

# **Computer Organization and Architecture**

**Prof. Indranil Sengupta**

**Dr. Sarani Bhattacharya**

**Department of Computer Science and Engineering**

**IIT Kharagpur**

# **Evolution of Computer Systems**

# Introduction

- Computers have become part and parcel of our daily lives.
  - They are everywhere (embedded systems?)
  - Laptops, tablets, mobile phones, intelligent appliances.
- It is required to understand how a computer works.
  - What are there inside a computer?
  - How does it work?
- We distinguish between two terms: *Computer Architecture* and *Computer Organization*.

- **Computer Organization:**

- Design of the components and functional blocks using which computer systems are built.
- **Analogy:** civil engineer's task during building construction (cement, bricks, iron rods, and other building materials).

- **Computer Architecture:**

- How to integrate the components to build a computer system to achieve a desired level of performance.
- **Analogy:** architect's task during the planning of a building (overall layout, floorplan, etc.).

# Historical Perspective

- Constant quest of building automatic computing machines have driven the development of computers.
  - **Initial efforts**: mechanical devices like pulleys, levers and gears.
  - **During World War II**: mechanical relays to carry out computations.
  - **Vacuum tubes developed**: first electronic computer called ENIAC.
  - **Semiconductor transistors developed** and journey of miniaturization began.
    - SSI → MSI → LSI → VLSI → ULSI → .... Billions of transistors per chip

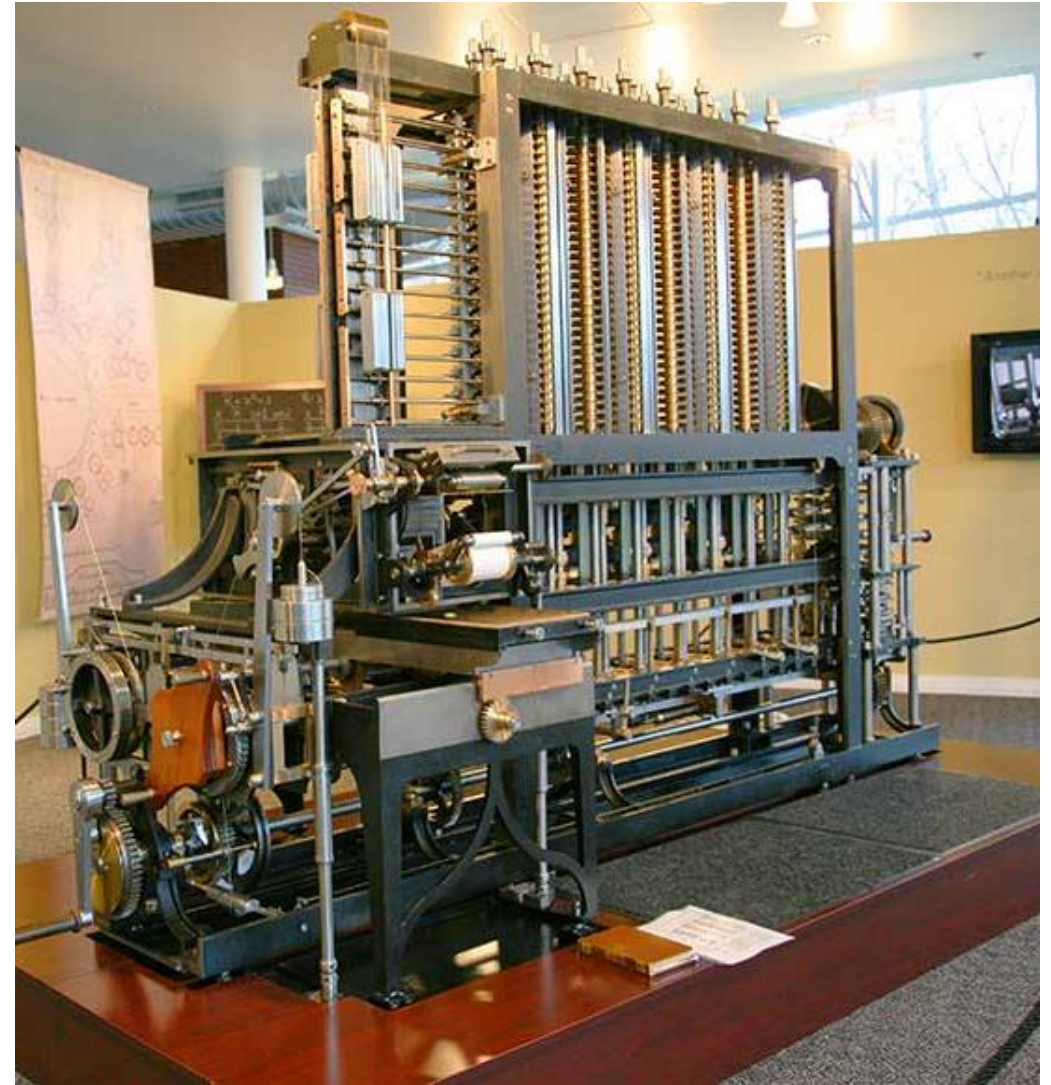
# PASCALINE (1642)

- Mechanical calculator invented by B. Pascal.
- Could add and subtract two numbers directly, and multiply and divide by repetition.



# Babbage Engine

- First automatic computing engine was designed by Charles Babbage in the 19<sup>th</sup> century, but he could not build it.
- The first complete Babbage engine was built in 2002, 153 years after it was designed.
  - 8000 parts.
  - Weighed 5 tons.
  - 11 feet in length.

