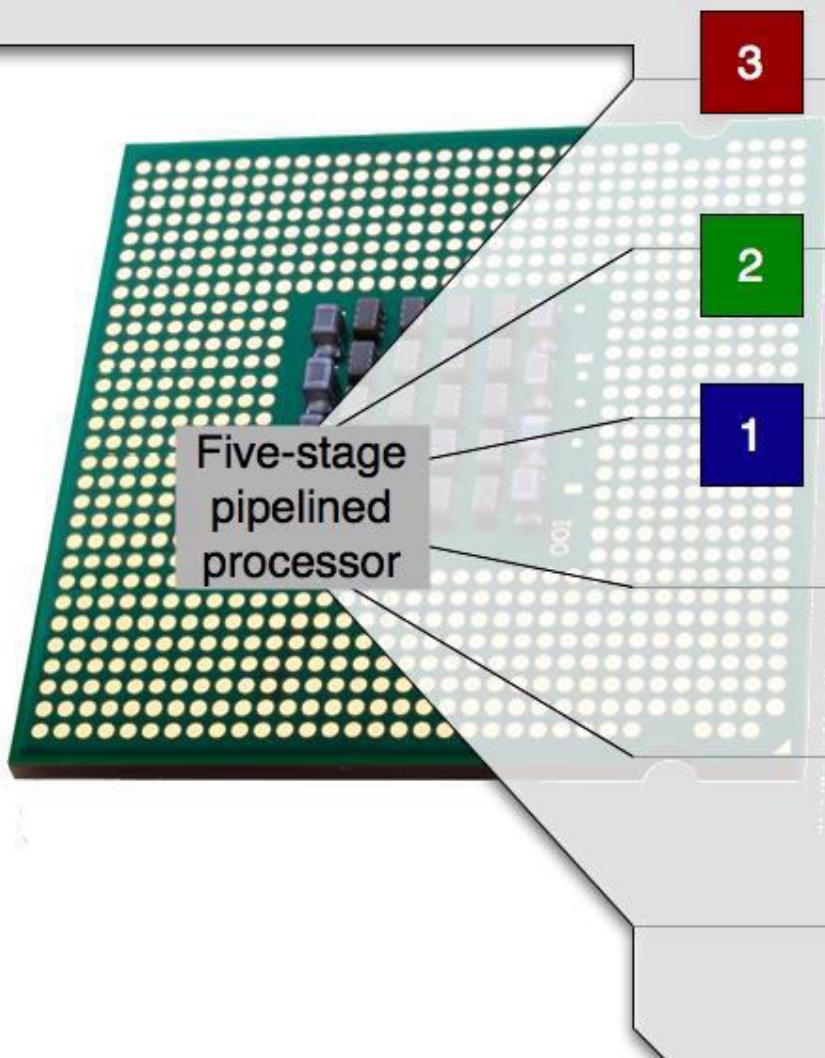
Execute

Operand write

Instructions in Memory



Instructions in Memory



Instruction read

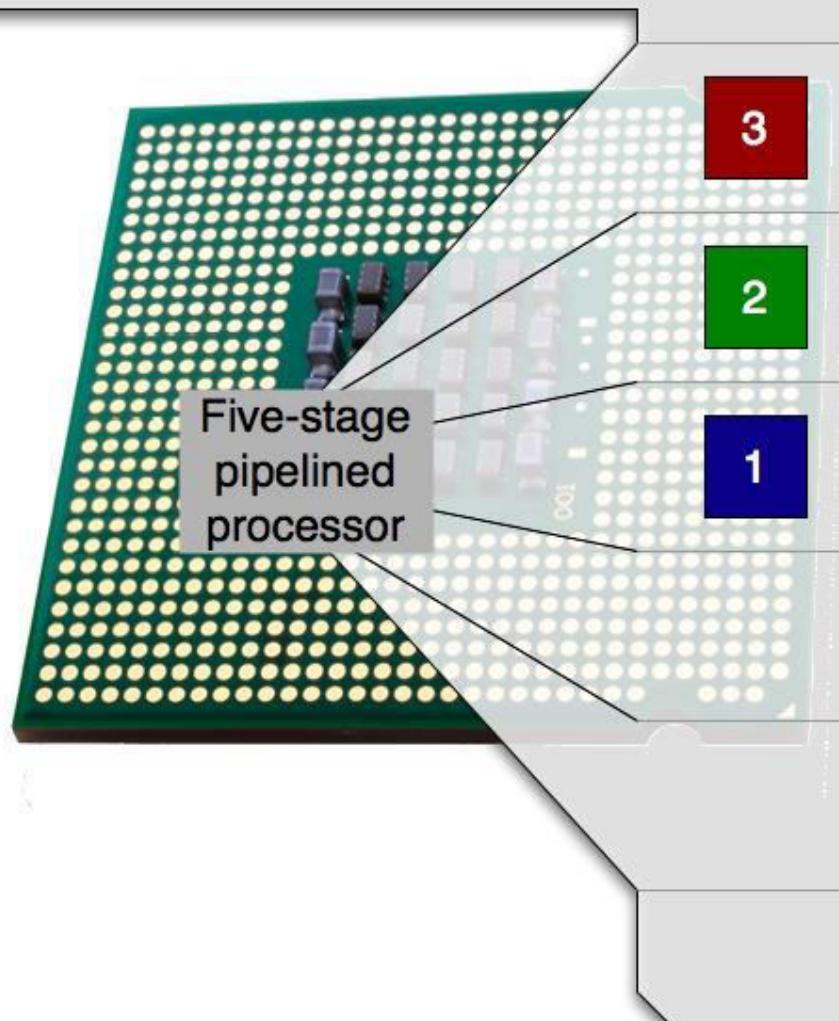
Decode

Operand read

Execute

Operand write

Instructions in Memory



Instruction read

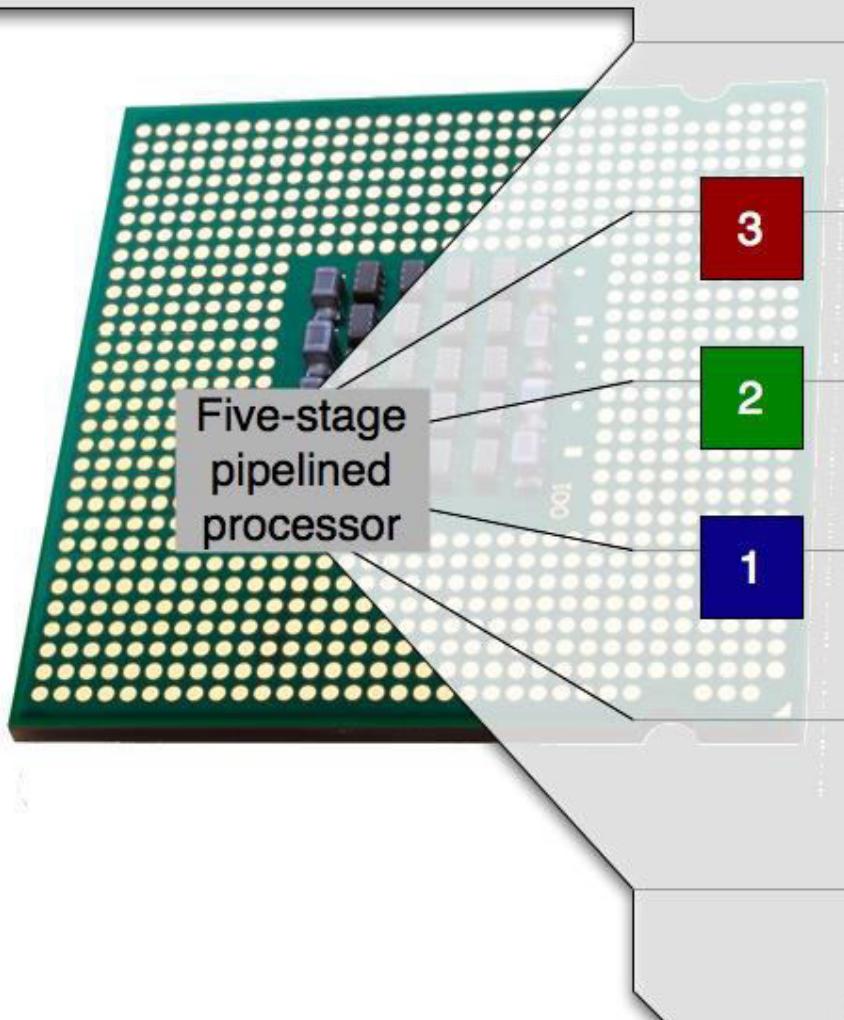
Decode

Operand read

Execute

Operand write

Instructions in Memory



Instruction read

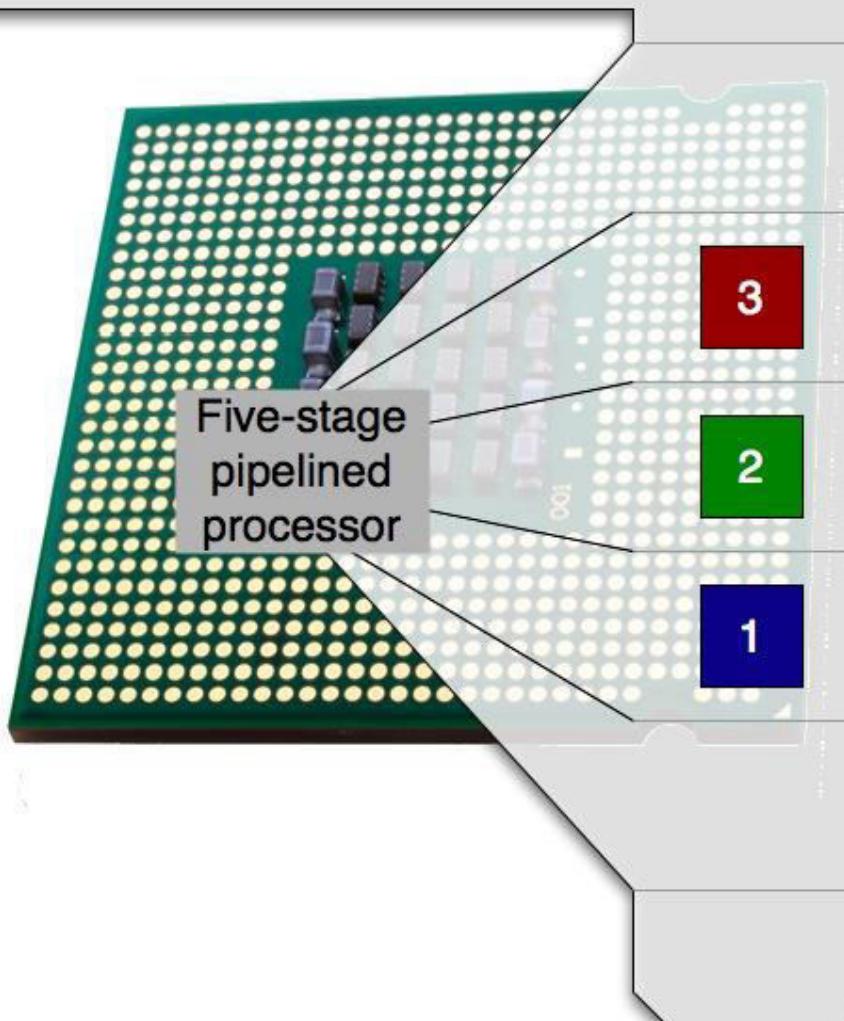
Decode

Operand read

Execute

Operand write

Instructions in Memory



Instruction read

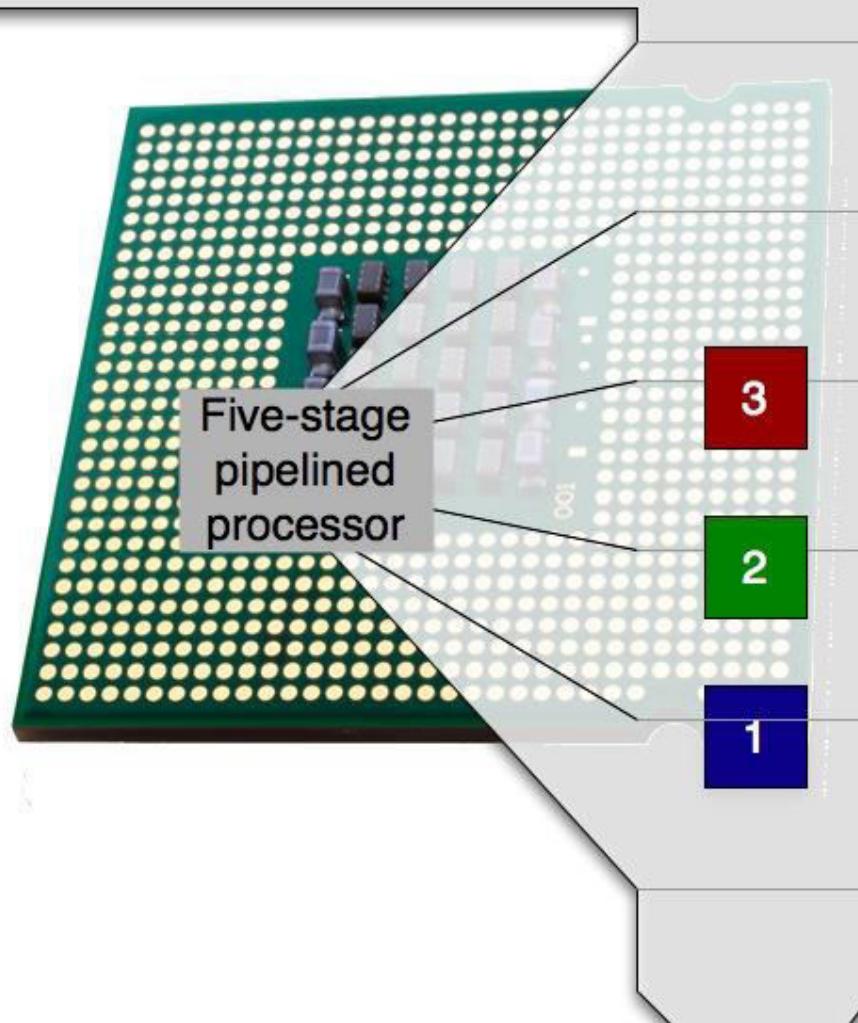
Decode

Operand read

Execute

Operand write

Instructions in Memory



Instruction read

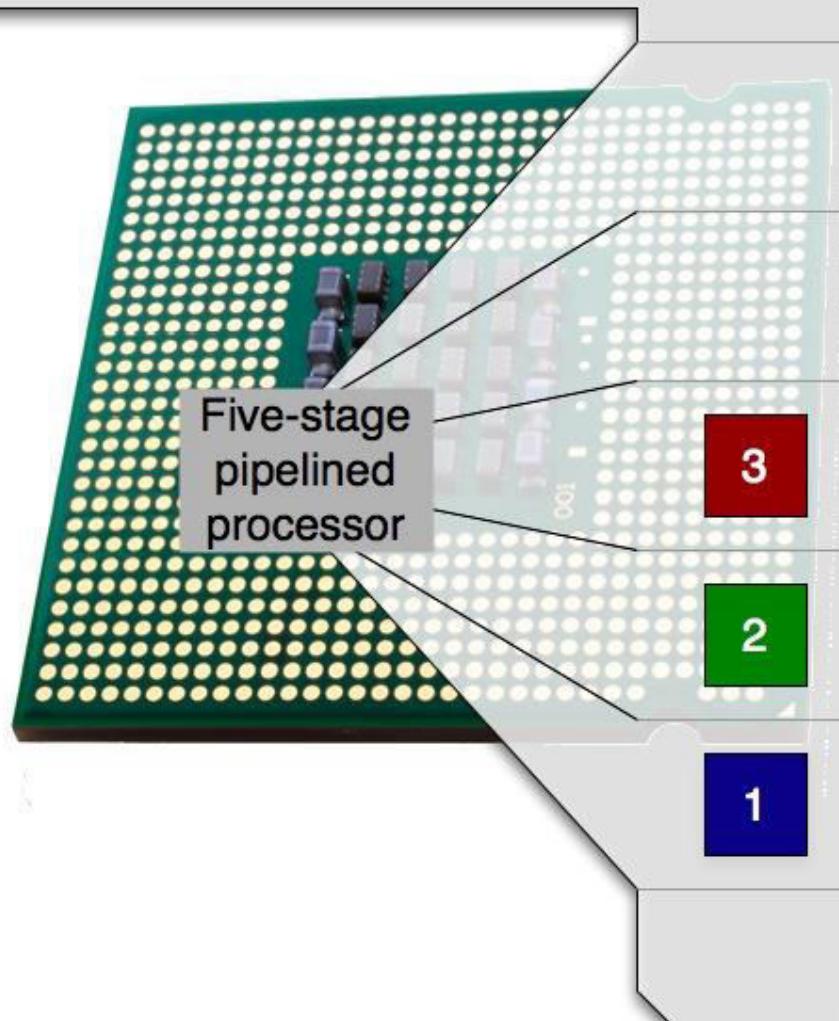
Decode

Operand read

Execute

Operand write

Instructions in Memory



Instruction read

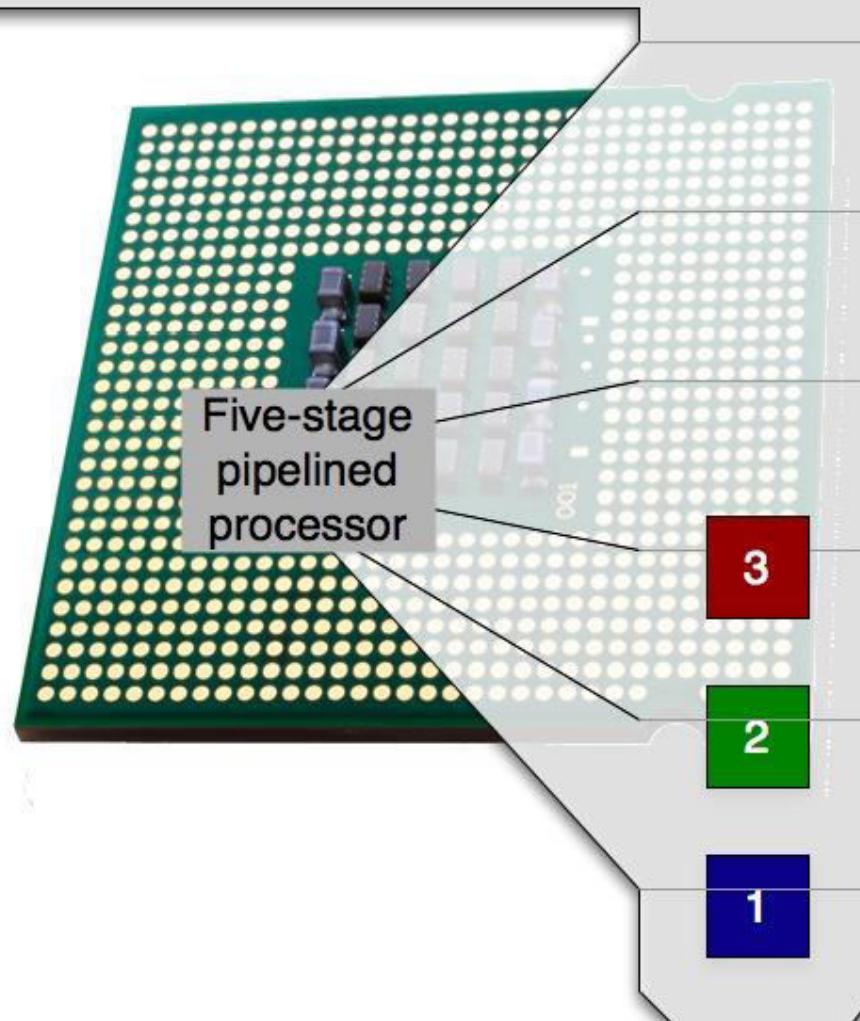
Decode

Operand read

Execute

Operand write

Instructions in Memory



Instruction read

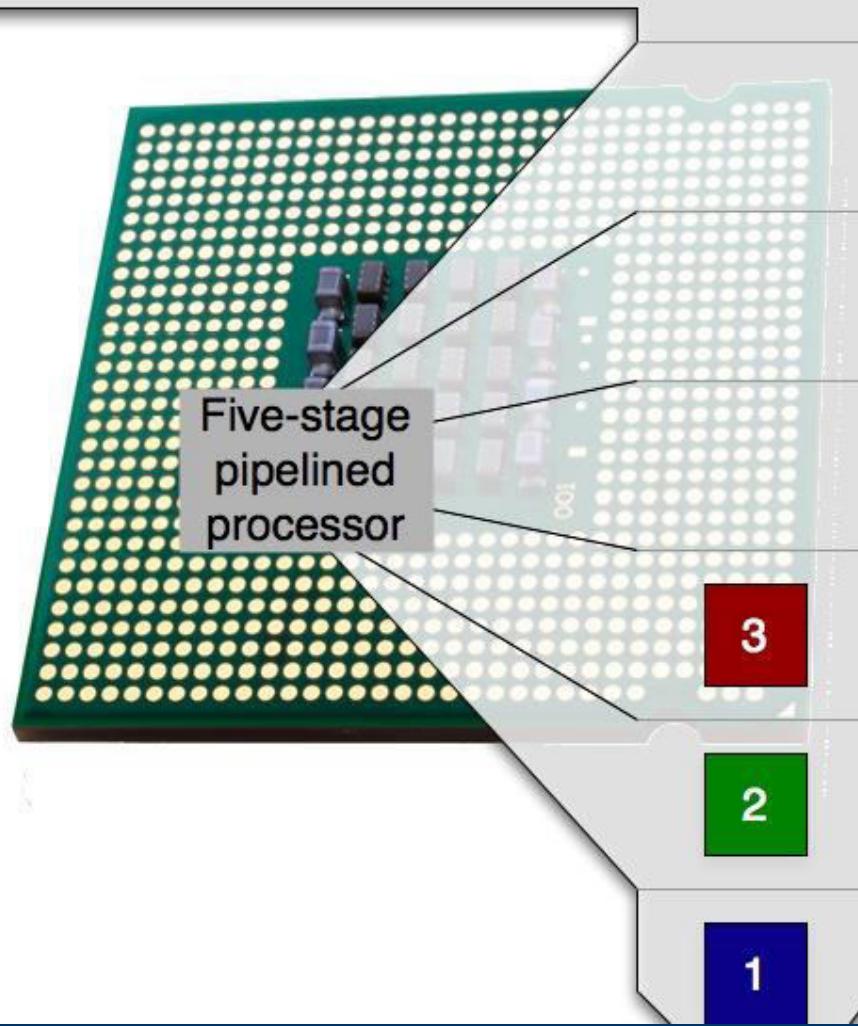
Decode

Operand read

Execute

Operand write

Instructions in Memory



Instruction read

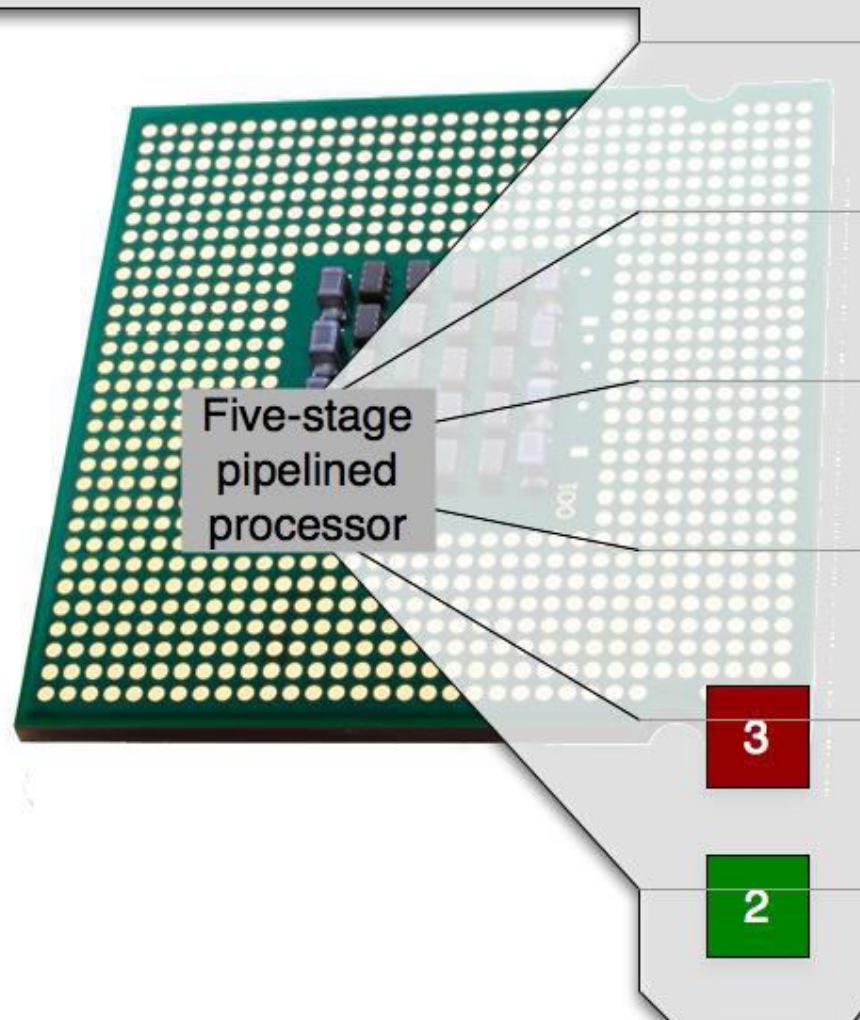
Decode

Operand read

Execute

Operand write

Instructions in Memory



Instruction read

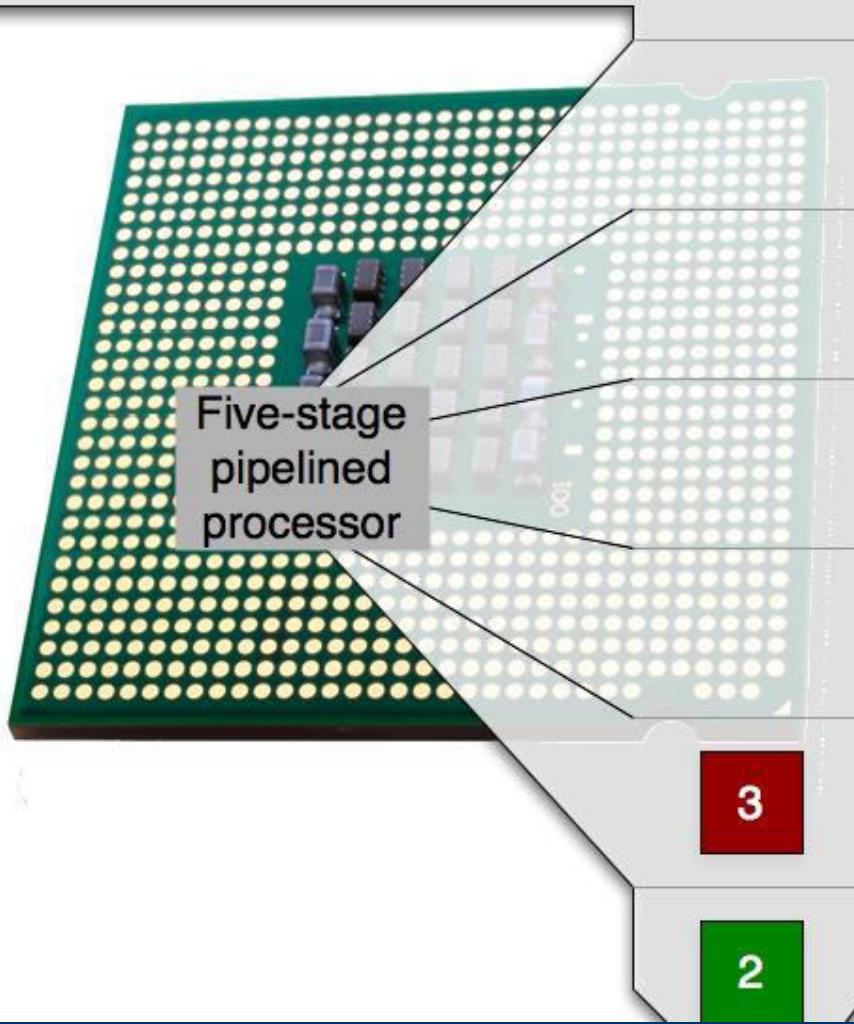
Decode

Operand read

Execute

Operand write

Instructions in Memory



Instruction read

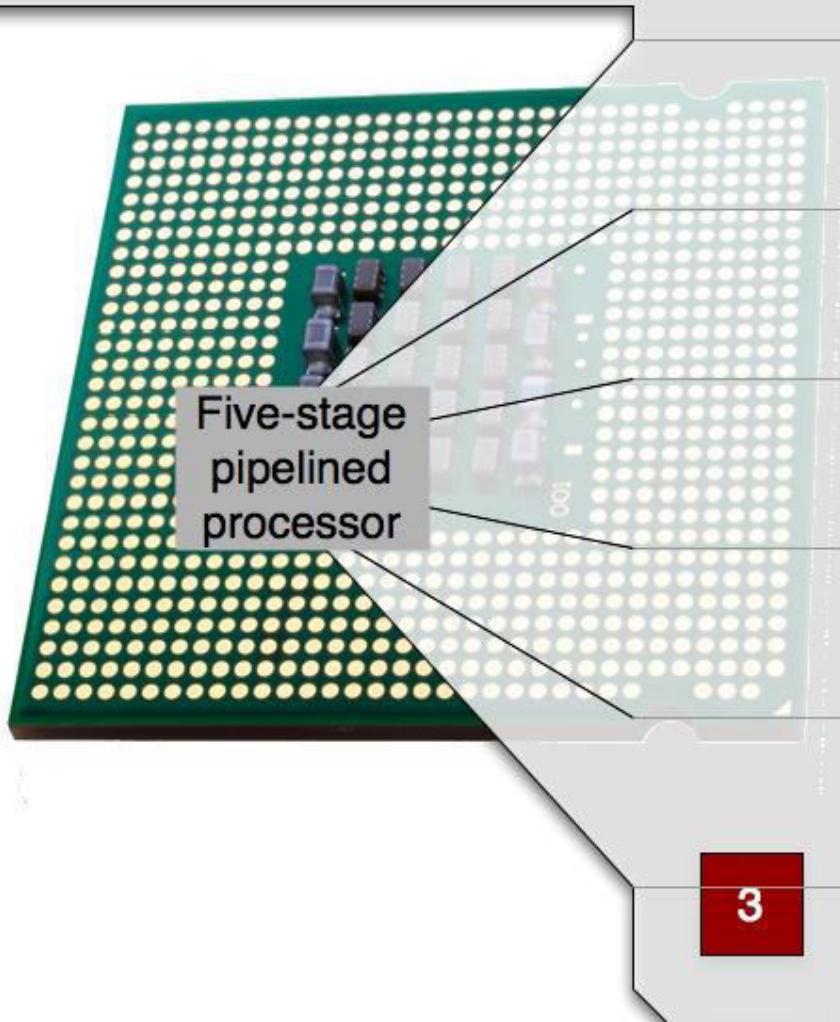
Decode

Operand read

Execute

Operand write

Instructions in Memory



Instruction read

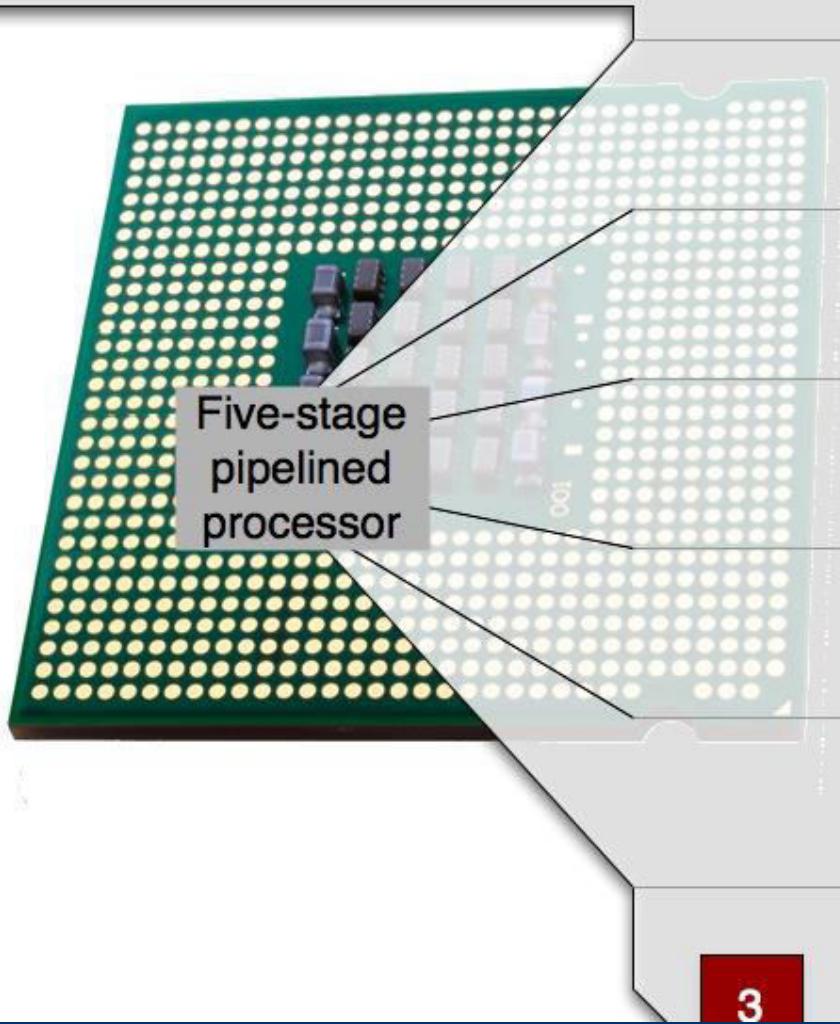
Decode

Operand read

Execute

Operand write

Instructions in Memory



Instruction read

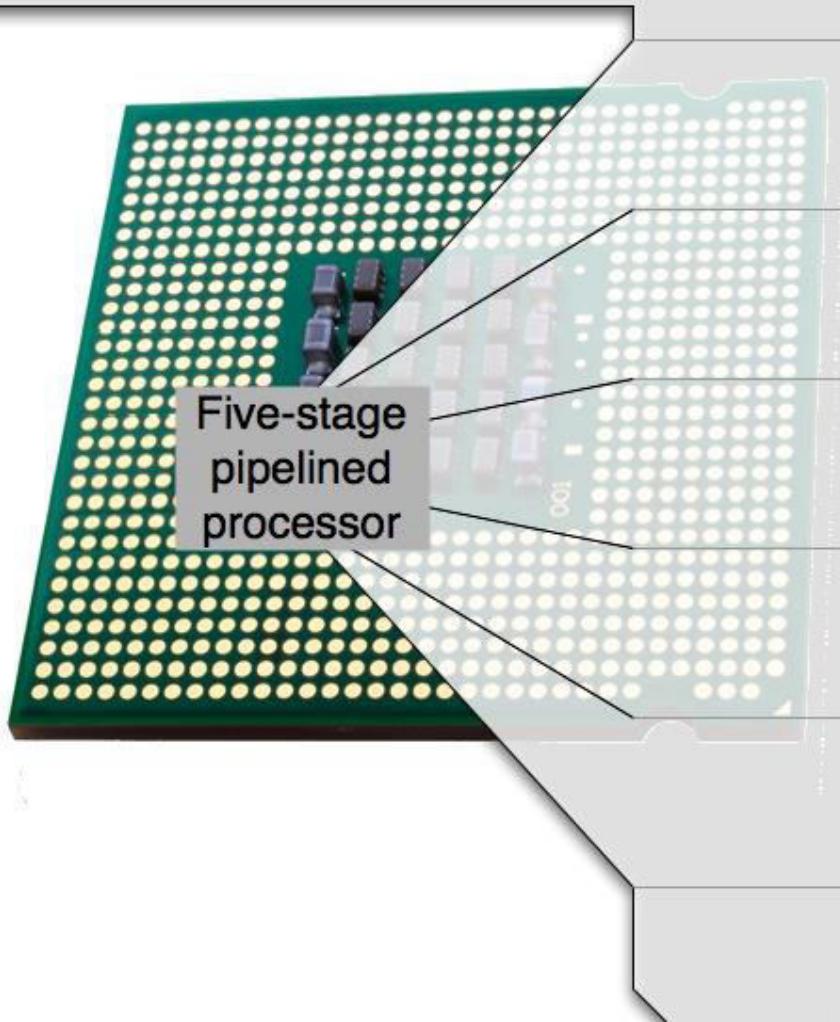
Decode

Operand read

Execute

Operand write

Instructions in Memory



Instruction read

Decode

Operand read

Execute

Operand write

What are the benefits of pipelining?

