



MODULE III



CONTENTS

- [?] Computer abstractions and technology – Introduction**
- [?] Computer architecture –8 Design features, Application program - layers of abstraction, Five key components of a computer, Technologies for building processors and memory, Performance.**
- [?] Instruction set principles – Introduction, Classifying instruction set architectures, Memory addressing, Encoding an instruction set.**

INTRODUCTION


- Although a common set of hardware technologies is used in computers ranging from smart home appliances to cell phones to the largest supercomputers, these different applications have different design requirements and employ the core hardware technologies in different ways.
- Broadly speaking, computers are used in three different classes of applications.





❖ **Personal computers (PCs)**

❖ **Servers**

❖ **Embedded computers**

- 
- ❑ **Personal Computer (PC):** A computer designed for use by an individual, usually incorporating a graphics display, a keyboard, and a mouse.
 - ❑ **Server:** A computer used for running larger programs for multiple users, often simultaneously, and typically accessed only via a network.
 - ❑ **Supercomputer:** A class of computers with the highest performance and cost; they are configured as servers and typically cost tens to hundreds of millions of dollars.
 - ❑ **Embedded Computer:** A computer inside another device used for running one predetermined application or collection of software.

- 
- ❑ Embedded computers include the microprocessors found in your car, the computers in a television set, and the networks of processors that control a modern airplane or cargo ship.
 - ❑ **Personal mobile devices (PMDs):** are small wireless devices to connect to the Internet; they rely on batteries for power, and software is installed by downloading apps. Conventional examples are smart phones and tablets.
 - ❑ **Cloud Computing:** refers to large collections of servers that provide services over the Internet; some providers rent dynamically varying numbers of servers as a utility.



❓ **Software as a Service (SaaS):** delivers software and data as a service over the Internet, usually via a thin program such as a browser that runs on local client devices, instead of binary code that must be installed, and runs wholly on that device. Examples include web search and social networking.

EIGHT GREAT IDEAS IN COMPUTER ARCHITECTURE

- ❓ ***Eight great ideas*** that computer architects have been invented in the last 60 years of computer design.
- ❓ These ideas are so powerful they have lasted long after the first computer that used them, with newer architects demonstrating their admiration by imitating their predecessors.

EIGHT GREAT IDEAS IN COMPUTER ARCHITECTURE



1. Design for Moore's Law

❓ The one constant for computer designers is rapid change, which is driven largely by Moore's Law. It states that integrated circuit resources double every 18–24 months. Moore's Law resulted from a 1965 prediction of such growth in IC capacity made by Gordon Moore, one of the founders of Intel.

❓ **Computer architects must anticipate where the technology will be when the design finishes rather than design for where it starts.**

EIGHT GREAT IDEAS IN COMPUTER ARCHITECTURE



2. Use Abstraction to Simplify Design

- ❓ A major productivity technique for hardware and software is to use abstractions to represent the design at different levels of representation; lower-level details are hidden to offer a simpler model at higher levels.

EIGHT GREAT IDEAS IN COMPUTER ARCHITECTURE



3. Make the Common Case Fast

- ❑ Making the common case fast will tend to enhance performance better than optimizing the rare case. Ironically, the common case is often simpler than the rare case and hence is often easier to enhance.
- ❑ This principle applies when determining how to spend resources , since the impact of improvement is higher if the occurrence is frequent.

EIGHT GREAT IDEAS IN COMPUTER ARCHITECTURE



4. Performance via Parallelism

- ❑ Since the dawn of computing, computer architects have created designs that get more performance by performing operations in parallel.
- ❑ System level: Multiple processors/disks
- ❑ Individual processor: Instruction level parallelism
- ❑ Digital design: set associative cache/ multiple banks of memory



5. Performance via Pipelining

- ❑ A particular pattern of parallelism is so prevalent in computer architecture that it merits its own name: pipelining.
- ❑ Pipelining is a technique where multiple instructions are overlapped during execution.
- ❑ Pipeline is divided into stages and these stages are connected with one another to form a pipe like structure.
- ❑ Pipelining increases the overall instruction throughput.

EIGHT GREAT IDEAS IN COMPUTER ARCHITECTURE



6. Performance via Prediction

- ❓ In some cases it can be faster on average to guess and start working rather than wait until you know for sure, assuming that the mechanism to recover from a misprediction is not too expensive and your prediction is relatively accurate.

EIGHT GREAT IDEAS IN COMPUTER ARCHITECTURE



7. Hierarchy of Memories

- ❓ Programmers want memory to be fast, large, and cheap, as memory speed often shapes performance, capacity limits the size of problems that can be solved, and the cost of memory today is often the majority of computer cost.
- ❓ Architects have found that they can address these conflicting demands with a hierarchy of memories, with the fastest, smallest, and most expensive memory per bit at the top of the hierarchy and the slowest, largest, and cheapest per bit at the bottom.
- ❓ We use a layered triangle icon to represent the memory hierarchy. The shape indicates speed, cost, and size: the closer to the top, the faster and more expensive per bit the memory; the wider the base of the layer, the bigger the memory.

EIGHT GREAT IDEAS IN COMPUTER ARCHITECTURE



8. Dependability via Redundancy

- ❓ Computers not only need to be fast; they need to be dependable. Since any physical device can fail, we make systems dependable by including redundant components that can take over when a failure occurs and to help detect failures.
- ❓ We use the tractor-trailer as our icon, since the dual tires on each side of its rear axels allow the truck to continue driving even when one tire fails.

LAYERS OF ABSTRACTION

- Layers of software are organized primarily in a hierarchical fashion, with applications being the outermost ring and a variety of systems software sitting between the hardware and applications software.




LAYERS OF ABSTRACTION

APPLICATION PROGRAM

- ❑ A typical application, such as a word processor or a large database system, may consist of millions of lines of code and rely on sophisticated software libraries that implement complex functions in support of the application.
- ❑ **Application programming** aims to produce **software** which provides services to the user directly (e.g. word processor).
- ❑ The hardware in a computer can only execute extremely simple low-level instructions.
- ❑ To go from a complex application to the simple instructions involves several layers of software that interpret or translate high-level operations into simple computer instructions, an example of the great idea of abstraction.

LAYERS OF ABSTRACTION

- ❑ There are many types of **systems software**, but two types of systems software are central to every computer system today: **an operating system and a compiler**.
- ❑ An operating system interfaces between a user's program and the hardware and provides a variety of services and supervisory functions.
- ❑ Among the most important functions are:
 - ❑ Handling basic input and output operations
 - ❑ Allocating storage and memory
 - ❑ Providing for protected sharing of the computer among multiple applications using it simultaneously.
- ❑ Examples of operating systems in use today are **Linux, iOS, and Windows**.

- 
- ❓ **Systems Software** → Software that provides services that are commonly useful, including operating systems, compilers, loaders, and assemblers.
 - ❓ **Operating System** → Supervising program that manages the resources of a computer for the benefit of the programs that run on that computer.
 - ❓ **Compiler** → A program that translates high-level language statements into assembly language statements.

FROM A HIGH-LEVEL LANGUAGE TO THE LANGUAGE OF HARDWARE

- ❓ The easiest signals for computers to understand are on and off, and so the computer alphabet is just two letters.
- ❓ The two symbols for these two letters are the numbers 0 and 1, and we commonly think of the computer language as numbers in base 2, or binary numbers. We refer to each “letter” as a **binary digit or bit**.
- ❓ Computers are slaves to our commands, which are called **instructions**.
- ❓ Instructions, which are just collections of bits that the computer understands and obeys, can be thought of as numbers.

❓ For example, the bits

1000110010100000 → tell one computer to add two numbers.

ASSEMBLER

❓ Assembler translates a symbolic version of an instruction into the binary version.

❓ For example, the programmer would write

add A,B → and the assembler would translate this notation into

1000110010100000

This instruction tells the computer to add the two numbers A and B.

ASSEMBLER

- ❑ The first programmers communicated to computers in binary numbers, but this was ***so tedious that they quickly invented new notations*** that were closer to the way humans think. At first, these notations were translated to binary by hand, but this process was still tiresome.
- ❑ Using the computer to help program the computer, the pioneers invented programs to translate from symbolic notation to binary. The **first of these programs was named an assembler.**

ASSEMBLER

- ❓ Assembler → A program that translates a symbolic version of instructions into the binary version.
- ❓ Assembly language → A symbolic representation of machine instructions.
- ❓ Machine language → A binary representation of machine instructions.
- ❓ High-level programming language → A portable language such as C, C++, Java, or Visual Basic that is composed of words and algebraic notation that can be translated by a compiler into assembly language.