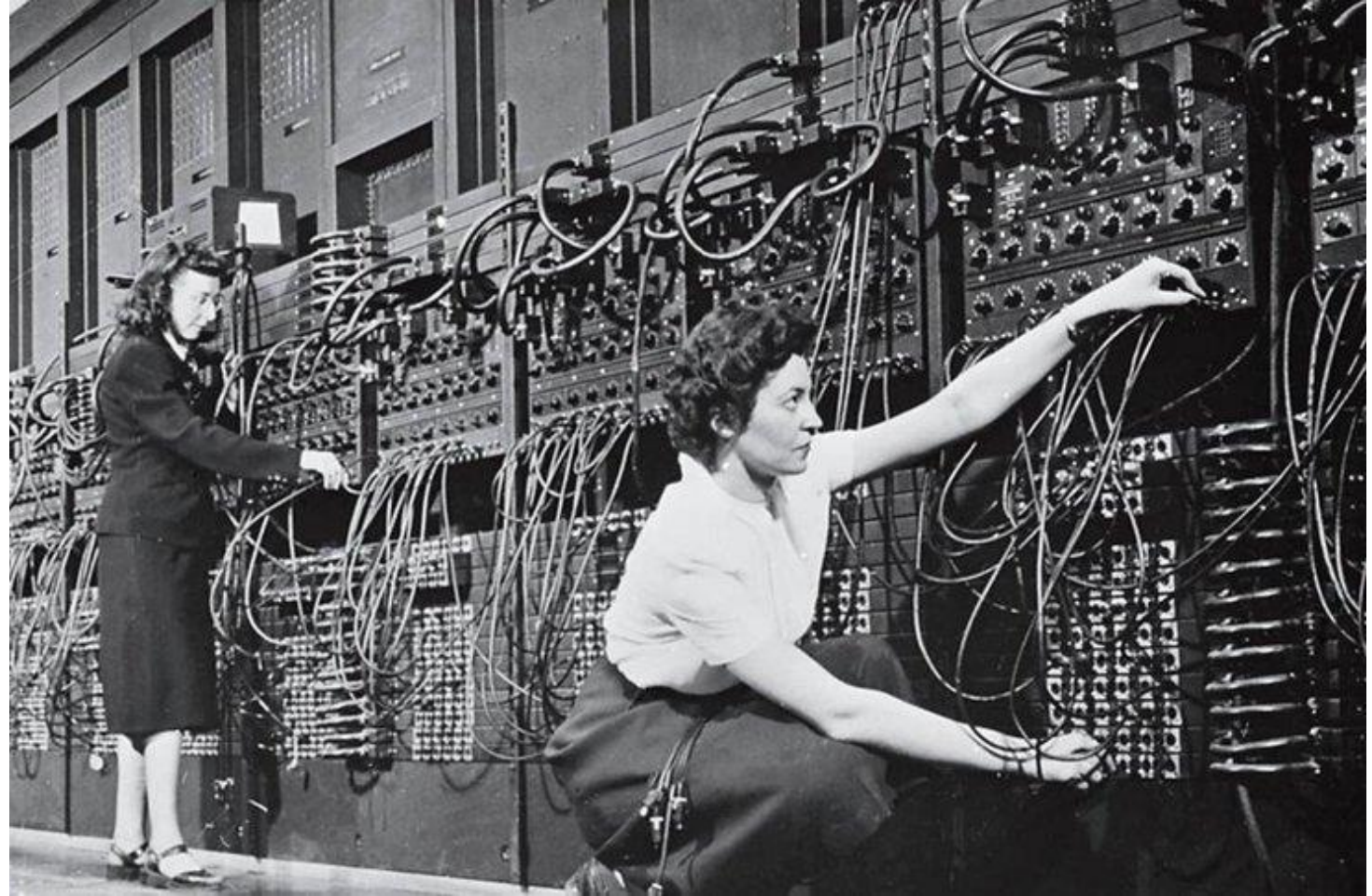




# ENIAC

(Electrical Numerical Integrator and Calculator)

- Developed at the University of Pennsylvania in 1945.
- Used 18,000 vacuum tubes, weighed 30 tons, and occupied a 30ft x 50ft space.





# Harvard Mark 1

- Built at the University of Harvard in 1944, with support from IBM.
- Used mechanical relays (switches) to represent data.
- It weighed 35 tons, and required 500 miles of wiring.



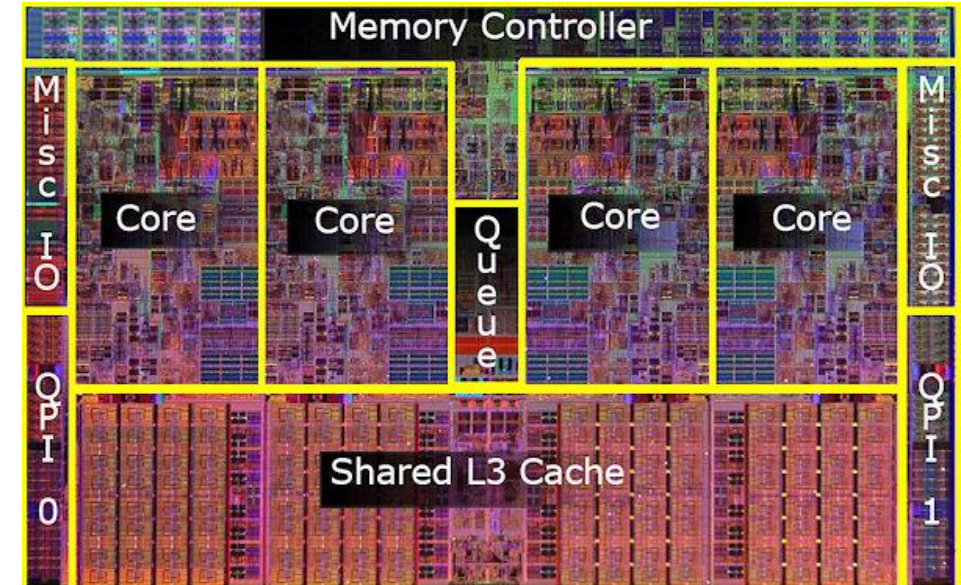
# IBM System/360

- Very popular mainframe computer of the 60s and 70s.
- Introduced many advanced architectural concepts that appeared in microprocessors several decades later.



# Intel Core i7

- A modern processor chip, that comes in dual-core, quad-core and 6-core variants.
- 64-bit processor that comes with various microarchitectures like Haswell, Nehalem, Sandy Bridge, etc.