- Uses most (if not all) of a processor's ability at all times
Efficiency goes up
- Relatively minor hardware changes give more completed cycles per second
Speed goes up

What can go wrong in pipelines?

- Data hazards
- Control hazards
- Structural hazards

What can go wrong...?

Data hazards

- Two instructions at different stages
One instruction needs the results of
the other instruction's operation

...

```
add    %or1, 5, %or2
add    %or2, 7, %or3
```

...

What can go wrong...?

Data hazards

- Well-designed programs can anticipate this, and spread the connected commands out with other operations

...

add	$\%r1, 5, \%or2$	(i1 fetch)
sub	$\%or4, 7, \%r8$	(i1 decode)
mov	$\%r5, \%r6$	(i1 read)
add	$\%or2, 7, \%r3$	(i1 execute, i2 fetch)

What can go wrong...?

Data hazards

- Hardware can detect some data hazards and “forward” the information to the instructions that need it, even as the data goes to the registers
- Otherwise, programs may simply have bad data - testing and good programming are the best cures!

What can go wrong...?

Control hazards

- Instructions are read sequentially from memory - even though loops may return to previous statements
- Instructions following the loop are already in the pipeline!
- Or, simple if/else blocks of code. Which block do you get instructions from?

What can go wrong...?

Control hazards

- At worst, the entire pipeline must be flushed: hardware can *stall* the influx of new instructions until the conditional is evaluated

What can go wrong...?

Control hazards

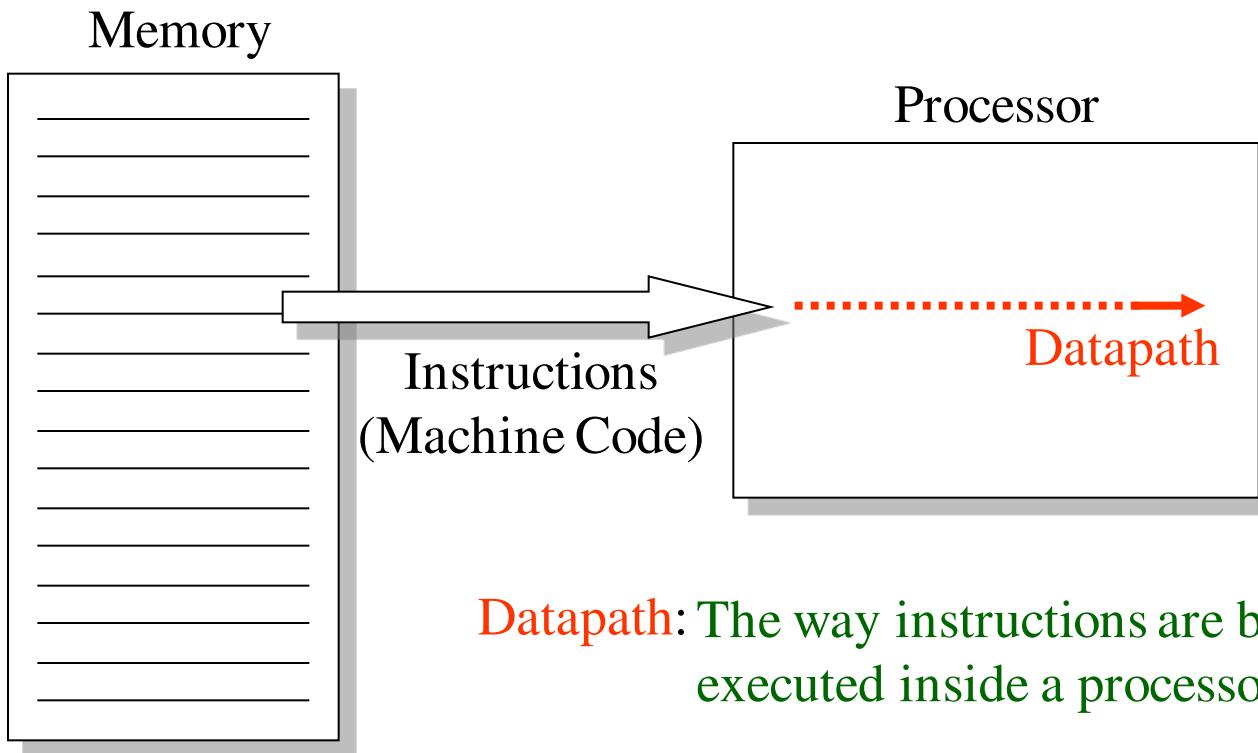
- *Branch prediction* inserts a couple instructions of its own to guess the outcome of the conditional
- If prediction is usually correct, fewer cycles lost on average
- If prediction is wrong, flush the pipeline and undo any improper commands

What can go wrong...?

Structure hazards

- Two different instructions try to use the same hardware resource
- One instruction will obviously get access first, second instruction must be stalled (maybe along with all other instructions behind it)
- Again, careful programming will prevent this

What is “datapath”?



Datapath: The way instructions are being executed inside a processor

Six different datapath architectures

(1) General-Purpose CISC/special-purpose RISC Processors:

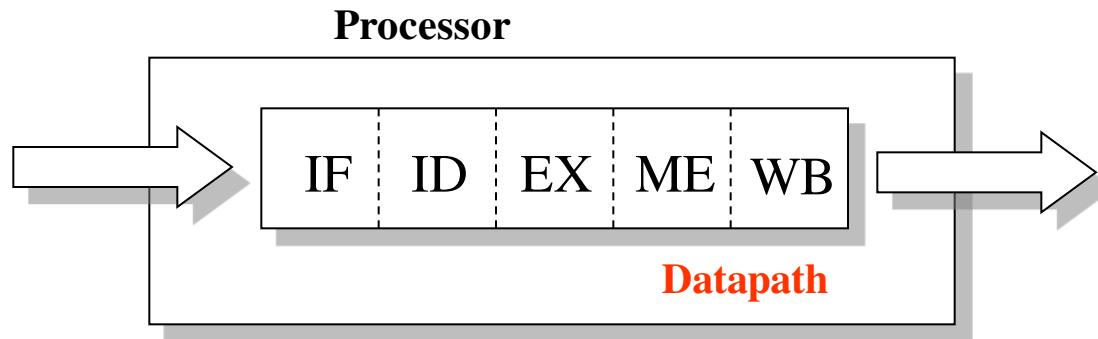
- ① Scalar Processors (i8086)
- ② Pipeline Processors (i80486)
- ③ Super-Scalar Processors
- ④ Super-Pipeline Processors (i80586 – P54)

(2) Super Computers/Mainframes:

- ⑤ Vector Processors
- ⑥ VLIW Processors

1. Scalar Datapath Processors

- The datapath includes the five circuit units
- All five units are implemented as single monolithic unit
- When an instruction is being executed, no other instruction can enter the datapath



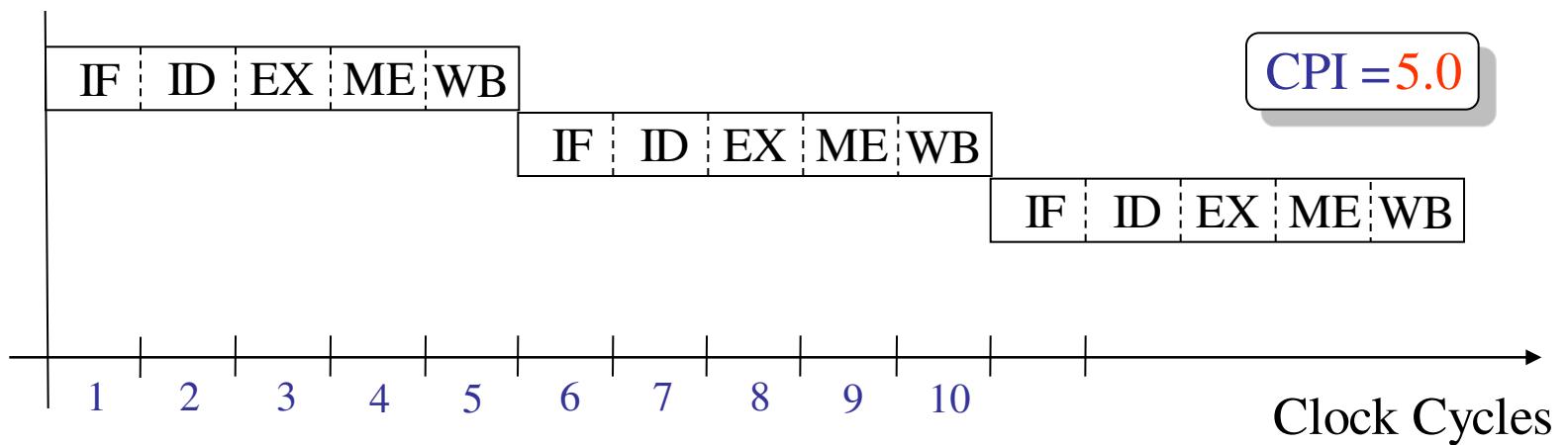
IF: Instruction Fetch **ID:** Instruction Decode **EX:** Execution

ME: Memory access **WB:** Write Back to registers

1. Scalar Datapath Processors (continued)

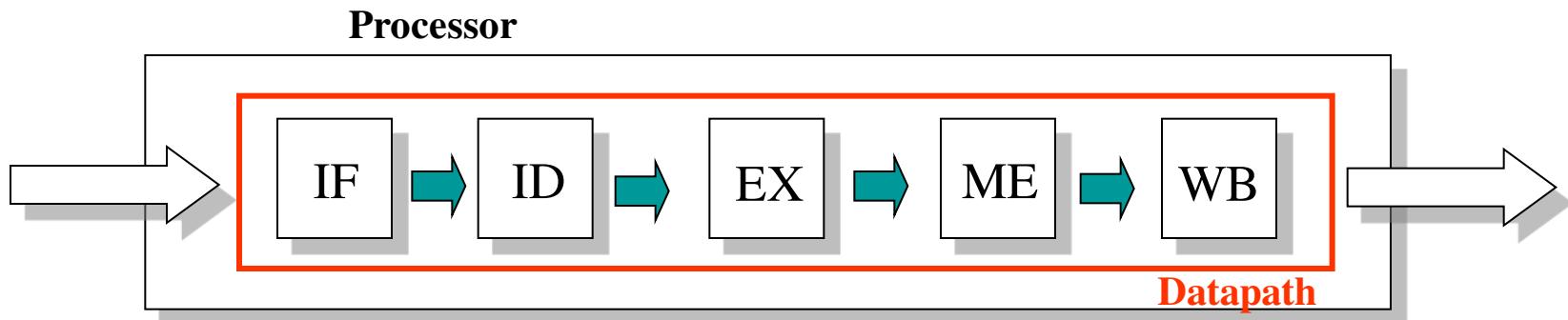
Representative processors in this generation

i4004, i8080, i8086, i80816, Z80, MC68000



2. Pipeline Datapath Processors

- All five units are implemented as independent units
- When an instruction is completed in a unit, the instruction can be forwarded to the next unit
- All five units can be occupied by different instructions

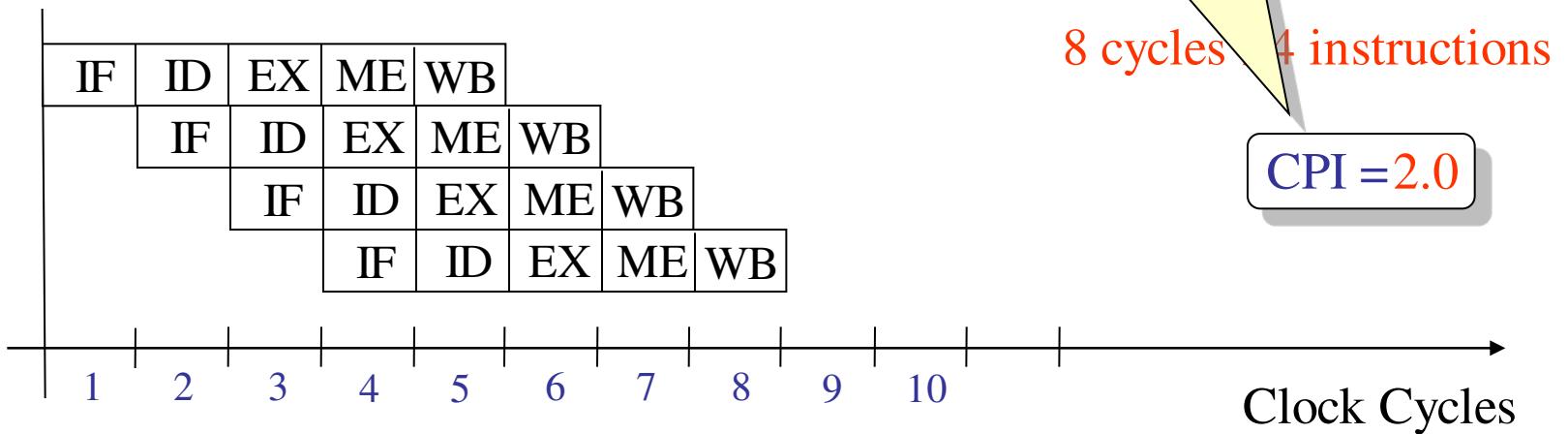


2. Pipeline Datapath Processors (continued)

Representative processors in this generation

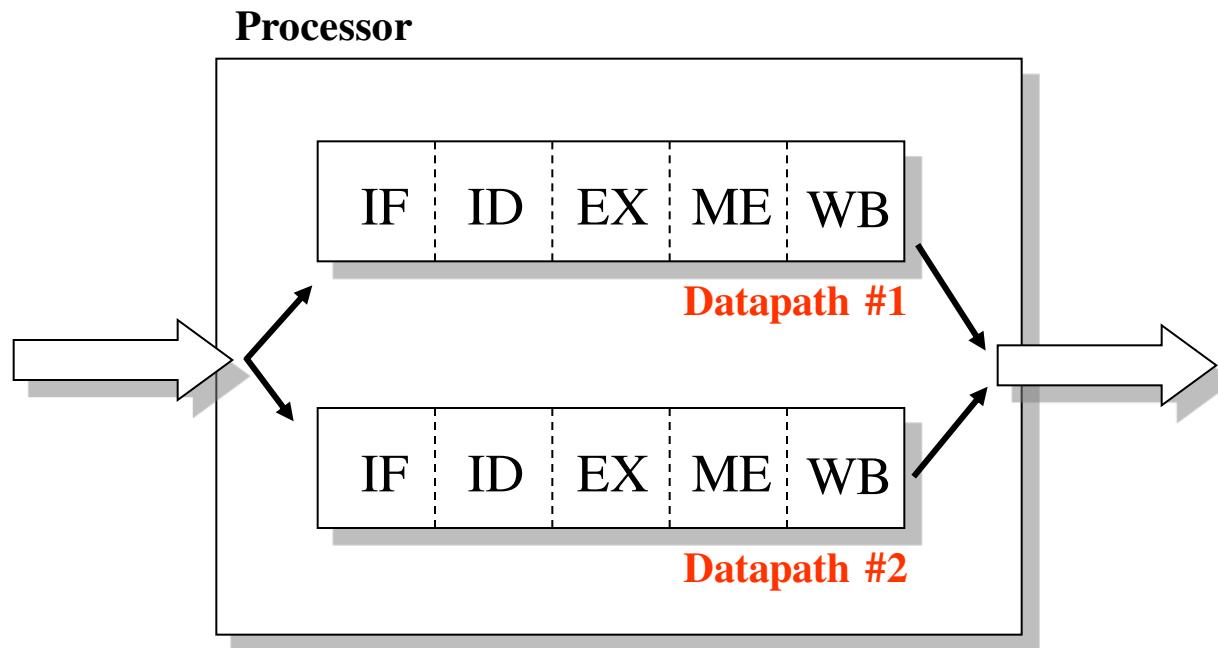
i80386, i40846, MC68040,

CPI for these only for
these four instructions

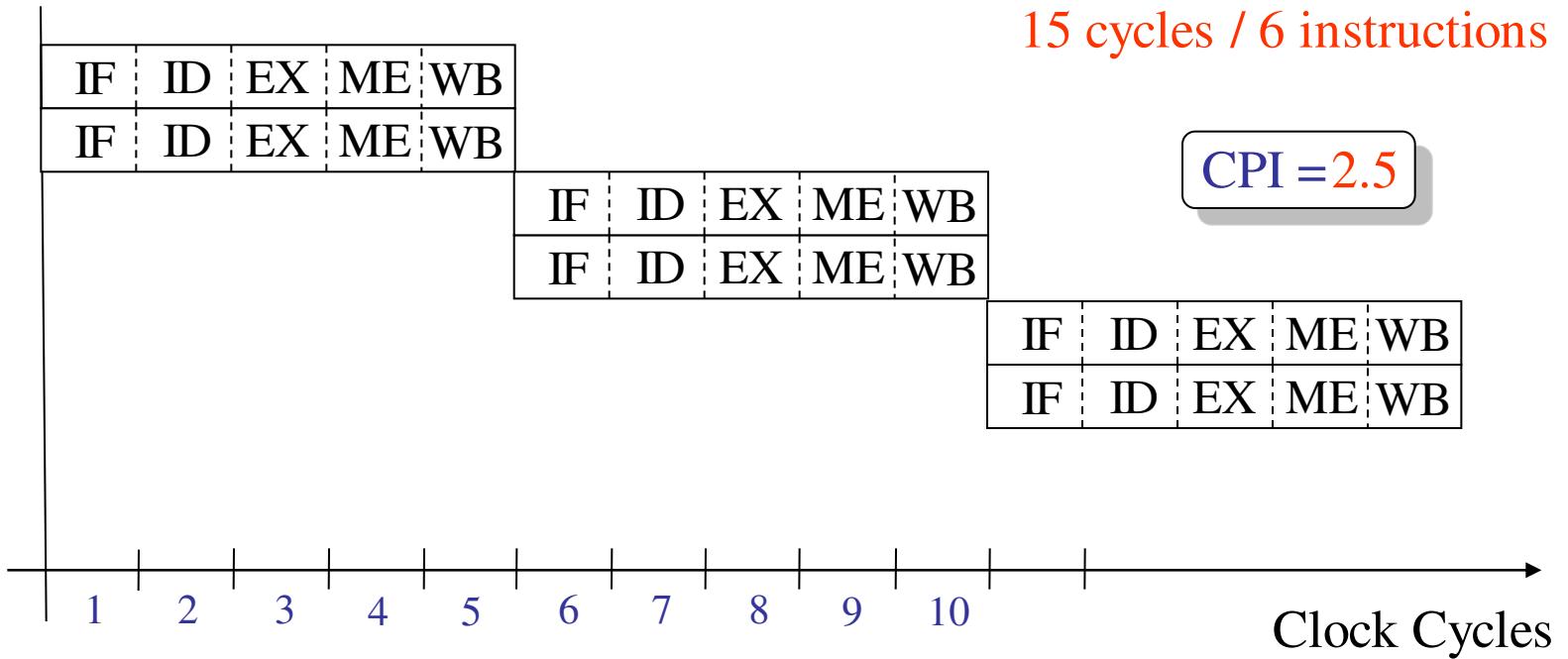


3. Super-Scalar Datapath Processors

- Multiple Scalar Datapath

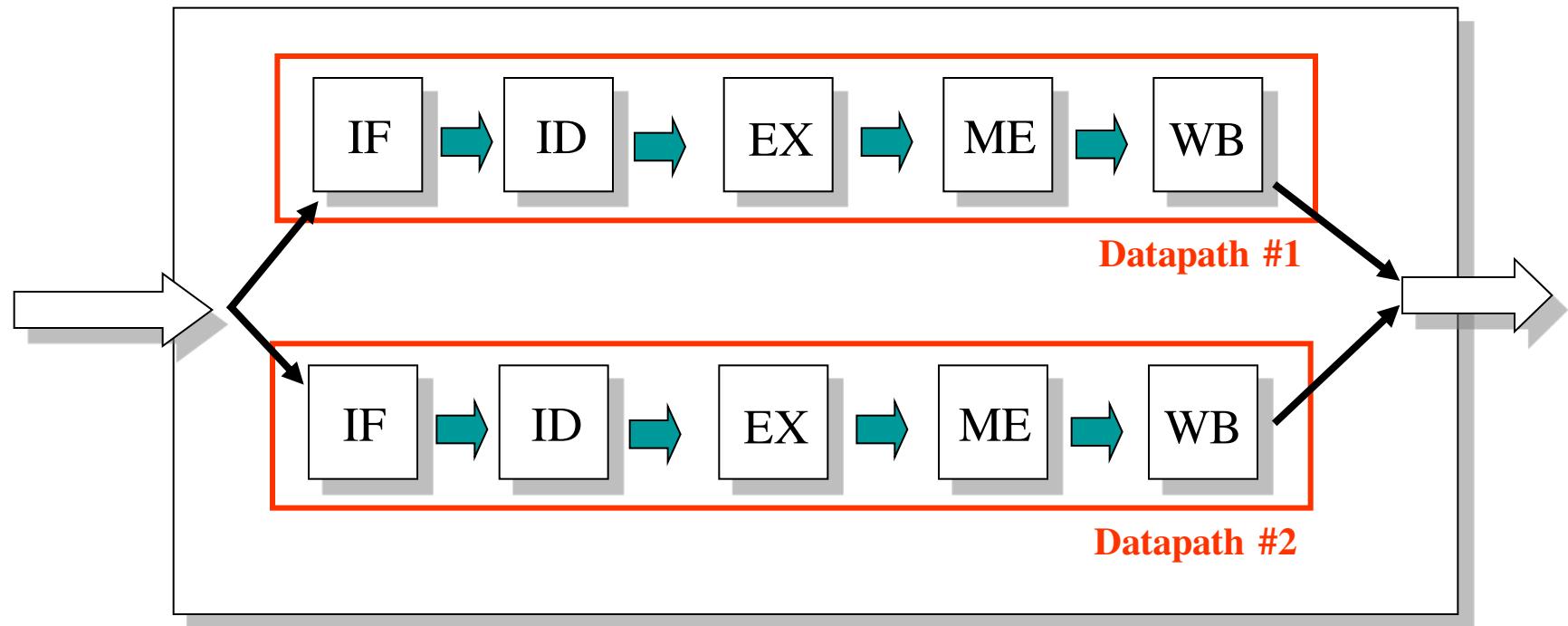


3. Super-Scalar Datapath Processors (continued)



4. Super-Pipeline Datapath Processors

- A combination of super-scalar and pipeline Processor



4. Super-Pipeline Datapath Processors (continued)

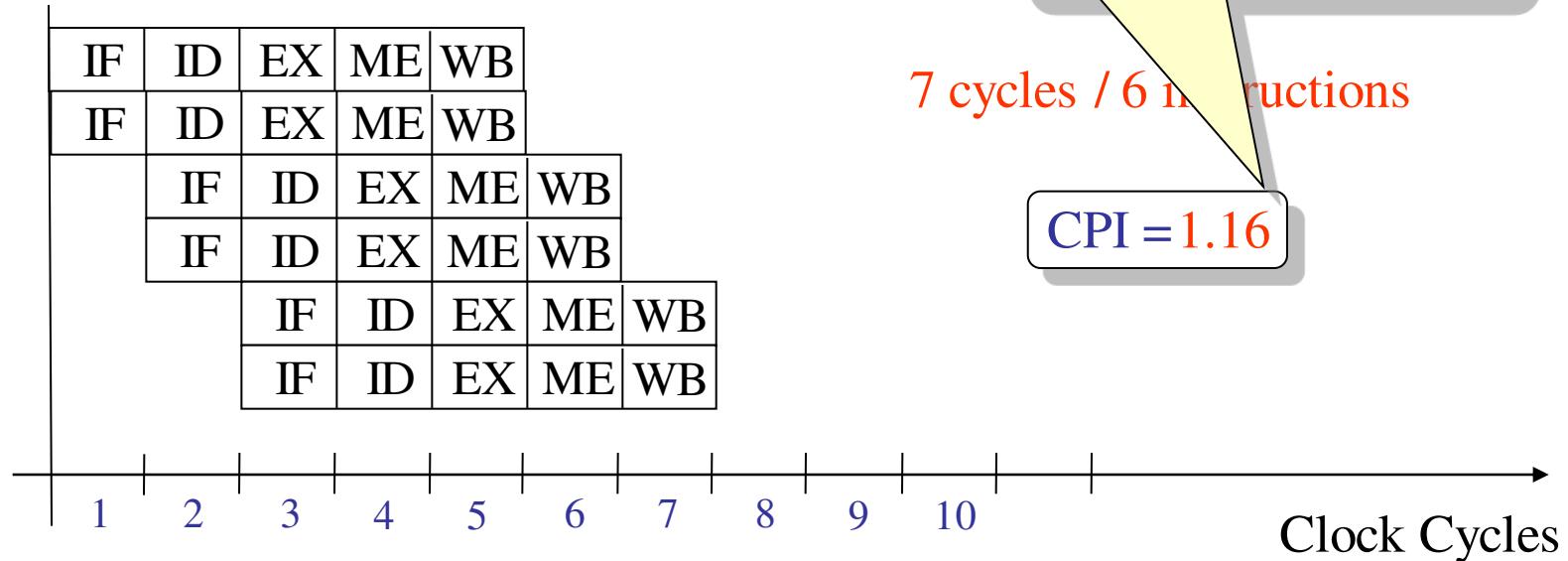
Representative processors in this generation

Pentiums (54, P55C), MIPS

CPI for these only for
these six instructions

7 cycles / 6 instructions

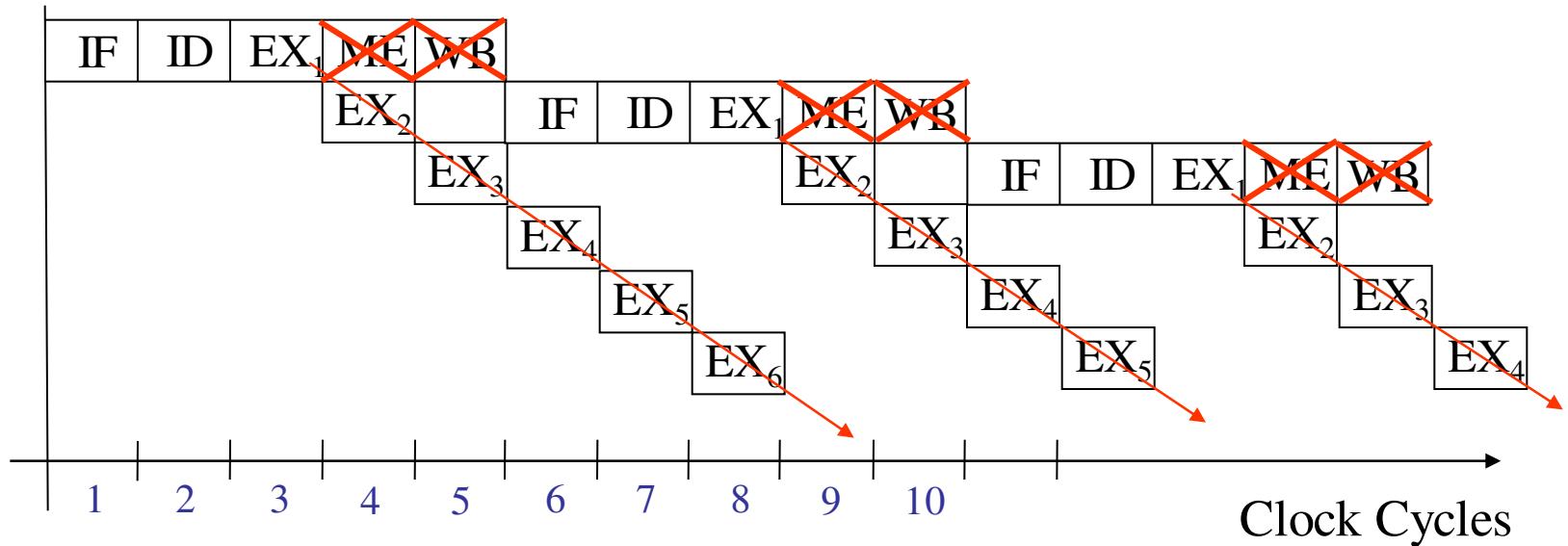
$$\text{CPI} = 1.16$$



5. Vector Datapath Processors (continued)

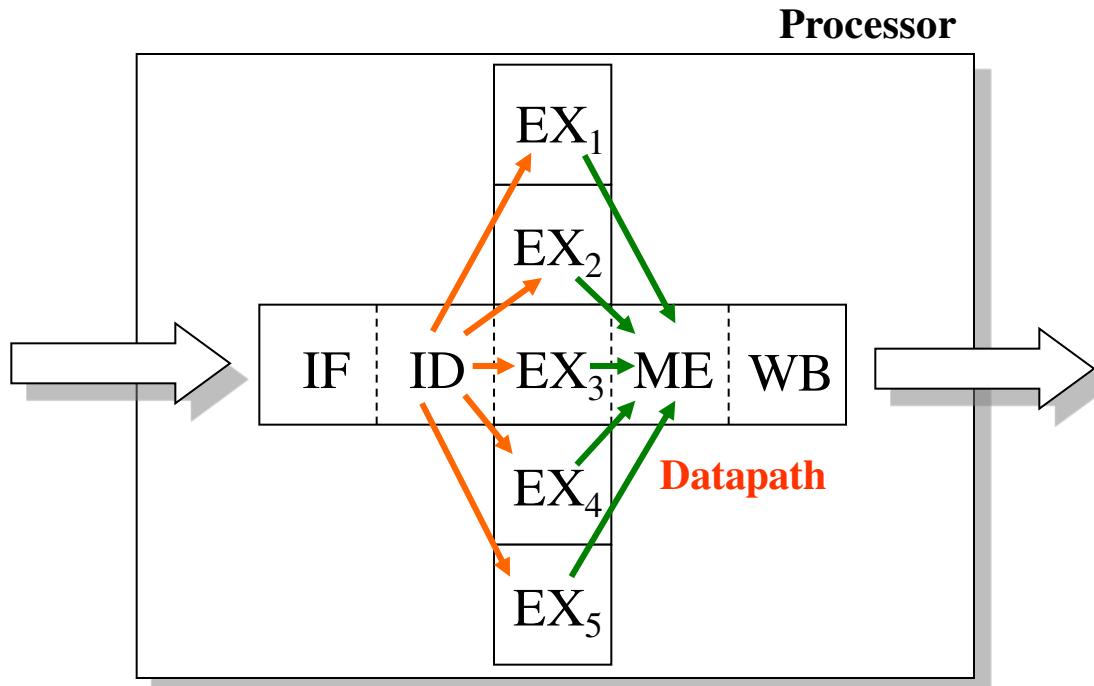
Representative processors in this generation

C-90 and Y-MP (Cray), VAX 9000 (Digital), ...



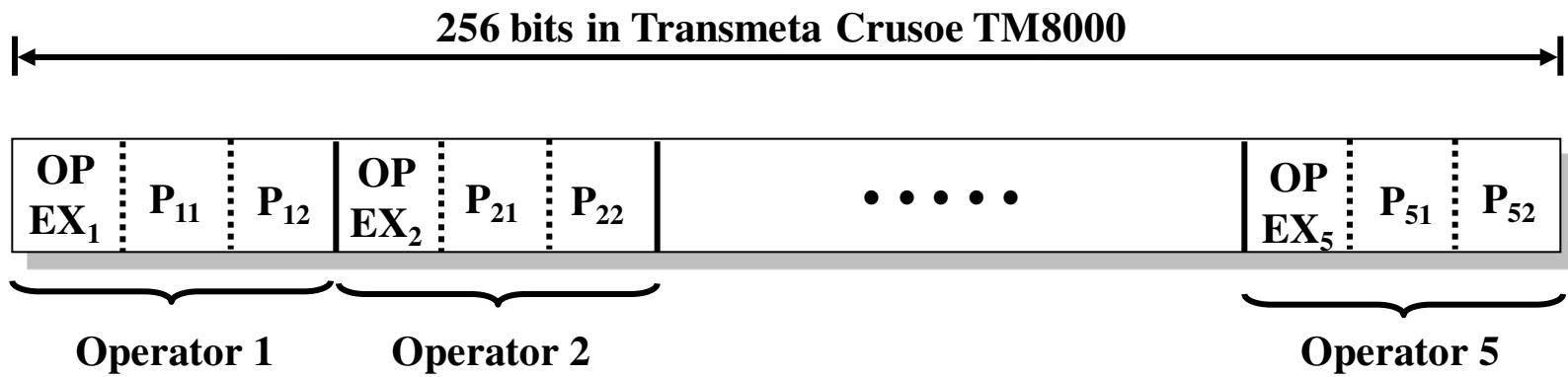
6. VLIW Datapath Processors (continued)

- An extension of scalar datapath architecture
- Multiple execution units



6. VLIW Datapath Processors (continued)

- Each instruction has to have multiple operations in it

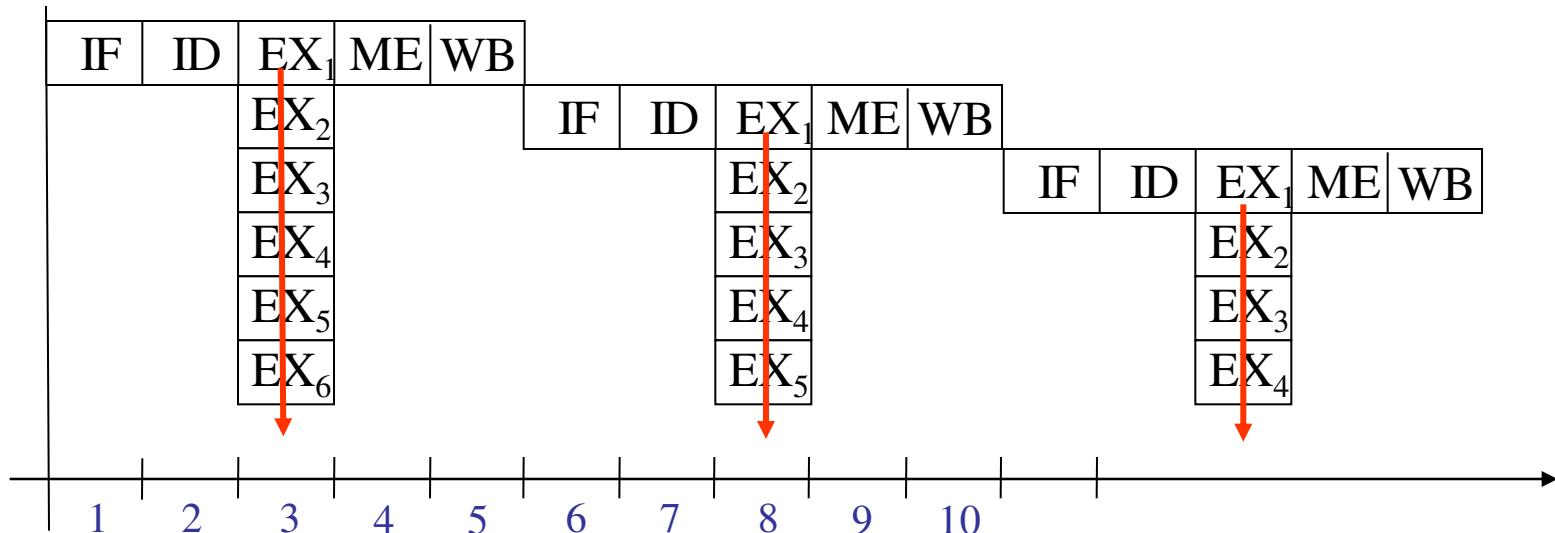


- Each operator corresponds to an instruction in scalar machine

6. VLIW Datapath Processors (continued)

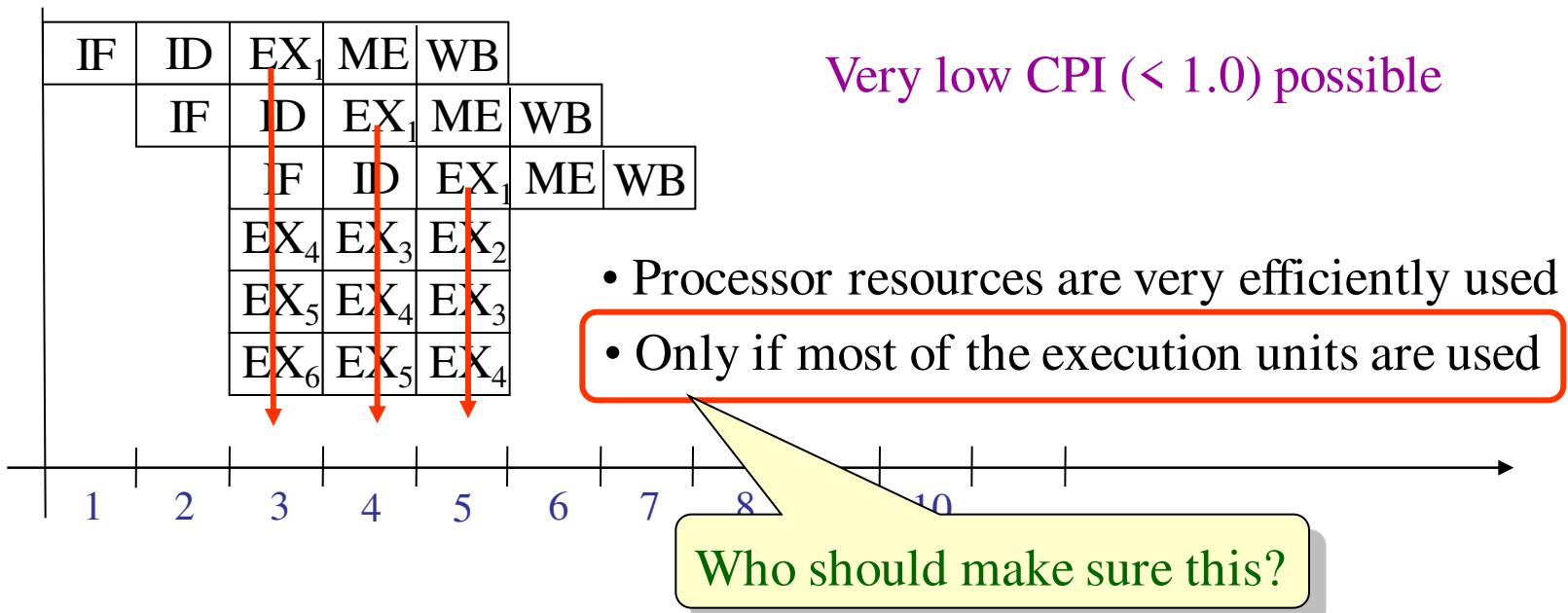
Representative processors in this generation

TI TMS320C6200, Philips TM1000, Transmeta Crusoe, ...