High-level
language
program
(in C)

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

Compiler


Assembly
language
program
(for MIPS)

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

Assembler

Binary machine
language
program
(for MIPS)

```
0000000001010001000000000100011000
000000000100000100001000000100001
10001101111000100000000000000000
1000111000010010000000000000000100
10101110000100100000000000000000
1010110111100010000000000000000100
000000111110000000000000000001000
```



❓ A compiler enables a programmer to write this high-level language expression:

A + B

❓ The compiler would compile it into this assembly language statement:

add A,B

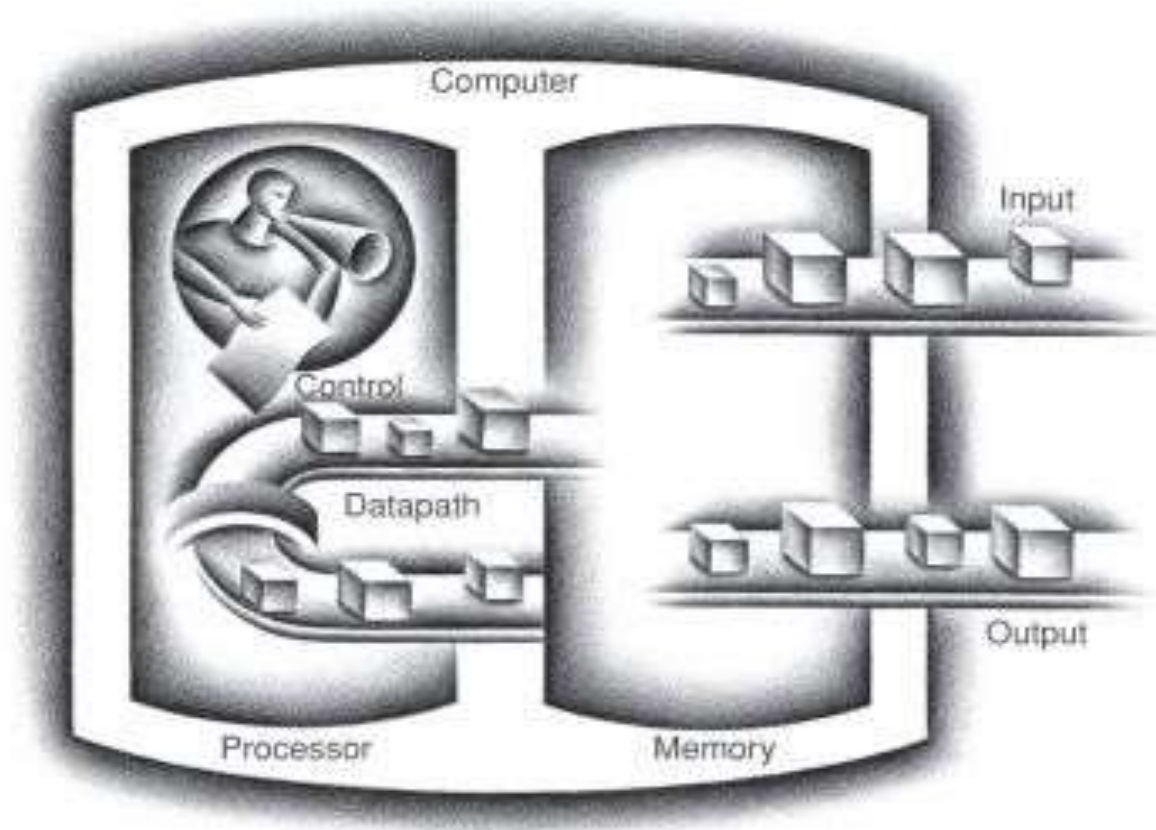


❓ High-level programming languages offer several important benefits:

- ❓ It allows the programmer to **think in a more natural language**, using English words and algebraic notation. Moreover, they allow languages to be designed according to their intended use. Hence, Fortran was designed for scientific computation, Cobol for business data processing, Lisp for symbol manipulation, and so on.
- ❓ The second advantage of programming languages is improved **programmer productivity**. One of the few areas of widespread agreement in software development is that it takes less time to develop programs when they are written in languages that require fewer lines to express an idea.
- ❓ The final advantage is that programming languages allow **programs to be independent of the computer on which they were developed**, since compilers and assemblers can translate high-level language programs to the binary instructions of any computer.

FIVE KEY COMPONENTS OF A COMPUTER

❓ The five classic components of a computer are *input, output, memory, datapath, and control*, with the last two sometimes combined and called the *processor*.



CPU


- ALU (arithmetic logic unit) – Performs calculations and comparisons (data changes)
- CU (control unit): performs fetch/execute cycle –

Functions:

- Moves data to and from CPU registers and other hardware components (no change in data)
- Accesses program instructions and issues commands to the ALU –

Subparts:



- Memory management unit: supervises fetching instructions and data
- I/O Interface: sometimes combined with memory management unit as Bus Interface Unit



A data path (also written as datapath) is a set of functional units ,such as arithmetic logic units or multipliers that perform data processing operations, registers, and buses.


Functional units carry out data processing operations.


Datapath, along with a control unit, make up the CPU (central processing unit) of a computer system.

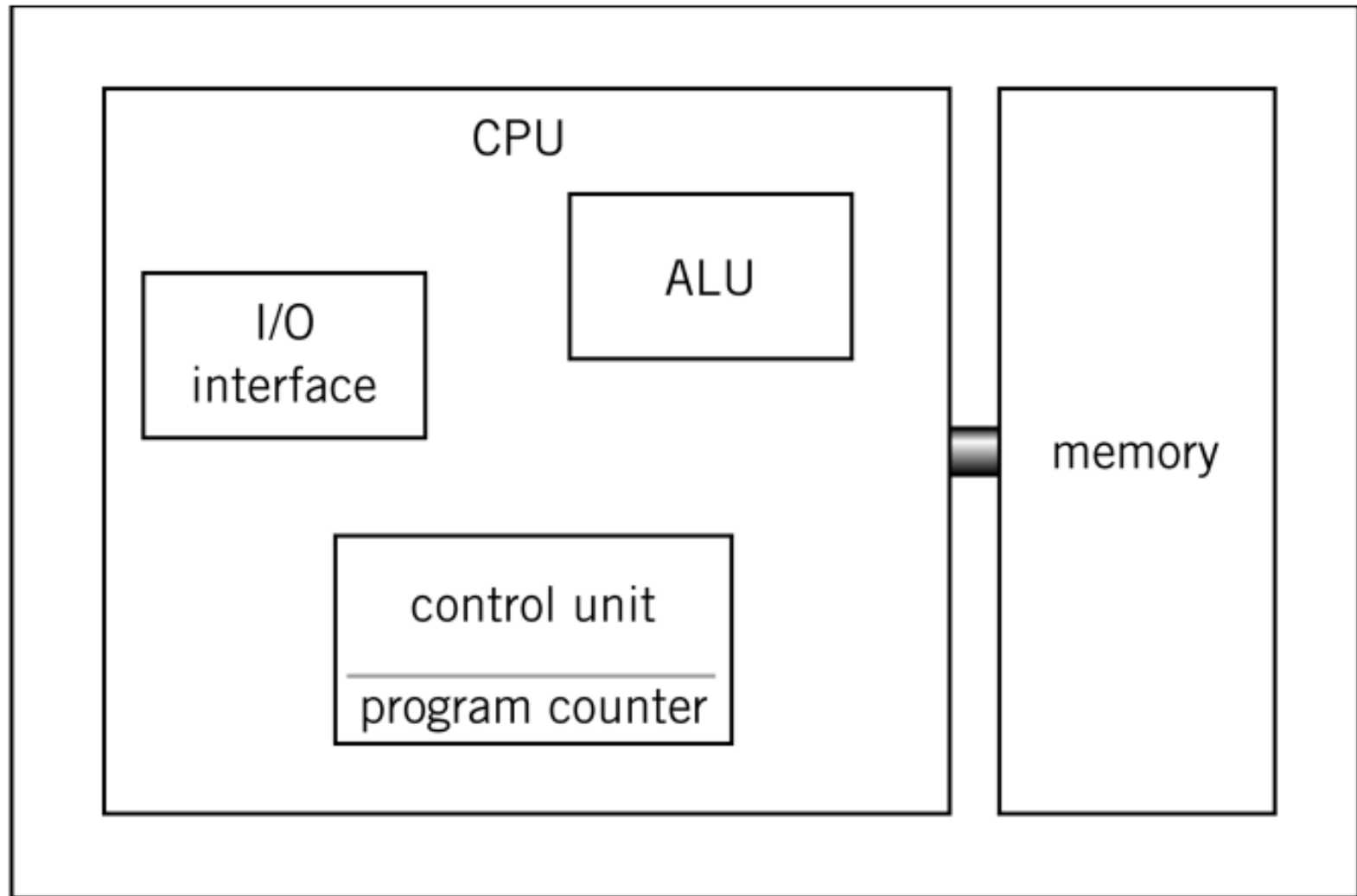


Registers – Example: Program counter (PC) or instruction pointer determines next instruction for execution Small, permanent storage locations within the CPU used for a particular purpose

- Manipulated directly by the Control Unit
- Wired for specific function
- Size in bits or bytes (not MB like memory)
- Can hold data, an address or an instruction


- 
- ❑ The memory is where *the programs are kept when they are running*; it also contains the data needed by the running programs.
 - ❑ The memory is built from DRAM chips. DRAM stands for dynamic random access memory.
 - ❑ Multiple DRAMs are used together to contain the instructions and data of a program.

- 
- ❑ Inside the processor is another type of memory—**cache memory**.
 - ❑ Cache memory → A small, fast memory that acts as a buffer for a slower, larger memory.
 - ❑ **Static Random Access Memory (SRAM)** → Also memory built as an integrated circuit, but faster and less dense than DRAM.



COMMUNICATING WITH OTHER COMPUTERS


- ❑ Networks interconnect whole computers, allowing computer users to extend the power of computing by including communication.
- ❑ Networks have become so popular that they are the backbone of current computer systems; a new personal mobile device or server without a network interface would be ridiculed.
- ❑ Networked computers have several major advantages:
 - ❑ **Communication**: Information is exchanged between computers at high speeds.
 - ❑ **Resource sharing**: Rather than each computer having its own I/O devices, computers on the network can share I/O devices.
 - ❑ **Nonlocal access**: By connecting computers over long distances, users need not be near the computer they are using.


- 
- ❑ Networks vary in length and performance, with the cost of communication increasing according to both the speed of communication and the distance that information travels. Perhaps the most popular type of network is **Ethernet**.
 - ❑ It can be up to a kilometer long and transfer at up to 40 gigabits per second. Its length and speed make Ethernet useful to connect computers on the same floor of a building; hence, it is an example of what is generically called a local area network.
 - ❑ **Local Area Network (LAN)** → A network designed to carry data within a geographically confined area, typically within a single building.
 - ❑ **Wide Area Network (WAN)** → A network extended over hundreds of kilometers that can span a continent.