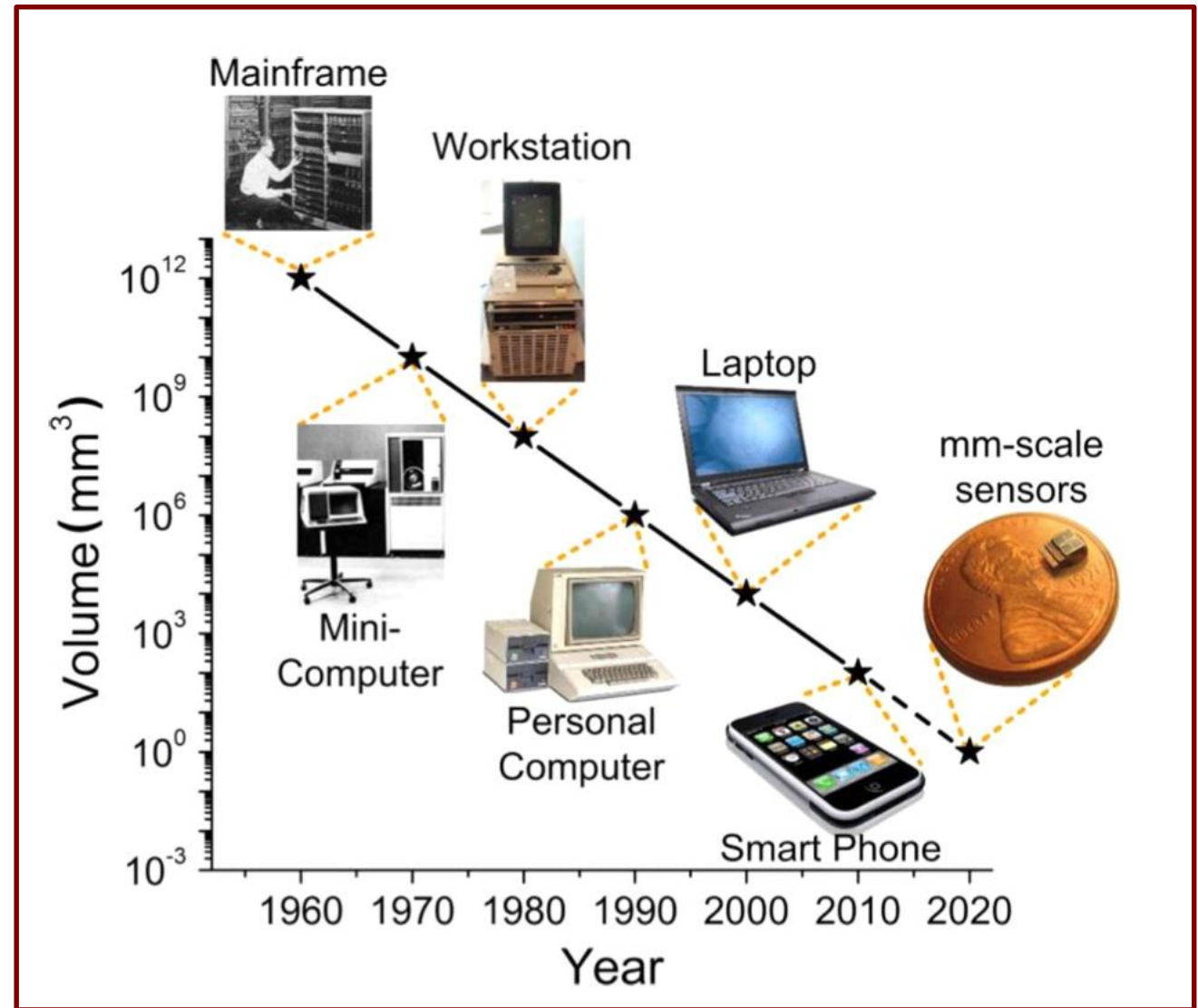
Generation	Main Technology	Representative Systems
First (1945-54)	Vacuum tubes, relays	Machine & assembly language ENIAC, IBM-701
Second (1955-64)	Transistors, memories, I/O processors	Batch processing systems, HLL IBM-7090
Third (1965-74)	SSI and MSI integrated circuits Microprogramming	Multiprogramming / Time sharing IBM 360, Intel 8008
Fourth (1975-84)	LSI and VLSI integrated circuits	Multiprocessors Intel 8086, 8088
Fifth (1984-90)	VLSI, multiprocessor on-chip	Parallel computing, Intel 486
Sixth (1990 onwards)	ULSI, scalable architecture, post-CMOS technologies	Massively parallel processors Pentium, SUN Ultra workstations



# Evolution of the Types of Computer Systems

## The future?

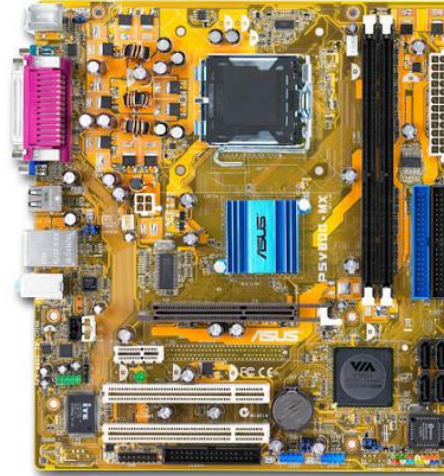
- Large-scale IoT based systems.
- Wearable computing.
- Intelligent objects.