6. VLIW Datapath Processors (continued)

VLIW can be pipelined

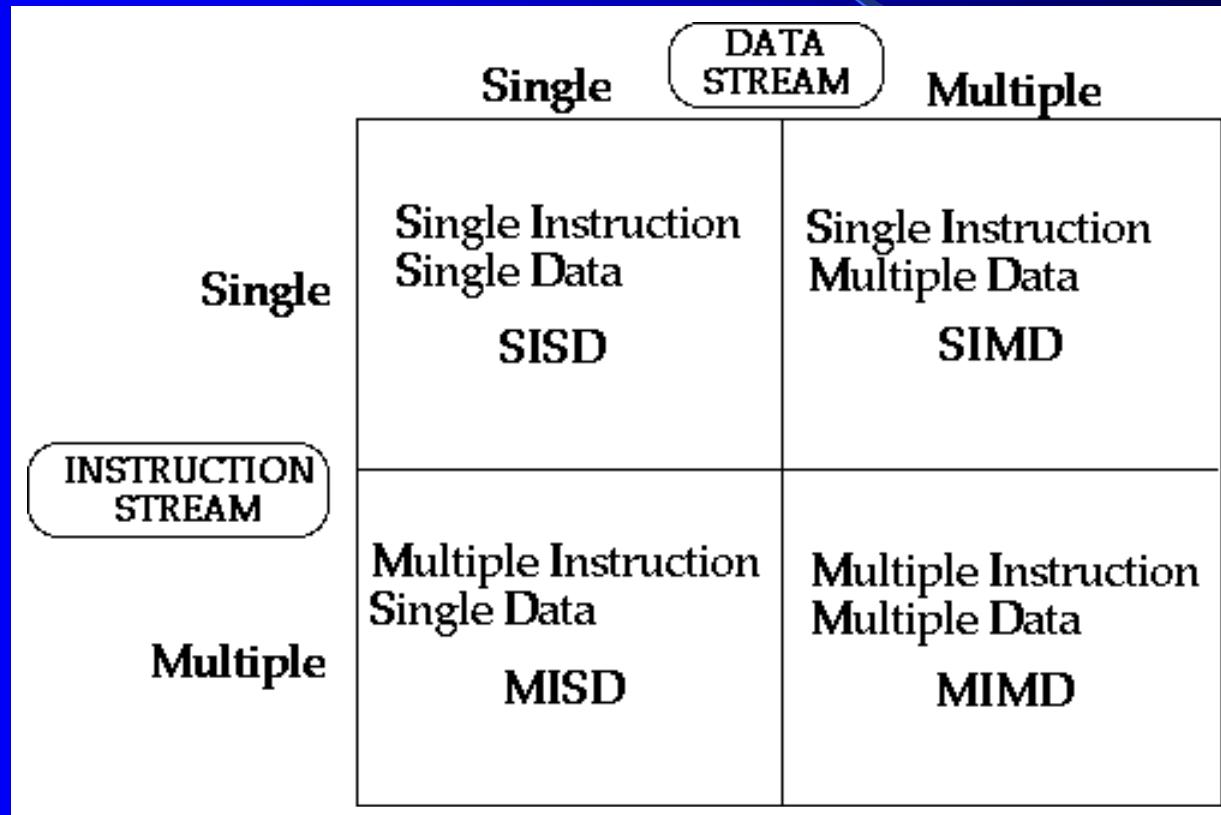


Organization of Multiprocessor Systems

- Flynn's Classification
 - Was proposed by researcher Michael J. Flynn in 1966.
 - It is the most commonly accepted taxonomy of computer organization.
 - In this classification, computers are classified by whether it processes a single instruction at a time or multiple instructions simultaneously, and whether it operates on one or multiple data sets.

Taxonomy of Computer Architectures

Simple Diagrammatic Representation



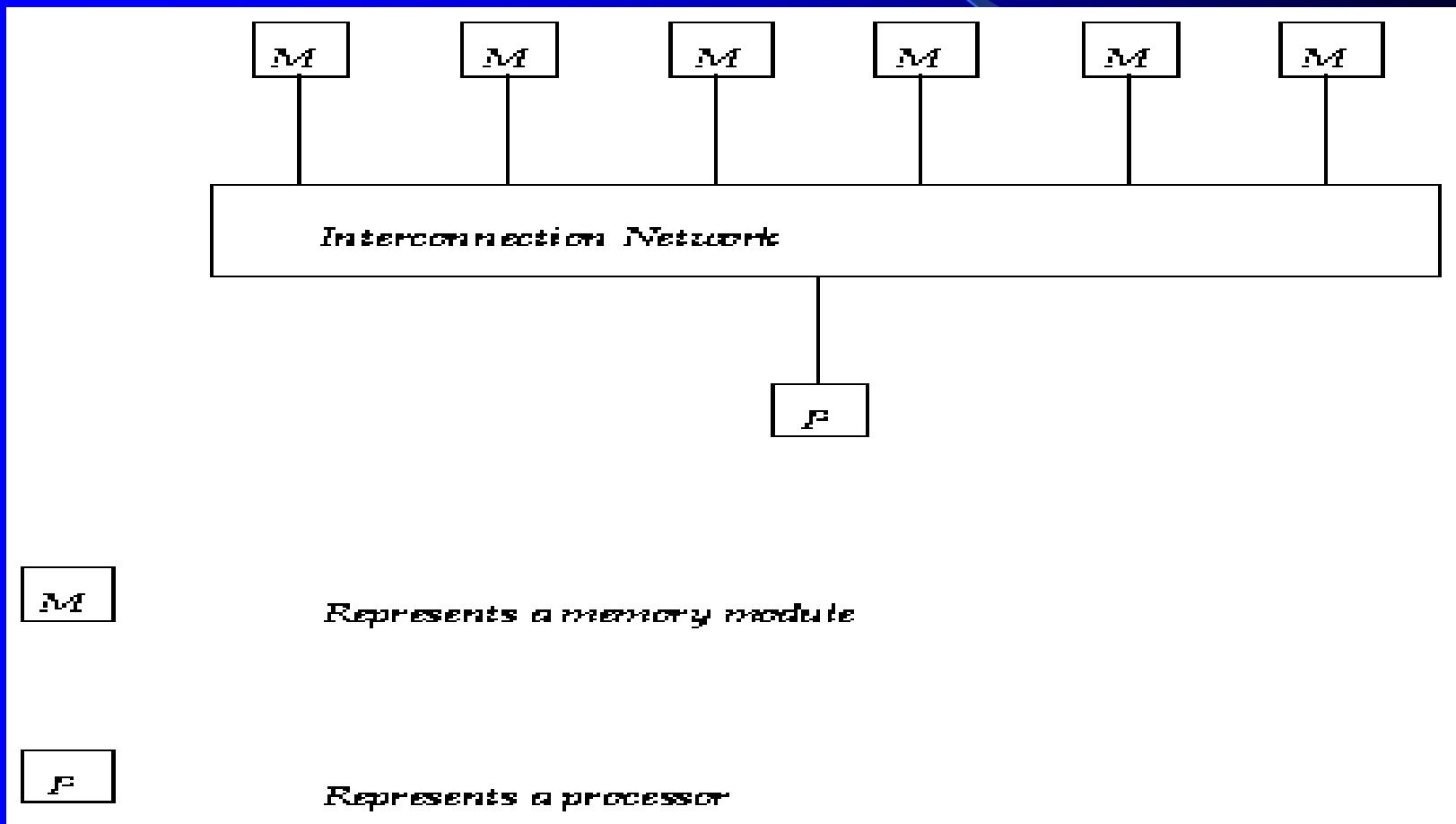
4 categories of Flynn's classification of multiprocessor systems by their instruction and data streams

Single Instruction, Single Data (SISD)

- SISD machines execute a single instruction on individual data values using a single processor.
- Based on traditional Von Neumann uniprocessor architecture, instructions are executed sequentially or serially, one step after the next.
- Until most recently, most computers are of SISD type.

SISD

Simple Diagrammatic Representation

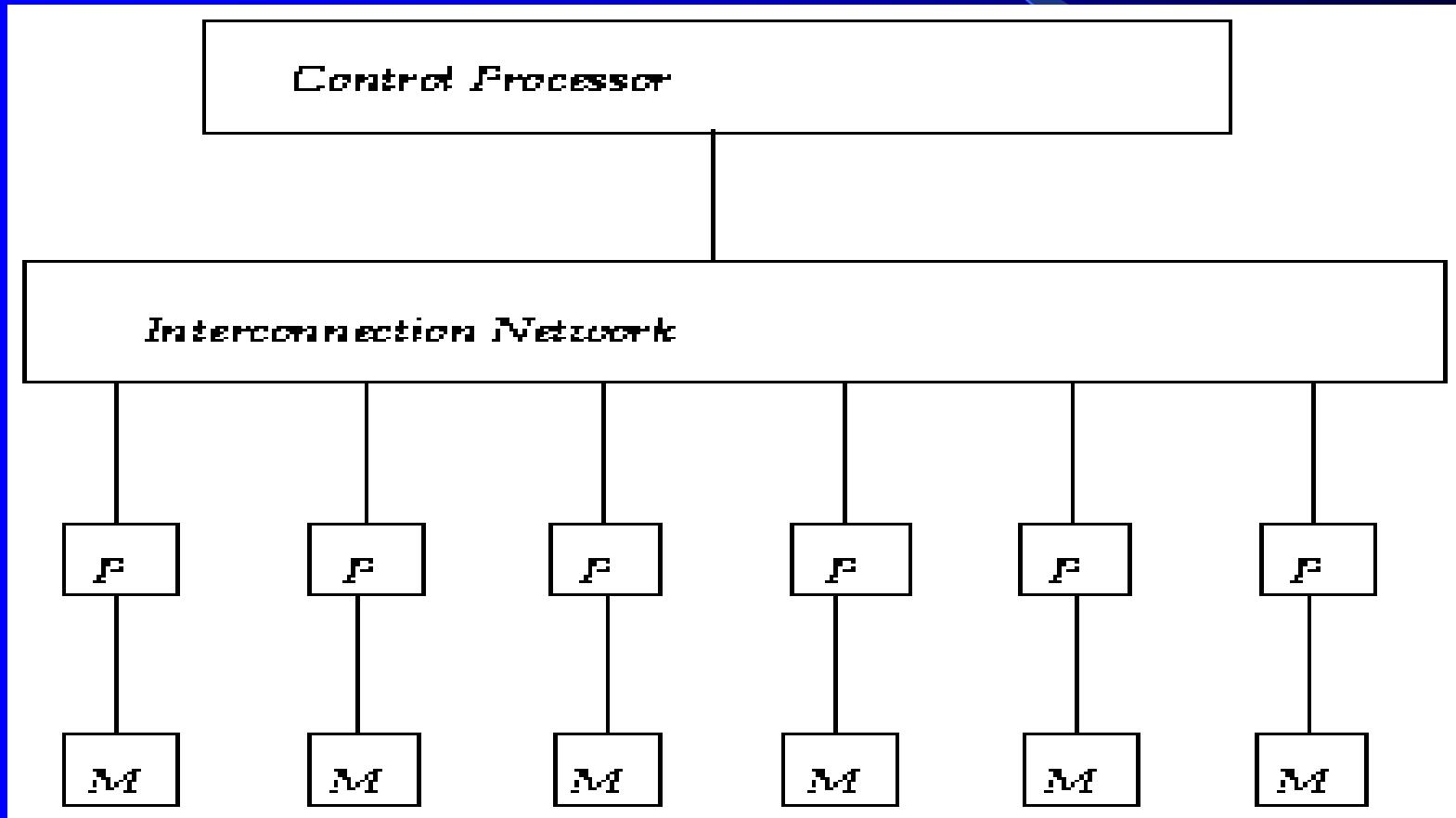


Single Instruction, Multiple Data (SIMD)

- An SIMD machine executes a single instruction on multiple data values simultaneously using many processors.
- Since there is only one instruction, each processor does not have to fetch and decode each instruction. Instead, a single control unit does the fetch and decoding for all processors.
- SIMD architectures include array processors.

SIMD

Simple Diagrammatic Representation

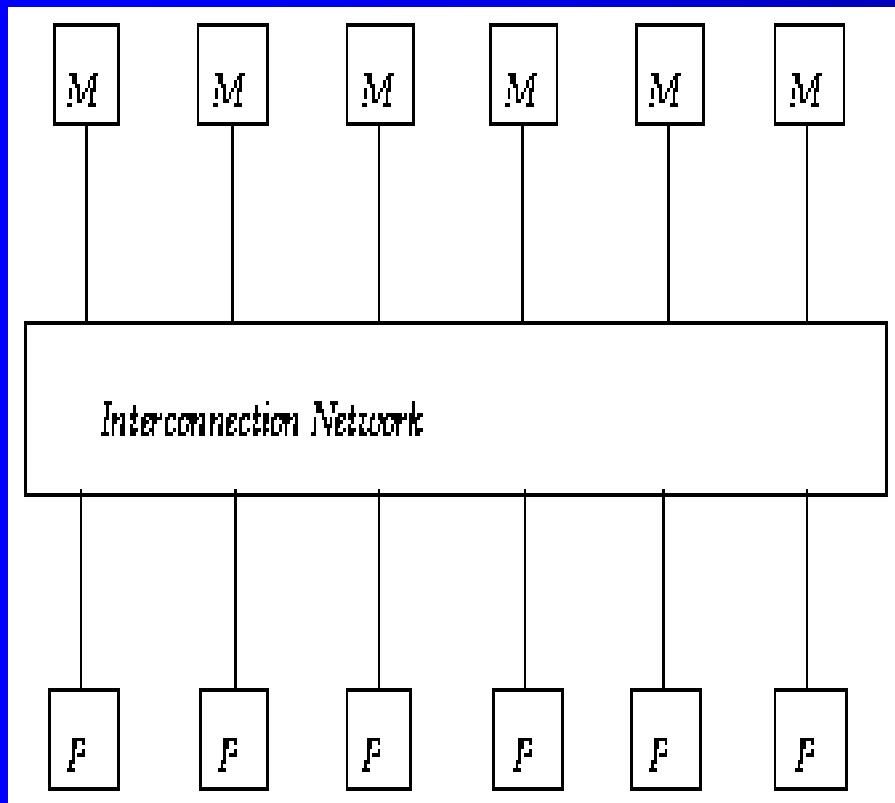


Multiple Instruction, Multiple Data (MIMD)

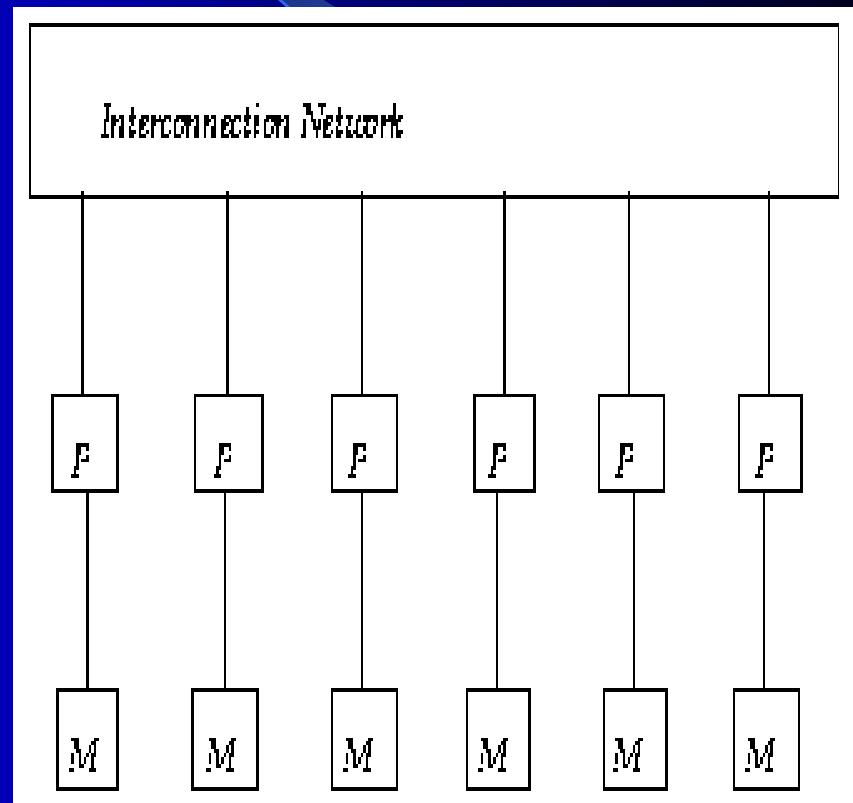
- MIMD machines are usually referred to as multiprocessors or multicomputers.
- It may execute multiple instructions simultaneously, contrary to SIMD machines.
- Each processor must include its own control unit that will assign to the processors parts of a task or a separate task.
- It has two subclasses: Shared memory and distributed memory

MIMD

Simple Diagrammatic Representation
(Shared Memory)



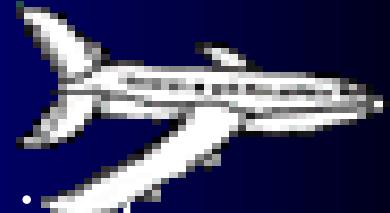
Simple Diagrammatic Representation(DistributedMemory)



Multiple Instruction, Single Data (MISD)

- This category does not actually exist. This category was included in the taxonomy for the sake of completeness.

Analogy of Flynn's Classifications



- An analogy of Flynn's classification is the check-in desk at an airport
 - SISD: a single desk
 - SIMD: many desks and a supervisor with a megaphone giving instructions that every desk obeys
 - MIMD: many desks working at their own pace, synchronized through a central database

System Topologies

Topologies

- A system may also be classified by its topology.
- A topology is the pattern of connections between processors.
- The cost-performance tradeoff determines which topologies to use for a multiprocessor system.

Topology Classification

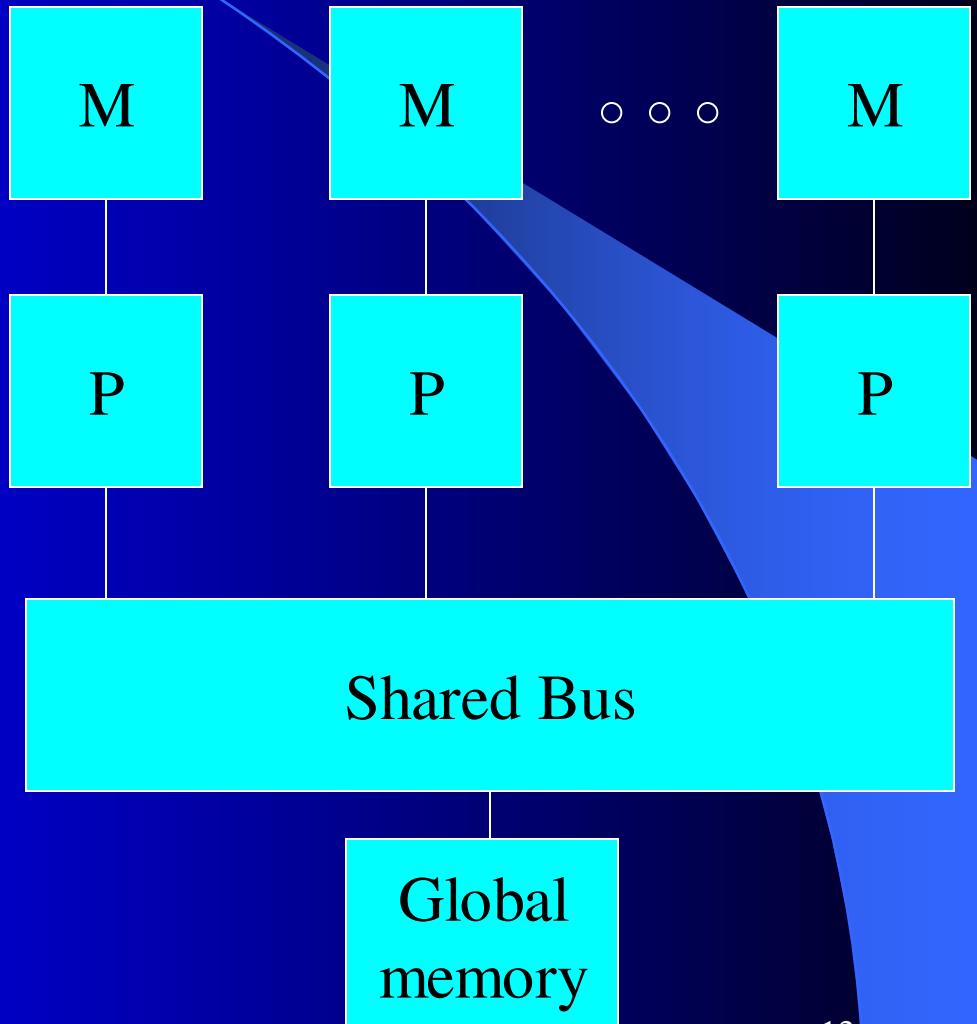
- A topology is characterized by its diameter, total bandwidth, and bisection bandwidth
 - Diameter – the maximum distance between two processors in the computer system.
 - Total bandwidth – the capacity of a communications link multiplied by the number of such links in the system.
 - Bisection bandwidth – represents the maximum data transfer that could occur at the bottleneck in the topology.



System Topologies

- Shared Bus Topology

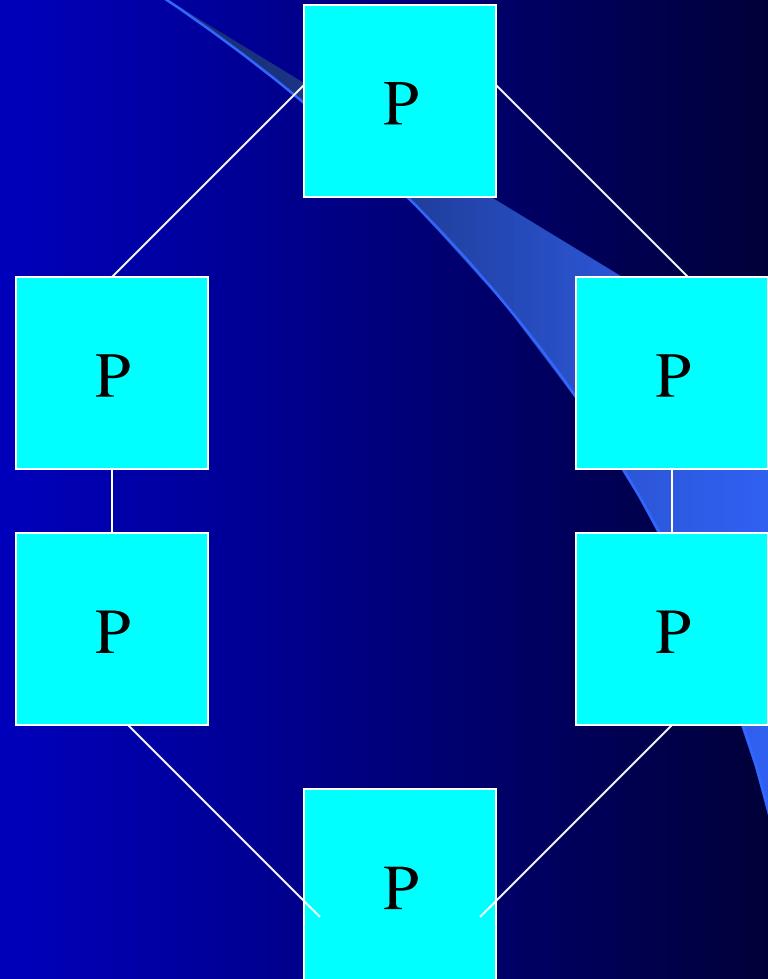
- Processors communicate with each other via a single bus that can only handle one data transmissions at a time.
- In most shared buses, processors directly communicate with their own local memory.



System Topologies

● Ring Topology

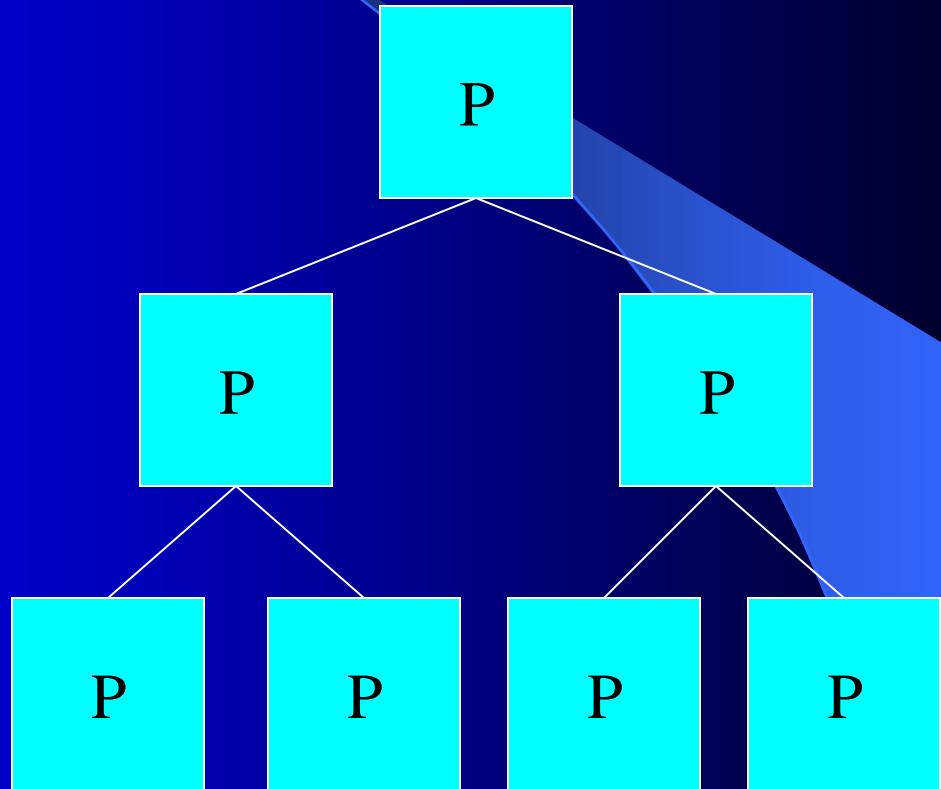
- Uses direct connections between processors instead of a shared bus.
- Allows communication links to be active simultaneously but data may have to travel through several processors to reach its destination.



System Topologies

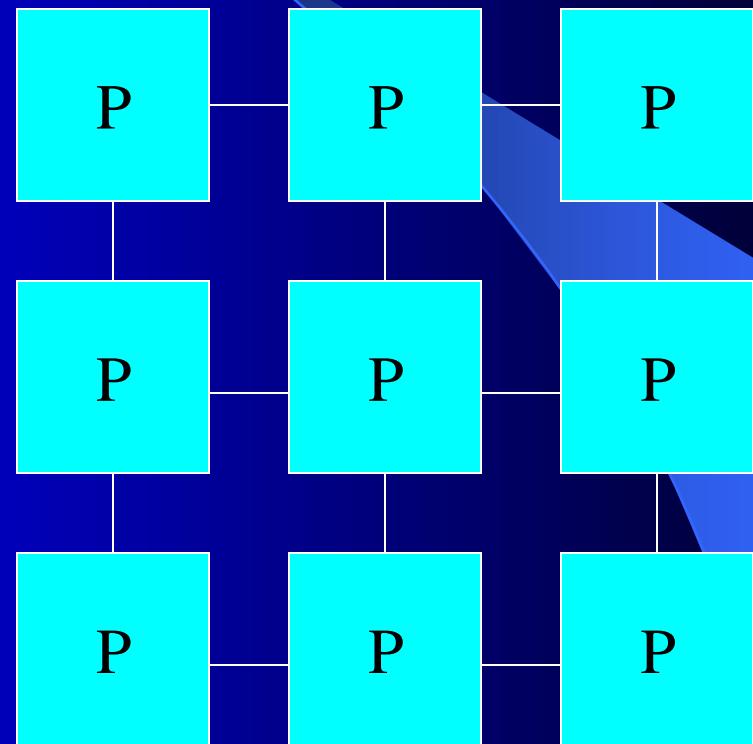
- Tree Topology

- Uses direct connections between processors; each having three connections.
- There is only one unique path between any pair of processors.



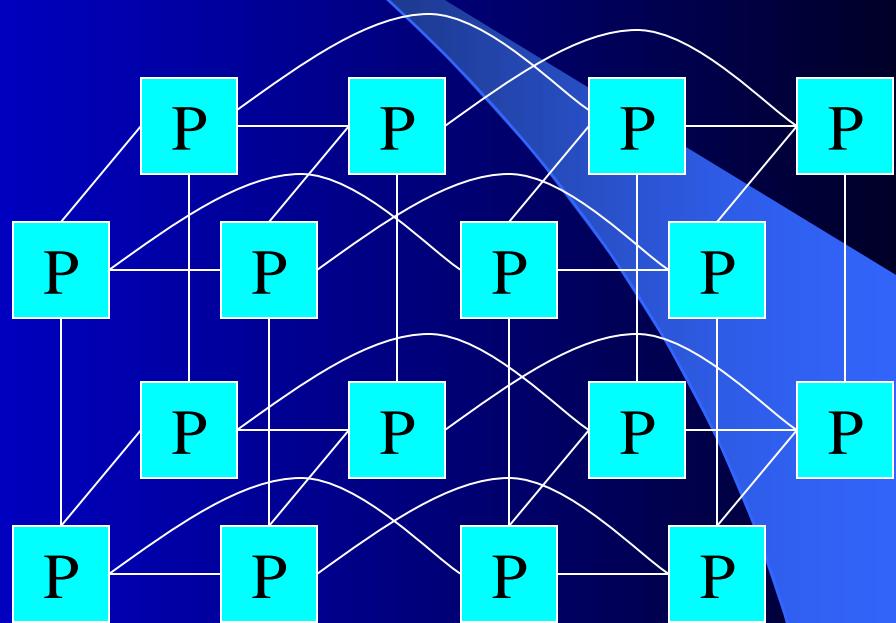
Systems Topologies

- Mesh Topology
 - In the mesh topology, every processor connects to the processors above and below it, and to its right and left.



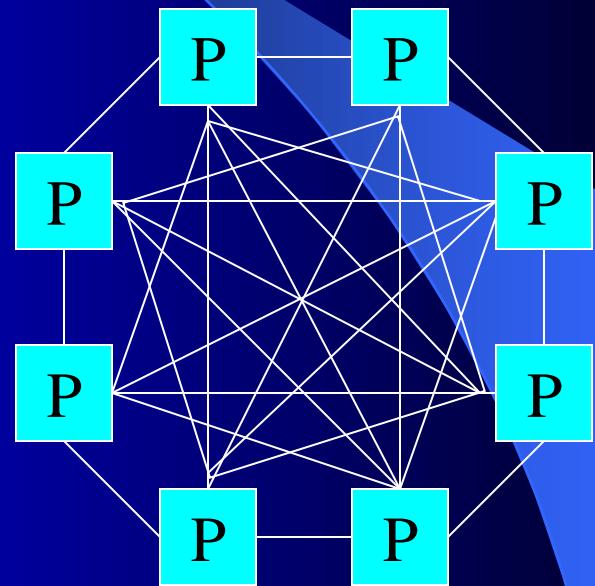
System Topologies

- Hypercube Topology
 - Is a multiple mesh topology.
 - Each processor connects to all other processors whose binary values differ by one bit. For example, processor 0(0000) connects to 1(0001) or 2(0010).



System Topologies

- Completely Connected Topology
 - Every processor has $n-1$ connections, one to each of the other processors.
 - There is an increase in complexity as the system grows but this offers maximum communication capabilities.