TECHNOLOGIES FOR BUILDING PROCESSORS AND MEMORY

- ❓ Processors and memory have improved at an incredible rate, because computer designers have long embraced the latest in electronic technology to try to win the race to design a better computer.
- ❓ Table below shows the technologies that have been used over time, with an estimate of the relative performance per unit cost for each technology.

Year	Technology used in computers	Relative performance/unit cost
1951	Vacuum tube	1
1965	Transistor	35
1975	Integrated circuit	900
1995	Very large-scale integrated circuit	2,400,000
2013	Ultra large-scale integrated circuit	250,000,000,000

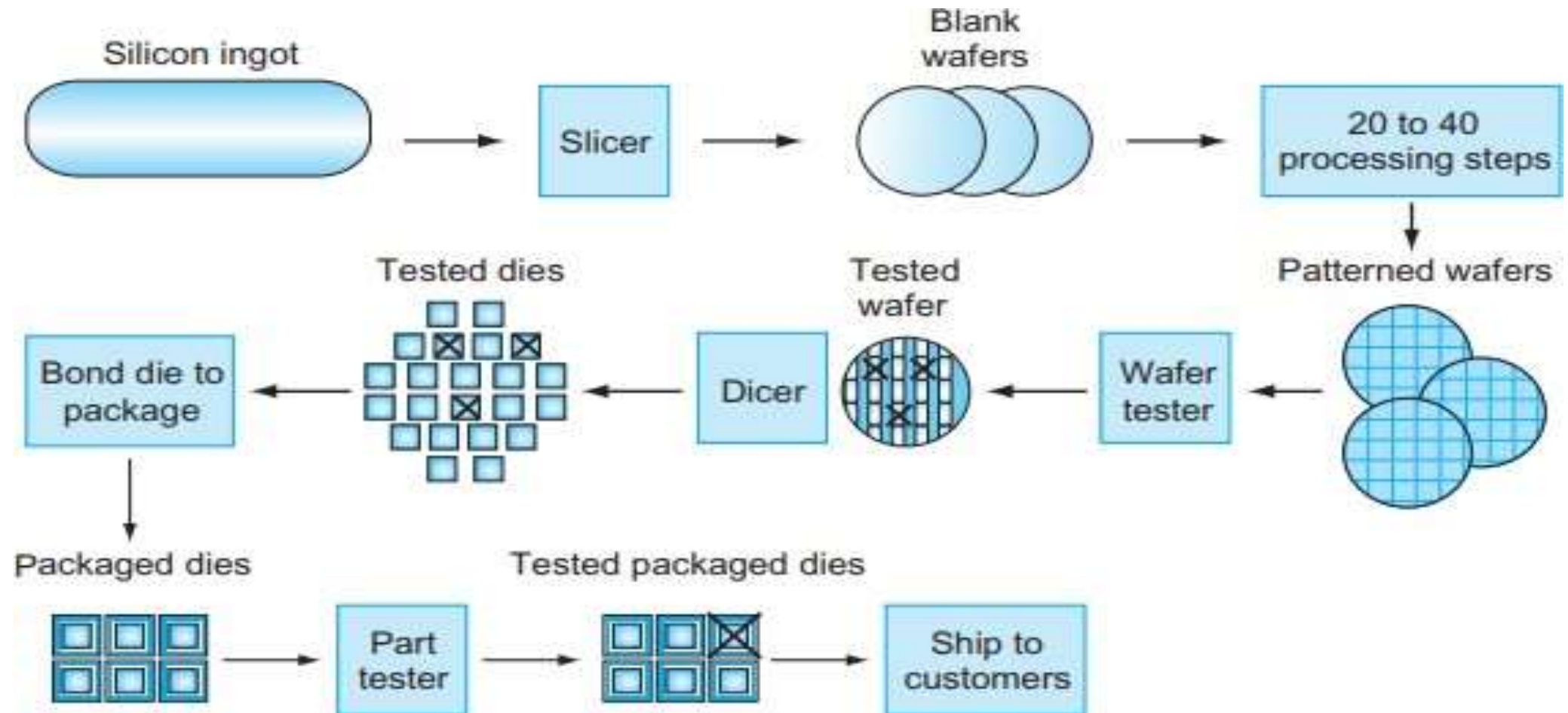
- 
- ❓ A **transistor** is simply an **on/off switch controlled by electricity**. The **integrated circuit (IC) combined dozens to hundreds of transistors** into a single chip.
 - ❓ **Very Large-Scale Integrated (VLSI) circuit** → A device containing hundreds of thousands to millions of transistors.




❓ To understand how manufacture integrated circuits, we start at the beginning. The manufacture of a chip **begins with silicon**, a substance found in sand. Because silicon does not conduct electricity well, it is called a semiconductor. With a special chemical process, it is possible to add materials to silicon that allow tiny areas to transform into one of three devices:

- ❓ Excellent conductors of electricity (using either microscopic copper or aluminum wire).
- ❓ Excellent insulators from electricity (like plastic sheathing or glass) .
- ❓ Areas that can conduct or insulate under special conditions (as a switch).

MANUFACTURING PROCESS FOR INTEGRATED CIRCUITS



- 
- ❓ **Silicon crystal ingot** → A rod composed of a silicon crystal that is between 8 and 12 inches in diameter and about 12 to 24 inches long.
 - ❓ **Wafer** → A slice from a silicon ingot no more than 0.1 inches thick, used to create chips.
 - ❓ wafers then go through a series of processing steps, during which patterns of chemicals are placed on each wafer.

- ❓ A single microscopic flaw in the wafer itself or in one of the dozens of patterning steps can result in that area of the wafer failing. These defects, as they are called, make it virtually impossible to manufacture a perfect wafer.
- ❓ The simplest way to cope with imperfection is to *place many independent components on a single wafer*. The patterned wafer is then chopped up, or diced, into these components, called dies and more informally known as chips.
- ❓ die → The individual rectangular sections that are cut from a wafer, more informally known as chips.
- ❓ Yield → The percentage of good dies from the total number of dies on the wafer.

- Once you've found good dies, they are connected to the input/output pins of a package, using a process called **bonding**. These packaged parts are tested a final time, since mistakes can occur in packaging, and then they are shipped to customers.
- Elaboration: The cost of an integrated circuit can be expressed in three simple equations:

$$\text{Cost per die} = \frac{\text{Cost per wafer}}{\text{Dies per wafer} \times \text{yield}}$$

$$\text{Dies per wafer} \approx \frac{\text{Wafer area}}{\text{Die area}}$$

$$\text{Yield} = \frac{1}{(1 + (\text{Defects per area} \times \text{Die area}/2))^2}$$

PERFORMANCE

- ❓ In most cases, we will need ***different performance metrics*** as well as different sets of applications to benchmark personal mobile devices, which are more focused on response time, versus servers, which are more focused on throughput.
- ❓ **Response time** → Also called execution time. The total time required for the computer to complete a task, including disk accesses, memory accesses, I/O activities, operating system overhead, CPU execution time, and so on.
- ❓ **Throughput** → Also called bandwidth. Another measure of performance, it is the number of tasks completed per unit time.

PERFORMANCE

❓ To maximize performance, we want to minimize response time or execution time for some task. Thus, we can relate performance and execution time for a computer X:

$$\text{Performance}_X = \frac{f}{\text{Execution time}_X}$$

This means that for two computers X and Y, if the performance of X is greater than the performance of Y, we have

$$\begin{aligned} \text{Performance}_X &> \text{Performance}_Y \\ \frac{1}{\text{Execution time}_X} &> \frac{1}{\text{Execution time}_Y} \\ \text{Execution time}_Y &> \text{Execution time}_X \end{aligned}$$

That is, the execution time on Y is longer than that on X, if X is faster than Y.

PERFORMANCE

In discussing a computer design, we often want to relate the performance of two different computers quantitatively. We will use the phrase “X is n times faster than Y”—or equivalently “X is n times as fast as Y”—to mean

$$\frac{\text{Performance}_X}{\text{Performance}_Y} = n$$

If X is n times as fast as Y, then the execution time on Y is n times as long as it is on X:

$$\frac{\text{Performance}_X}{\text{Performance}_Y} = \frac{\text{Execution time}_Y}{\text{Execution time}_X} = n$$

PERFORMANCE - EXAMPLE

If computer A runs a program in 10 seconds and computer B runs the same program in 15 seconds, how much faster is A than B?

We know that A is n times as fast as B if

$$\frac{\text{Performance}_A}{\text{Performance}_B} = \frac{\text{Execution time}_B}{\text{Execution time}_A} = n$$

Thus the performance ratio is

$$\frac{15}{10} = 1.5$$

and A is therefore 1.5 times as fast as B.

In the above example, we could also say that computer B is 1.5 times *slower than* computer A, since

$$\frac{\text{Performance}_A}{\text{Performance}_B} = 1.5$$

means that

$$\frac{\text{Performance}_A}{1.5} = \text{Performance}_B$$

MEASURING PERFORMANCE

- ❓ Time is the measure of computer performance: the computer that performs the same amount of work in the least time is the fastest. Program execution time is measured in seconds per program.
- ❓ Computers are often shared, however, and a processor may work on several programs simultaneously.
- ❓ **CPU execution time** or simply **CPU time** is the time the CPU spends computing for the task and does not include time spent waiting for I/O or running other programs.
- ❓ CPU time can be further divided into the CPU time spent in the program, called **user CPU time**, and the CPU time spent in the operating system performing tasks on behalf of the program, called **system CPU time**.