# Evolution of PC form factors over the years



Standard-ATX



Micro-ATX



Mini-ITX



Nano-ITX



Pico-ITX





# Inside a laptop

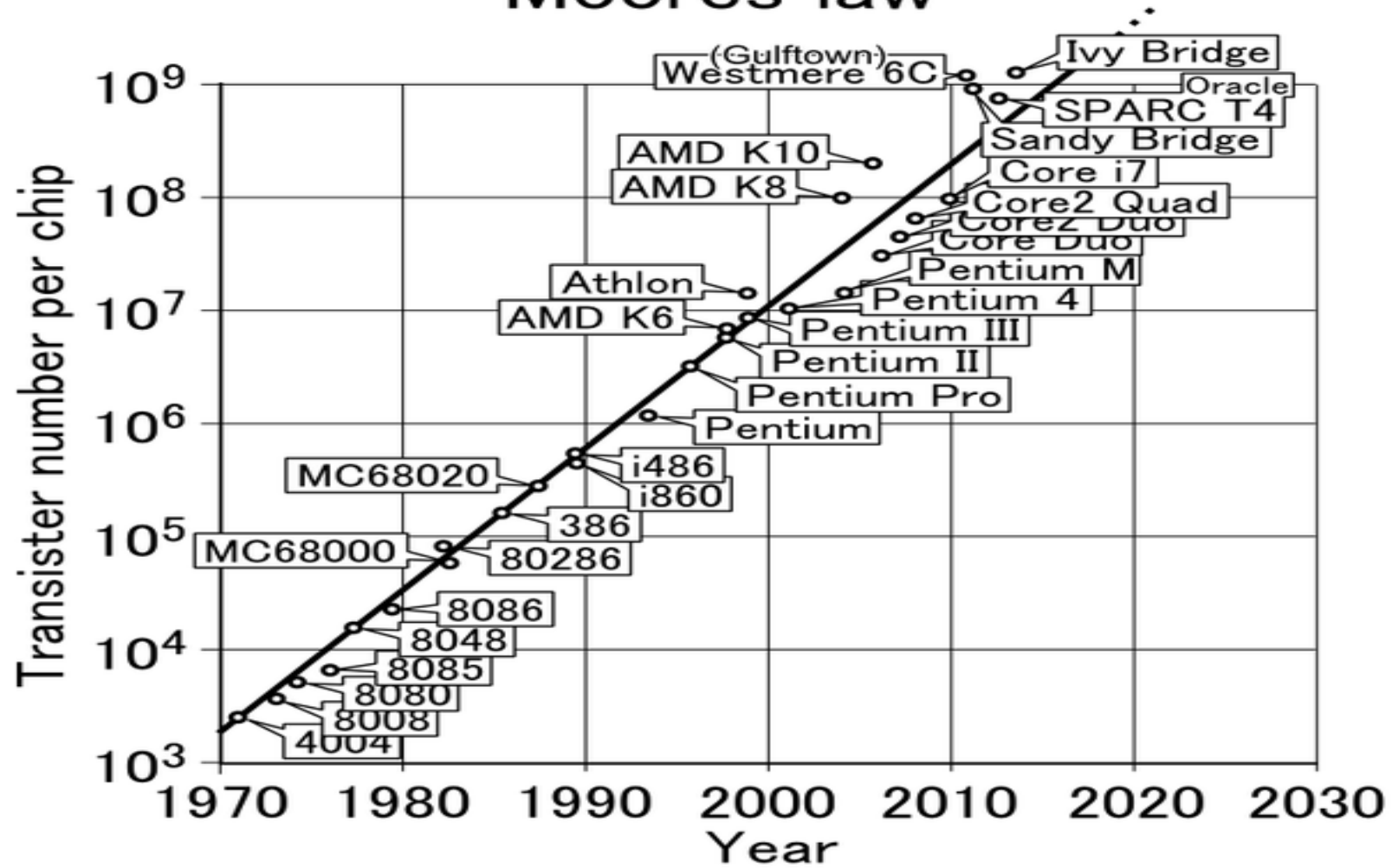
- Miniaturization in feature sizes of all parts.
- Hard drive getting replaced by flash-based memory devices.
- Cooling is a major issue.



# Moore's Law

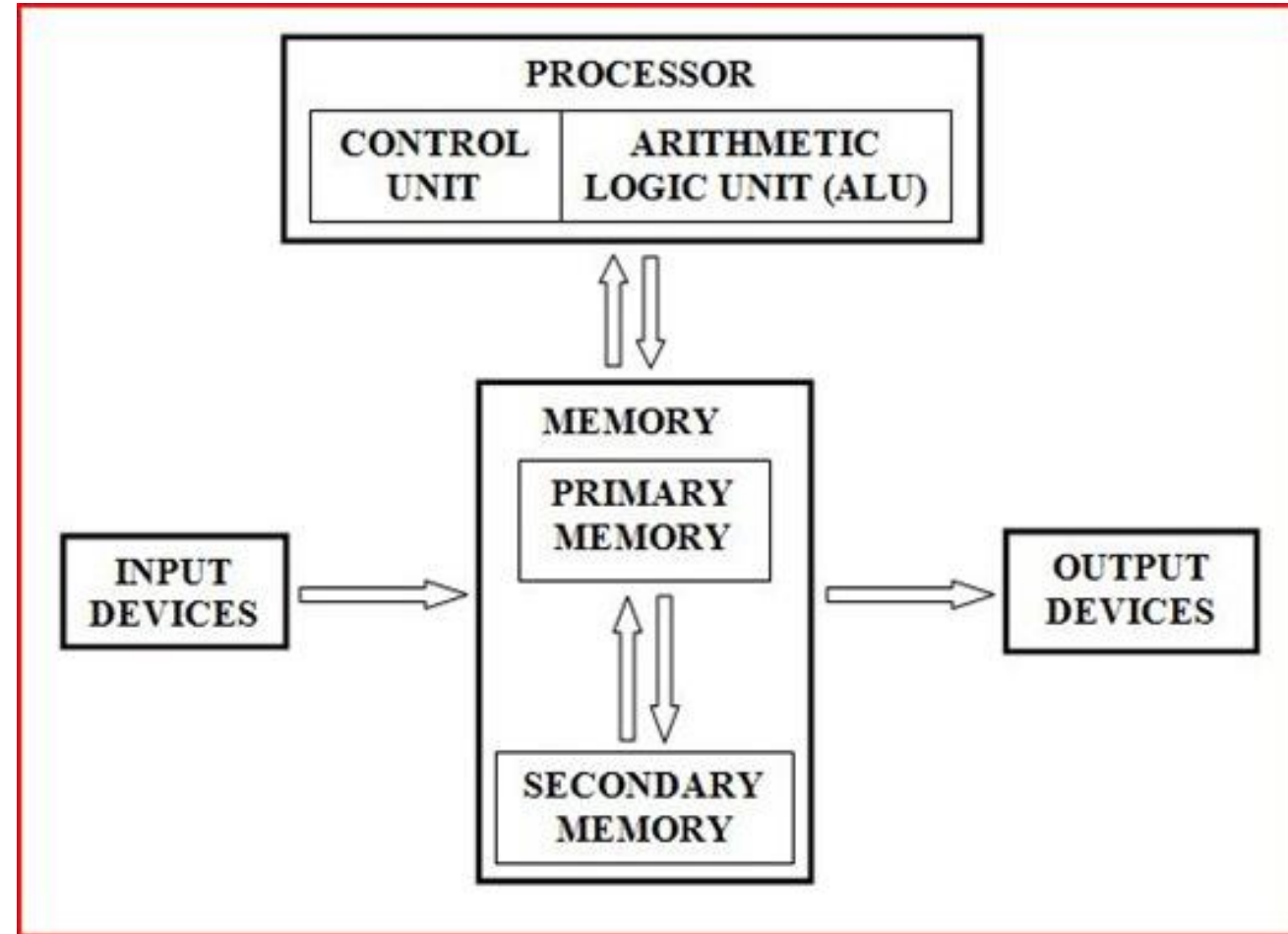
- Refers to an observation made by Intel co-founder Gordon Moore in 1965. He noticed that the number of transistors per square inch on integrated circuits had doubled every year since their invention.
- Moore's law predicts that this trend will continue into the foreseeable future.
- Although the pace has slowed, the number of transistors per square inch has since doubled approximately every 18 months. This is used as the current definition of Moore's law.