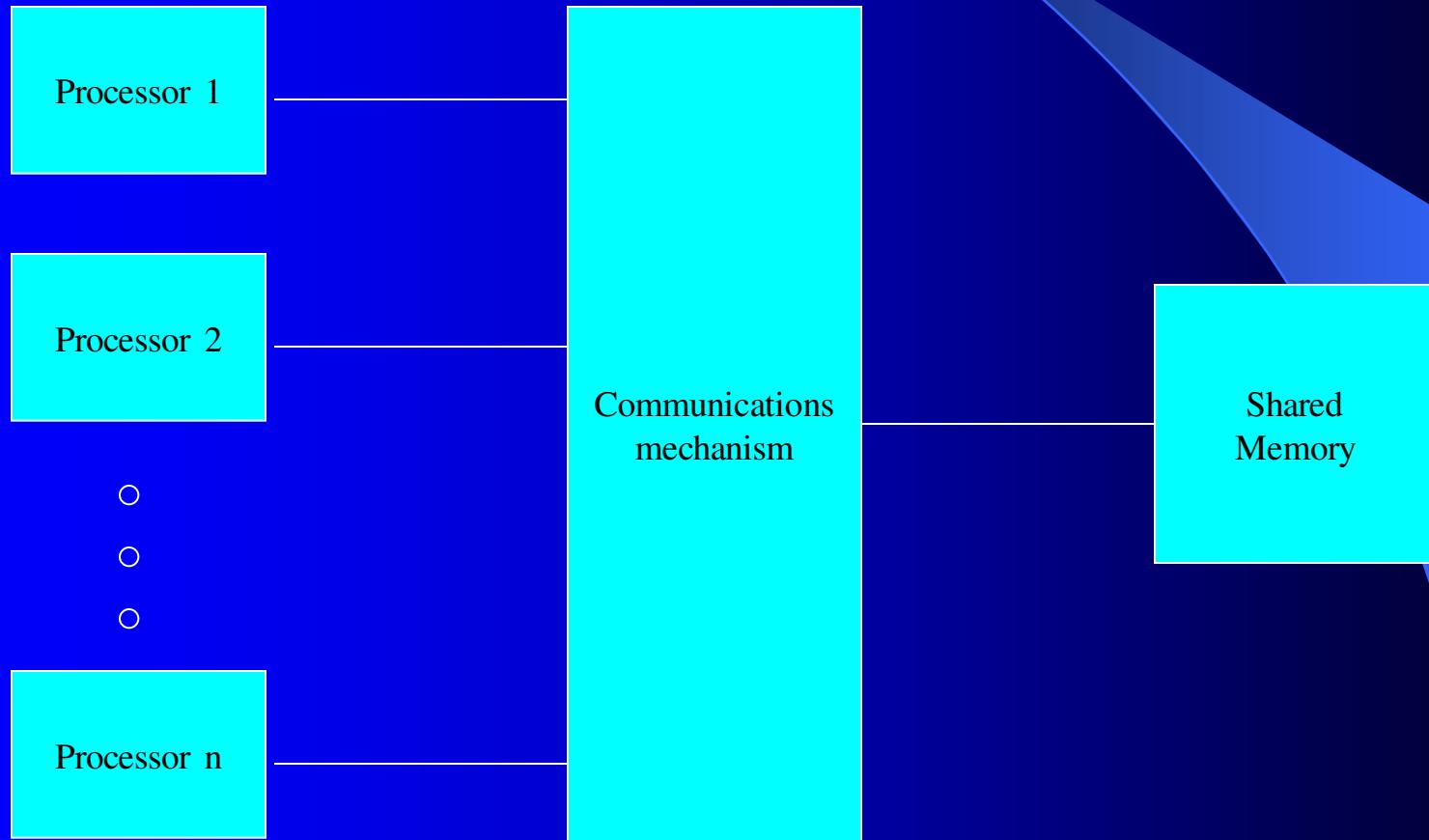
MIMD System Architectures

- Finally, the architecture of a MIMD system, contrast to its topology, refers to its connections to its system memory.
- A systems may also be classified by their architectures. Two of these are:
 - Uniform memory access (UMA)
 - Nonuniform memory access (NUMA)

Uniform memory access (UMA)

- The UMA is a type of symmetric multiprocessor, or SMP, that has two or more processors that perform symmetric functions. UMA gives all CPUs equal (uniform) access to all memory locations in shared memory. They interact with shared memory by some communications mechanism like a simple bus or a complex multistage interconnection network.

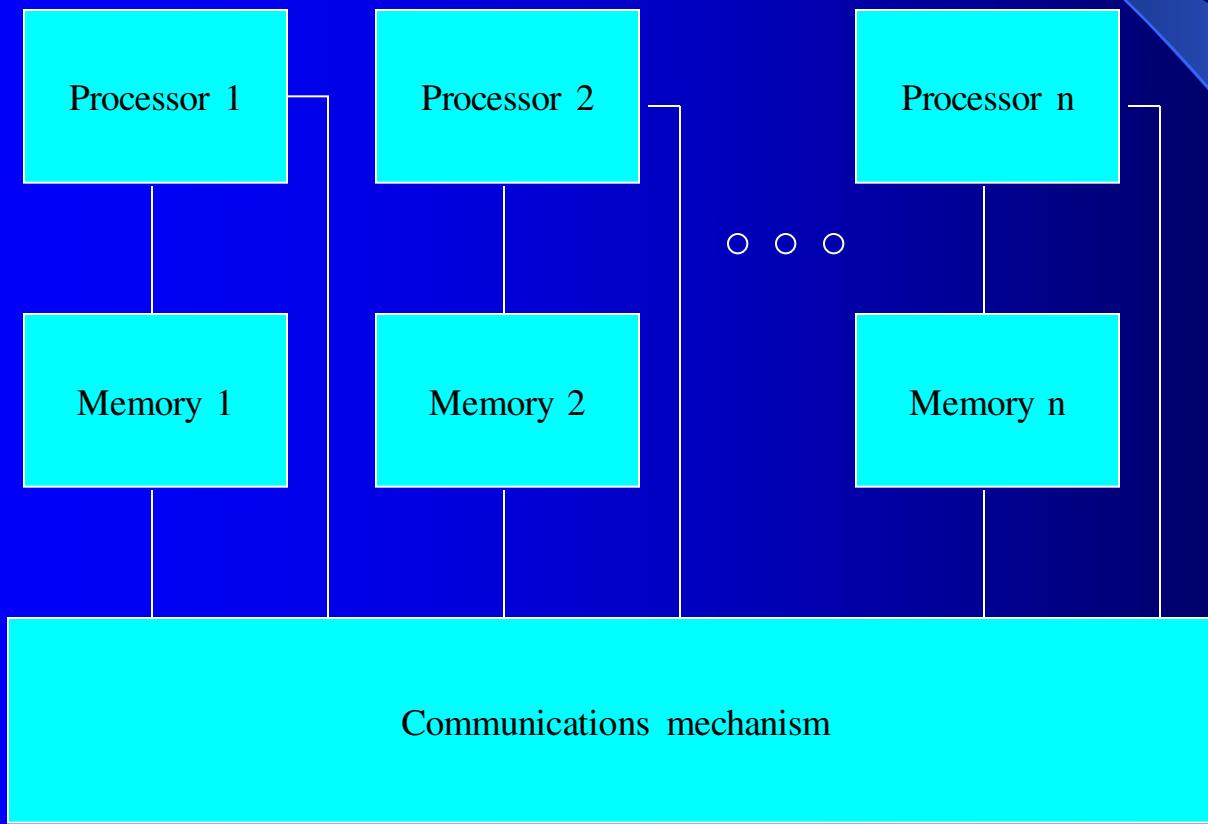
Uniform memory access (UMA) Architecture



Nonuniform memory access (NUMA)

- NUMA architectures, unlike UMA architectures do not allow uniform access to all shared memory locations. This architecture still allows all processors to access all shared memory locations but in a nonuniform way, each processor can access its local shared memory more quickly than the other memory modules not next to it.

Nonuniform memory access (NUMA) Architecture

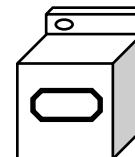
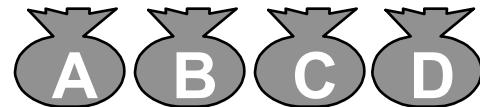


What is Pipelining?

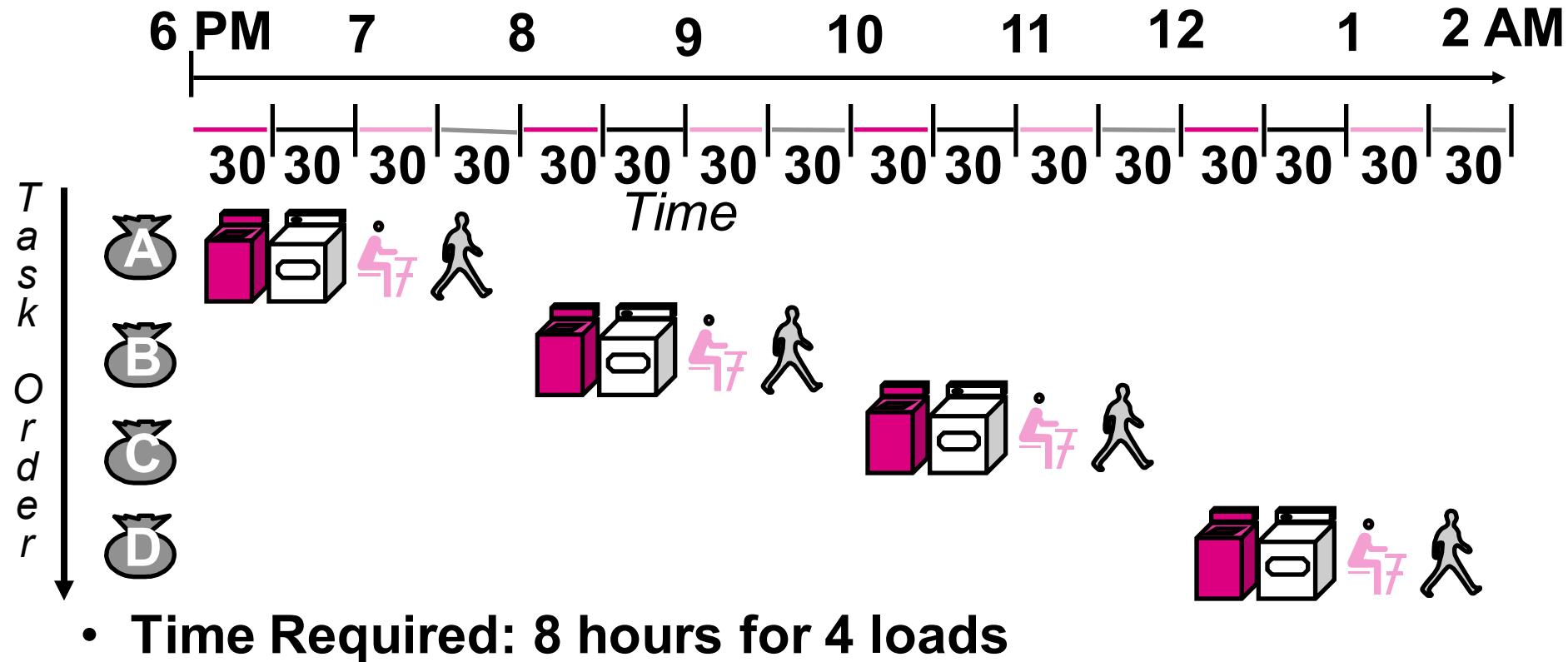
- A way of speeding up execution of instructions
- *Key idea:*
overlap execution of multiple instructions

The Laundry Analogy

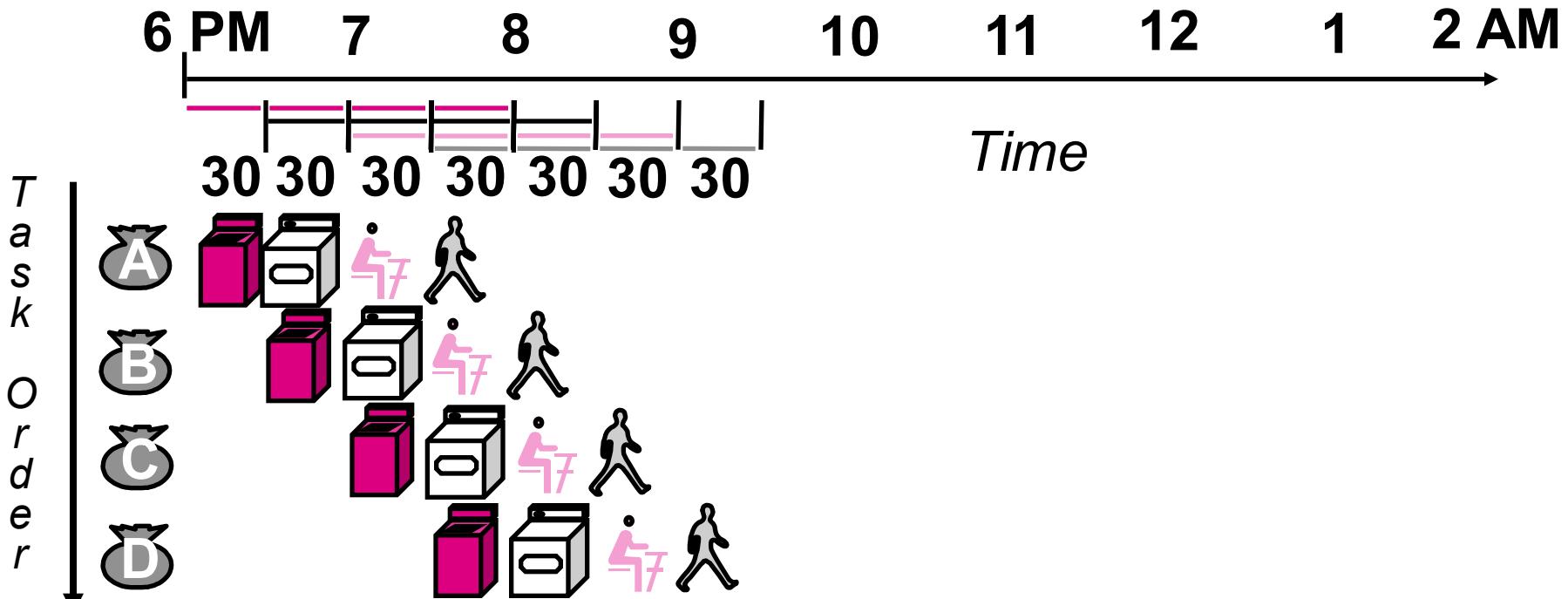
- Ann, Brian, Cathy, Dave each have one load of clothes to wash, dry, and fold
- Washer takes 30 minutes
- Dryer takes 30 minutes
- “Folder” takes 30 minutes
- “Stasher” takes 30 minutes to put clothes into drawers



If we do laundry sequentially...

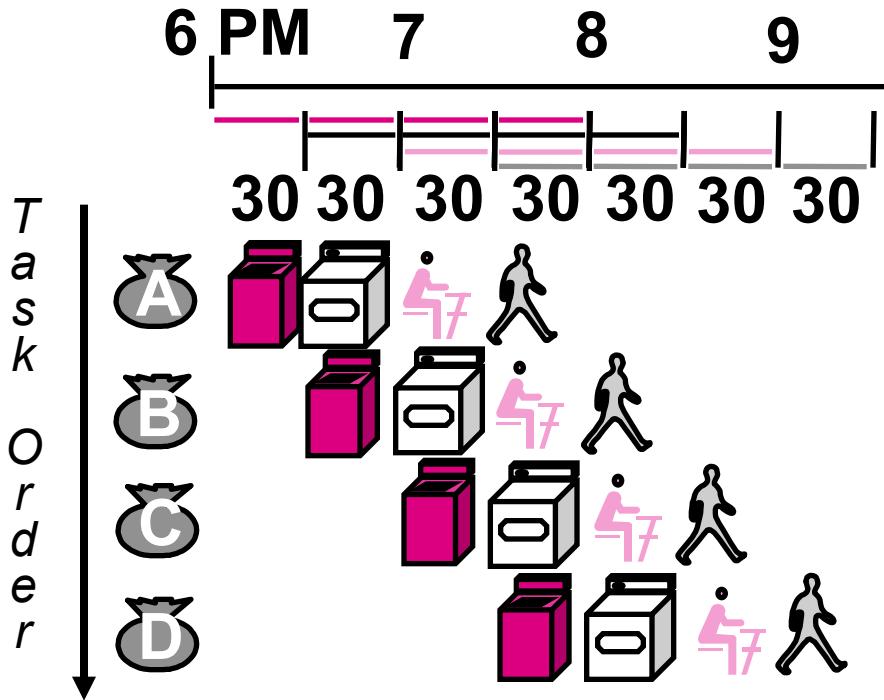


To Pipeline, We Overlap Tasks



- Time Required: 3.5 Hours for 4 Loads

To Pipeline, We Overlap Tasks



- Pipelining doesn't help **latency** of single task, it helps **throughput** of entire workload
- Pipeline rate limited by **slowest** pipeline stage
- **Multiple** tasks operating simultaneously
- Potential speedup = **Number pipe stages**
- Unbalanced lengths of pipe stages reduces speedup
- Time to “**fill**” pipeline and time to “**drain**” it reduces speedup

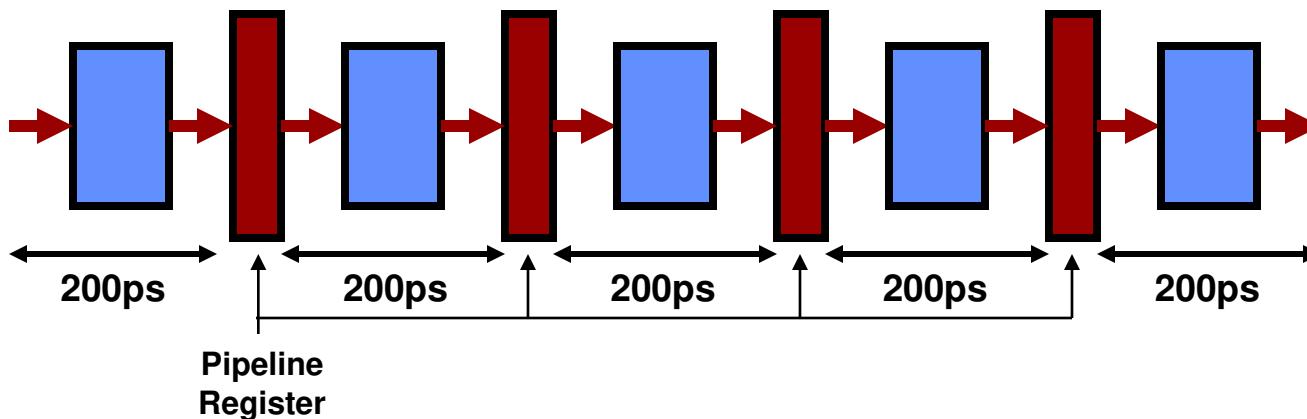
Pipelining a Digital System

1 nanosecond = 10^{-9} second
1 picosecond = 10^{-12} second

- Key idea: break big computation up into pieces

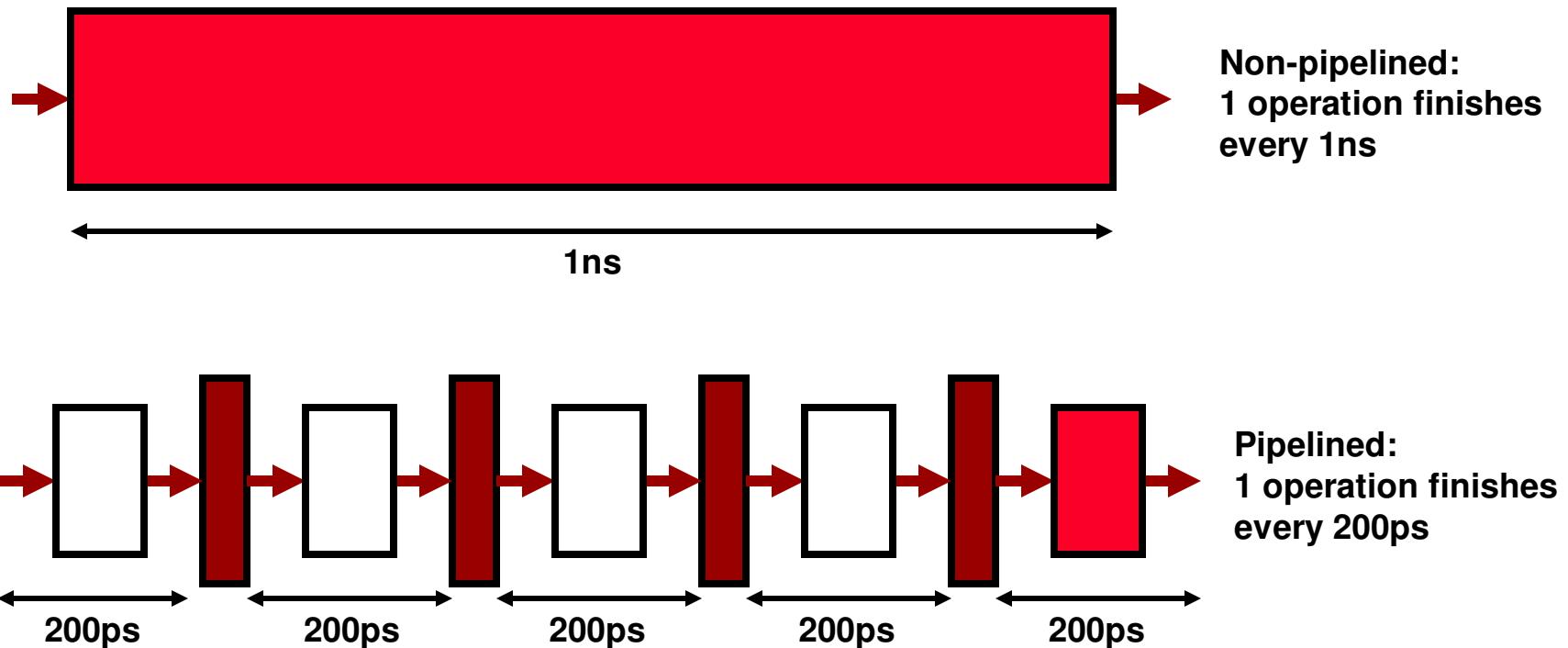


- Separate each piece with a pipeline register



Pipelining a Digital System

- Why do this? Because it's faster for repeated computations



Comments about pipelining

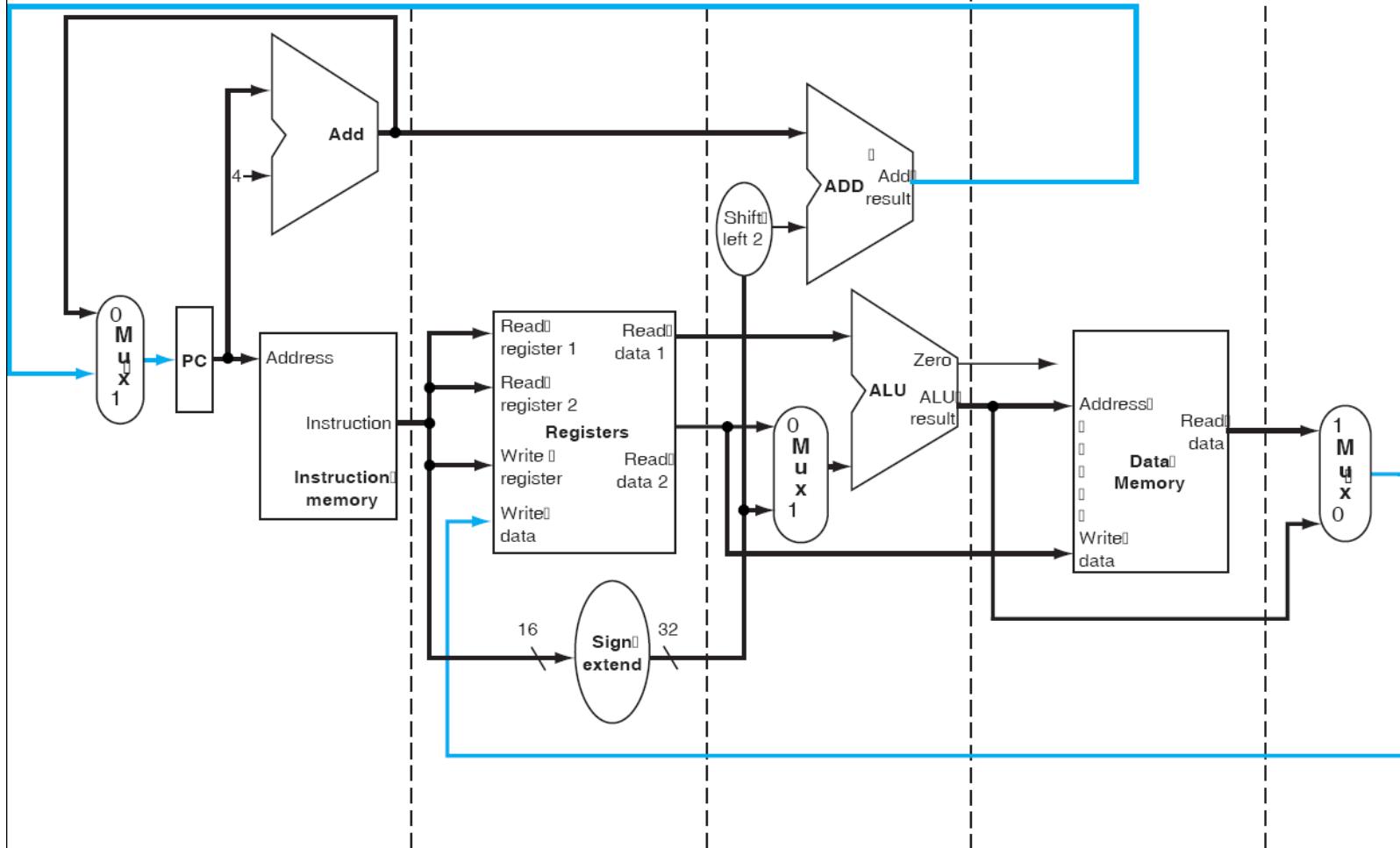
- Pipelining increases **throughput**, but not **latency**
 - Answer available every 200ps, BUT
 - A single computation still takes 1ns
- Limitations:
 - Computations must be divisible into stage size
 - Pipeline registers add overhead

Pipelining a Processor

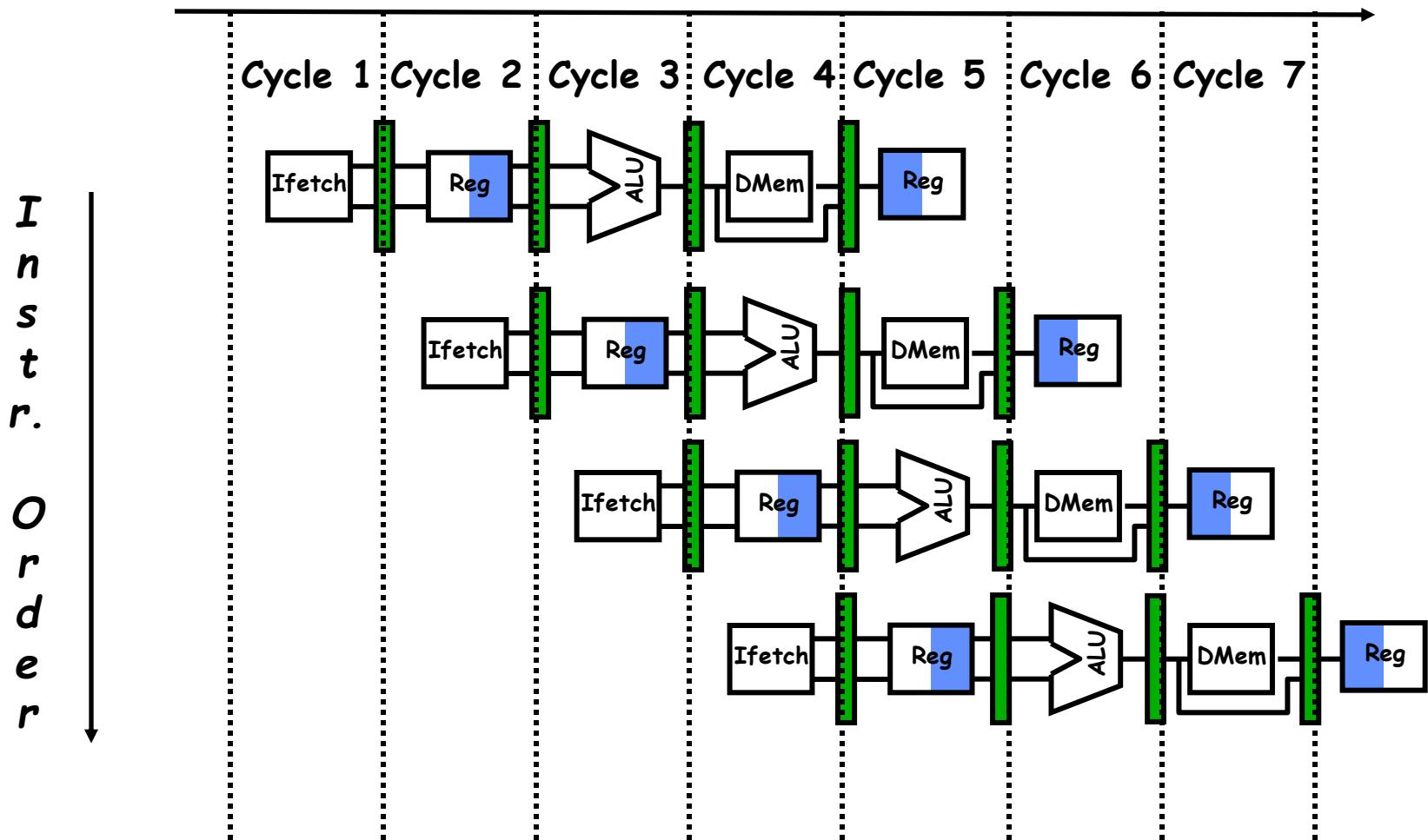
- **Recall the 5 steps in instruction execution:**
 1. Instruction Fetch (**IF**)
 2. Instruction Decode and Register Read (**ID**)
 3. Execution operation or calculate address (**EX**)
 4. Memory access (**MEM**)
 5. Write result into register (**WB**)
- **Review: Single-Cycle Processor**
 - All 5 steps done in a single clock cycle
 - Dedicated hardware required for each step

Review - Single-Cycle Processor

IF: Instruction fetch | ID: Instruction decode// register file read | EX: Execute// address calculation | MEM: Memory access | WB: Write back

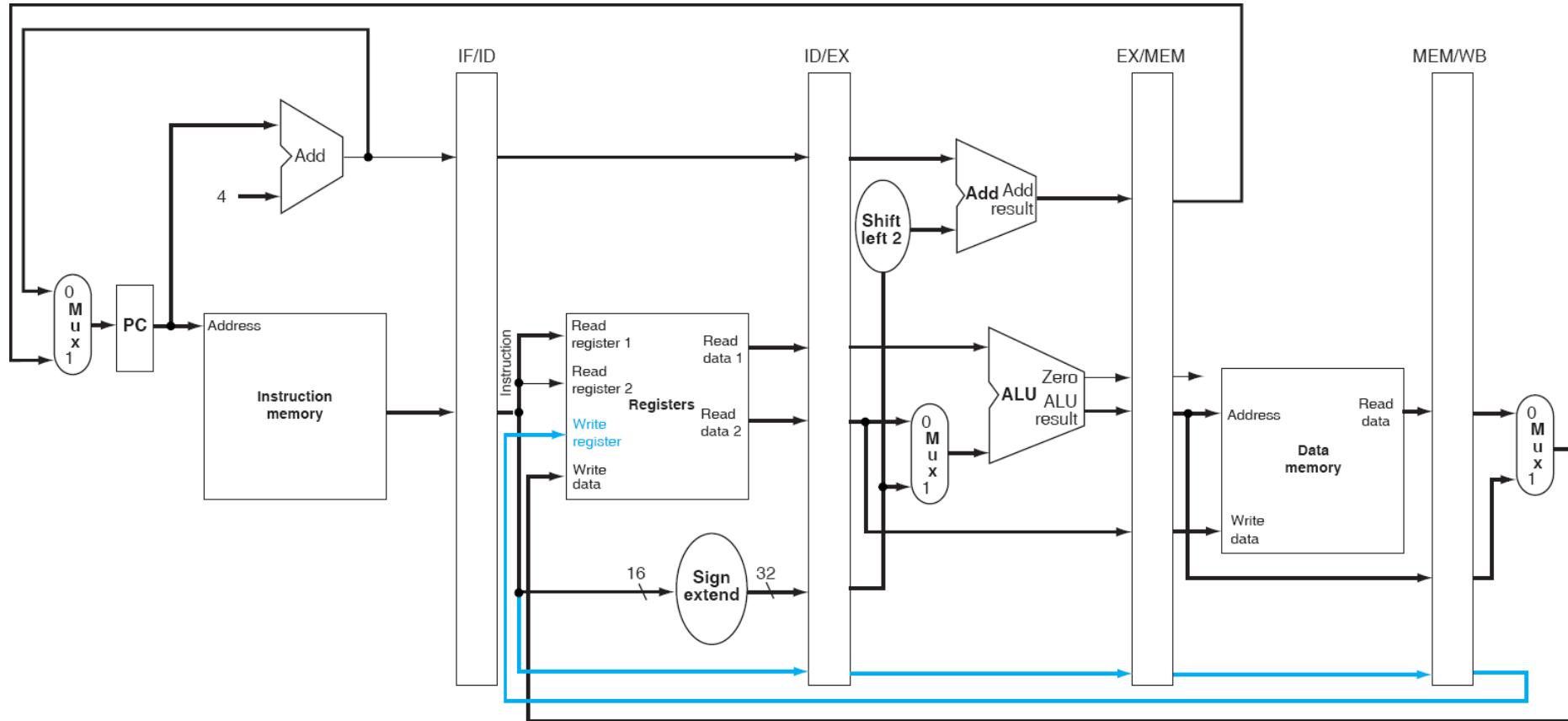


The Basic Pipeline For MIPS



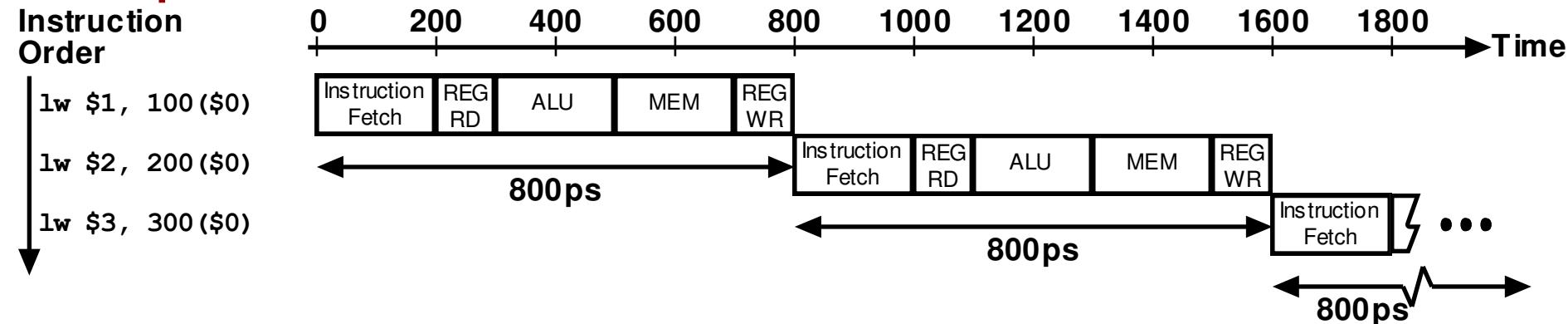
What do we need to add to actually split the datapath into stages?