MEASURING PERFORMANCE

- ❓ All computers are constructed using a clock that determines when events take place in the hardware.
- ❓ These discrete time intervals are called clock cycles (or ticks, clock ticks, clock periods, clocks, cycles).
- ❓ Designers refer to the length of a clock period both as the time for a complete clock cycle (e.g., 250 picoseconds, or 250 ps) and as the clock rate (e.g., 4 gigahertz, or 4 GHz), which is the inverse of the clock period.

CPU PERFORMANCE AND ITS FACTORS

$$\text{CPU execution time for a program} = \text{CPU clock cycles for a program} \times \text{Clock cycle time}$$

Alternatively, because clock rate and clock cycle time are inverses,


$$\text{CPU execution time for a program} = \frac{\text{CPU clock cycles for a program}}{\text{Clock rate}}$$

This formula makes it clear that the hardware designer can improve performance by ***reducing the number of clock cycles*** required for a program or the length of the clock cycle.

INSTRUCTION PERFORMANCE

- ❓ The performance equations above did not include any reference to the number of instructions needed for the program.
- ❓ However, since the compiler clearly generated instructions to execute, and the computer had to execute the instructions to run the program, the execution time must depend on the number of instructions in a program.
- ❓ One way to think about execution time is that it equals the number of instructions executed multiplied by the average time per instruction. Therefore, the number of clock cycles required for a program can be written as,

$$\text{CPU clock cycles} = \text{Instructions for a program} \times \text{Average clock cycles per instruction}$$



❓ The term **clock cycles per instruction**, which is the average number of clock cycles each instruction takes to execute, is often abbreviated as **CPI**. Since different instructions may take different amounts of time depending on what they do, CPI is an average of all the instructions executed in the program.

THE CLASSIC CPU PERFORMANCE EQUATION

❓ The basic performance equation in terms of instruction count (the number of instructions executed by the program), CPI, and clock cycle time:

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

or, since the clock rate is the inverse of clock cycle time:

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

INSTRUCTION SET PRINCIPLES

- ❑ In order to use the hardware of a computer, we must speak its language.
- ❑ The words of a computer language are called instructions, and its vocabulary is called an instruction set.
- ❑ Instruction set of a computer → the portion of the computer visible to the assembly level programmer or to the compiler writer.
- ❑ The ISA serves as the boundary between software and hardware.

CLASSIFYING INSTRUCTION SET ARCHITECTURES

The ISA of a processor can be described using 5 categories:

1. Operand Storage in the CPU

❓ Where are the operands kept other than in memory?

2. Number of explicit named operands

❓ How many operands are named in a typical instruction.

3. Operand location

❓ Can any ALU instruction operand be located in memory? Or must all operands be kept internally in the CPU?



4. Operations

❓ What operations are provided in the ISA.

5. Type and size of operands

❓ What is the type and size of each operand and how is it specified?



The 3 most common types of ISAs are:

- ❑ ***Stack*** - The operands are implicitly on top of the stack.
- ❑ ***Accumulator*** - One operand is implicitly the accumulator.
- ❑ ***General Purpose Register (GPR)*** - All operands are explicitly mentioned, they are either registers or memory locations.

ACCUMULATOR BASED MACHINES

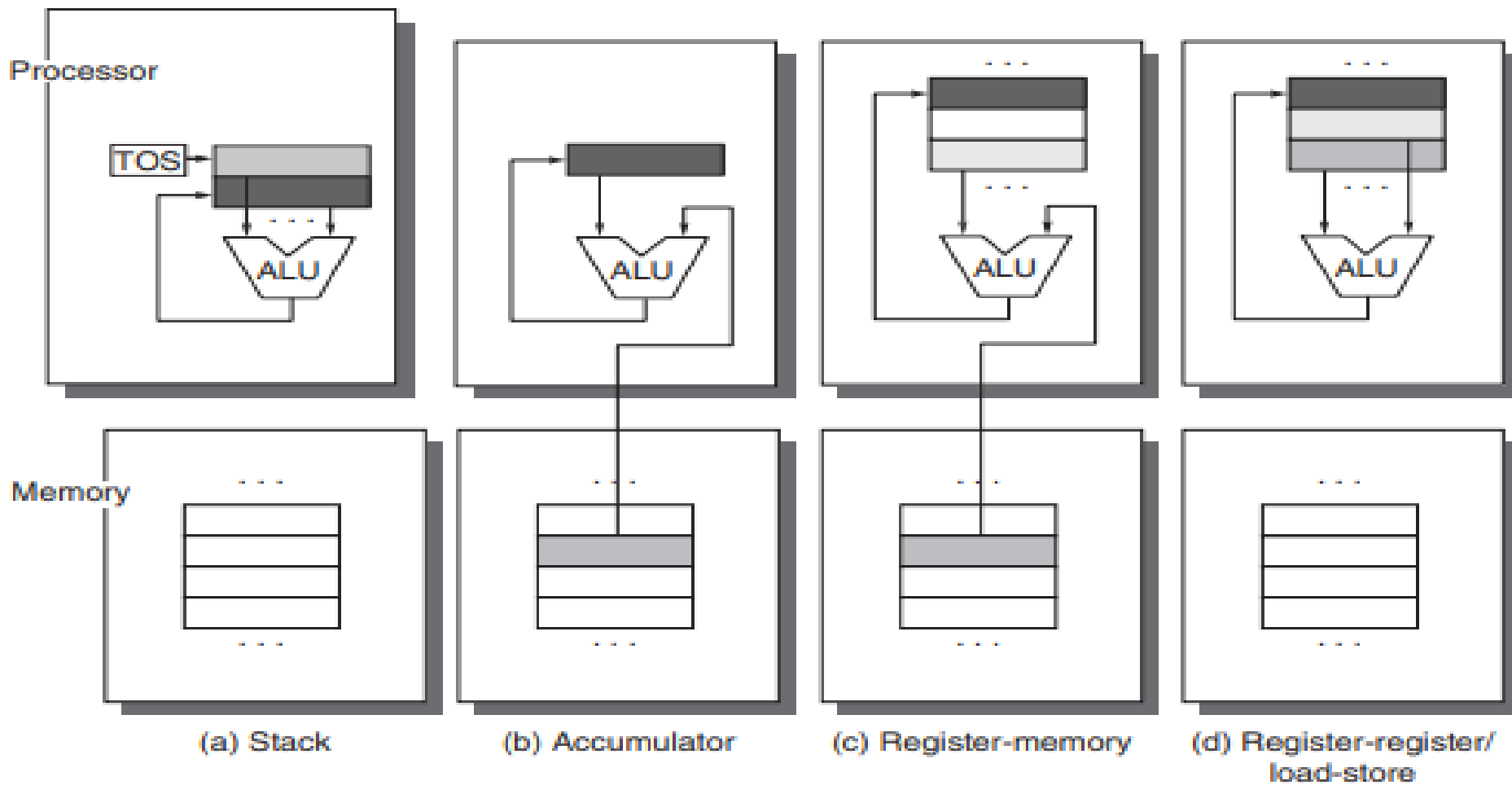
- ❓ Accumulator based machines use special registers called the **accumulators** to hold one source operand and also the result of the arithmetic or logic operations performed. Thus the accumulator registers collect (or 'accumulate') data.
- ❓ Since the accumulator holds one of the operands, one more register may be required to hold the address of another operand.
- ❓ Accumulator machines employ a very small number of accumulator registers, generally only one. These machines were useful at the time when memory was quite expensive.
- ❓ However, now that the memory is relatively inexpensive, these are not considered very useful, and their use is severely limited for the computation of expressions with many operands.

STACK BASED MACHINES

- ❑ A stack is a group of registers organized as a last-in-first-out (LIFO) structure.
- ❑ In such a structure, the operands stored first, through the push operation, can only be accessed last, through a pop operation; the order of access to the operands is reverse of the storage operation.
- ❑ Arithmetic and logic operations successively pick operands from the top-of-the-stack (TOS), and push the results on the TOS at the end of the operation.
- ❑ In stack based machines, operand addresses need not be specified during the arithmetic or logical operations. Therefore, these machines are also called 0-address machines.


GENERAL-PURPOSE-REGISTER MACHINES


- ❑ In general purpose register machines, a number of registers are available within the CPU.
- ❑ These registers do not have dedicated functions, and can be employed for a variety of purposes.
- ❑ To identify the register within an instruction, a small number of bits are required in an instruction word. For example, to identify one of the 64 registers of the CPU, a 6-bit field is required in the instruction.
- ❑ CPU registers are faster than cache memory. Registers are also easily and more effectively used by the compiler compared to other forms of internal storage.
- ❑ Registers can also be used to hold variables, thereby reducing memory traffic.





? Figure shows how the code sequence $C = A + B$ would typically appear in these three classes of instruction sets. The explicit operands may be accessed directly from memory or may need to be first loaded into temporary storage, depending on the class of architecture and choice of specific instruction.

Stack	Accumulator	Register (register-memory)	Register (load-store)
Push A	Load A	Load R1,A	Load R1,A
Push B	Add B	Add R3,R1,B	Load R2,B
Add	Store C	Store R3,C	Add R3,R1,R2
Pop C			Store R3,C


- 
- ❓ As the figures show, there are really **two classes of register computers**.
 - ❓ One class can access memory as part of any instruction, called **register-memory architecture**, and the other can access memory only with load and store instructions, called **load-store architecture**.
 - ❓ A third class, not found in computers shipping today, keeps all operands in memory and is called a **memory-memory architecture**.


- 
- ❓ Although most early computers used stack or accumulator-style architectures, virtually every new architecture designed after 1980 uses a **load-store register architecture**.
 - ❓ The major reasons for the emergence of general-purpose register (GPR) computers are twofold. First, registers—like other forms of storage ***internal to the processor***—are ***faster than memory***. Second, ***registers are more efficient***.