# Moore's law



# Simplified Block Diagram of a Computer System

- All instructions and data are stored in memory.
- An instruction and the required data are brought into the processor for execution.
- Input and Output devices interface with the outside world.
- Referred to as *von-Neumann architecture*.



- **Inside the Processor**

- Also called *Central Processing Unit* (CPU).
- Consists of a *Control Unit* and an *Arithmetic Logic Unit* (ALU).
  - All calculations happen inside the ALU.
  - The Control Unit generates sequence of control signals to carry out all operations.
- The processor fetches an instruction from memory for execution.
  - An instruction specifies the exact operation to be carried out.
  - It also specifies the data that are to be operated on.
  - A program refers to a set of instructions that are required to carry out some specific task (e.g. sorting a set of numbers).

- **What is the role of ALU?**

- It contains several registers, some general-purpose and some special-purpose, for temporary storage of data.
- It contains circuitry to carry out logic operations, like AND, OR, NOT, shift, compare, etc.
- It contains circuitry to carry out arithmetic operations like addition, subtraction, multiplication, division, etc.
- During instruction execution, the data (operands) are brought in and stored in some registers, the desired operation carried out, and the result stored back in some register or memory.



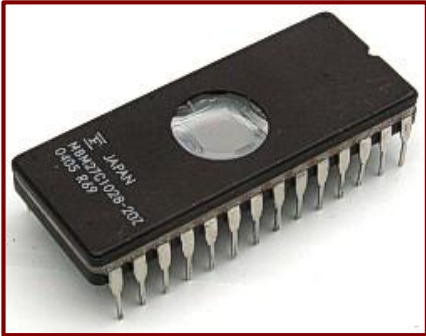
- **What is the role of control unit?**

- Acts as the nerve center that senses the states of various functional units and sends control signals to control their states.
- To carry out a specific operation (say,  $R1 \leftarrow R2 + R3$ ), the control unit must generate control signals in a specific sequence.
  - Enable the outputs of registers R2 and R3.
  - Select the addition operation.
  - Store the output of the adder circuit into register R1.
- When an instruction is fetched from memory, the operation (called *opcode*) is decoded by the control unit, and the control signals issued.

- **Inside the Memory Unit**

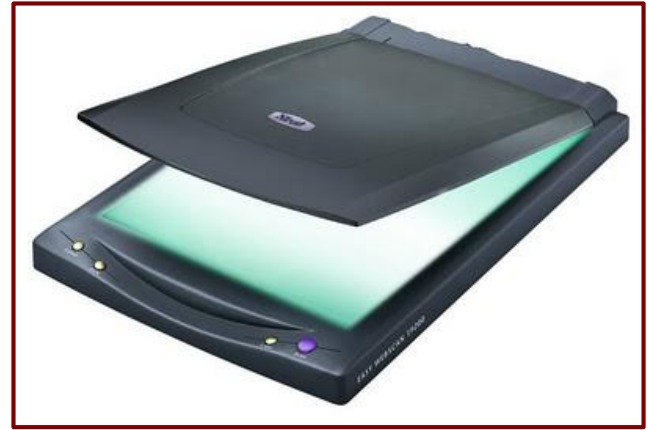
- Two main types of memory subsystems.
  - *Primary or Main memory*, which stores the active instructions and data for the program being executed on the processor.
  - *Secondary memory*, which is used as a backup and stores all active and inactive programs and data, typically as files.
- The processor only has direct access to the primary memory.
- In reality, the memory system is implemented as a hierarchy of several levels.
  - L1 cache, L2 cache, L3 cache, primary memory, secondary memory.
  - Objective is to provide faster memory access at affordable cost.

- Various different types of memory are possible.
  - a) Random Access Memory (RAM), which is used for the cache and primary memory sub-systems. Read and Write access times are independent of the location being accessed.
  - b) Read Only Memory (ROM), which is used as part of the primary memory to store some fixed data that cannot be changed.
  - c) Magnetic Disk, which uses direction of magnetization of tiny magnetic particles on a metallic surface to store data. Access times vary depending on the location being accessed, and is used as secondary memory.
  - d) Flash Memory, which is replacing magnetic disks as secondary memory devices. They are faster, but smaller in size as compared to disk.



# Input Unit

- Used to feed data to the computer system from the external environment.
  - Data are transferred to the processor/memory after appropriate encoding.
- Common input devices:
  - Keyboard
  - Mouse
  - Joystick
  - Camera





# Output Unit

- Used to send the result of some computation to the outside world.
- Common output devices:
  - LCD/LED screen
  - Printer and Plotter
  - Speaker / Buzzer
  - Projection system