Basic Pipelined Processor

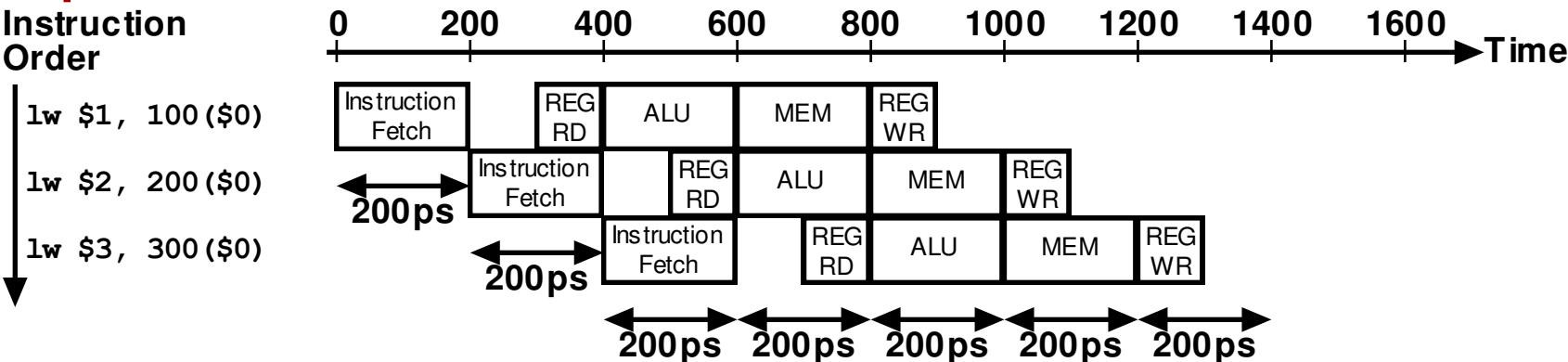


Single-Cycle vs. Pipelined Execution

Non-Pipelined



Pipelined



Speedup

- Consider the unpipelined processor introduced previously. Assume that it has a 1 ns clock cycle and it uses 4 cycles for ALU operations and branches, and 5 cycles for memory operations, assume that the relative frequencies of these operations are 40%, 20%, and 40%, respectively. Suppose that due to clock skew and setup, pipelining the processor adds 0.2ns of overhead to the clock. Ignoring any latency impact, how much speedup in the instruction execution rate will we gain from a pipeline?

Average instruction execution time

$$\begin{aligned} &= 1 \text{ ns} * ((40\% + 20\%)*4 + 40\%*5) \\ &= 4.4 \text{ ns} \end{aligned}$$

Speedup from pipeline

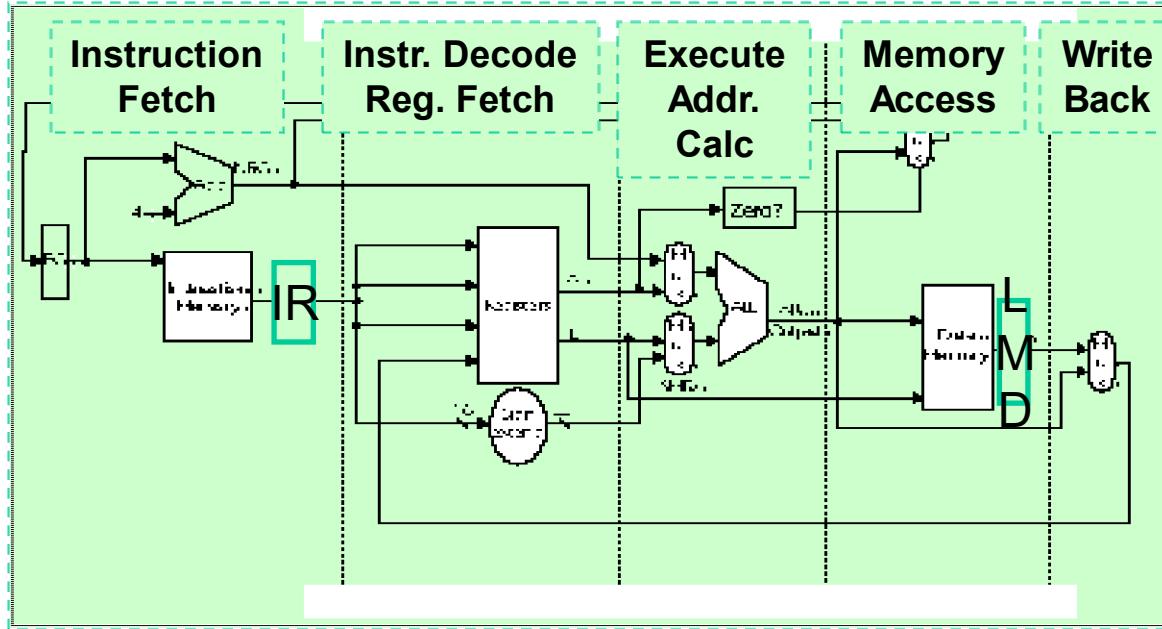
$$\begin{aligned} &= \text{Average instruction time unpiplined}/\text{Average instruction time pipelined} \\ &= 4.4 \text{ ns}/1.2 \text{ ns} = 3.7 \end{aligned}$$

Comments about Pipelining

- **The good news**
 - Multiple instructions are being processed at same time
 - This works because stages are isolated by registers
 - Best case speedup of N
- **The bad news**
 - Instructions interfere with each other - **hazards**
 - » Example: different instructions may need the same piece of hardware (e.g., memory) in same clock cycle
 - » Example: instruction may require a result produced by an earlier instruction that is not yet complete

What Is Pipelining

MIPS Functions



Passed To Next Stage

$IR \leftarrow \text{Mem}[PC]$

$NPC \leftarrow PC + 4$

Instruction Fetch (IF):

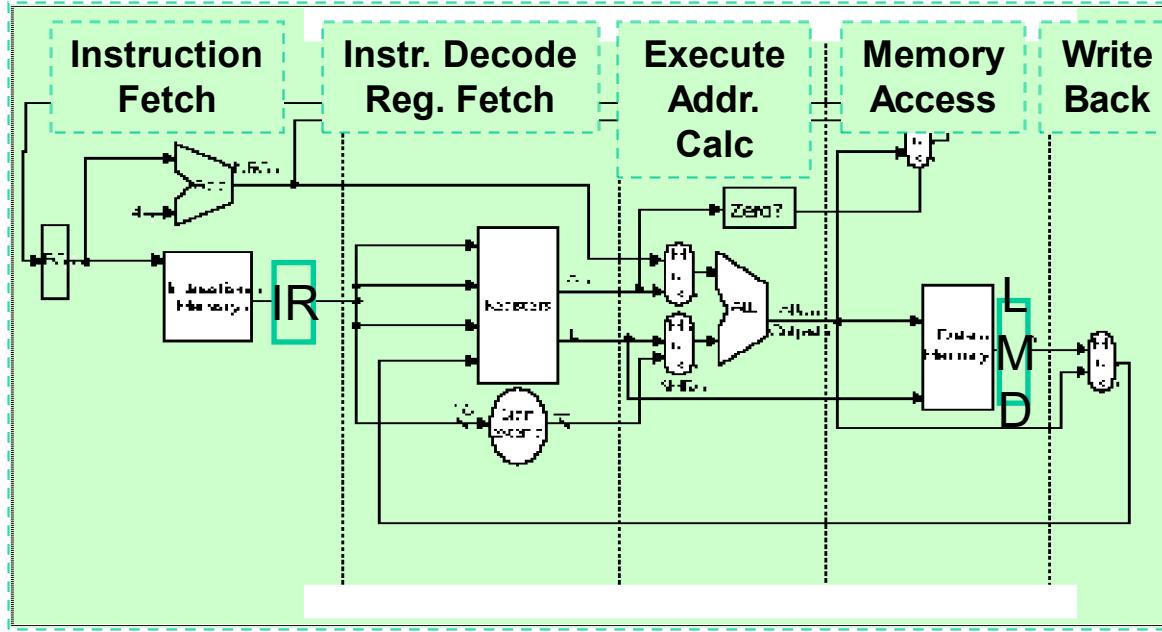
Send out the PC and fetch the instruction from memory into the instruction register (IR); increment the PC by 4 to address the next sequential instruction.

IR holds the instruction that will be used in the next stage.

NPC holds the value of the next PC.

What Is Pipelining

MIPS Functions

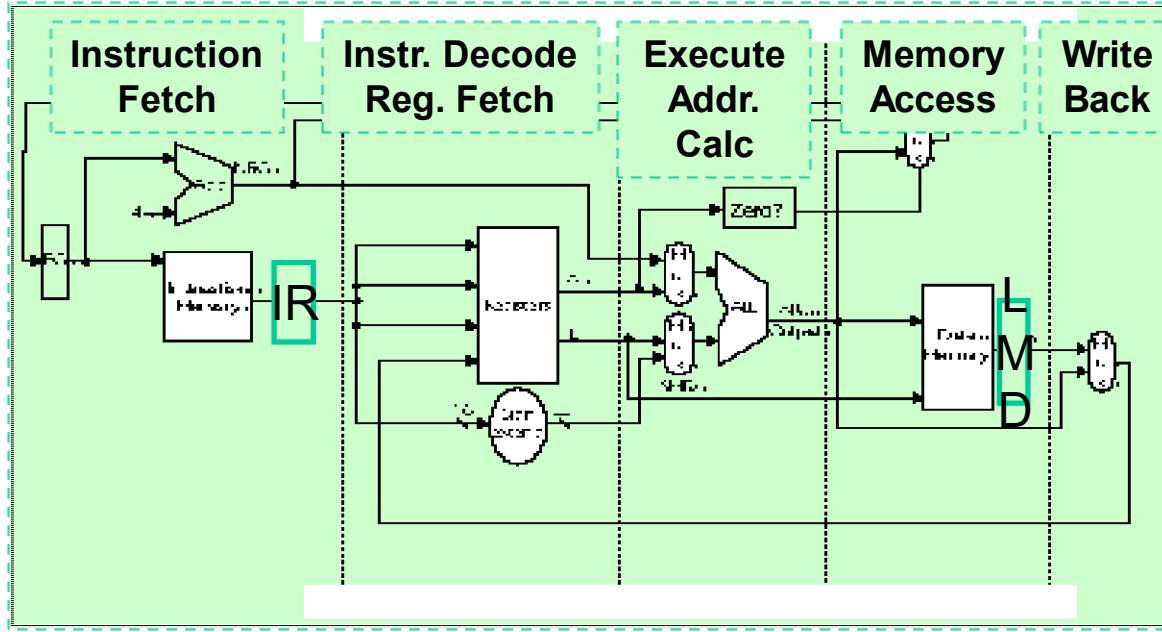


Instruction Decode/Register Fetch Cycle (ID):

Decode the instruction and access the register file to read the registers. The outputs of the general purpose registers are read into two temporary registers (A & B) for use in later clock cycles. We extend the sign of the lower 16 bits of the Instruction Register.

What Is Pipelining

MIPS Functions



Passed To Next Stage
A <- A func. B
cond = 0;

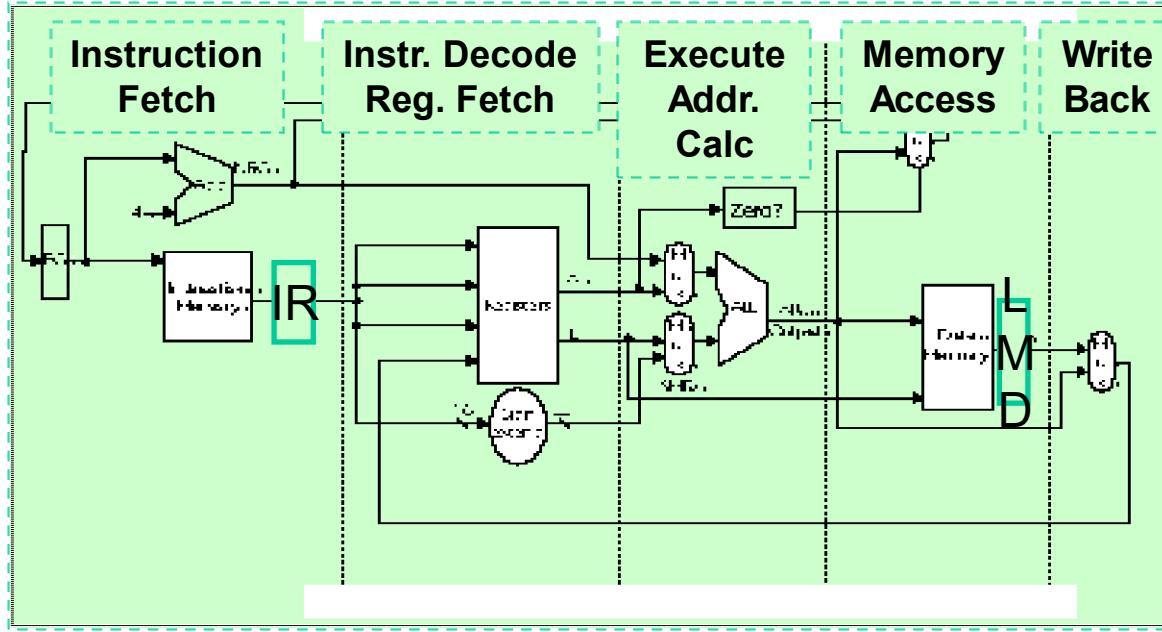
Execute Address Calculation (EX):

We perform an operation (for an ALU) or an address calculation (if it's a load or a Branch).

If an ALU, actually do the operation. If an address calculation, figure out how to obtain the address and stash away the location of that address for the next cycle.

What Is Pipelining

MIPS Functions



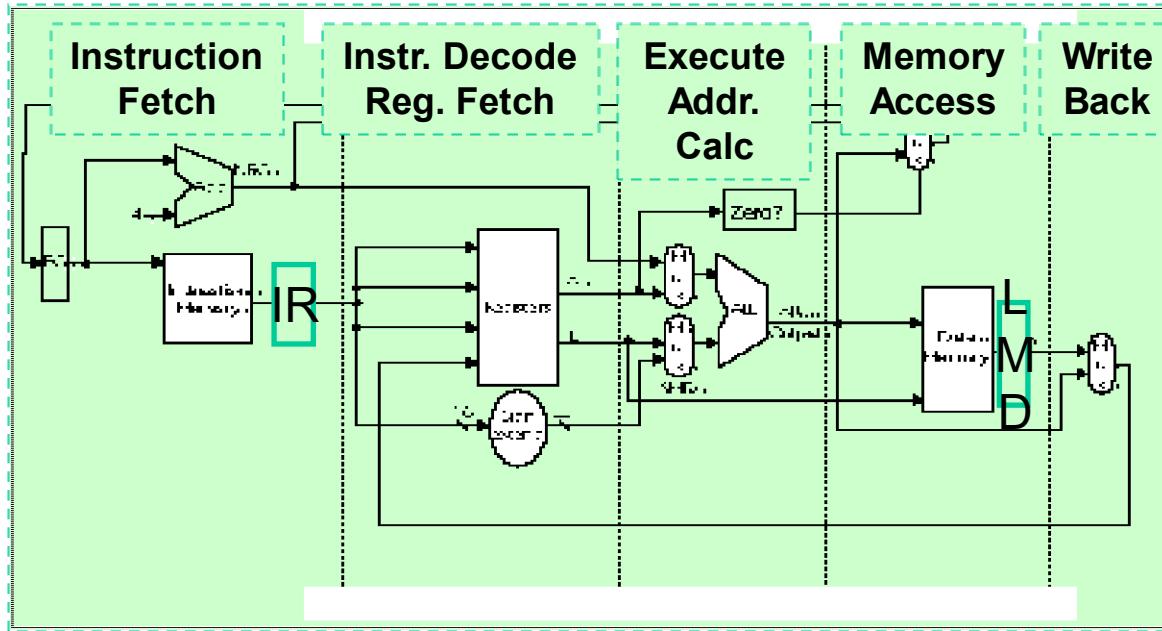
Passed To Next Stage
 $A = \text{Mem}[\text{prev. } B]$
or
 $\text{Mem}[\text{prev. } B] = A$

MEMORY ACCESS (MEM):

If this is an ALU, do nothing.
If a load or store, then access memory.

What Is Pipelining

MIPS Functions



Passed To Next Stage
Regs <- A, B;

WRITE BACK (WB):

Update the registers from either the ALU or from the data loaded.

Pipeline Hazards

- **Limits to pipelining: Hazards prevent next instruction from executing during its designated clock cycle**
 - Structural hazards: two different instructions use same h/w in same cycle
 - Data hazards: Instruction depends on result of prior instruction still in the pipeline
 - Control hazards: Pipelining of branches & other instructions that change the PC

Summary - Pipelining Overview

- Pipelining increase throughput (but not latency)
- Hazards limit performance
 - Structural hazards
 - Control hazards
 - Data hazards

Pipeline Hurdles

Definition

- conditions that lead to incorrect behavior if not fixed
- Structural hazard
 - two different instructions use same h/w in same cycle
- Data hazard
 - two different instructions use same storage
 - must appear as if the instructions execute in correct order
- Control hazard
 - one instruction affects which instruction is next

Resolution

- Pipeline interlock logic detects hazards and fixes them
- simple solution: stall
- increases CPI, decreases performance
- better solution: partial stall
- some instruction stall, others proceed better to stall early than late

Structural Hazards

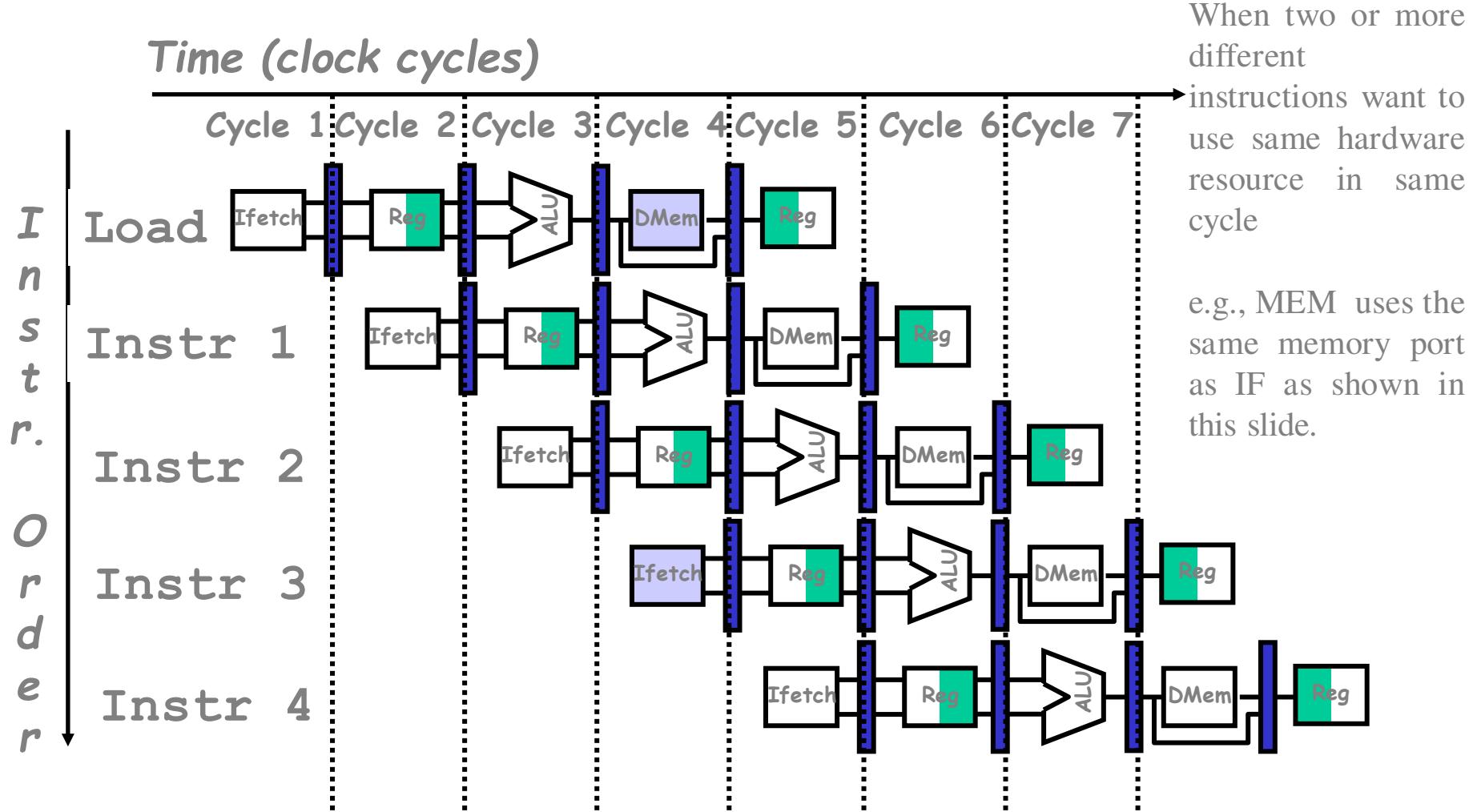


Figure 3.6

Structural Hazards

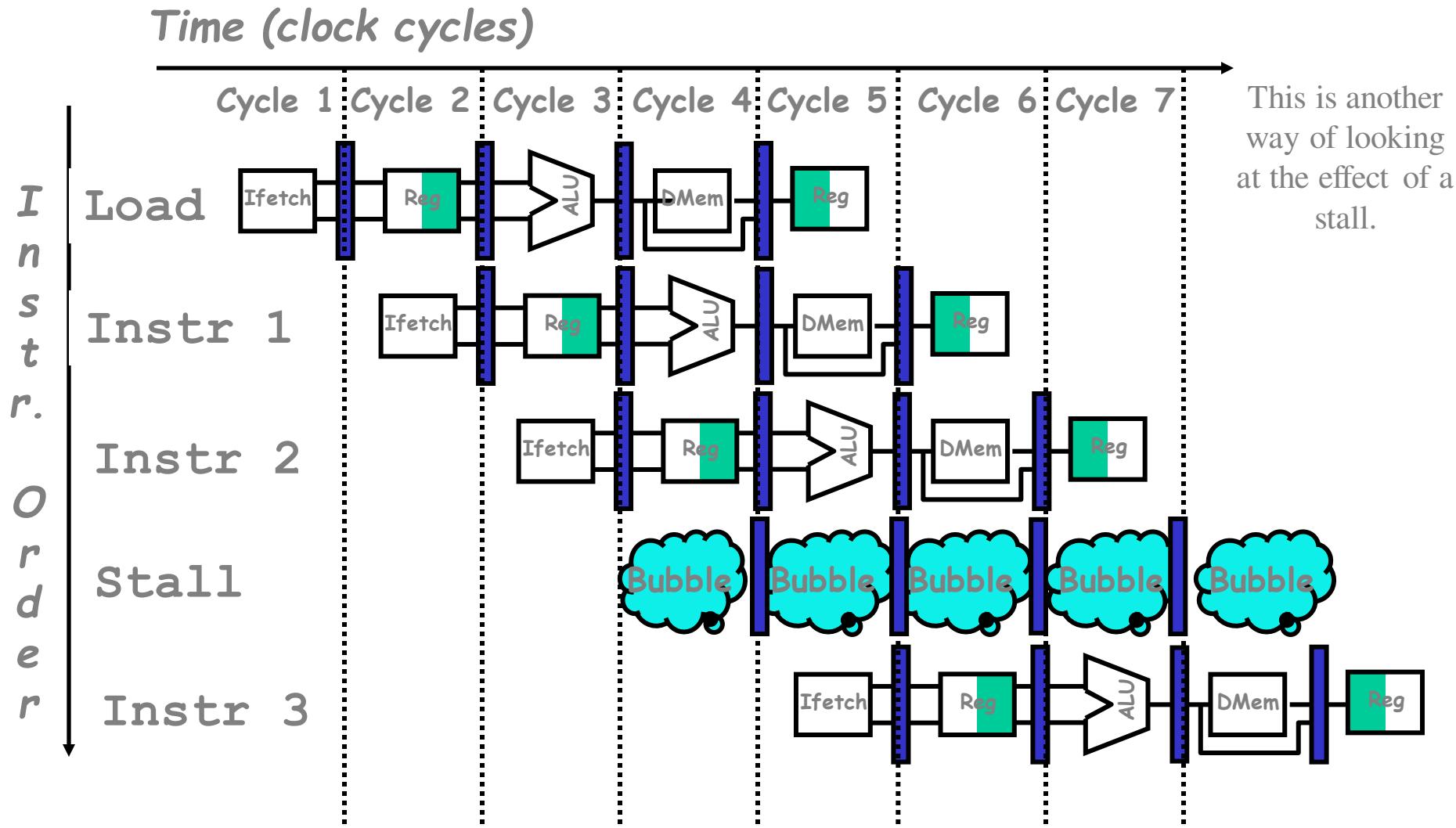


Figure 3.7

Structural Hazards

Instruction	Clock cycle number									
	1	2	3	4	5	6	7	8	9	10
Load instruction	IF	ID	EX	MEM	WB					
Instruction j + 1	IF	ID	EX	MEM	WB					
Instruction j + 2		IF	ID	EX	MEM	WB				
Instruction j + 3			stall	IF	ID	EX	MEM	WB		
Instruction j + 4				IF	ID	EX	MEM	WB		
Instruction j + 5					IF	ID	EX	MEM		
Instruction j + 6						IF	ID	EX		

This is another way to represent the stall we saw on the last few pages.

Structural Hazards

Dealing with Structural Hazards

Stall

- low cost, simple
- Increases CPI
- use for rare case since stalling has performance effect

Pipeline hardware resource

- useful for multi-cycle resources
- good performance
- sometimes complex e.g., RAM

Replicate resource

- good performance
- increases cost (+ maybe interconnect delay)
- useful for cheap or divisible resources

Structural Hazards

Structural hazards are reduced with these rules:

- Each instruction uses a resource at most once
- Always use the resource in the same pipeline stage
- Use the resource for one cycle only

Many RISC ISA's designed with this in mind

Sometimes very complex to do this. For example, memory of necessity is used in the IF and MEM stages.

Some common Structural Hazards:

- Memory - we've already mentioned this one.
- Floating point - Since many floating point instructions require many cycles, it's easy for them to interfere with each other.
- Starting up more of one type of instruction than there are resources. For instance, the PA-8600 can support two ALU + two load/store instructions per cycle - that's how much hardware it has available.

Structural Hazards

This is the example on Page 144.

We want to compare the performance of two machines. Which machine is faster?

- Machine A: Dual ported memory - so there are no memory stalls
- Machine B: Single ported memory, but its pipelined implementation has a 1.05 times faster clock rate

Assume:

- Ideal CPI = 1 for both
- Loads are 40% of instructions executed

$$\begin{aligned}\text{SpeedUp}_A &= \text{Pipeline Depth} / (1 + 0) \times (\text{clock}_{\text{unpipe}} / \text{clock}_{\text{pipe}}) \\ &= \text{Pipeline Depth}\end{aligned}$$

$$\begin{aligned}\text{SpeedUp}_B &= \text{Pipeline Depth} / (1 + 0.4 \times 1) \\ &\quad \times (\text{clock}_{\text{unpipe}} / (\text{clock}_{\text{unpipe}} / 1.05)) \\ &= (\text{Pipeline Depth} / 1.4) \times 1.05 \\ &= 0.75 \times \text{Pipeline Depth}\end{aligned}$$

$$\text{SpeedUp}_A / \text{SpeedUp}_B = \text{Pipeline Depth} / (0.75 \times \text{Pipeline Depth}) = 1.33$$

- Machine A is 1.33 times faster

Data Hazards

A.1 What is Pipelining?

A.2 The Major Hurdle of Pipelining- Structural Hazards

- Structural Hazards
 - Data Hazards
 - Control Hazards

A.3 How is Pipelining Implemented

A.4 What Makes Pipelining Hard to Implement?

A.5 Extending the MIPS Pipeline to Handle Multi-cycle Operations

These occur when at any time, there are instructions active that need to access the same data (memory or register) locations.

Where there's real trouble is when we have:

instruction A
instruction B

and B manipulates (reads or writes) data before A does. This violates the order of the instructions, since the architecture implies that A completes entirely before B is executed.

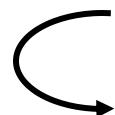
Data Hazards

Execution Order is:

Instr_I
Instr_J

Read After Write (RAW)

Instr_J tries to read operand before Instr_I writes it



I: add r1, r2, r3
J: sub r4, r1, r3

- Caused by a “Dependence” (in compiler nomenclature). This hazard results from an actual need for communication.

Data Hazards

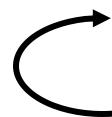
Execution Order is:

Instr_I
Instr_J

Write After Read (WAR)

Instr_J tries to write operand before Instr_I reads it

- Gets wrong operand



I: sub r4, r1, r3
J: add r1, r2, r3
K: mul r6, r1, r7

- Called an “anti-dependence” by compiler writers.
This results from reuse of the name “r1”.
- **Can't happen in MIPS 5 stage pipeline because:**
 - All instructions take 5 stages, and
 - Reads are always in stage 2, and
 - Writes are always in stage 5

Data Hazards

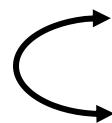
Execution Order is:

Instr_I
Instr_J

Write After Write (WAW)

Instr_J tries to write operand before Instr_I writes it

- Leaves wrong result (Instr_I not Instr_J)



I: sub r1,r4,r3
J: add r1,r2,r3
K: mul r6,r1,r7

- Called an “output dependence” by compiler writers
This also results from the reuse of name “r1”.
- Can’t happen in MIPS 5 stage pipeline because:
 - All instructions take 5 stages, and
 - Writes are always in stage 5
- Will see WAR and WAW in later more complicated pipes

Data Hazards

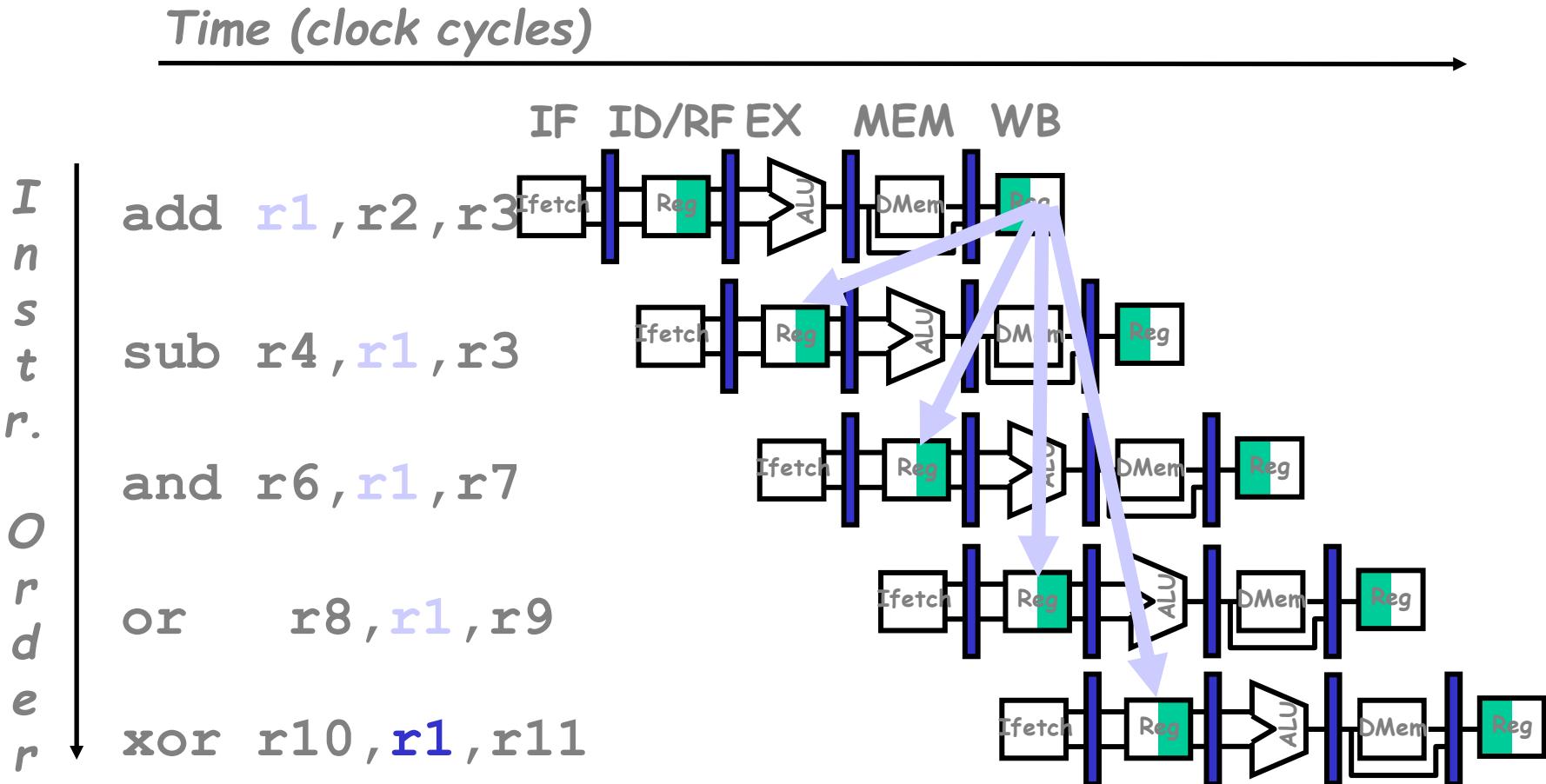
Simple Solution to RAW

- Hardware detects RAW and stalls
- Assumes register written then read each cycle
 - + low cost to implement, simple
 - reduces IPC
 - Try to minimize stalls

Minimizing RAW stalls

- Bypass/forward/shortcircuit (We will use the word “forward”)
 - Use data before it is in the register
 - + reduces/avoids stalls
 - complex
 - Crucial for common RAW hazards

Data Hazards



The use of the result of the ADD instruction in the next three instructions causes a hazard, since the register is not written until after those instructions read it.

Data Hazards

Forwarding To Avoid Data Hazard

Forwarding is the concept of making data available to the input of the ALU for subsequent instructions, even though the generating instruction hasn't gotten to WB in order to write the memory or registers.