- 
- ❓ For example, on a register computer the expression $(A * B) - (B * C) - (A * D)$ may be evaluated by doing the multiplications in any order.
 - ❓ Nevertheless, on a stack computer the hardware must evaluate the expression in only one order, since operands are hidden on the stack, and it may have to load an operand multiple times



❓ More importantly, registers can be used to hold variables. When variables are allocated to registers, the memory traffic reduces, the program speeds up (since registers are faster than memory), and the code density improves (since a register can be named with fewer bits than can a memory location).

- 
- ❓ **Two major instruction set characteristics divide GPR architectures.** Both characteristics concern the nature of operands for a typical arithmetic or logical instruction (ALU instruction).
 - ❓ The first concerns **whether an ALU instruction has two or three operands.**
 - ❓ In the three-operand format, the instruction **contains one result operand and two source operands.** In the two-operand format, **one of the operands is both a source and a result for the operation.**



❓ The second distinction among GPR architectures concerns *how many of the operands may be memory addresses in ALU instructions*. The number of memory operands supported by a typical ALU instruction may vary from none to three.

Figure shows combinations of these two attributes with examples of computers. Although there are seven possible combinations, three serve to classify nearly all existing computers. As we mentioned earlier, these three are load-store (also called register-register), register-memory, and memory-memory.

Number of memory addresses	Maximum number of operands allowed	Type of architecture	Examples
0	3	Load-store	Alpha, ARM, MIPS, PowerPC, SPARC, SuperH, TM32
1	2	Register-memory	IBM 360/370, Intel 80x86, Motorola 68000, TI TMS320C54x
2	2	Memory-memory	VAX (also has three-operand formats)
3	3	Memory-memory	VAX (also has two-operand formats)

Figure A.3 Typical combinations of memory operands and total operands per typical ALU instruction with examples of computers. Computers with no memory reference per ALU instruction are called load-store or register-register computers. Instructions with multiple memory operands per typical ALU instruction are called register-memory or memory-memory, according to whether they have one or more than one memory operand.

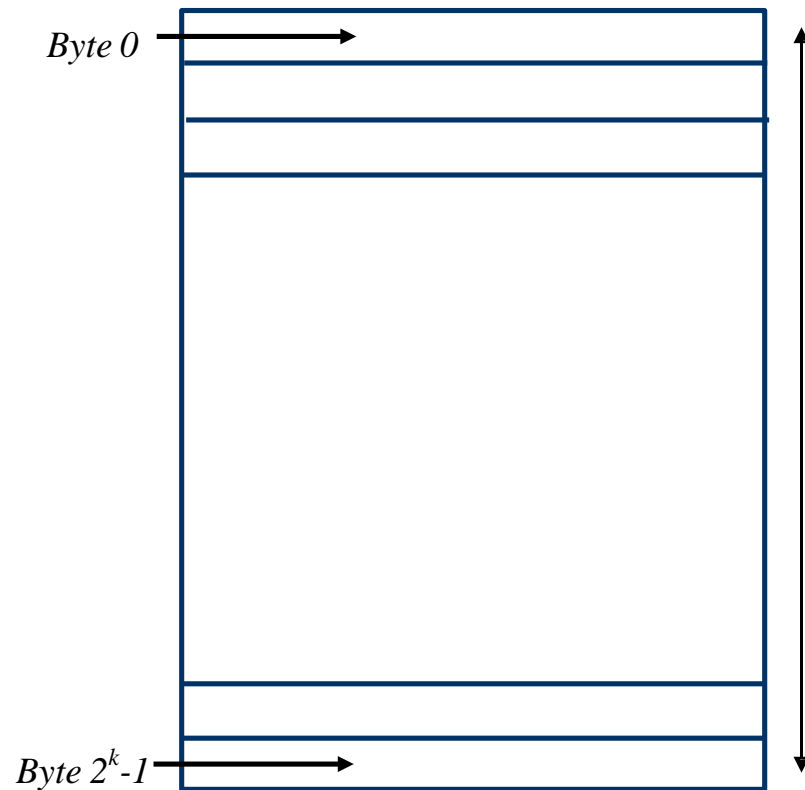
ADVANTAGES AND DISADVANTAGES

Type	Advantages	Disadvantages
Register-register (0, 3)	Simple, fixed-length instruction encoding. Simple code generation model. Instructions take similar numbers of clocks to execute (see Appendix C).	Higher instruction count than architectures with memory references in instructions. More instructions and lower instruction density lead to larger programs.
Register-memory (1, 2)	Data can be accessed without a separate load instruction first. Instruction format tends to be easy to encode and yields good density.	Operands are not equivalent since a source operand in a binary operation is destroyed. Encoding a register number and a memory address in each instruction may restrict the number of registers. Clocks per instruction vary by operand location.
Memory-memory (2, 2) or (3, 3)	Most compact. Doesn't waste registers for temporaries.	Large variation in instruction size, especially for three-operand instructions. In addition, large variation in work per instruction. Memory accesses create memory bottleneck. (Not used today.)

MEMORY ADDRESSING

- ❓ *Information is stored in the memory as a **collection of bits**.*
- ❓ *Collection of 8 bits known as a **“byte”***
- ❓ *Bytes are grouped into **words** → **Collection of bits stored or retrieved simultaneously**.*


BYTE ADDRESSABILITY




- Memory is viewed as a sequence of bytes.
- Address of the first byte is 0
- Address of the last byte is $2^k - 1$, where k is the number of bits used to hold memory address
- E.g. when $k = 16$,
Address of the first byte is 0
Address of the last byte is 65535

BIG-ENDIAN & LITTLE-ENDIAN ASSIGNMENTS

- ❓ Endianness is a term that describes the order in which a sequence of bytes is stored in computer memory.
- ❓ Byte address can be assigned across words in 2 ways:
 1. Big-endian : Lower byte addresses are used for the most significant bytes of the words.
 2. Little-endian: Lower byte addresses are used for the less significant bytes of the word.



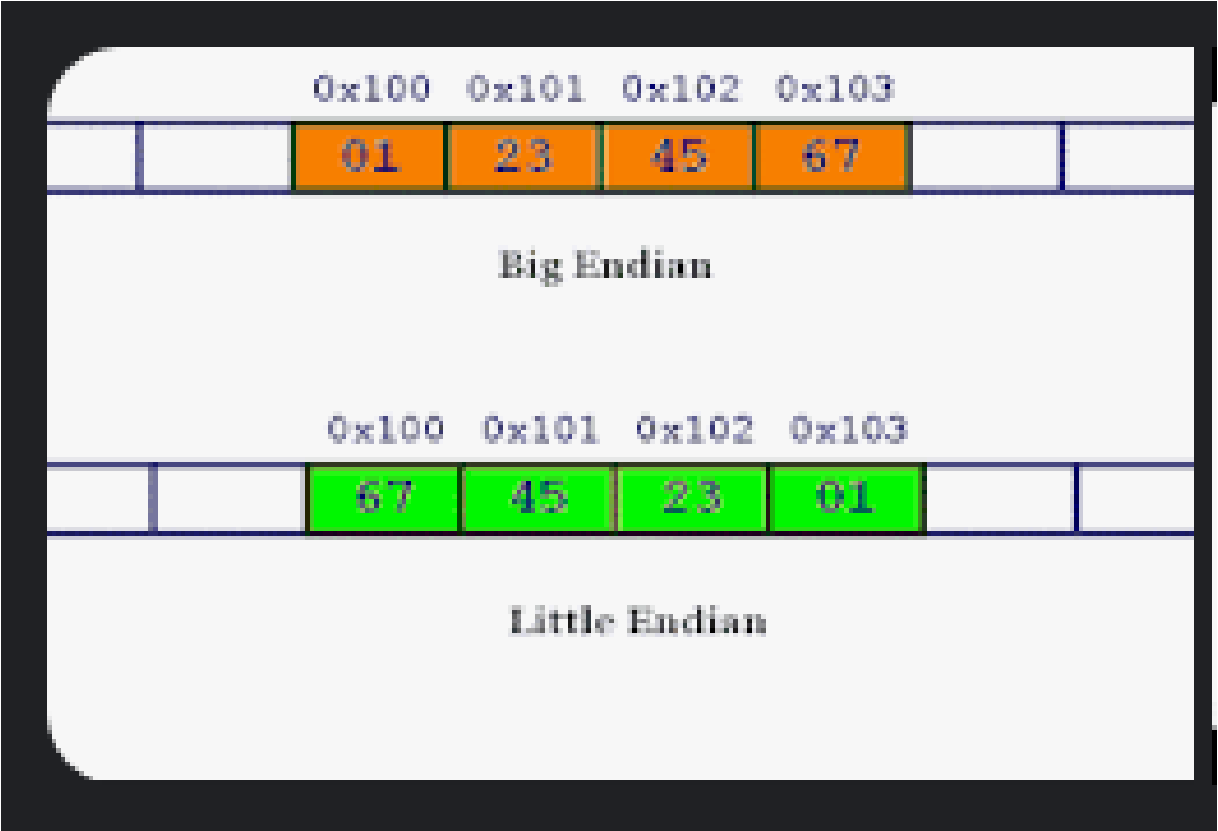
Big-endian is an order in which the "big end" (most significant value in the sequence) is stored first, at the lowest storage address. Little-endian is an order in which the "little end" (least significant value in the sequence) is stored first.



In a big-endian computer, the two bytes required for the hexadecimal number 4F52 would be stored as 4F52 in storage.