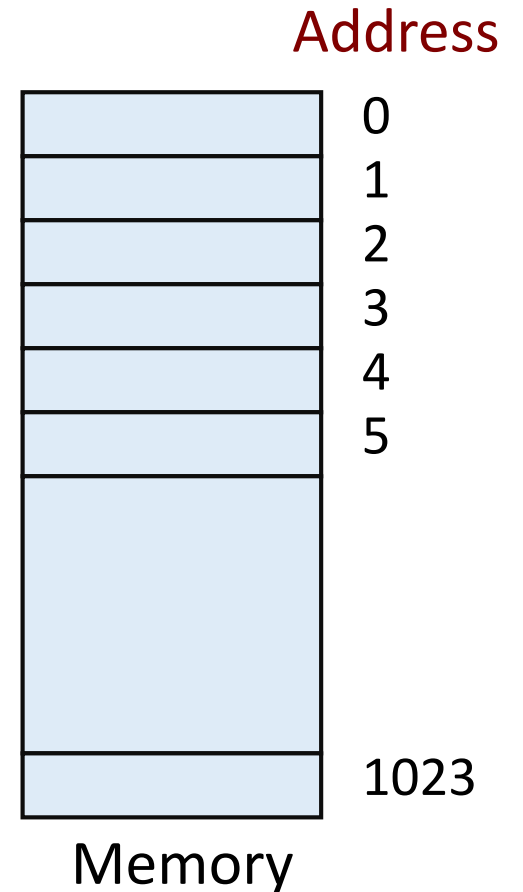
# **Basic Operation of a Computer**

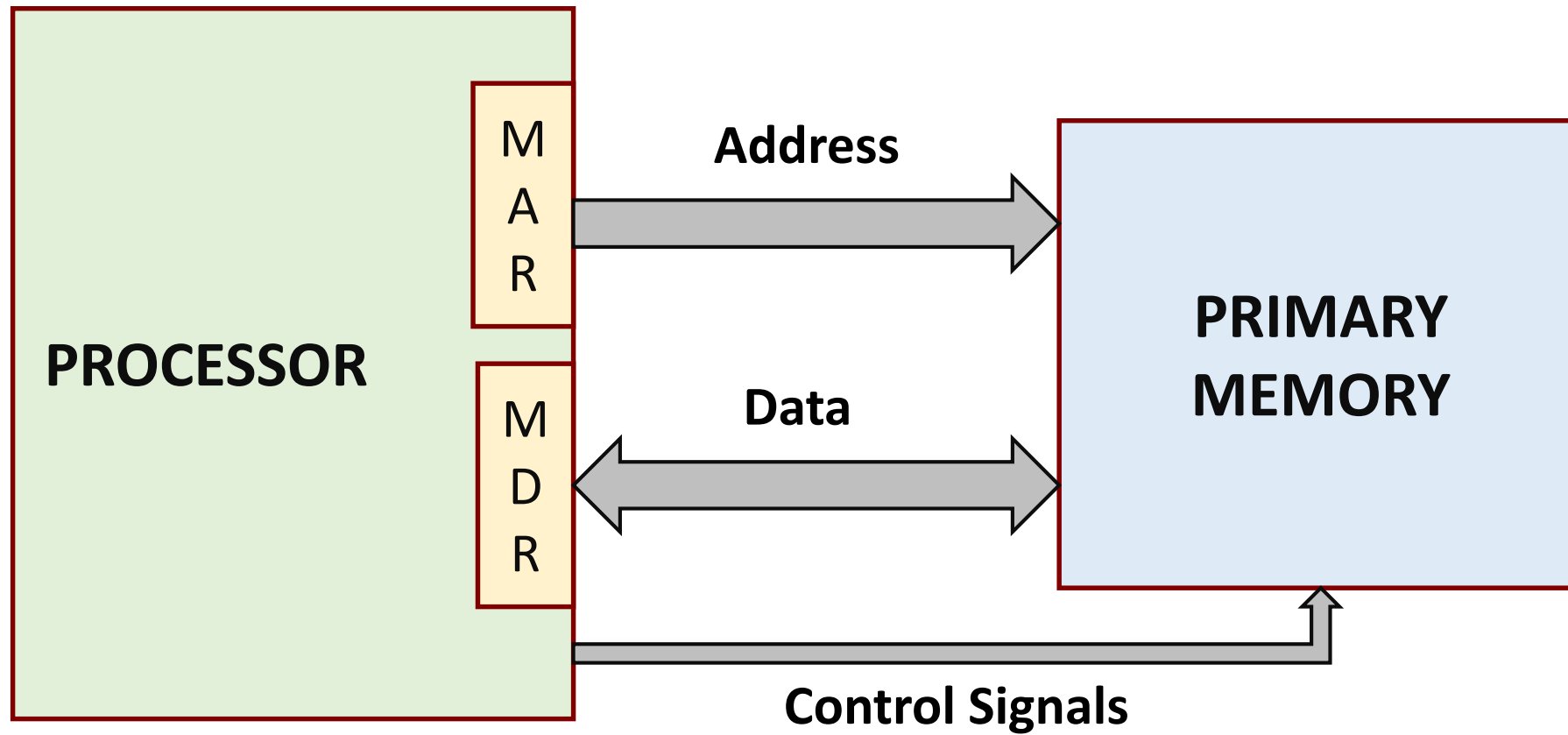
# Introduction

- The basic mechanism through which an instruction gets executed shall be illustrated.
- May be recalled:
  - ALU contains a set of registers, some general-purpose and some special-purpose.
  - First we briefly explain the functions of the special-purpose registers before we look into some examples.

# For Interfacing with the Primary Memory

- Two special-purpose registers are used:
  - **Memory Address Register (MAR)**: Holds the address of the memory location to be accessed.
  - **Memory Data Register (MDR)**: Holds the data that is being written into memory, or will receive the data being read out from memory.
- Memory considered as a linear array of storage locations (bytes or words) each with unique address.