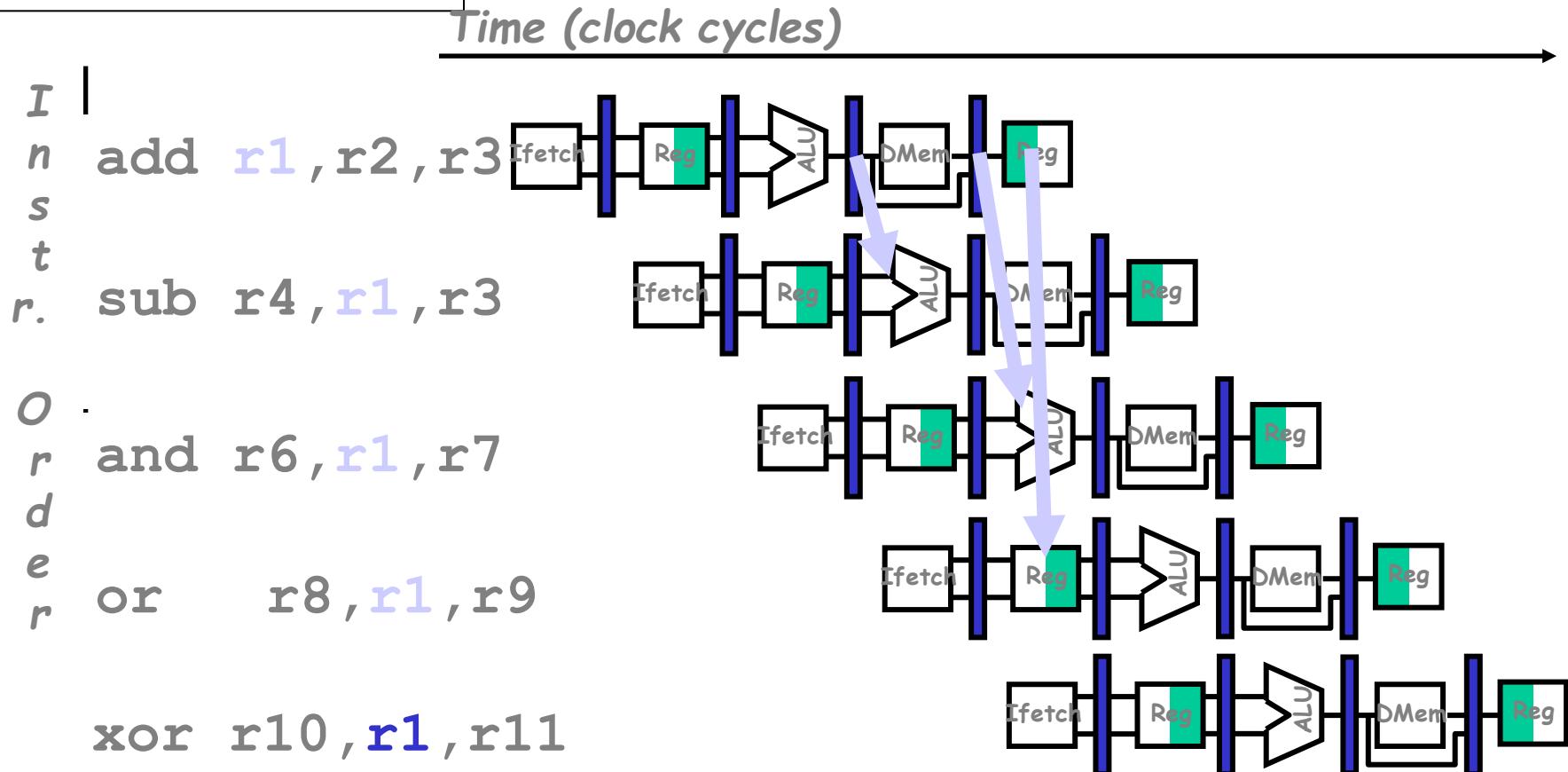
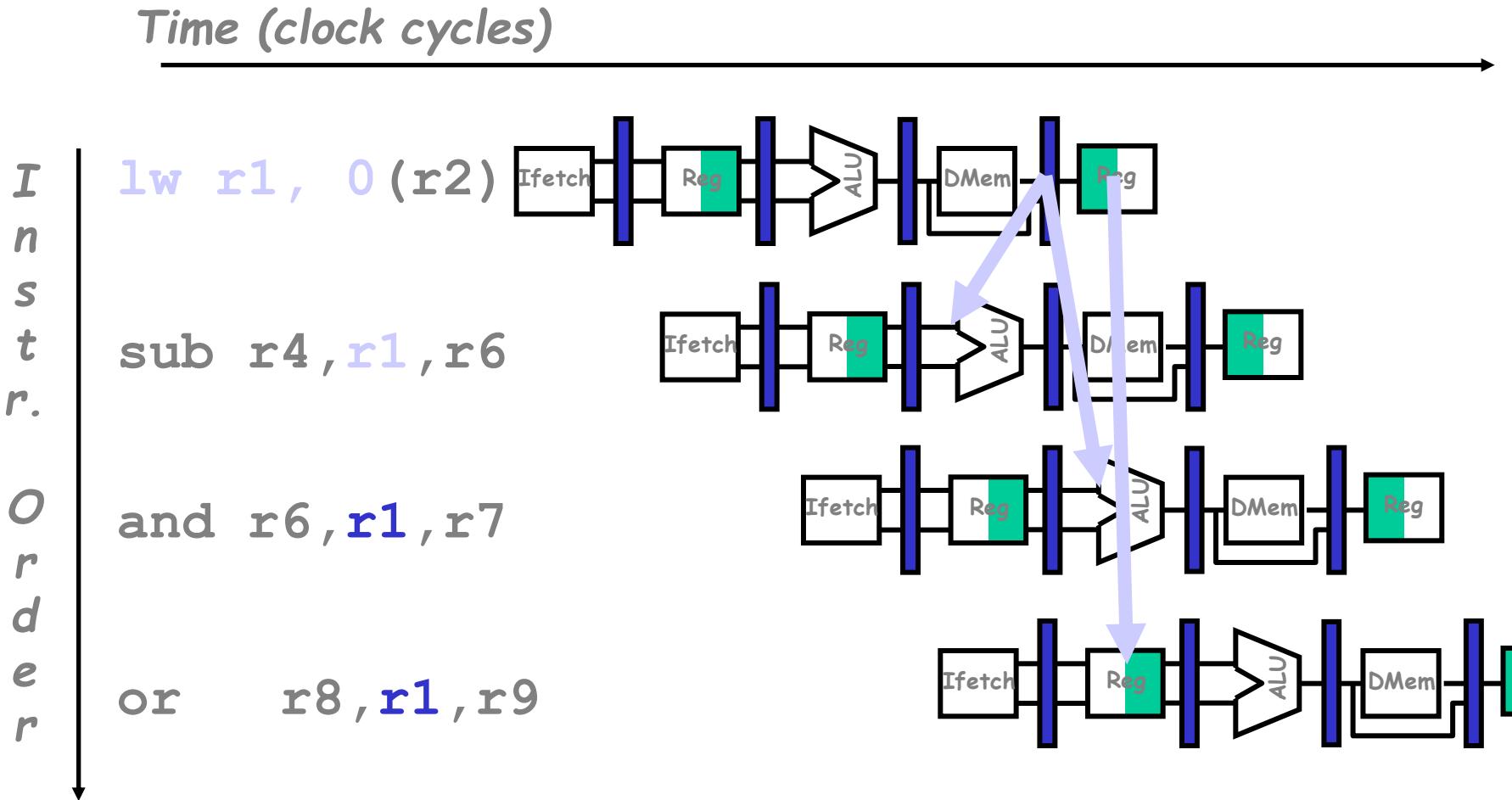


Figure 3.10
08-Jul-2024_multiprocessor

Data Hazards

The data isn't loaded until after the MEM stage.

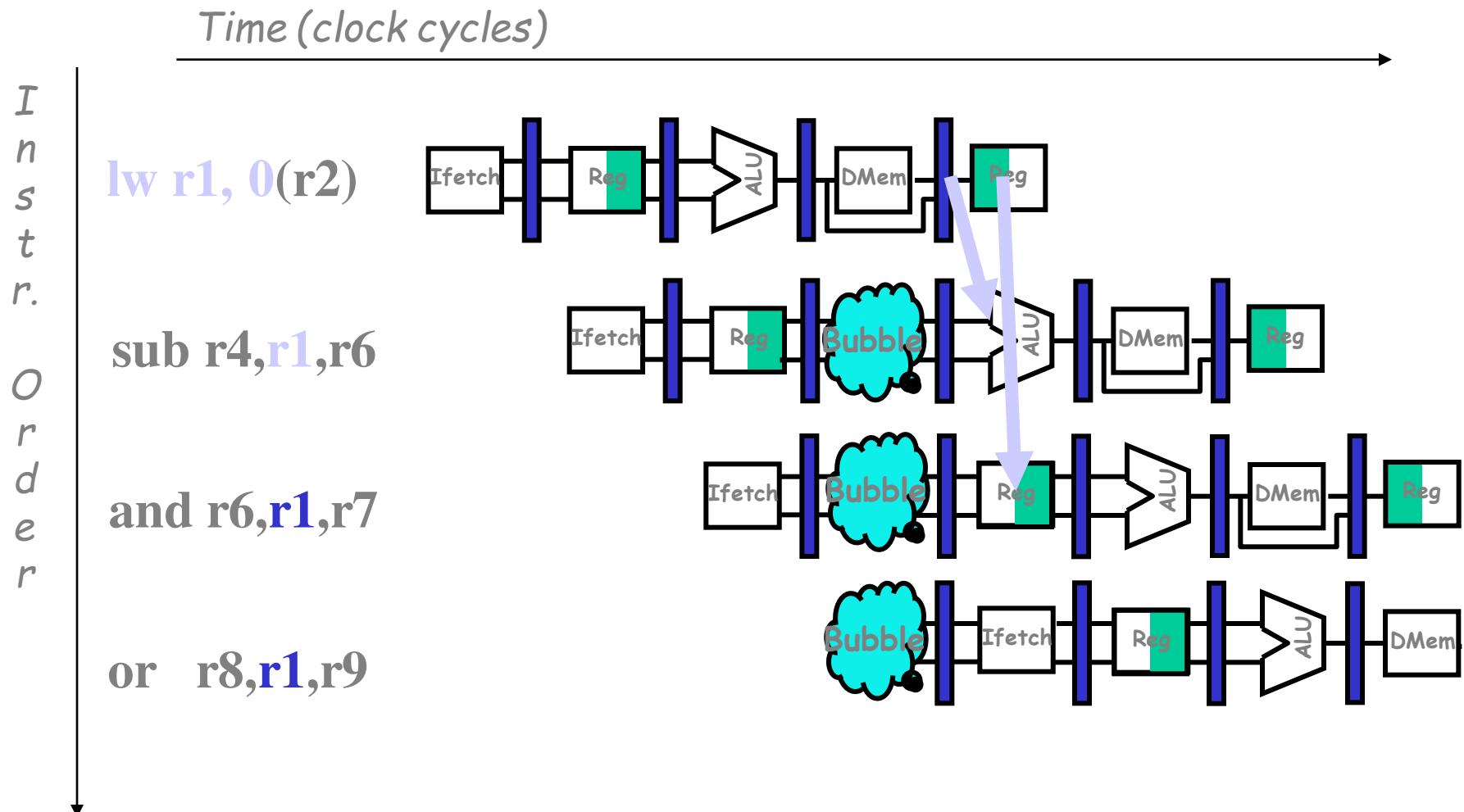


There are some instances where hazards occur, even with forwarding.

Figure 3.12
08-Jul-2024_multiprocessor

Data Hazards

The stall is necessary as shown here.



There are some instances where hazards occur, even with forwarding.

Figure 3.13
08-Jul-2024_multiprocessor

Data Hazards

This is another representation of the stall.

LW R1, 0(R2)	IF	ID	EX	MEM	WB			
SUB R4, R1, R5		IF	ID	EX	MEM	WB		
AND R6, R1, R7			IF	ID	EX	MEM	WB	
OR R8, R1, R9				IF	ID	EX	MEM	WB

LW R1, 0(R2)	IF	ID	EX	MEM	WB			
SUB R4, R1, R5		IF	ID	stall	EX	MEM	WB	
AND R6, R1, R7			IF	stall	ID	EX	MEM	WB
OR R8, R1, R9				stall	IF	ID	EX	MEM

Data Hazards

Pipeline Scheduling

Instruction scheduled by compiler - move instruction in order to reduce stall.

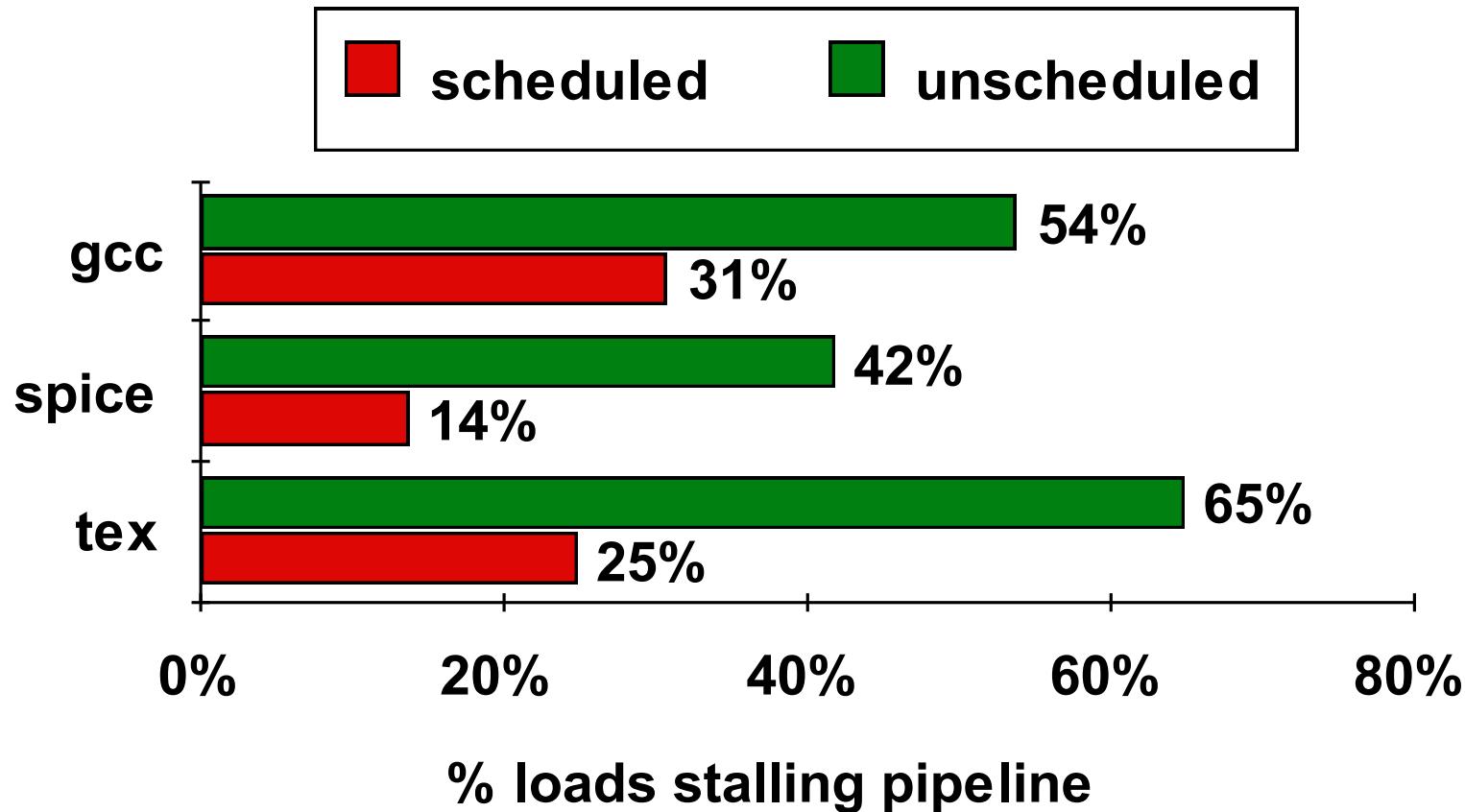
lw Rb, b	code sequence for a = b+c before scheduling
	lw Rc, c
	Add Ra, Rb, Rc stall
	sw a, Ra
lw Re, e	code sequence for d = e+f before scheduling
	lw Rf, f
	sub Rd, Re, Rf stall
	sw d, Rd

Arrangement of code after scheduling.

lw Rb, b
lw Rc, c
lw Re, e
Add Ra, Rb, Rc
lw Rf, f
sw a, Ra
sub Rd, Re, Rf
sw d, Rd

Data Hazards

Pipeline Scheduling



Control Hazards

A.1 What is Pipelining?

A.2 The Major Hurdle of Pipelining-
Structural Hazards

- Structural Hazards
 - Data Hazards
 - Control Hazards

A.3 How is Pipelining Implemented

A.4 What Makes Pipelining Hard to
Implement?

A.5 Extending the MIPS Pipeline to
Handle Multi-cycle Operations

A control hazard is when we need to find the destination of a branch, and can't fetch any new instructions until we know that destination.

Control Hazards

Control Hazard on Branches Three Stage Stall

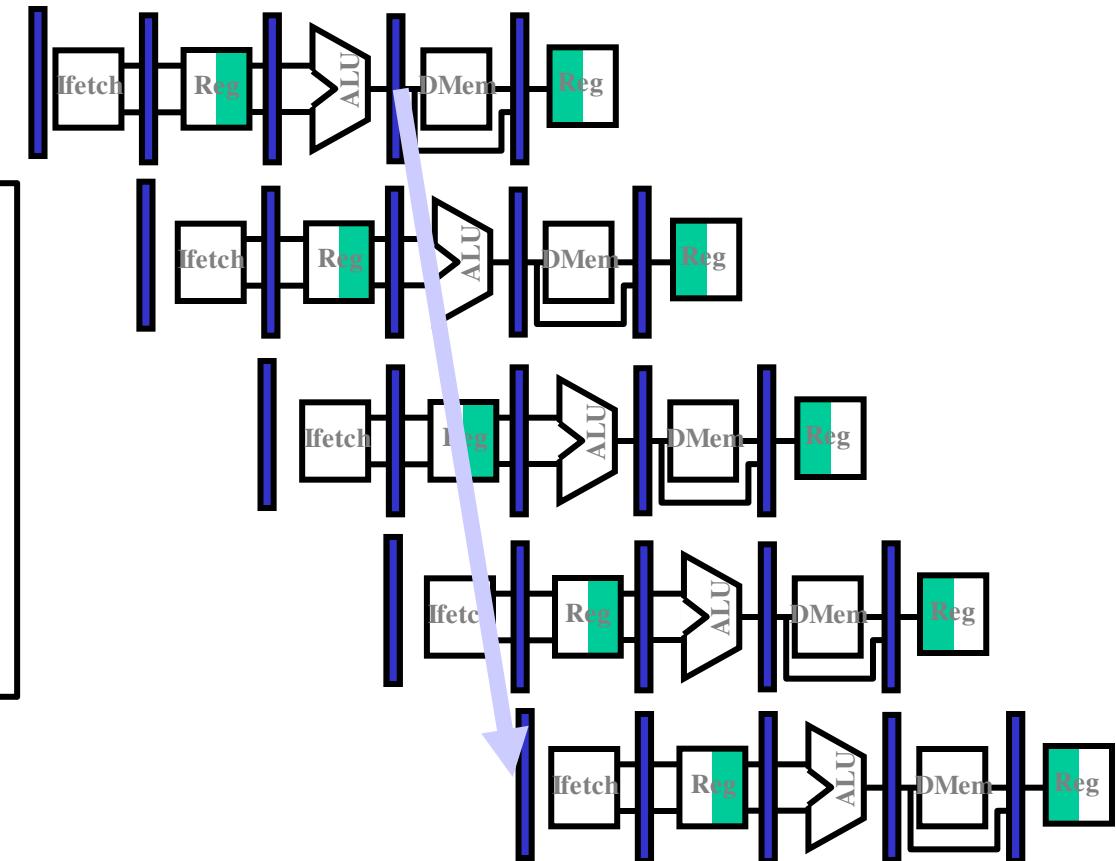
10: beq r1,r3,36

14: and r2,r3,r5

18: or r6,r1,r7

22: add r8,r1,r9

36: xor r10,r1,r11



Control Hazards

Branch Stall Impact

- **If CPI = 1, 30% branch, Stall 3 cycles => new CPI = 1.9!**
(Whoa! How did we get that 1.9????)
- **Two part solution to this dramatic increase:**
 - Determine branch taken or not sooner, AND
 - Compute taken branch address earlier
- **MIPS branch tests if register = 0 or ^ 0**
- **MIPS Solution:**
 - Move Zero test to ID/RF stage
 - Adder to calculate new PC in ID/RF stage
 - must be fast
 - can't afford to subtract
 - compares with 0 are simple
 - Greater-than, Less-than test signbit, but not-equal must OR all bits
 - more general compares need ALU
 - 1 clock cycle penalty for branch versus 3

In the next chapter, we'll look at ways to avoid the branch all together.

Control Hazards

Five Branch Hazard Alternatives

#1: Stall until branch direction is clear

#2: Predict Branch Not Taken

- Execute successor instructions in sequence
- “Squash” instructions in pipeline if branch actually taken
- Advantage of late pipeline state update
- 47% MIPS branches not taken on average
- PC+4 already calculated, so use it to get next instruction

#3: Predict Branch Taken

- 53% MIPS branches taken on average
- But haven't calculated branch target address in MIPS
 - MIPS still incurs 1 cycle branch penalty
 - Other machines: branch target known before outcome

Control Hazards

Five Branch Hazard Alternatives

#4: Execute Both Paths

#5: Delayed Branch

- Define branch to take place **AFTER** a following instruction

branch instruction
sequential successor₁
sequential successor₂
.....
sequential successor_n
branch target if taken

Branch delay of length *n*

- 1 slot delay allows proper decision and branch target address in 5 stage pipeline
- MIPS uses this

Control Hazards

Delayed Branch

- **Where to get instructions to fill branch delay slot?**
 - Before branch instruction
 - From the target address: only valuable when branch taken
 - From fall through: only valuable when branch not taken
 - Cancelling branches allow more slots to be filled
- **Compiler effectiveness for single branch delay slot:**
 - Fills about 60% of branch delay slots
 - About 80% of instructions executed in branch delay slots useful in computation
 - About 50% ($60\% \times 80\%$) of slots usefully filled
- **Delayed Branch downside: 7-8 stage pipelines, multiple instructions issued per clock (superscalar)**

Control Hazards

Evaluating Branch Alternatives

$$\text{Pipeline speedup} = \frac{\text{Pipeline depth}}{1 + \text{Branch frequency} \times \text{Branch penalty}}$$

<i>Scheduling scheme</i>	<i>Branch penalty</i>	<i>CPI</i>	<i>speedup v. unpipelined</i>	<i>Speedup v. stall</i>
Stall pipeline	3	1.42	3.5	1.0
Predict taken	1	1.14	4.4	1.26
Predict not taken	1	1.09	4.5	1.29
Delayed branch	0.5	1.07	4.6	1.31

Conditional & Unconditional = 14%,

65% change PC

Control Hazards

Pipelining Introduction Summary

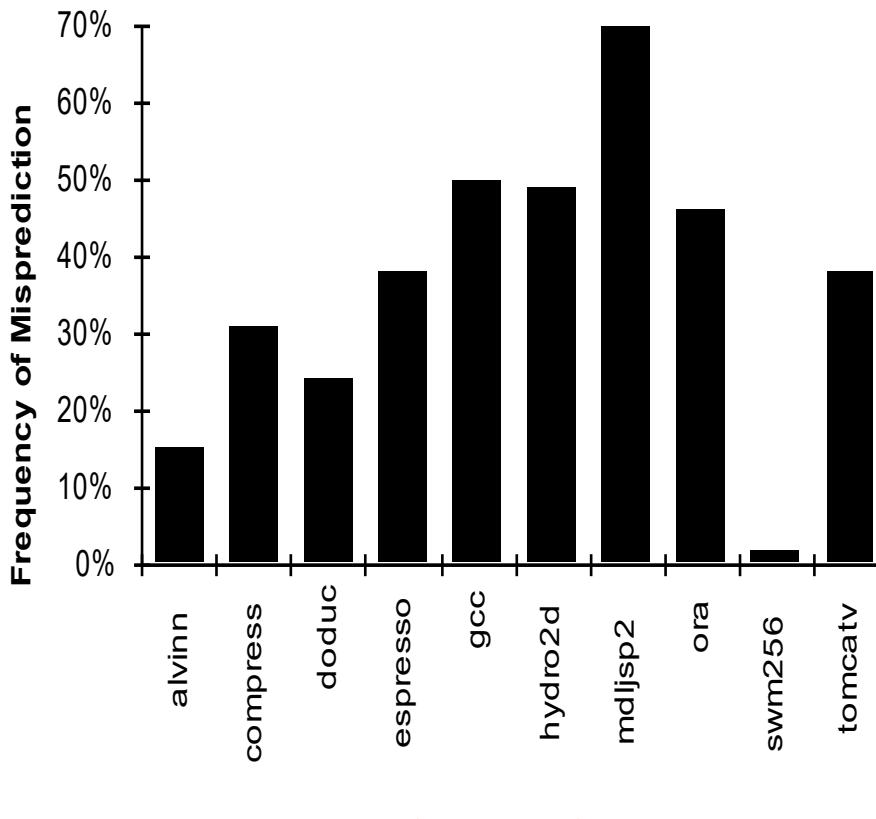
- Just overlap tasks, and easy if tasks are independent
- Speed Up Pipeline Depth; if ideal CPI is 1, then:

$$\text{Speedup} = \frac{\text{Pipeline Depth}}{1 + \text{Pipeline stall CPI}} \times \frac{\text{Clock Cycle Unpipelined}}{\text{Clock Cycle Pipelined}}$$

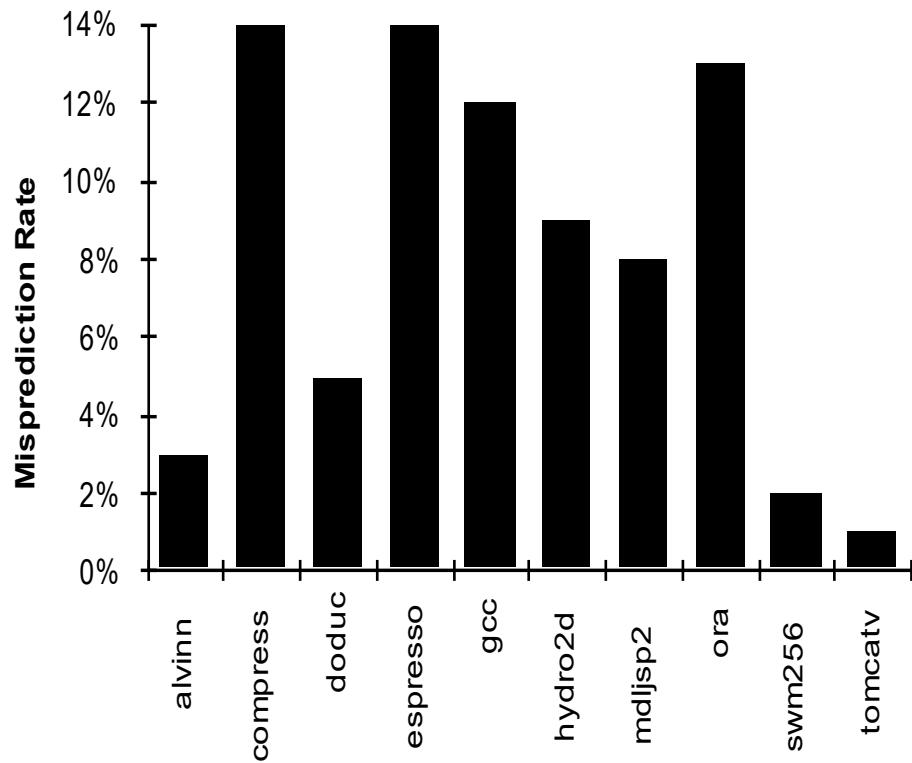
- Hazards limit performance on computers:
 - Structural: need more HW resources
 - Data (RAW,WAR,WAW): need forwarding, compiler scheduling
 - Control: delayed branch, prediction

Control Hazards

The compiler can program what it thinks the branch direction will be. Here are the results when it does so.



Compiler “Static” Prediction of Taken/Untaken Branches



Always taken

Taken backwards
Not Taken Forwards

Control Hazards

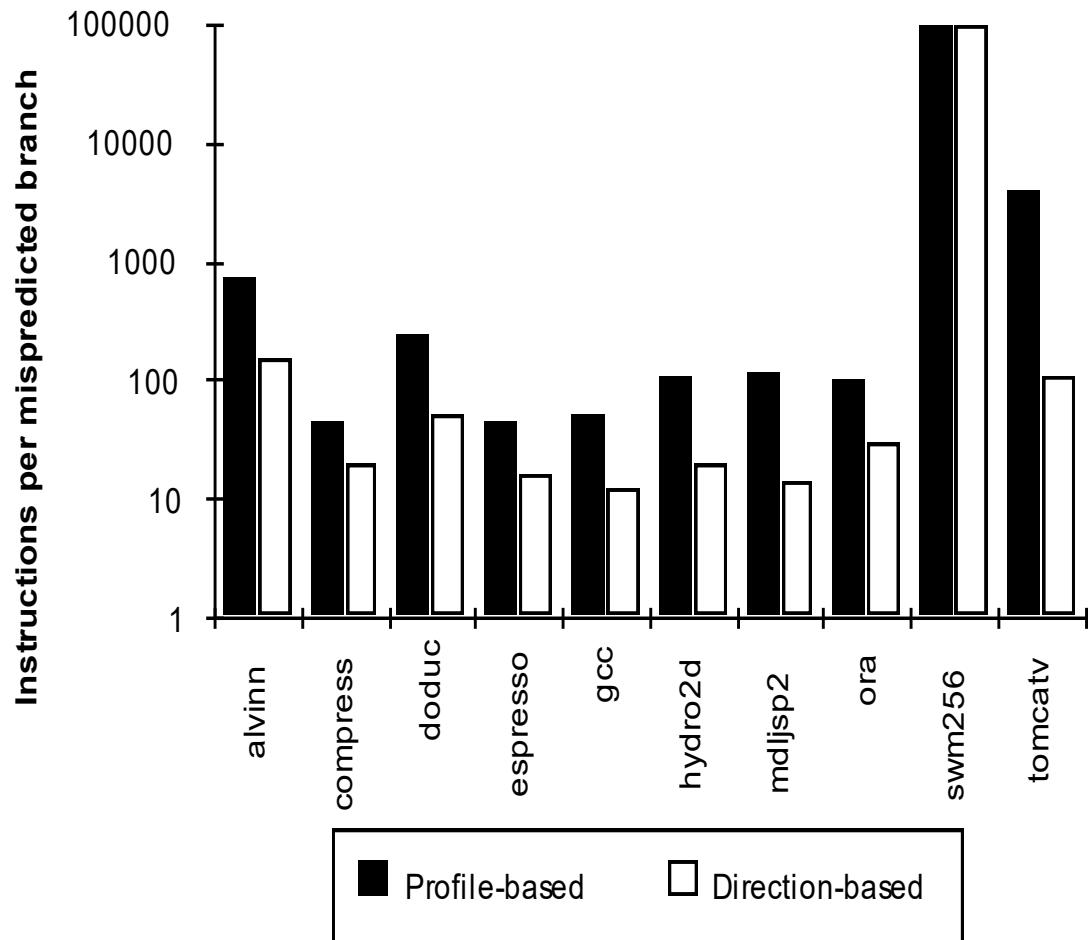
Compiler “Static” Prediction of Taken/Untaken Branches

- **Improves strategy for placing instructions in delay slot**
- **Two strategies**
 - Backward branch predict taken, forward branch not taken
 - Profile-based prediction: record branch behavior, predict branch based on prior run

Control Hazards

Evaluating Static Branch Prediction Strategies

- Misprediction ignores frequency of branch
- “Instructions between mispredicted branches” is a better metric



An example execution sequence

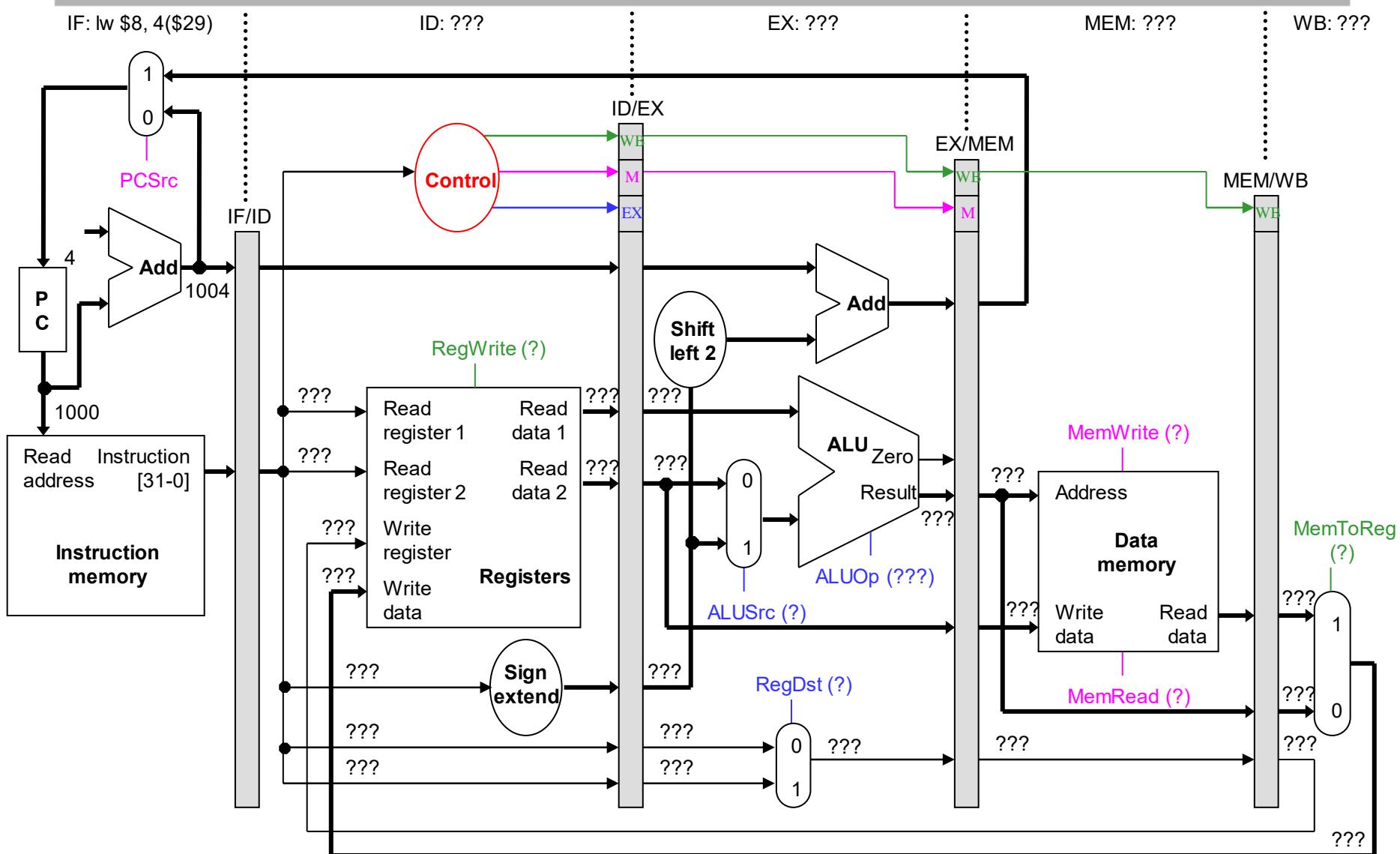
- Here's a sample sequence of instructions to execute.

addresses
in decimal

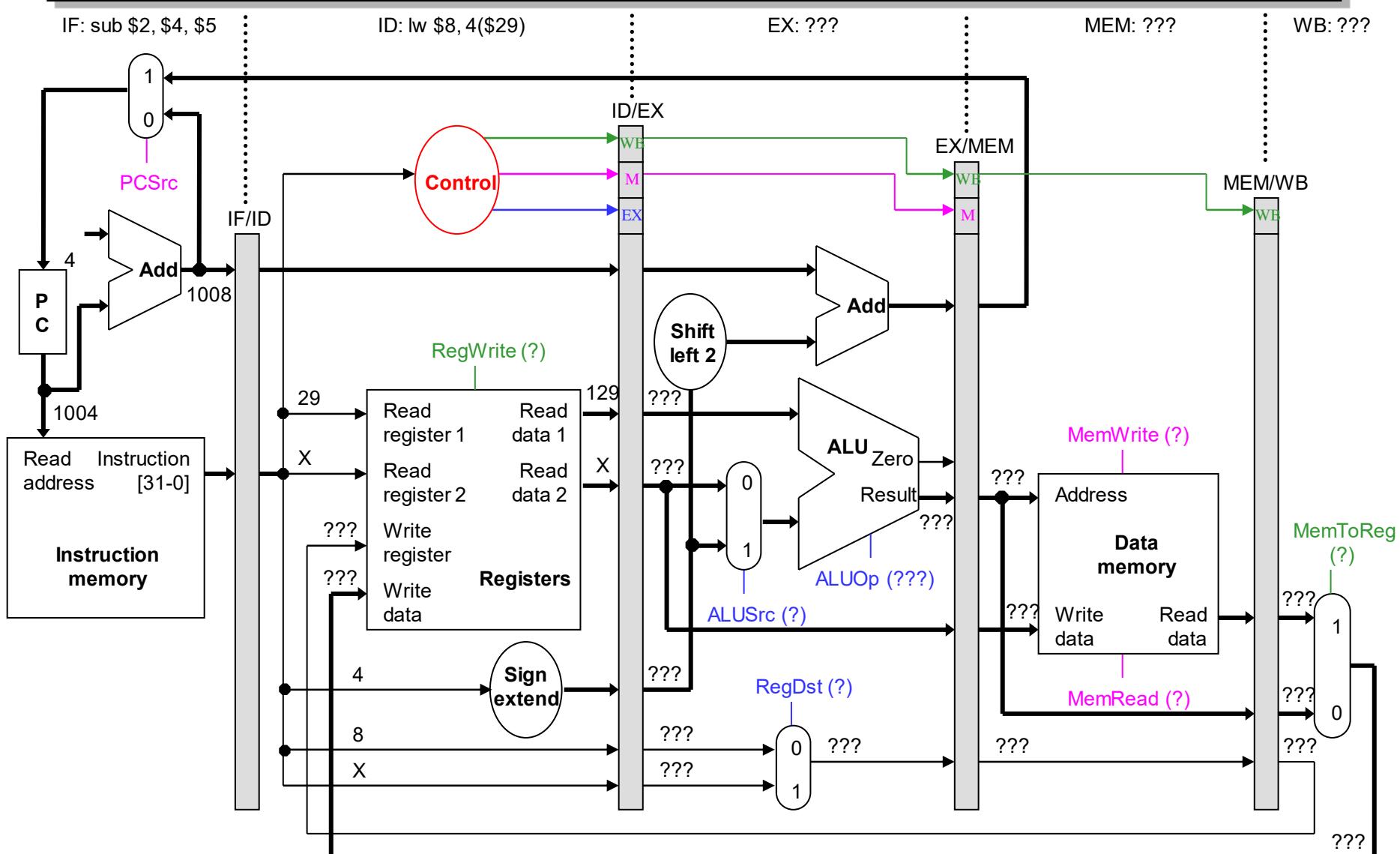
```
1000: lw $8, 4($29)
1004: sub $2, $4, $5
1008: and $9, $10, $11
1012: or $16, $17, $18
1016: add $13, $14, $0
```

- We'll make some assumptions, just so we can show actual data values.
 - Each register contains its number plus 100. For instance, register \$8 contains 108, register \$29 contains 129, and so forth.
 - Every data memory location contains 99.
- Our pipeline diagrams will follow some conventions.
 - An X indicates values that aren't important, like the constant field of an R-type instruction.
 - Question marks ?? indicate values we don't know, usually resulting from instructions coming before and after the ones in our example.