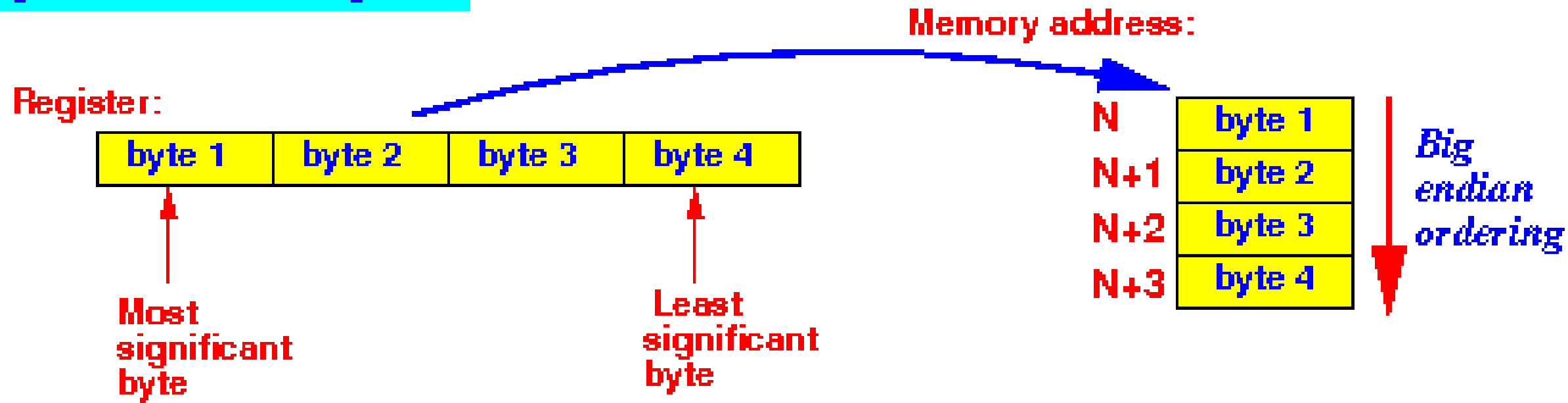
For example, if 4F is stored at storage address 1000, 52 will be at address 1001.

In a little-endian system, it would be stored as 524F, with 52 at address 1000 and 4F at 1001.

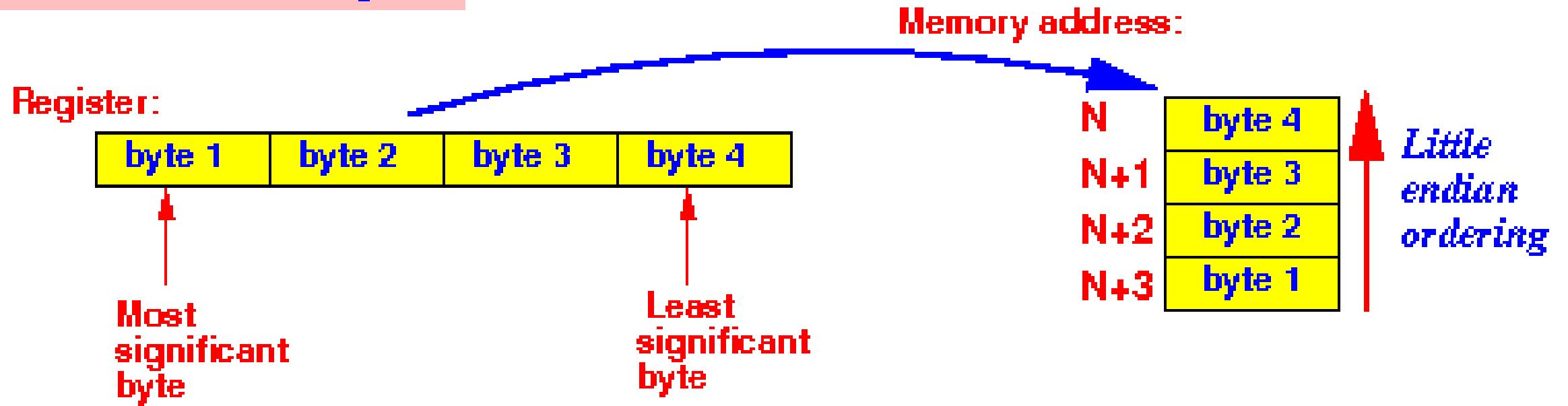




Big endian ordering



Little endian ordering



❓ Number of bits in a word → word length.

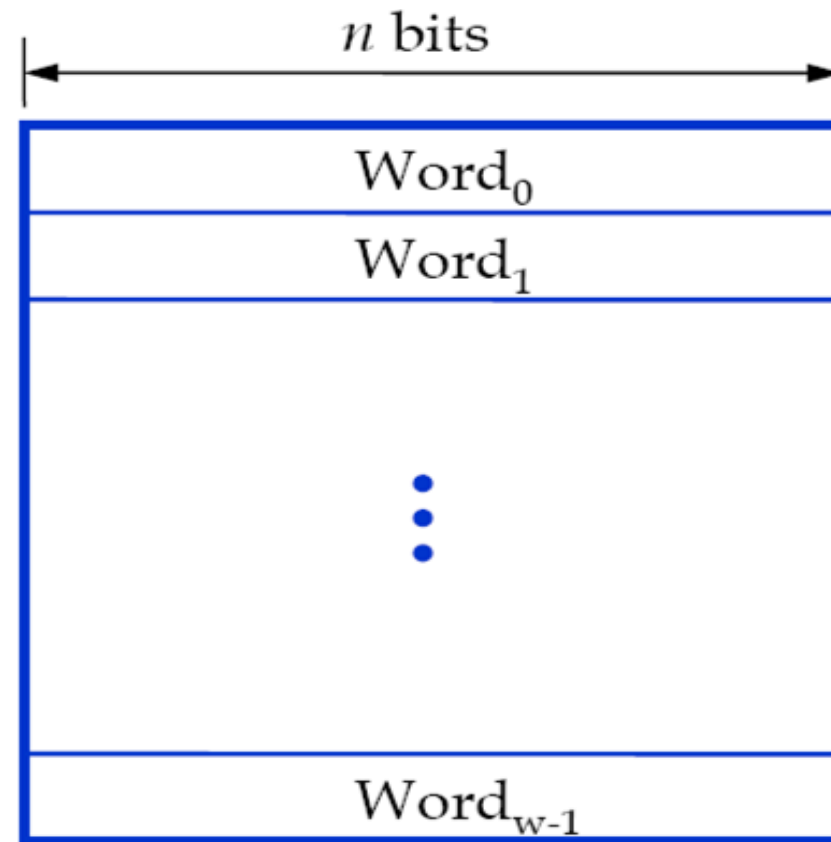
❓ Word length can be 16 to 64 bits.

❓ *Word length can also be expressed as number of bytes .*

❓ *Word length of 16 bits → word length of 2 bytes.*

❓ *Words may be 2 bytes (older architectures), 4 bytes (current architectures), or 8+ bytes (modern architectures).*

? *Memory Words*



Byte Addressability

- ❑ *Accessing the memory to obtain information requires specifying the “address” of the memory location.*
- ❑ *Assigning addresses to each bit is impractical.*
- ❑ *Addresses are assigned to a single byte.*
- ❑ *The most practical assignment is to have successive addresses refer to successive byte locations in memory → “Byte addressable memory”.*

ADDRESSING MODES

- ❓ The **different ways** in which the **location of an operand is specified in an instruction** are referred to as **addressing modes**.
- ❓ When a memory location is used, the actual memory address specified by the addressing mode is called the **effective address**.

-
- The operands of the instructions can be located either in the main memory or in the CPU registers.
 - If the operand is placed in the main memory, then the instruction provides the location address in the operand field.
 - Many methods are followed to specify the operand address.
 - They are called addressing modes.

Addressing Modes— The term addressing modes refers to the way in which the operand of an instruction is specified. The addressing mode specifies a rule for interpreting or modifying the address field of the instruction before the operand is actually executed.

Addressing modes for 8086 instructions are divided into two categories:

- 1) Addressing modes for data
- 2) Addressing modes for branch

ADDRESSING MODES

Generic Addressing Modes:

- ☐ Register mode
- ☐ Absolute mode
- ☐ Immediate mode
- ☐ Indirect mode
- ☐ Index mode
- ☐ Relative mode
- ☐ Auto-increment mode
- ☐ Auto-decrement mode

ADDRESSING MODES

Name	Assembler syntax	Addressing function
Immediate	#Value	Operand=Value
Register	Ri	EA=Ri
Absolute (Direct)	LOC	EA=LOC
Indirect	(Ri)	EA=[Ri]
	(LOC)	EA=[LOC]
Index	X(Ri)	EA=[Ri]+X
Base with index	(Ri, Rj)	EA=[Ri]+[Rj]
Base with index and offset	X(Ri, Rj)	EA=[Ri]+[Rj]+X
Relative	X(PC)	EA=[PC]+X
Autoincrement	(Ri)+	EA=[Ri]; Increment Ri
Autodecrement	-(Ri)	Decrement Ri; EA=[Ri]

ADDRESSING MODES

Implementation of Variables and Constants

- ❑ Variables and constants are the simplest data types.
- ❑ **Representing a variable:** Allocating a register or memory location to hold its value.
- ❑ **Representing a constants:** Address and data constants can be given explicitly in the instruction.

ADDRESSING MODES

Implementation of Variables and Constants

Variables:

- ❑ The value can be changed as needed using the appropriate instructions. There are 2 accessing modes to access the variables. They are,
 - ❑ Register Mode
 - ❑ Absolute Mode

ADDRESSING MODES

1. Register mode

- ❑ **Operand** is the contents of a processor register.
- ❑ **Name (Address)** of the register (its Name) is given in the instruction.
- ❑ E.g. **Clear R1** or **Move R1, R2**

In this mode, the operands are in registers that reside within the CPU. The specific register is selected from a register field in the instruction.

In register addressing the operand is placed in one of 8 bit or 16 bit general purpose registers. The data is in the register that is specified by the instruction.

Here one register reference is required to access the data.