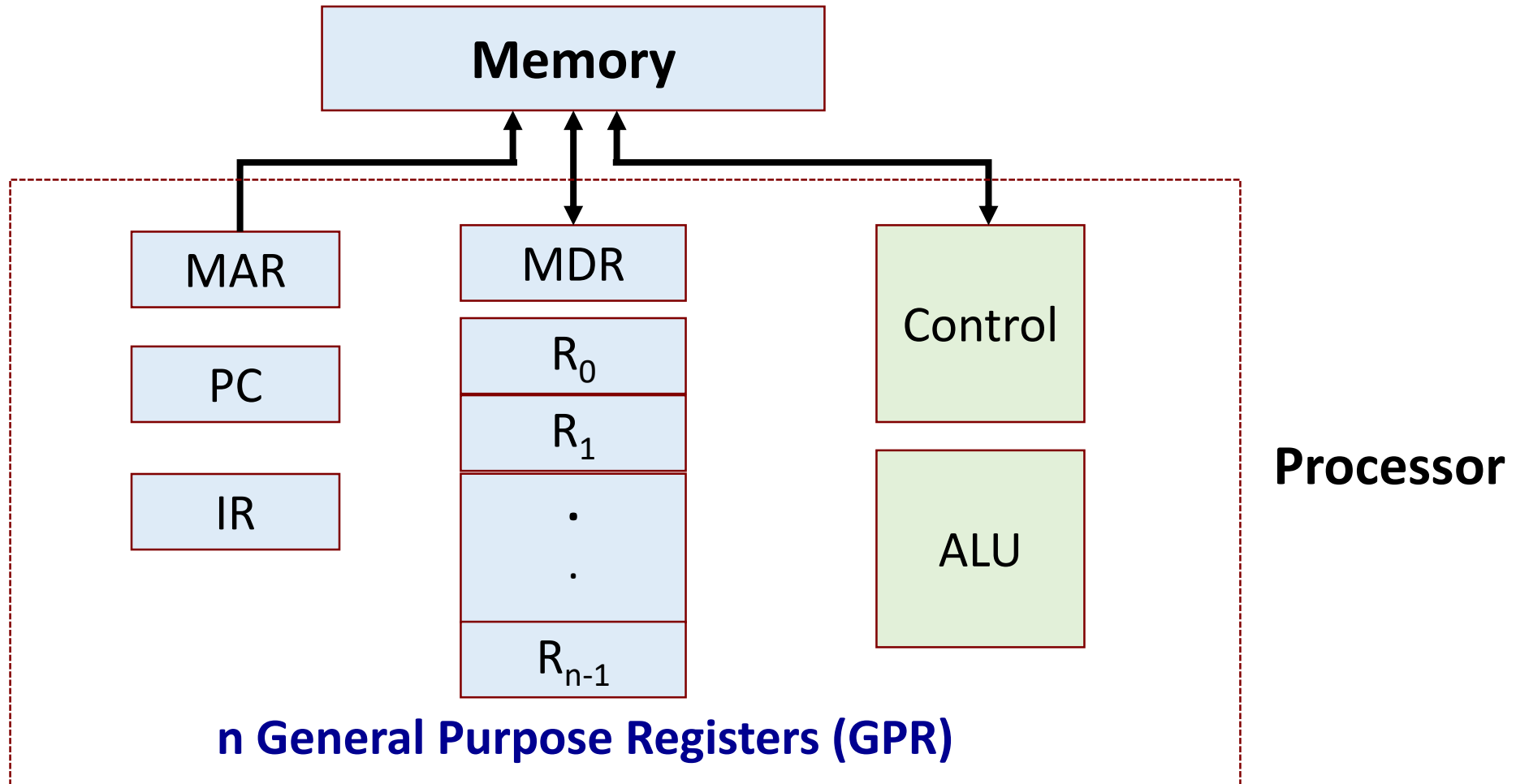


- To read data from memory
  - Load the memory address into MAR.
  - Issue the control signal READ.
  - The data read from the memory is stored into MDR.
- To write data into memory
  - Load the memory address into MAR.
  - Load the data to be written into MDR.
  - Issue the control signal WRITE.

# For Keeping Track of Program / Instructions

- Two special-purpose registers are used:
  - **Program Counter (PC)**: Holds the memory address of the next instruction to be executed.
    - Automatically incremented to point to the next instruction when an instruction is being executed.
  - **Instruction Register (IR)**: Temporarily holds an instruction that has been fetched from memory.
    - Need to be decoded to find out the instruction type.
    - Also contains information about the location of the data.

# Architecture of the Example Processor



# Example Instructions

- We shall illustrate the process of instruction execution with the help of the following two instructions:

a) **ADD R1, LOCA**

Add the contents of memory location LOCA (i.e. address of the memory location is LOCA) to the contents of register R1.

$$R1 \leftarrow R1 + \text{Mem}[\text{LOCA}]$$

b) **ADD R1, R2**

Add the contents of register R2 to the contents of register R1.

$$R1 \leftarrow R1 + R2$$

## Execution of *ADD R1,LOCA*

- Assume that the instruction is stored in memory location 1000, the initial value of R1 is 50, and LOCA is 5000.
- Before the instruction is executed, PC contains 1000.
- Content of PC is transferred to MAR.  $MAR \leftarrow PC$
- READ request is issued to memory unit.
- The instruction is fetched to MDR.  $MDR \leftarrow Mem[MAR]$
- Content of MDR is transferred to IR.  $IR \leftarrow MDR$
- PC is incremented to point to the next instruction.  $PC \leftarrow PC + 4$
- The instruction is decoded by the control unit.

ADD R1	5000
--------	------

- LOCA (i.e. 5000) is transferred (from IR) to MAR.
- READ request is issued to memory unit.
- The data is fetched to MDR.
- The content of MDR is added to R1.

$MAR \leftarrow IR[Operand]$

$MDR \leftarrow Mem[MAR]$

$R1 \leftarrow R1 + MDR$

The steps being carried out are called **micro-operations**:

$MAR \leftarrow PC$

$MDR \leftarrow Mem[MAR]$

$IR \leftarrow MDR$

$PC \leftarrow PC + 4$

$MAR \leftarrow IR[Operand]$

$MDR \leftarrow Mem[MAR]$

$R1 \leftarrow R1 + MDR$



R1

125

Address	Content
1000	ADD R1, LOCA
1004	...

5000	75
------	----

LOCA

1. PC = 1000
2. MAR = 1000
3. PC = PC + 4 = 1004
4. MDR = ADD R1, LOCA
5. IR = ADD R1, LOCA
6. MAR = LOCA = 5000
7. MDR = 75
8. R1 = R1 + MDR = 50 + 75 =  
125

## Execution of *ADD R1,R2*

- Assume that the instruction is stored in memory location 1500, the initial value of R1 is 50, and R2 is 200.
- Before the instruction is executed, PC contains 1500.
- Content of PC is transferred to MAR.
- READ request is issued to memory unit.
- The instruction is fetched to MDR.
- Content of MDR is transferred to IR.
- PC is incremented to point to the next instruction.
- The instruction is decoded by the control unit.
- R2 is added to R1.