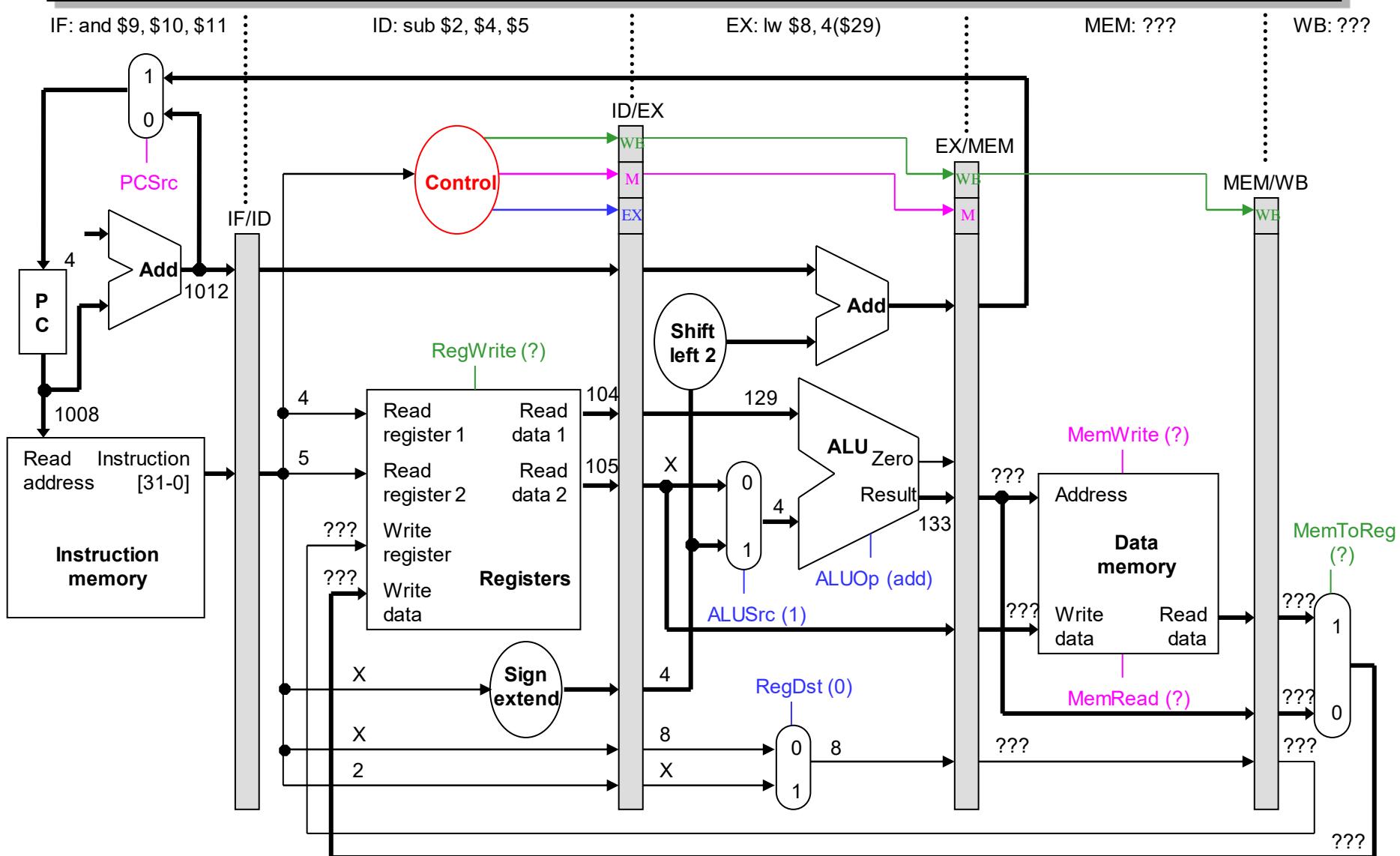
Cycle 1 (filling)



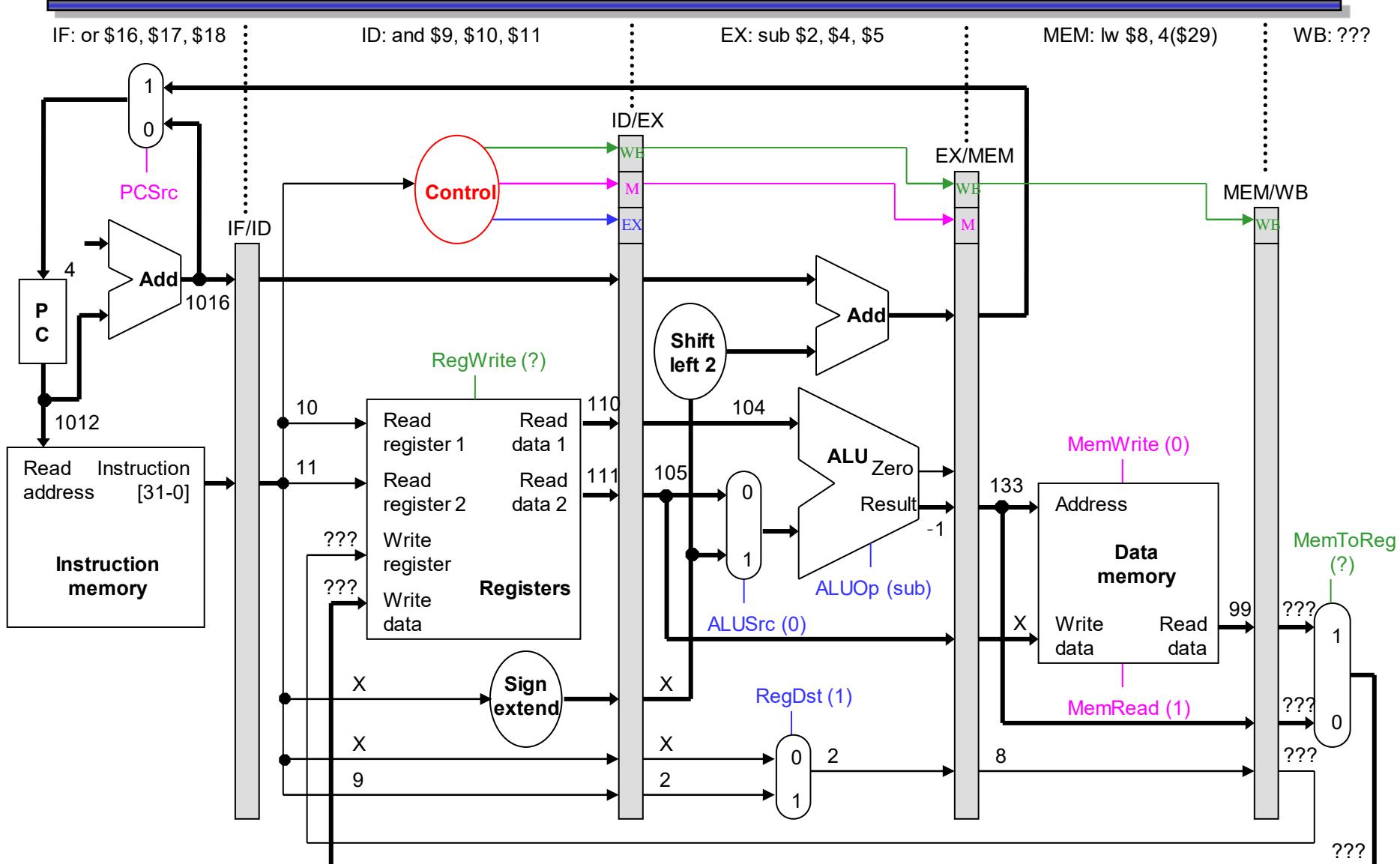
Cycle 2



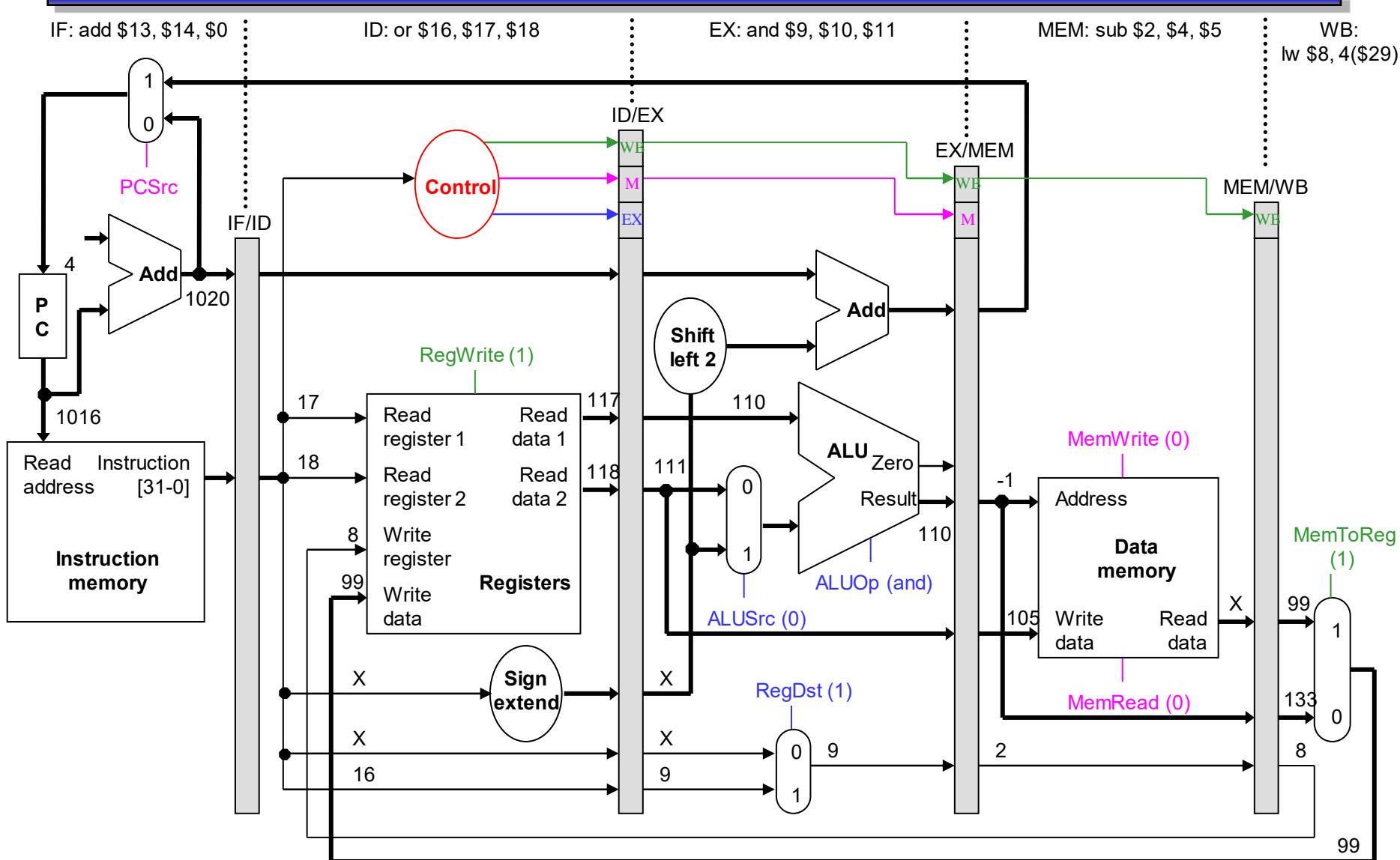
Cycle 3



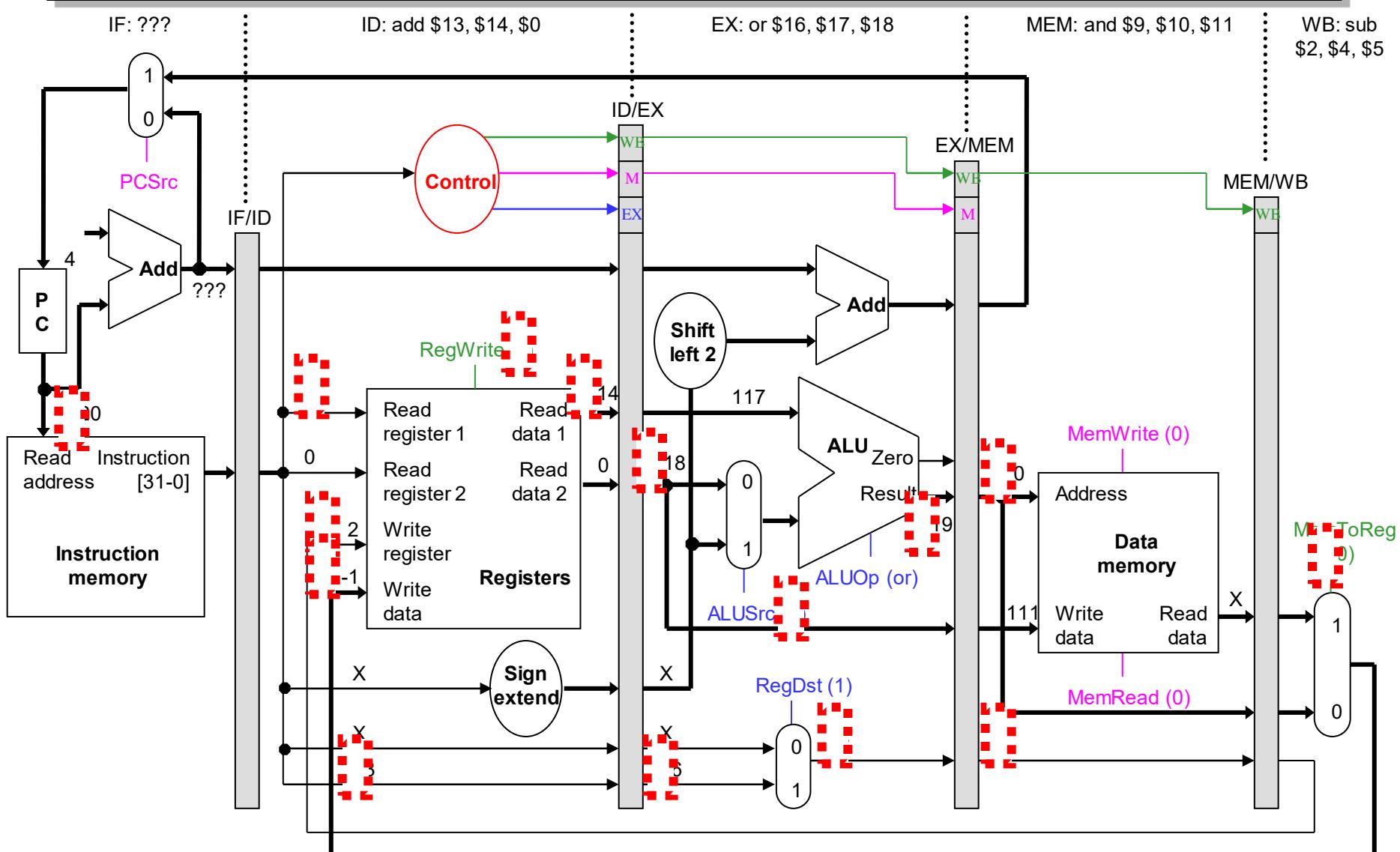
Cycle 4



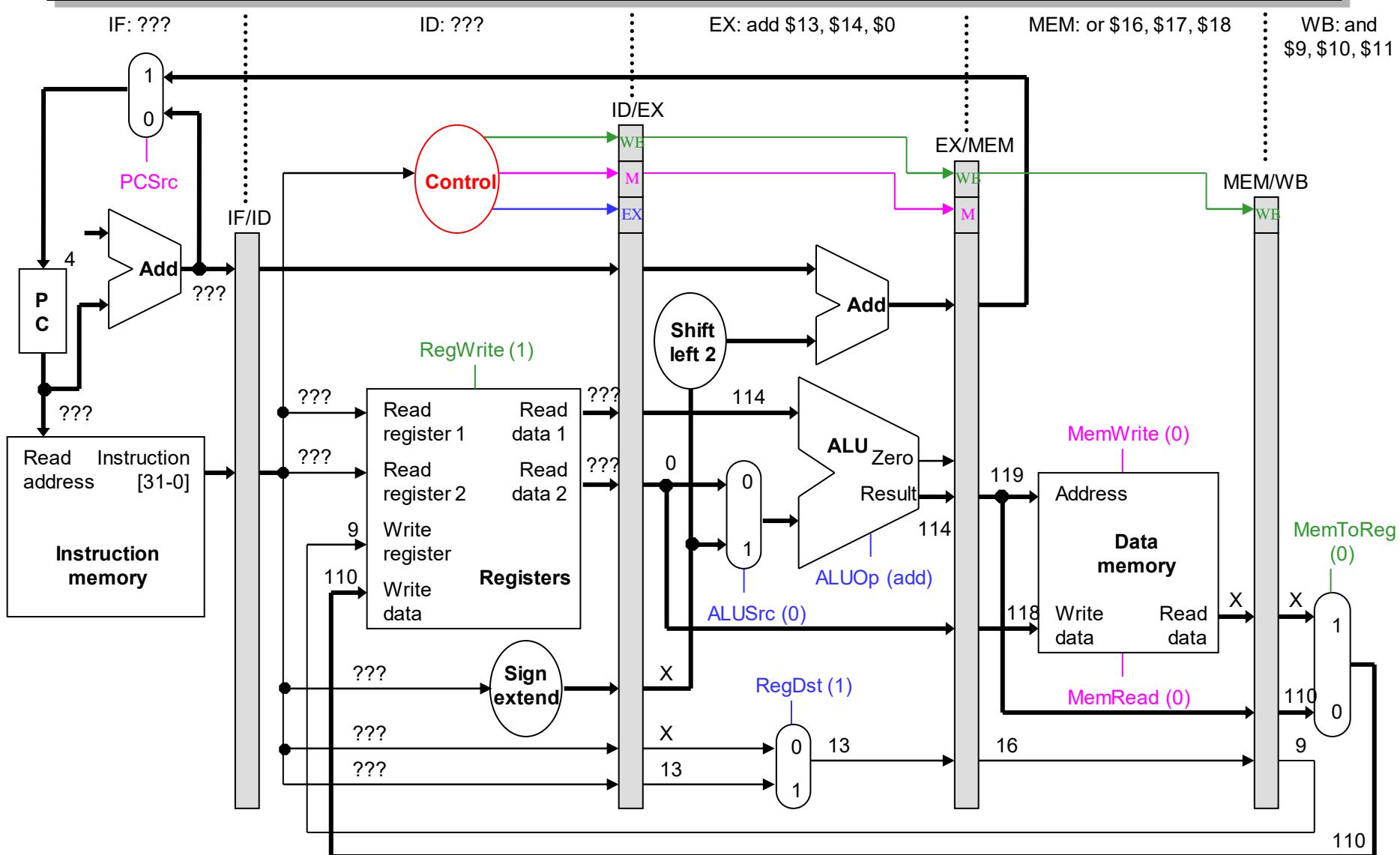
Cycle 5 (full)



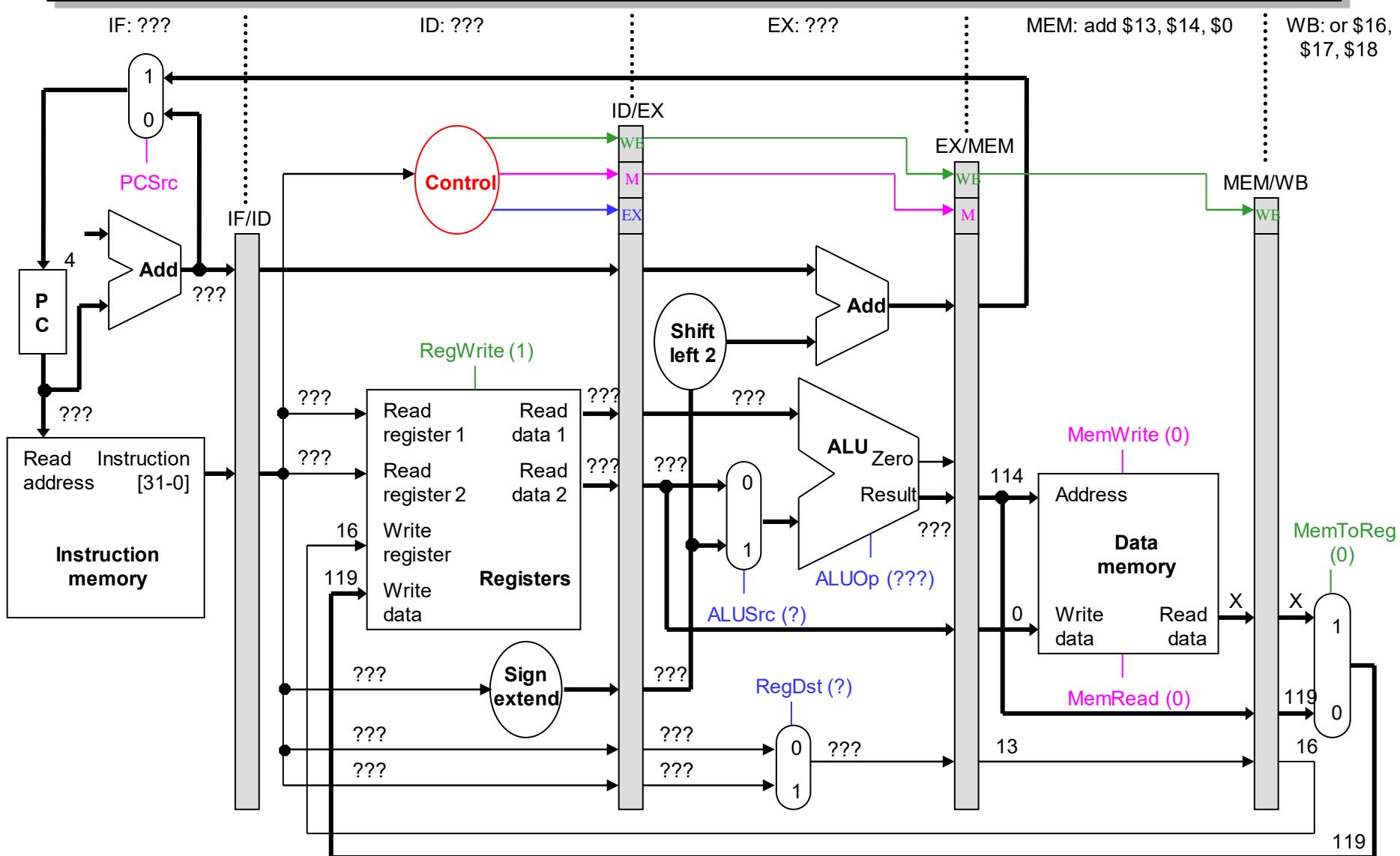
Cycle 6 (emptying)



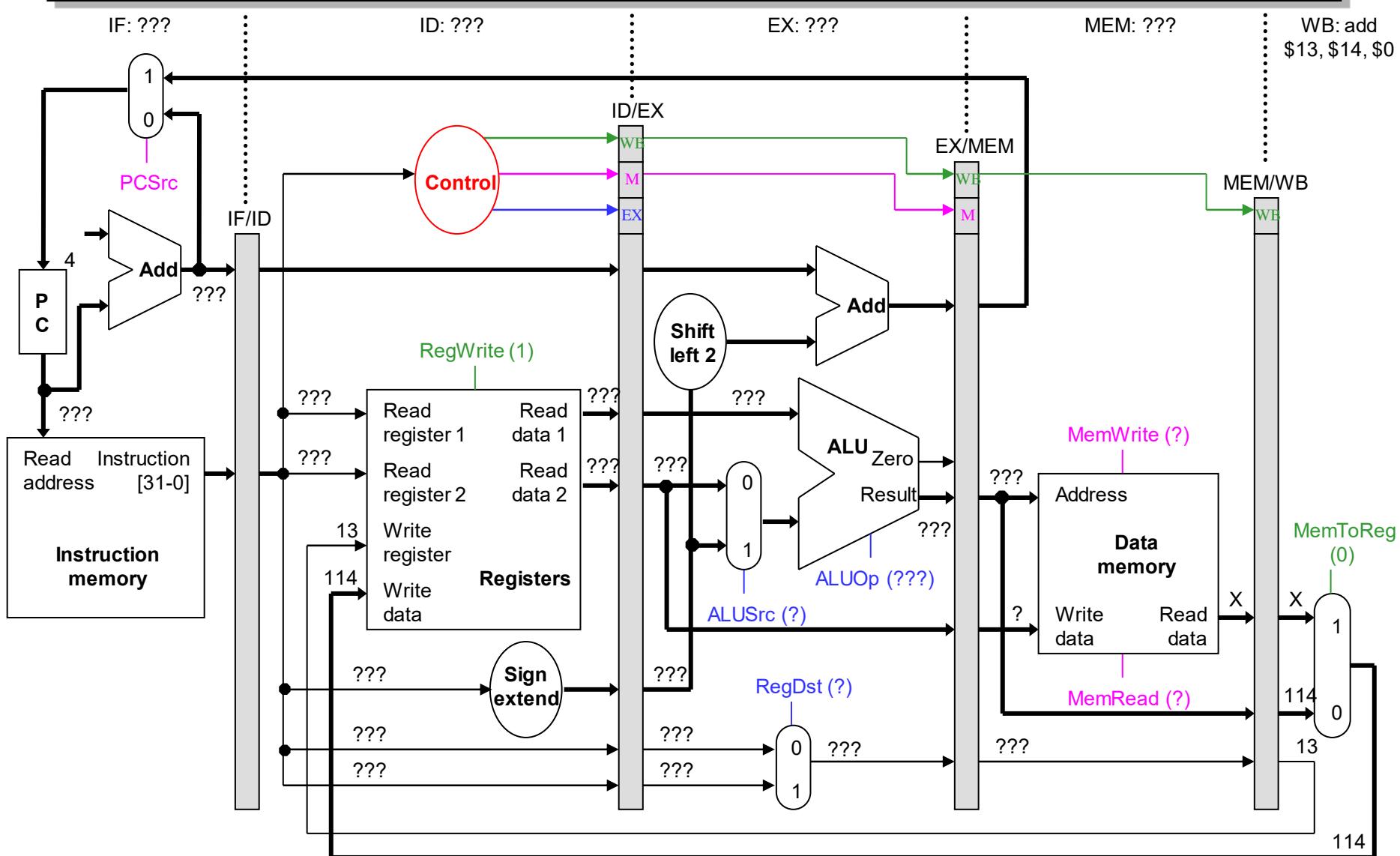
Cycle 7



Cycle 8



Cycle 9



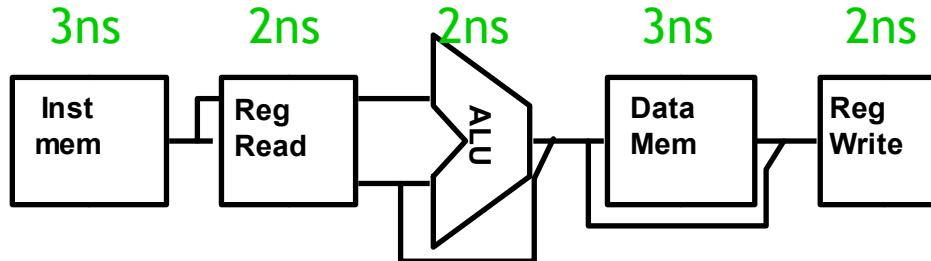
That's a lot of diagrams there

	Clock cycle								
	1	2	3	4	5	6	7	8	9
lw	\$t0, 4(\$sp)	IF	ID	EX	MEM	WB			
sub	\$v0, \$a0, \$a1		IF	ID	EX	MEM	WB		
and	\$t1, \$t2, \$t3			IF	ID	EX	MEM	WB	
or	\$s0, \$s1, \$s2				IF	ID	EX	MEM	WB
add	\$t5, \$t6, \$0					IF	ID	EX	MEM
									WB

- Compare the last nine slides with the pipeline diagram above.
 - You can see how instruction executions are overlapped.
 - Each functional unit is used by a *different* instruction in each cycle.
 - The pipeline registers save control and data values generated in previous clock cycles for later use.
 - When the pipeline is full in clock cycle 5, all of the hardware units are utilized. This is the ideal situation, and what makes pipelined processors so fast.
- Try to understand this example or the similar one in the book at the end of Section 6.3.

Performance Revisited

- Assuming the following functional unit latencies:



- What is the cycle time of a **single-cycle implementation**?
 - What is its throughput?
- What is the cycle time of a ideal **pipelined implementation**?
 - What is its steady-state throughput?
- How much faster is pipelining?

9. The Memory Hierarchy (3)

Main Memory

Main memory is the name given to the level below the cache(s) in the memory hierarchy. There is a large variety of dimensions, but a smaller one in speed due to the fact that vendors use the same chips to build memory arrays. A main memory may have a few MBytes for a typical Personal Computer, tens to hundreds of MBytes for a workstation, hundreds of MBytes to GBytes for supercomputers. The capacity of main memory has continuously increased over the years, as prices have dramatically dropped. The main memory must satisfy the cache requests as quickly as possible (the main memory should have a low latency), and must provide sufficient bandwidth for I/O devices and for vector units (if it is the case).

The **access time**, is defined as the time between the moment the read command is issued and the moment the requested data is at outputs.

The **cycle time** is defined as the minimum time between successive accesses to memory. The cycle time is usually greater than the access time.

9.1 DRAM/SRAM

To access data in a memory chip with a capacity of NxM bits, one must provide a number of addresses equal to:

$$\log_2 N$$

N is the number of “words” each chip has; each “word” is M bits wide. As the technology improved, the packaging costs become a real concern, as the number of address lines got greater and greater.

Example 9.1 ADDRESS LINES:

Which is the number of address lines needed for a 4 Mbit memory chip:

- a) organized as 4Mx1;
- b) organized as 1Mx4?

Answer:

a) $4 M = 2^{22}$ hence the number of address lines is

$$\log_2 2^{22} = 22$$

b) $1 M = 2^{20}$ therefore the number of address lines is

$$\log_2 2^{20} = 20$$

To keep memory chips cheap, the solution adopted for Dynamic RAM (DRAM) integrated circuits was to multiplex the address, thus reducing the number of pins for addresses to half, and adding two new control lines: **RAS** (Row Address Strobe), which loads into an internal buffer half of the address supplied by the control on the address lines, and **CAS** (Column Address Strobe) which handles the second half of the address.

Example 9.2 NUMBER OF PINS:

How many pins has a 1Mx1 memory chip:

- a) in DRAM technology;
- b) in SRAM technology?

Answer:

Organization is important because it says how many pins are needed for data input and data output lines; it still does not say everything about the chip, more precisely it does not say if the input and output lines are the same or are separate.

Let's suppose that the chip in this example has one input line and one output line.

a) $1 M = 2^{20}$

$$n_A = \log_2 2^{20} = 20 \text{ addresses}$$

a)For DRAM the number of address lines (pins) is half of the address size:

- 10 address lines
- 1 RAS
- 1 CAS
- 1 WE (Write Enable)
- 1 Din (the data input line)
- 1 Dout (the data output line)
- 2 for power supply

Total 17 pins needed for 1Mx1 DRAM. A real circuit has 18 pins, with one pin unused but devoted to use as an address line in the 4Mx1 chips.

b)For SRAM the number of address lines is the same as the address size:

- 20 address lines
- 1 WE
- 1 Din
- 1 Dout
- 2 for power supply

Total 25 pins needed for a 1Mx1 SRAM which fits in a 26 pin chip.

The reason for which SRAM address lines are not multiplexed is **speed**; the package is however more expensive than the package for a DRAM with the same capacity.

Another problem the designer faces, when using DRAM circuits is the refresh: each row in a DRAM circuit has to be accessed within some time interval (say 2 milliseconds), to prevent data from getting lost. This is a consequence of the dynamic technology, where data is stored as electric charge in a small capacitor: due to unavoidable losses in the dielectric of the capacitor, the charge decreases in time; the purpose of refresh is to periodically restore the charge on the capacitors (a charged capacitor stores a 1) thus preserving data.

The refresh requirement implies that the memory will be sometimes unavailable, being busy to do refresh: usually it takes a memory cycle to do a refresh to one row in DRAM. The number of refresh cycles in the critical time interval is a circuit specification, and can be found in the circuit's data sheet. Multiplexed address also mean an involved timing for DRAM memories; the cycle time is always larger than the access time.

SRAM use more transistors per memory-bit to prevent the loss of data. The cycle time for a static memory is very close to the access time, and is in the range of nanoseconds to tens of nanoseconds.

With the today's technology the maximum capacity of DRAMs is 16 times larger than that of SRAMs; the cycle time for a SRAM is, on the other hand, 8 to 16 times shorter than that of a DRAM.

As we saw in chapter 8, the faster a CPU is, the higher the miss penalty is. DRAM capacity has increased, over the last decade, quadrupling every three years. Unfortunately this is not the case with the DRAM performance: the 64 Kbit circuit, introduced in 1980, had a cycle time of 250 ns, while the 4 Mbit circuit introduced in 1989 has a cycle time of 165 ns; the capacity is 64 times larger but the performance is only 51% better (figures for the access time are quite similar).