Example:

MOV AX,CX

(move the contents of CX register to AX register)

ADDRESSING MODES

2. Absolute mode(Direct Mode) –symbol []

- ❑ Operand is in a memory location.
- ❑ Address of the memory location is given explicitly in the instruction.
- ❑ E.g. Clear A or Move LOC, R2
- ❑ Also called as “Direct mode” in some assembly languages.
- ❑ The operand resides in memory and its address is given directly by the address field of the instruction. In a branch-type instruction, the address field specifies the actual branch address.

Here only one memory reference operation is required to access the data.



Example:

ADD AL,[0301] / /add the contents of offset address 0301 to AL

ADDRESSING MODES

3. Immediate mode –Symbol

- ❑ **Operand** is given explicitly in the instruction.
- ❑ Move R0,200 immediate
- ❑ It places the value 200 in the register R0. The immediate mode used to specify the value of source operand.
- ❑ In assembly language, the immediate subscript is not appropriate so '#' symbol is used. It can be re-written as,
- ❑ Move R0 ,#200
- ❑ **Register, Absolute and Immediate** modes contained either the **address of the operand or the operand itself.**



In this mode data is present in address field of instruction .Designed like one address instruction format.

Note:Limitation in the immediate mode is that the range of constants are restricted by size of address field.

In other words, an immediate-mode instruction has an operand field instead of an address field.

Immediate-mode instructions are beneficial for initializing registers to a constant value.



Example:

MOV AL, 35H

(move the data 35H into AL register)

ADDRESSING MODES

□ Immediate mode

□ Eg: $A=B+6$

□ MOV R1,B

□ ADD R1, #6

□ MOV A,R1

ADDRESSING MODES

Indirection and Pointers

- ❑ Some instructions provide information from which the memory address of the operand can be determined.
- ❑ That is, they provide the “Effective Address” of the operand.
- ❑ They do not provide the operand or the address of the operand explicitly.

ADDRESSING MODES

❓ Instruction does not give the operand or its address explicitly. Instead it provides information from which the new address of the operand can be determined. This address is called effective Address(EA) of the operand.

ADDRESSING MODES

4. Indirect mode

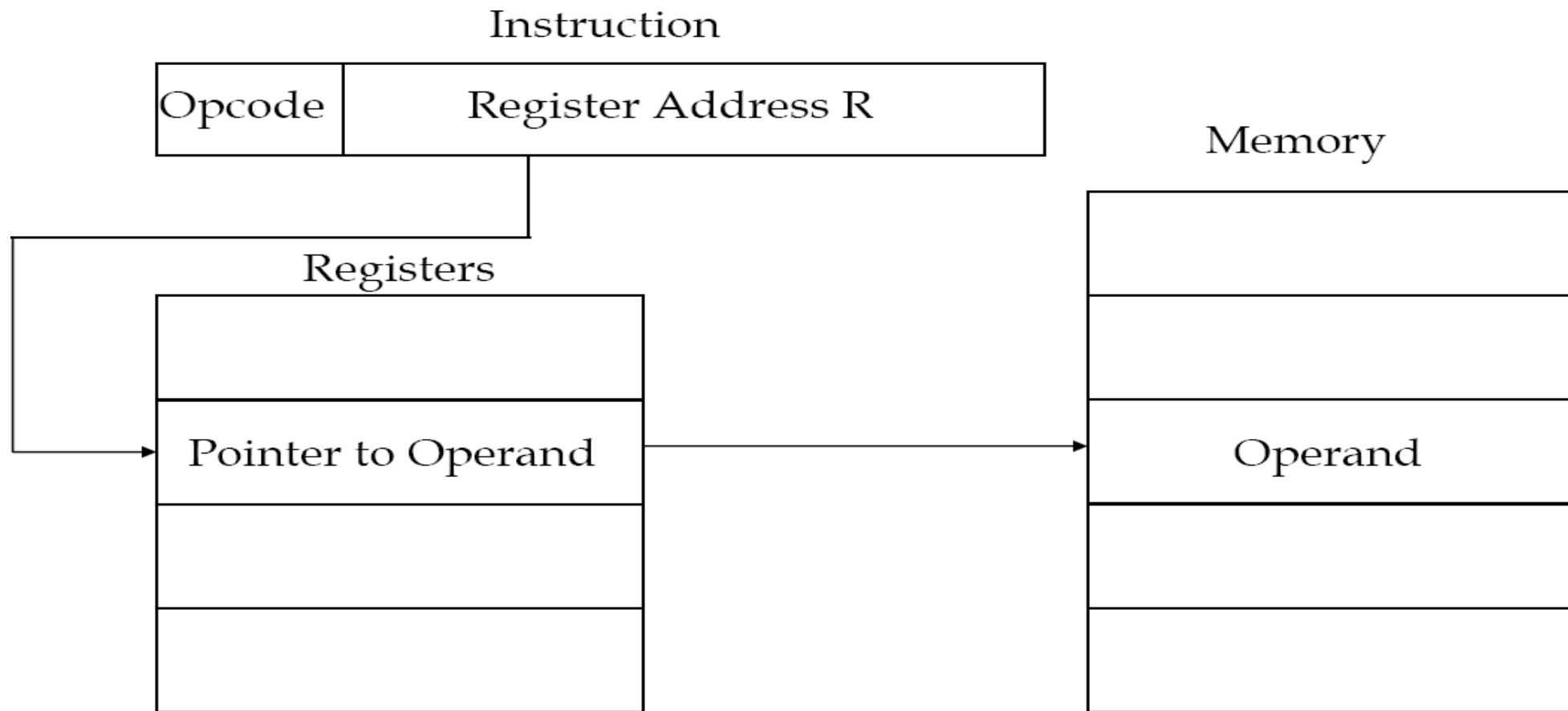
- ❑ The **effective address** of the **operand** is the **contents** of a **register** or **memory location** appears in the instruction.
- ❑ **Indirection** is denoted by placing the **name** of the **register** or the **memory** address given in the instruction in **parentheses**.
- ❑ The **register** or memory **location** that contains the **address** of an **operand** is called a **pointer**.
- ❑ $(R_i) \rightarrow EA = [R_i]$
- ❑ $(LOC) \rightarrow EA = [LOC]$



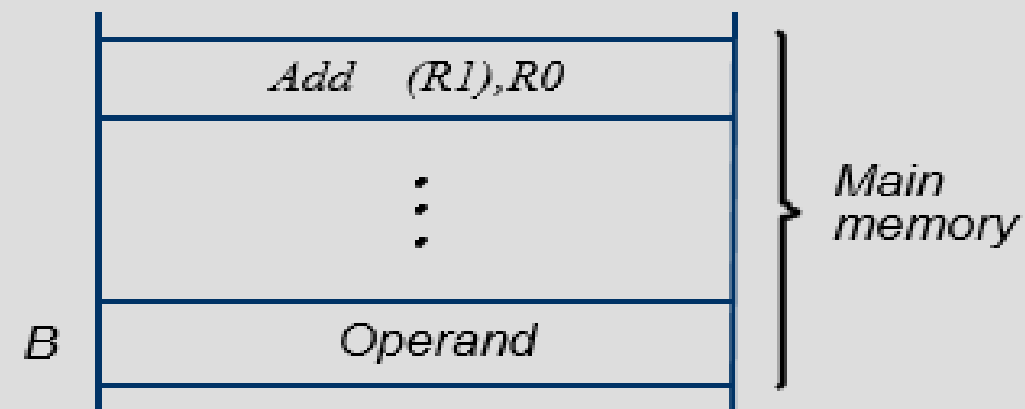
Here two register reference is required to access the data.

MOV AX, [BX](move the contents of memory location s
addressed by the register BX to the register AX)

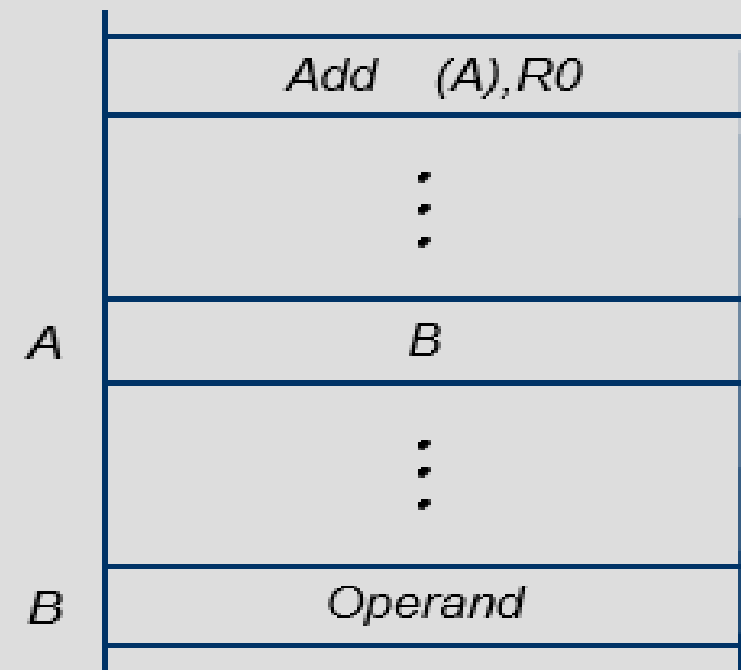
Register Indirect Addressing Diagram



ADDRESSING MODES



- Register *R1* contains Address *B*
- Address *B* has the operand



- Address *A* contains Address *B*
- Address *B* has the operand


R1 and ***A*** are called “***pointers***”

ADDRESSING MODES

Indexing and arrays

5. Index mode:

- ❑ The effective address of the operand is generated by adding a constant value to the contents of a register.
- The constant value uses either special purpose or general purpose register.
- This register is referred to as an index register.