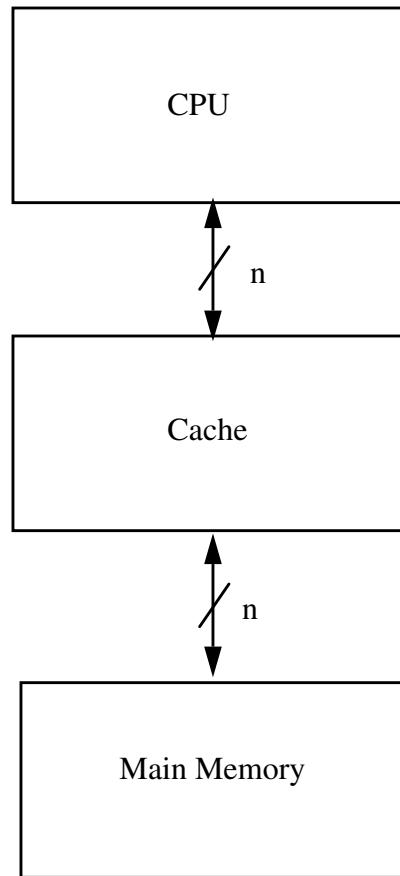


FIGURE 9.1 Connecting the Main memory to the CPU and cache; all buses have the same width.

9.2 Possible Organizations for Main Memory

We shall compare different memory organizations based on the following assumptions:

- all transfers are multiple of word (1 word = 4 bytes);
- 1 clock cycle to send the address;
- 10 clock cycles for the access time;
- 1 clock cycle for a bus transfer of the accessed item.

The simple and cheap approach for memory organization is to have transfers, between all levels of the memory hierarchy, the same size, as depicted in Figure 9.1.

Example 9.3 MEMORY ORGANIZATION:

Compute the miss penalty and the memory bandwidth for a word organized memory system. The cache block size is 8 words (32 bytes).

Answer:

For each word in the block the address must be transmitted (1 clock cycle), a fixed amount of time has to be spent waiting (10 clock cycles), and each word has to be transferred into the cache (1 clock cycle); therefore the miss penalty is:

$$\text{miss_penalty} = 8 * (1 + 10 + 1) = 96 \text{ clock cycles}$$

The memory bandwidth is:

$$\text{memory_bandwidth} = \frac{\text{bytes_transferred}}{\text{clock_cycles}}$$

$$\text{memory_bandwidth} = \frac{32}{96} = 0.33 \text{ bytes/clock cycle}$$

There are two parameters that can be modified to obtain a larger memory bandwidth: to increase the number of bytes transferred in the same amount of time, and to decrease the number of clock cycles necessary to complete a block transfer. The two possibilities correspond to two different memory organizations: a wider memory and interleaved memory respectively.

Wider Main Memory

The basic organization of a wider memory is presented in Figure 9.2. The data bus between the cache and main memory is wider than the bus between the cache and the CPU (the size of this one is the size of the

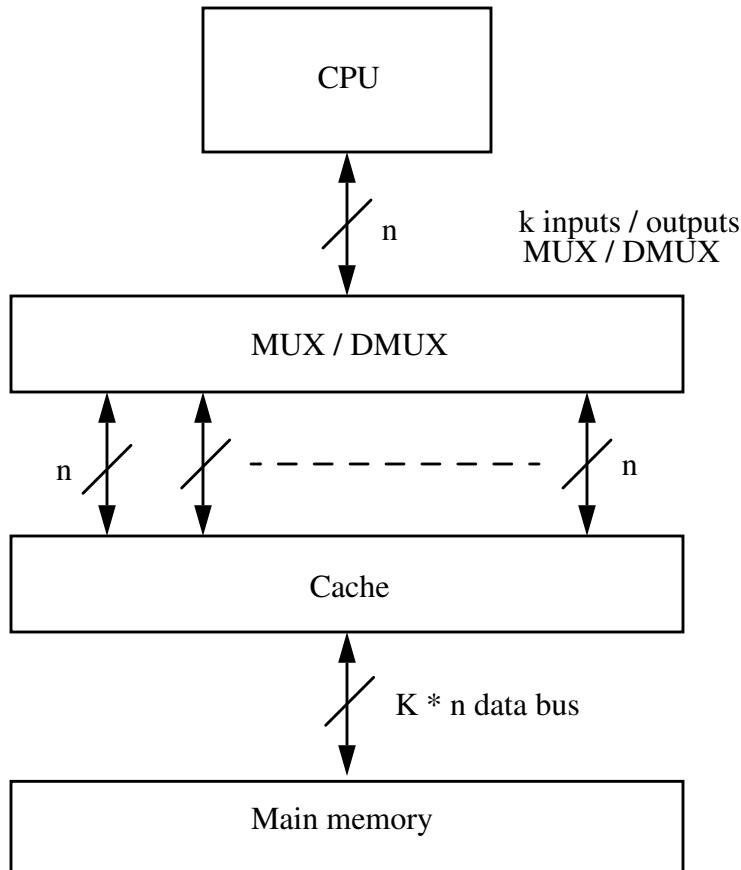


FIGURE 9.2 Wide memory organization. $k * n$ bits are transferred between main memory and the cache, but only n bits between CPU and the cache.

datapath). In this case the address no longer indicates a word in main memory but a block; it is, in other words, the block-frame address that comes to main memory.

Example 9.4 MISS PENALTY AND MEMORY BANDWIDTH:

Calculate the miss penalty and the bus bandwidth for a wide memory organization. The cache line is 8 words wide, and the data bus is also 8 words wide.

Answer:

It takes one clock cycle to transmit the address, 10 clock cycles to access a line in the memory (8 words are accessed at once), and one more clock cycle to transfer the line into the cache; hence, the miss penalty is:

miss_penalty = 1 + 10 + 1 = 12 clock cycles

and the memory bandwidth is:

$$\text{memory_bandwidth} = \frac{\text{bytes_transferred}}{\text{clock_cycles}}$$

$$\text{memory_bandwidth} = \frac{32}{12} = 2.67 \text{ bytes/clock cycle}$$

The memory bandwidth is 8 times larger than for the word organized memory.

There is a price to be paid for better performance: because the CPU reads only n bits at a time and the cache is n*k bits organized, there must be a multiplexer between the cache and the CPU. Incidentally this is the case with cache represented in Figure 8.5 where we have a multiplexer that selects the proper word from the block (note also that, if the memory is byte addressable the scheme gets more involved, in that the type of item being addressed must be stated).

Another problem with the wide memories is related to the price paid by the customer: if the chips available for expansion have a NxM capacity, then the number of circuits the user must add into the system is a multiple of:

$$n*k / M$$

because the user has to add complete lines to the memory for expansion.

Interleaved Memory

The memory can be organized in banks, each bank one word wide, such that transfers between the cache and main memory are word wide, but several words can be read at once and then transferred one after the other to the cache. Figure 9.3 presents a interleaved memory organization.

Having an interleaved memory there is only one address that the CPU has to supply to main memory: this address will access a word in the bank number:

(address) modulo (number of banks)

while the other banks will access words at addresses successive to the address issued by the CPU.

Example 9.5 MISS PENALTY AND MEMORY BANDWIDTH:

Compute the miss penalty and the memory bandwidth for a 4 bank main memory; each bank is one word wide. The cache is 8 words wide.

Answer:

Because the memory has 4 banks, there will be 4 words read in a single burst; this takes one clock cycle to send the address, 10 clock cycles waiting time for the memories to access date, and 4 clock cycles to read the four word coming from the four banks. Since the cache block is 8 words wide this process has to be repeated; therefore, the miss penalty is:

$$\text{miss_penalty} = 2 * (1 + 10 + 4 * 1) = 30 \text{ clock cycles}$$

and the memory bandwidth is:

$$\text{memory_bandwidth} = \frac{\text{bytes_transferred}}{\text{clock_cycles}}$$

$$\text{memory_bandwidth} = \frac{32}{30} = 1.1 \text{ bytes/clock cycle (roughly)}$$

Interleaving gives fairly good performance as compared with a word organized main memory, still having the advantage that it does not require a wider data bus. The manner in which addresses are mapped to banks affect the memory's behavior; mapping word addresses to banks, with banks word wide, is a natural solution for the today's 32 bit machines that access the most frequent words in the memory.

An interleaved memory behaves fine in the case of cache misses due to the fact that words are read sequentially, and transferred one at a time to the cache; it is also attractive for write-back caches, as the words in a block are written sequentially into the memory, with the price of a single access time.

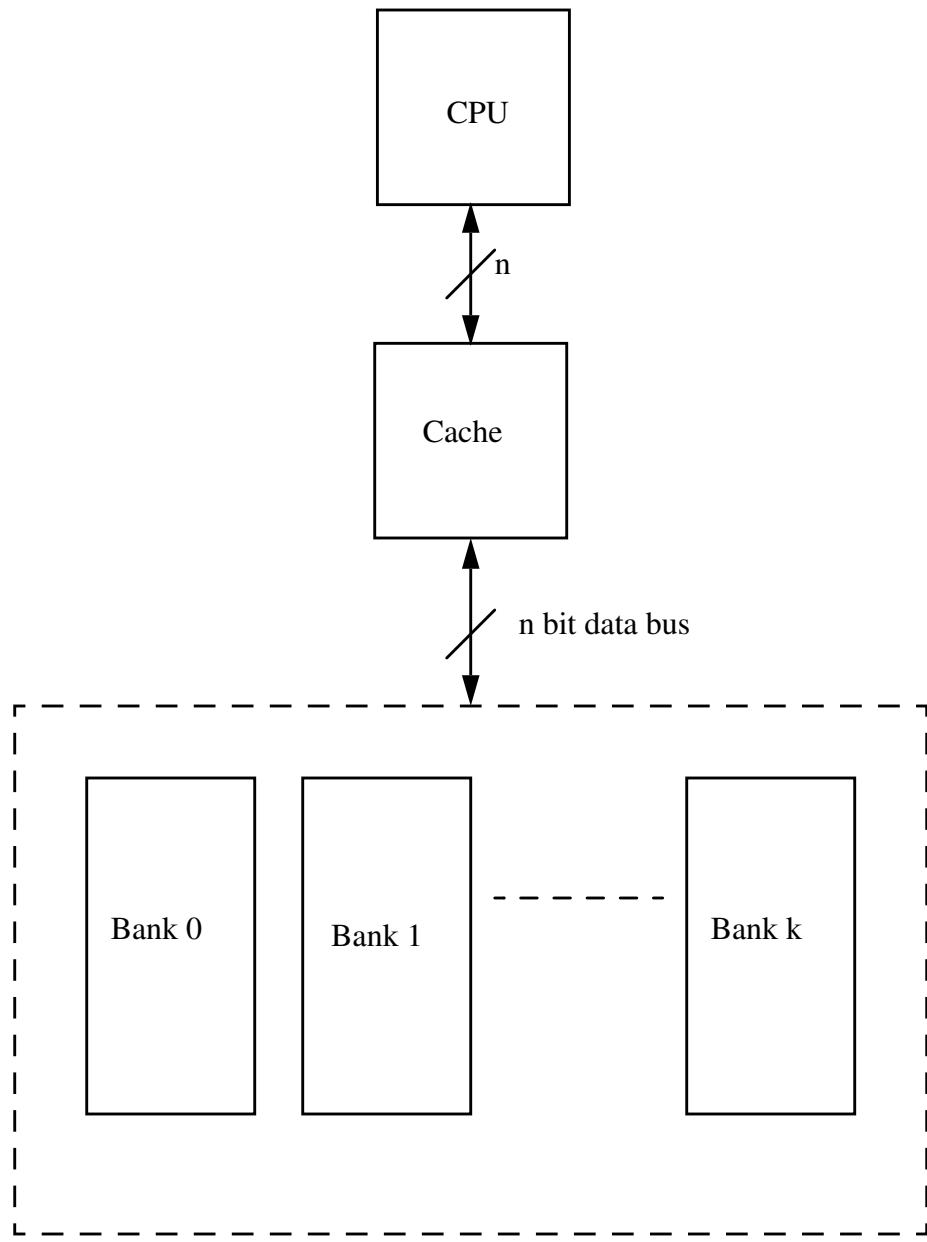


FIGURE 9.3 Interlaced memory organization. CPU, cache and each memory bank have the same width.

There is however the same drawback as for the wide memory organization, the price that the user must pay to increase the system's memory capacity. In this case the user must add the same number of chips in each bank, for a memory upgrade. Moreover, as the memory chips get larger capacities, it is more difficult to organize the chips in banks, as the following example suggests.

Example 9.6 MEMORY CAPACITY AND NUMBER OF CHIPS:

The maximum memory a PC can address is 16 MBytes; you have to design a four bank interleaved memory for this system. Each bank is byte organized. How many chips are necessary, using:

- a) 1Mx1 chips;
- b) 4Mx1 chips;
- c) 16Mx1 chips?

Answer:

The capacity of a bank is:

$$\text{bank_capacity} = \frac{\text{memory_capacity}}{\text{number_of_banks}}$$

$$\text{bank_capacity} = \frac{16\text{MB}}{4} = 4\text{MB/bank}$$

a)

$$n_a = (4 \text{ banks}) * \left(\frac{4\text{Mx8}}{1\text{Mx1}} \text{circuits per bank} \right) = 4 * 32 = 128 \text{ circuits}$$

b)

$$n_b = (4 \text{ banks}) * \left(\frac{4\text{Mx8}}{4\text{Mx1}} \text{circuits per bank} \right) = 4 * 8 = 32 \text{ circuits}$$

c)

$$n_a = (4 \text{ banks}) * \left(\frac{4\text{Mx8}}{16\text{Mx1}} \text{circuits per bank} \right) = 4 * 2 = 8 \text{ circuits ??}$$

We are in trouble because we need at least 8 circuits per bank to ensure the proper number of inputs and outputs (using 16Mx1 circuits); using 8 such circuits per bank means a capacity of:

$$8 * 16\text{Mx1} = 16 \text{ MByte/bank}$$

If the system can access only 16 MB, it results that 3/4 of the main memory is inaccessible.

Given the actual conditions in which the CPU performance increases at a faster pace than the memory performance, memory organizations that reduce the cache penalty tend to become common place.

Exercises

9.1 A memory hierarchy is being designed for a system. The following possibilities have to be investigated:

- a) cache block size is 1 word, miss rate is 20%
- b) cache block size is 4 words, miss rate is 10%
- c) cache block size is 8 words, miss rate is 2%

In every case there are 1.4 memory accesses per instruction. Which is the best choice for the main memory? State your assumptions.

9.2 Explore the possibility of using some of the features the new DRAM circuits offer, to improve the memory performance: here are some of the standard DRAM improvements you might want to consider:

- a) nibble mode;
- b) page mode;
- c) static column.

9.3 A new idea being studied is to move the cache closer to the memory, more precisely on the same memory chip die; this is tempting because, in the case of read, a whole row is accessed: for a $1M \times 1$ DRAM memory a row is 1024 bits wide (supposing the die is square). How do you think could this improve the memory performance? For a good introduction in this, you could consider a series of articles in the IEEE Spectrum, October 1992.

Performance Metrics

Why study performance metrics?

- determine the benefit/lack of benefit of designs
- computer design is too complex to intuit performance & performance bottlenecks
- have to be careful about what you mean to measure & how you measure it

What you should get out of this discussion

- good metrics for measuring computer performance
- what they should be used for
- what metrics you shouldn't use & how metrics are misused

Performance of Computer Systems

Many different factors to take into account when determining performance:

- Technology
 - circuit speed (clock, MHz)
 - processor technology (how many transistors on a chip)
- Organization
 - type of processor (ILP)
 - configuration of the memory hierarchy
 - type of I/O devices
 - number of processors in the system
- Software
 - quality of the compilers
 - organization & quality of OS, databases, etc.

“Principles” of Experimentation

Meaningful metrics

execution time & component metrics that explain it

Reproducibility

machine configuration, compiler & optimization level, OS, input

Real programs

no toys, kernels, synthetic programs

SPEC is the norm (integer, floating point, graphics, webserver)

TPC-B, TPC-C & TPC-D for database transactions

Simulation

long executions, **warm start** to mimic **steady-state** behavior

usually applications only; some OS simulation

simulator “validation” & internal checks for accuracy

Metrics that Measure Performance

Raw speed: peak performance (never attained)

Execution time: time to execute one program from beginning to end

- the “performance bottom line”
- wall clock time, response time
- Unix time function: 13.7u 23.6s 18:27 3%

Throughput: total amount of work completed in a given time

- transactions (database) or packets (web servers) / second
- an indication of how well hardware resources are being used
- good metrics for chip designers or managers of computer systems

(Often improving execution time will improve throughput & vice versa.)

Component metrics: subsystem performance, e.g., memory behavior

- help explain how execution time was obtained
- pinpoints performance bottlenecks

Execution Time

$$\text{Performance}_a = 1 / (\text{Execution Time}_a)$$

Processor A is faster than processor B, i.e.,

$$\text{Execution Time}_A < \text{Execution Time}_B$$

$$\text{Performance}_A > \text{Performance}_B$$

Relative Performance

$$\text{Performance}_A / \text{Performance}_B$$

$$= n$$

$$= \text{ExecutionTime}_B / \text{ExecutionTime}_A$$

performance of A is n times greater than B

execution time of B is n times longer than A

CPU Execution Time

The time the CPU spends executing an application

- no memory effects
- no I/O
- no effects of multiprogramming

$$\text{CPUExecutionTime} = \text{CPUClockCycles} * \text{ClockCycleTime}$$

Cycle time (clock period) is measured in time or rate

- clock cycle time = 1/clock cycle rate

$$\text{CPUExecutionTime} = \text{CPUClockCycles} / \text{ClockCycleRate}$$

- clock cycle rate of 1 MHz = cycle time of 1 μs
- clock cycle rate of 1 GHz = cycle time of 1 ns

CPI

$$CPUClockCycles = \text{NumberOfInstructions} * CPI$$

Average number of clock cycles per instruction

- throughput metric
- component metric, not a measure of performance
- used for processor organization studies, given a fixed compiler & ISA

Can have different CPIs for classes of instructions

e.g., floating point instructions take longer than integer instructions

$$CPUClockCycles = \sum_1^n (CPI_i \times C_i)$$

where CPI_i = CPI for a particular class of instructions

where C_i = the number of instructions of the i^{th} class that have been executed

Improving part of the architecture can improve a CPI_i

- Talk about the contribution to CPI of a class of instructions

CPU Execution Time

$$\text{CPUExecutionTime} = \text{numberofInstructions} * \text{CPI} * \text{clockCycleTime}$$

To measure:

- execution time: depends on all 3 factors
 - time the program
- number of instructions: determined by the ISA
 - programmable hardware counters
 - profiling
 - count number of times each basic block is executed
 - instruction sampling
- CPI: determined by the ISA & implementation
 - simulator: interpret (in software) every instruction & calculate the number of cycles it takes to simulate it
- clock cycle time: determined by the implementation & process technology

Factors are interdependent:

- RISC: increases instructions/program, but decreases CPI & clock cycle time because the instructions are simple
- CISC: decreases instructions/program, but increases CPI & clock cycle time because many instructions are more complex

Metrics Not to Use

MIPS (millions of instructions per second)

$$\text{instruction count} / \text{execution time} * 10^6 = \\ \text{clock rate} / (\text{CPI} * 10^6)$$

- instruction set-dependent (even true for similar architectures)
- implementation-dependent
- compiler technology-dependent
- program-dependent
- + intuitive: the higher, the better

MFLOPS (millions of floating point operations per second)

$$\text{floating point operations} / (\text{execution time} * 10^6)$$

- + FP operations are independent of FP instruction implementation
- different machines implement different FP operations
- different FP operations take different amounts of time
- only measures FP code

static metrics (code size)

Means

Measuring the performance of a workload

- **arithmetic:** used for averaging execution times

$$\left(\sum_{i=1}^n time_i \right) \times \frac{1}{n}$$

- **harmonic:** used for averaging rates ("the average of", as opposed to "the average statistic of")

$$\frac{p}{\left(\sum_{i=1}^p \frac{1}{rate_i} \right)}$$

- **weighted means:** the programs are executed with different frequencies, for example:

$$\left(\sum_{i=1}^n time_i \times weight_i \right) \times \frac{1}{n}$$

Means

	FP Ops	Time (secs)		
		Computer A	Computer B	Computer C
program 1	100	1	10	20
program 2	100	1000	100	20
total		1001	110	40
arith mean		500.5	55	20

	FP Ops	Rate (FLOPS)		
		Computer A	Computer B	Computer C
program 1	100	100	10	5
program 2	100	.1	1	5
harm mean		.2	1.5	5
arith mean		50.1	5.5	5

Computer C is ~25 times faster than A when measuring execution time

Still true when measuring MFLOPS(a rate) with the harmonic mean

Speedup

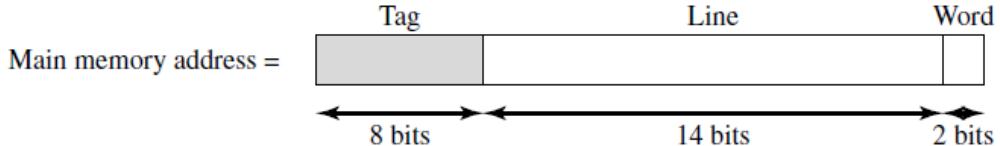
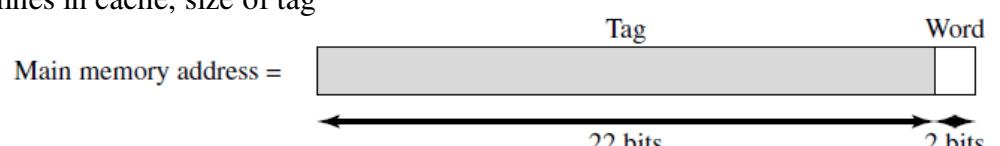
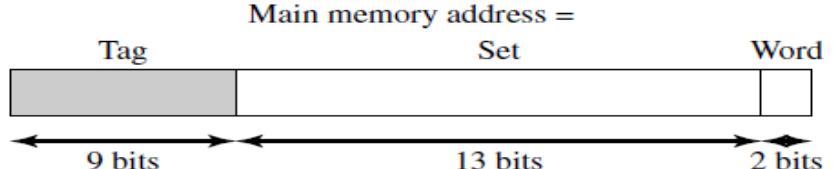
$$\text{Speedup} = \frac{\text{Execution Time}_{\text{beforeImprovement}}}{\text{ExecutionTime}_{\text{afterImprovement}}}$$

Amdahl's Law:

Performance improvement from speeding up a part of a computer system is limited by the proportion of time the enhancement is used.



Problem Set

- c. Tag, Set, and Word values for a two-way set-associative cache, where tag – 9 bits, set – 13 bits, word – 2 bits
16. List the following values:
- d. For the direct cache from the below Fig: address length, number of addressable units, block size, number of blocks in main memory, number of lines in cache, size of tag
- Main memory address = 
- e. For the associative cache from below Figure: address length, number of addressable units, block size, number of blocks in main memory, number of lines in cache, size of tag
- Main memory address = 
- f. For the two-way set-associative cache example of Figure 4.15: address length, number of addressable units, block size, number of blocks in main memory, number of lines in set, number of sets, number of lines in cache, size of tag
- Main memory address = 
17. Consider a 32-bit microprocessor that has an on-chip 16-KByte four-way set-associative cache. Assume that the cache has a line size of four 32-bit words. Draw a block diagram of this cache showing its organization and how the different address fields are used to determine a cache hit/miss. Where in the cache is the word from memory location ABCDE8F8 mapped?
18. Consider a machine with a byte addressable main memory of 216 bytes and block size of 8 bytes. Assume that a direct mapped cache consisting of 32 lines is used with this machine.
- g. How is a 16-bit memory address divided into tag, line number, and byte number?
- h. Into what line would bytes with each of the following addresses be stored?
- 0001 0001 0001 1011
 1100 0011 0011 0100
 1101 0000 0001 1101
 1010 1010 1010 1010
- i. Suppose the byte with address 0001 1010 0001 1010 is stored in the cache. What are the addresses of the other bytes stored along with it?
- j. How many total bytes of memory can be stored in the cache?
- k. Why the tag is also stored in the cache?

19. A set-associative cache has a block size of four 16-bit words and a set size of 2. The cache can accommodate a total of 4096 words. The main memory size that is cacheable is $64K \times 32$ bits. Show how the processor's addresses are interpreted?
20. Consider a memory system that uses a 32-bit address to address at the byte level, plus a cache that uses a 64-byte line size.
1. Assume a direct mapped cache with a tag field in the address of 20 bits. Show the address format and determine the following parameters: number of addressable units, number of blocks in main memory, number of lines in cache, size of tag.
 - m. Assume an associative cache. Show the address format and determine the following parameters: number of addressable units, number of blocks in main memory, number of lines in cache, size of tag.
 - n. Assume a four-way set-associative cache with a tag field in the address of 9 bits. Show the address format and determine the following parameters: number of addressable units, number of blocks in main memory, number of lines in set, number of sets in cache, number of lines in cache, size of tag.
21. Consider a computer with the following characteristics: total of 1Mbyte of main memory; word size of 1 byte; block size of 16 bytes; and cache size of 64 Kbytes.
- o. For the main memory addresses of F0010, 01234, and CABBE, give the corresponding tag, cache line address, and word offsets for a direct-mapped cache.
 - p. Give any two main memory addresses with different tags that map to the same cache slot for a direct-mapped cache.
 - q. For the main memory addresses of F0010 and CABBE, give the corresponding tag and offset values for a fully-associative cache.
 - r. For the main memory addresses of F0010 and CABBE, give the corresponding tag, cache set, and offset values for a two-way set-associative cache.
22. Consider the following code:
- ```
for (i = 0; i < 20; i++)
 for (j = 0; j < 10; j++)
 a[i] = a[i] * j
```
- a. Give one example of the spatial locality in the code.
  - b. Give one example of the temporal locality in the code.
23. Consider a memory system with the following parameters:  
 $T_c = 100\text{ns}$ ;  $T_m = 1200\text{ns}$   
If the effective access time is 10% greater than the cache access time, what is the hit ratio H for look through cache?
24. Consider a look through cache with an access time of 1 ns and a hit ratio of H 0.95. Suppose that we can change the cache design (size of cache, cache organization) such that we increase H to 0.97, but increase access time to 1.5 ns. What conditions must be met for this change to result in improved performance?
25. A computer employs RAM chips of  $128 \times 8$  and ROM chips of  $512 \times 8$ . The computer system needs  $256 \times 16$  of RAM,  $1024 \times 16$  of ROM, and two interface units with 256 registers each.

Show the chip layout for the given specifications.

26. A computer employs RAM chips of  $128 \times 8$  and ROM chips of  $512 \times 8$ . The computer system needs  $512 * 8$  of RAM,  $512 \times 16$  of ROM, and two interface units with 256 registers each.

Show the chip layout for the given specifications.

27. If the received data is 101110011001. Determine whether single bit error or more than one bit error occurs. If there is an error in single bit, perform error correction.

28. If the received data is 001101100111. Determine whether single bit error or more than one bit error occurs. If there is an error in single bit, perform error correction.

29. Calculate the check bits for the data bits 101100110101.

30. Compare the performance of FIFO, LRU and optimal page replacement algorithms in terms of hit ratio considering three pages to be in memory at a time for the following sequence.

2 3 2 1 5 2 4 5 3 2 5 2

## ANSWER KEY

|                     |            |                                                                       |                                                        |                               |               |
|---------------------|------------|-----------------------------------------------------------------------|--------------------------------------------------------|-------------------------------|---------------|
| Name of Examination |            | <b>Final Assessment Test (FAT), Fall 2019-20 Semester, (NOV 2019)</b> |                                                        |                               |               |
| <b>Slot: G2</b>     |            | Course Mode: <b>CBL / PBL / RBL</b>                                   |                                                        | <b>Class Number (s): 0616</b> |               |
| Course Code:        | CSE2001    | Course Title:                                                         | <b>COMPUTER ARCHITECTURE AND ORGANIZATION</b>          |                               |               |
| Emp. ID :           | 16386      | Faculty Name:                                                         | D.RUBY                                                 |                               | School: SCOPE |
| Contact No.         | 9043542311 | Email:                                                                | <a href="mailto:ruby.d@vit.ac.in">ruby.d@vit.ac.in</a> |                               |               |

1. The processor A, B C and D has a 2 GHz clock frequency. Find the total execution time for the programme with instruction mix given below. If the CPI of arithmetic instruction was doubled what would be the impact on the execution time of all the processors.

| Processors | Instruction mix / Processor |              |        | CPI        |              |        |
|------------|-----------------------------|--------------|--------|------------|--------------|--------|
|            | Arithmetic                  | Load / Store | Branch | Arithmetic | Load / Store | Branch |
| A          | 2560                        | 1280         | 256    | 1          | 4            | 2      |
| B          | 1280                        | 640          | 128    | 1          | 4            | 2      |
| C          | 640                         | 320          | 64     | 1          | 4            | 2      |
| D          | 320                         | 160          | 32     | 1          | 4            | 2      |

Find the total execution time for this program on A, B, C & D processors. Assume that each processor has a 2 GHz clock frequency. If the CPI of arithmetic instructions was doubled, what would the impact be on the execution time of the program on A, B, C & D processors?

**ANS:** i) Total execution time:

$$A = 4.096 \mu s, B = 2.048 \mu s, C = 1.024 \mu s, D = 0.512 \mu s$$

ii) Execution time after CPI been doubled

$$A = 5.376 \mu s, B = 2.688 \mu s, C = 1.344 \mu s, D = 0.672 \mu s$$

2. Calculate M multiplied by Q using Booth's Algorithm where M = -11 and Q = 27.

**ANS:**

| n | A                | Q                | Q <sub>0</sub> | Comments                           |
|---|------------------|------------------|----------------|------------------------------------|
| 6 | 000000           | 011011           | 0              | Initial Values                     |
| 5 | 001011<br>000101 | 011011<br>101101 | 0<br>1         | A <- A-M<br>Arithmetic Shift Right |

|   |        |        |   |                        |
|---|--------|--------|---|------------------------|
| 4 | 000010 | 110110 | 1 | Arithmetic Shift Right |
| 3 | 110111 | 110110 | 1 | A<-A+M                 |
|   | 111011 | 111011 | 0 | Arithmetic Shift Right |
| 2 | 000110 | 111011 | 0 | A<- A-M                |
|   | 000011 | 011101 | 1 | Arithmetic Shift Right |
| 1 | 000001 | 101110 | 1 | Arithmetic Shift Right |
| 0 | 110110 | 101110 | 1 | A<-A+M                 |
|   | 111011 | 010111 | 0 | Arithmetic Shift Right |

3. Write a program to evaluate the expression  $X = (A+B)*(C/D)$  with one address, two address and three address Instructions.

**ANS:**

| ONE ADDRESS | TWO ADDRESS | THREE ADDRESS |
|-------------|-------------|---------------|
| LOAD A      | MOV T, A    | ADD X, A, B   |
| ADD B       | ADD T, B    | DIV T, C, D   |
| STORE T     | MOV S, C    | MUL X, X, T   |
| LOAD C      | DIV S, D    |               |
| DIV D       | MUL T, S    |               |
| MUL T       | MOV X, T    |               |
| STORE X     |             |               |

4. A. Give a block diagram for a 8M X 32 memory using 512K X 8 RAM chips.

**ANS:**

- 16 rows (of four 512×8 chips) are needed.
- Address lines A18–0 are connected to all chips.
- Address lines A22–19 are connected to a 4-bit decoder to select one of the 16 rows.

B. Multiply the numbers  $(0.5)_{10}$  and  $(-0.4375)_{10}$  using binary floating point multiplication.

$$\text{ANS: } (1.000 \times 2^{-1}) \times (-1.110 \times 2^{-2}) \\ = -1.110 \times 2^{-3}$$

5. The following is a list of 32-bit memory address references, given as word addresses of 8-bit each. 1, 134, 212, 1, 135, 213, 162, 161, 2, 44, 41, 221. For the above references, identify the binary address and the index address given a direct-mapped cache with initially 2-word blocks and a total size of blocks. Assuming the cache to be empty initially, list the hit or miss for cache references.

**ANS:** i) Binary Address: 00000001, 10000110, 11010100, 00000001, 10000111, 11010101, 10100010, 10100001, 00000010, 00101100, 00101001, 11011101

ii) Index: Binary address mod 16

iii) Hit/Miss: M, M, M, H, M, M, M, M, M, M.

6. Consider a two level memory hierarchy of the form (L1, L2) where L1 is connected directly to the CPU. Determine the average cost per bit and average access time for the data given below.

| Level | Capacity | Cost   | Access time | Hit ratio |
|-------|----------|--------|-------------|-----------|
| L1    | 1024     | 0.1000 | $10^{-8}$   | 0.9000    |
| L2    | $2^{16}$ | 0.0100 | $10^{-6}$   | -         |

**ANS:**

$$\text{Average Cost (C)} = (C_1 S_1 + C_2 S_2) / (S_1 + S_2)$$

$$= ((0.1 \times 1024) + (0.01 \times 2^{16})) / (1024 + 2^{16})$$

$$= 0.011384615$$

Average access time

$$T_A = h T_{A1} + (1 - h) T_{A2}$$

$$= 0.9000 \times 10^{-8} + (1 - 0.9000) \times 10^{-6}$$

$$= 1.09 \times 10^{-7}$$

7. It is necessary to transfer 512 words from a backup store to a memory section starting from address 1000 and the transfer is by means of DMA. i) What are the initial values that the CPU must transfer to the DMA controller. ii) Give step by step account of the actions taken during the input of the first two words.

**ANS:**

- i) CPU initiates DMA by Transferring: 512 to the word count register. 1000 to the DMA address register. Bits to the control register to specify a write operation.
- ii)
1. I/O device sends a “DMA request”.
  2. DMA sends BR (bus request) to CPU.
  3. CPU responds with a BG (bus grant).
  4. Contents of DMA address register are placed in address bus.
  5. DMA sends “DMA acknowledge” to I/O device and enables the write control line to memory.
  6. Data word is placed on data bus by I/O device.
  7. Increment DMA address register by 1 and Decrement DMA word count register by 1.

8. Repeat steps 4-7 for each data word Transferred.
8. List and explain the levels of RAID. What is the distinction between parallel access and independent access in context of RAID?

**ANS:**

RAID Level 0: Non-Redundant

RAID Level 1: Mirrored

RAID Level 2: Redundancy through hamming code

RAID Level 3: Bit-interleaved parity

RAID Level 4: Block level parity

RAID Level 5: Block-level distributed parity

9. Assume a pipeline with four stages: fetch instruction (FI), decode instruction and calculate addresses (DA), fetch operand (FO), and execute (EX). Draw a diagram for a sequence of 7 instructions, in which the third instruction is a branch that is taken and in which there are no data dependencies.

**ANS:**

|    | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 |
|----|----|----|----|----|----|----|----|----|----|----|
| I1 | FI | DA | FO | EX |    |    |    |    |    |    |
| I2 |    | FI | DA | FO | EX |    |    |    |    |    |
| I3 |    |    | FI | DA | FO | EX |    |    |    |    |
| I4 |    |    |    | FI | DA | FO |    |    |    |    |
| I5 |    |    |    |    | FI | DA |    |    |    |    |
| I6 |    |    |    |    |    | FI |    |    |    |    |
| I7 |    |    |    |    |    |    | FI | DA | FO | EX |

10. A. Discuss the difference between tightly coupled multiprocessors and loosely coupled multiprocessors from the viewpoint of hardware organization and programming techniques.

**ANS:**

Tightly coupled multiprocessors require that all processes in the system have access to a common global memory. In loosely coupled multiprocessors, the memory is distributed and a mechanism is required to provide message-passing between the processors. Tightly coupled systems are easier to program since no special steps are required to make shared data available to two or more processors. A loosely coupled system requires that sharing of data be implemented by the messages.

- B. What is the use of parity bits in an error correction code? How many check bits are needed if the Hamming error correction code is used to detect single bit errors in a 2048-bit data word?

**ANS:** Need K check bits such that  $2048 + K \leq 2^K - 1$ .

The minimum value of K that satisfies this condition is 12.