The operand's offset is the sum of the content of an index register SI or DI and an 8 bit or 16 bit displacement.

Example:

MOV AX, [SI + 05]

ADDRESSING MODES

□ Index mode:

☐ Index mode symbolically denoted as $X(R_i)$

☐ $X \rightarrow$ constant value contained in the instruction (offset or displacement).

☐ $R_i \rightarrow$ the name of the register involved.

☐ The effective address of the operand is given by, $EA = X + [R_i]$

☐ The index register R_1 contains the address of a new location and the value of X defines an offset (also called a displacement).

☐ The contents of the index register are not changed in the process of generating the effective address.

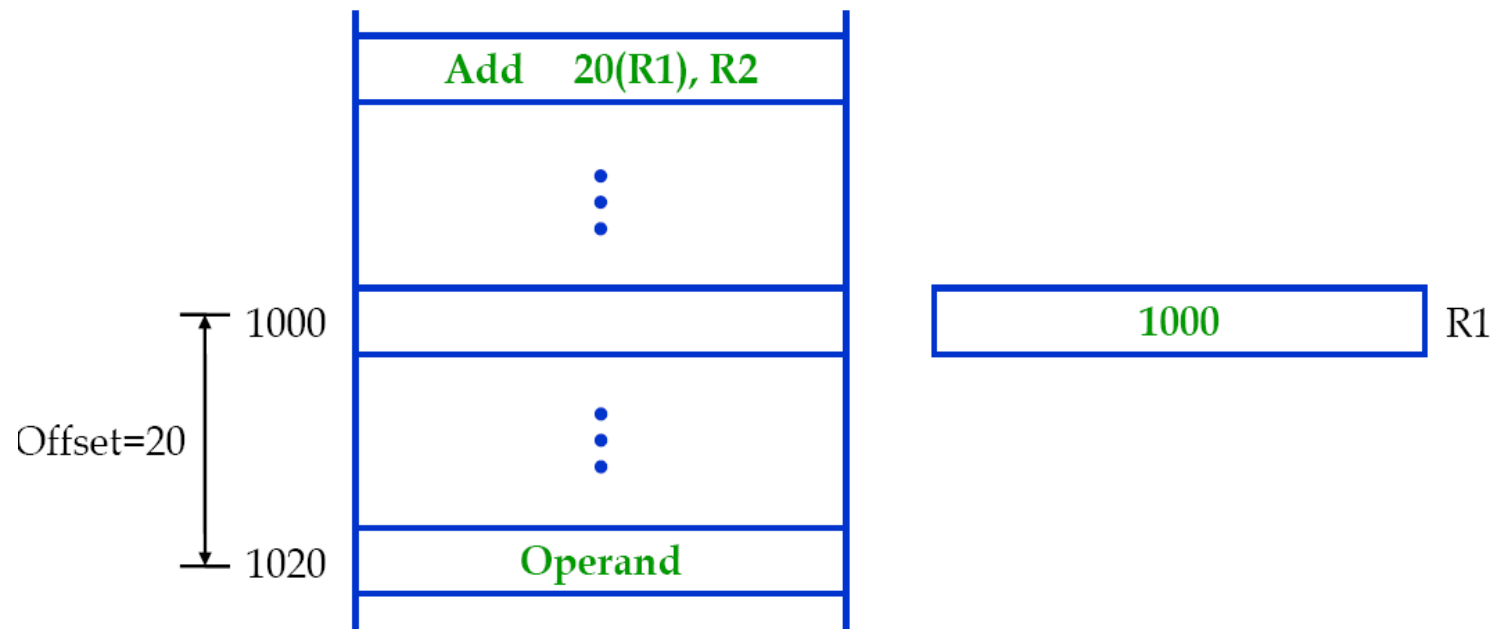
ADDRESSING MODES

MOV R0, 20(R1)

To find operand,

- First goto Reg R1 (using the content address)
- Read from R1-1000
- Add the content 1000 with offset 20 get the result.
- $1000+20=1020$

Offset is given as a constant



ADDRESSING MODES

6. Relative Mode

- Effective Address of the operand is generated by adding a constant value to the contents of the Program Counter (PC).

$$X(PC) \rightarrow EA=[PC]+X$$

- Same as Index Mode \rightarrow the index register is the PC instead of a general purpose register.
- Useful for specifying target addresses in branch instructions
- Addressed location is “relative” to the PC \rightarrow “Relative Mode”

Auto-increment/decrement mode

In this mode, the instruction specifies a register which points to a memory address that contains the operand. However, after the address stored in the register is accessed, the address is incremented or decremented, as specified. The next operand is found by the new value stored in the register.

ADDRESSING MODES

❑ Autodecrement mode

- Effective address of the operand is the contents of a register specified in the instruction.

Decrement R_i ;
 $EA = [R_i]$

- Before accessing the operand, the contents of this register are automatically decremented to point to the previous consecutive memory location.

$-(R1)$

Addressing mode	Example instruction	Meaning	When used
Register	Add R4,R3	$\text{Regs}[R4] \leftarrow \text{Regs}[R4] + \text{Regs}[R3]$	When a value is in a register.
Immediate	Add R4,#3	$\text{Regs}[R4] \leftarrow \text{Regs}[R4] + 3$	For constants.
Displacement	Add R4,100(R1)	$\text{Regs}[R4] \leftarrow \text{Regs}[R4] + \text{Mem}[100 + \text{Regs}[R1]]$	Accessing local variables (+ simulates register indirect, direct addressing modes).
Register indirect	Add R4,(R1)	$\text{Regs}[R4] \leftarrow \text{Regs}[R4] + \text{Mem}[\text{Regs}[R1]]$	Accessing using a pointer or a computed address.
Indexed	Add R3,(R1 + R2)	$\text{Regs}[R3] \leftarrow \text{Regs}[R3] + \text{Mem}[\text{Regs}[R1] + \text{Regs}[R2]]$	Sometimes useful in array addressing: R1 = base of array; R2 = index amount.
Direct or absolute	Add R1,(1001)	$\text{Regs}[R1] \leftarrow \text{Regs}[R1] + \text{Mem}[1001]$	Sometimes useful for accessing static data; address constant may need to be large.
Memory indirect	Add R1,@(R3)	$\text{Regs}[R1] \leftarrow \text{Regs}[R1] + \text{Mem}[\text{Mem}[\text{Regs}[R3]]]$	If R3 is the address of a pointer p , then mode yields $*p$.
Autoincrement	Add R1,(R2)+	$\text{Regs}[R1] \leftarrow \text{Regs}[R1] + \text{Mem}[\text{Regs}[R2]]$ $\text{Regs}[R2] \leftarrow \text{Regs}[R2] + d$	Useful for stepping through arrays within a loop. R2 points to start of array; each reference increments R2 by size of an element, d .
Autodecrement	Add R1,-(R2)	$\text{Regs}[R2] \leftarrow \text{Regs}[R2] - d$ $\text{Regs}[R1] \leftarrow \text{Regs}[R1] + \text{Mem}[\text{Regs}[R2]]$	Same use as autoincrement. Autodecrement/-increment can also act as push/pop to implement a stack.
Scaled	Add R1,100(R2)[R3]	$\text{Regs}[R1] \leftarrow \text{Regs}[R1] + \text{Mem}[100 + \text{Regs}[R2] + \text{Regs}[R3] * d]$	Used to index arrays. May be applied to any indexed addressing mode in some computers.

ENCODING AN INSTRUCTION SET

How to encode instructions as binary values?

❓ Instructions consist of:

❓ **operation** (opcode) e.g. MOV

❓ **operands** (number depends on operation)

❓ operands specified using addressing modes.

❓ addressing mode may include **addressing information**.


❓ e.g. registers, constant values




❓ Encoding of instruction must include opcode, operands & addressing information.

Encoding:

- ❓ represent entire instruction as a **binary value**.
- ❓ **number of bytes** needed depends on how much information must be encoded.
- ❓ instructions are **encoded by assembler**.

- 
- ❑ Encoding of instruction *depends on the range of addressing modes and the degree of independence between opcodes and addressing modes.*
 - ❑ Some older computers have one to five operands with 10 addressing modes for each operand. For such a large number of combinations, typically a separate address specifier is needed for each operand.
 - ❑ The address specifier tells *what addressing mode is used to access the operand.*

- 
- ❑ When encoding the instructions, the number of registers and the number of addressing modes both have a significant impact on the size of instructions, as the register field and addressing mode field may appear many times in a single instruction.
 - ❑ In fact, for most instructions many more bits are consumed in encoding addressing modes and register fields than in specifying the opcode.



❓ The architect must balance several competing forces when encoding the instruction set:

- ❓ The desire to have as many registers and addressing modes as possible.

- ❓ The impact of the size of the register and addressing mode fields on the average instruction size and hence on the average program size.

- ❓ A desire to have instructions encoded into lengths that will be easy to handle in a pipelined implementation. As a minimum, the architect wants instructions to be in multiples of bytes, rather than an arbitrary bit length.

POPULAR CHOICES FOR ENCODING THE INSTRUCTION SET

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

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

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

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

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

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

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

(c) Hybrid (e.g., IBM 360/370, MIPS16, Thumb, TI TMS320C54x)





❓ Three basic variations in instruction encoding:

❓ variable length

❓ fixed length

❓ hybrid

- 
- 
- ❑ The **variable format** allows virtually all addressing modes to be with all operations. This style is best when there are many addressing modes and operations.
 - ❑ The **fixed format** combines the operation and the addressing mode into the opcode. Often fixed encoding will have only a single size for all instructions; it works best when there are few addressing modes and operations.
 - ❑ The **hybrid approach** has multiple formats specified by the opcode, adding one or two fields to specify the addressing mode and one or two fields to specify the operand address.



THANK YOU