ADD R1, R2

$MAR \leftarrow PC$

$MDR \leftarrow Mem[MAR]$

$IR \leftarrow MDR$

$PC \leftarrow PC + 4$

$R1 \leftarrow R1 + R2$

R1    250

R2    200

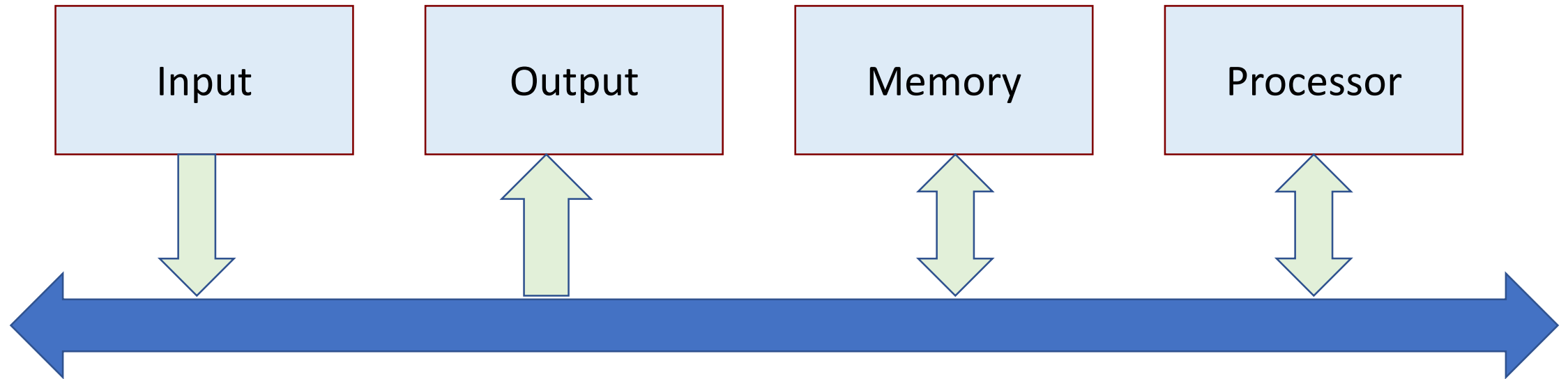
Address	Instruction
1500	ADD R1, R2
1504	...

1. PC = 1500
2. MAR = 1500
3. PC = PC + 4 = 1504
4. MDR = ADD R1, R2
5. IR = ADD R1, R2
6. R1 = R1 + R2 = 250

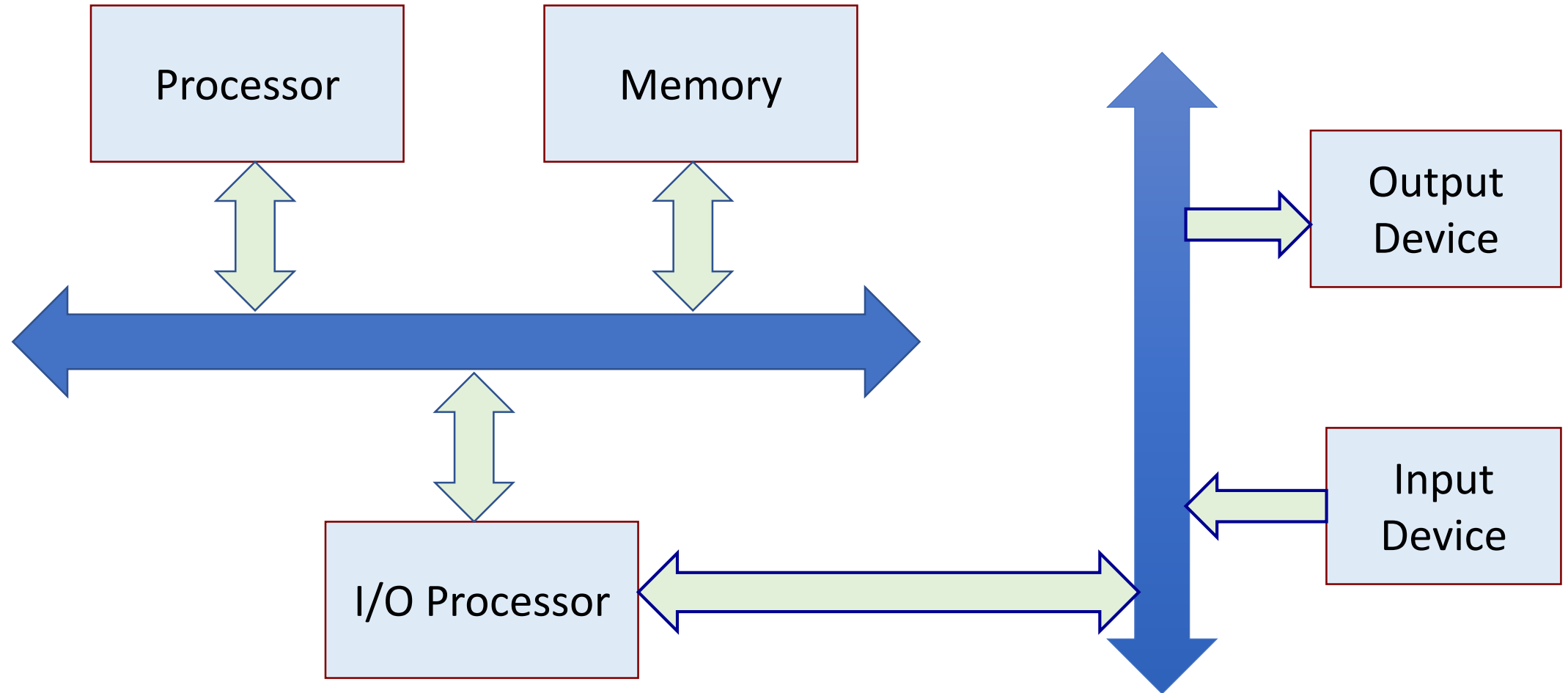
# Bus Architecture

- The different functional modules must be connected in an organized manner to form an operational system.
- Bus refers to a group of lines that serves as a connecting path for several devices.
- The simplest way to connect the functional unit is to use the single bus architecture.
  - Only one data transfer allowed in one clock cycle.
  - For multi-bus architecture, parallelism in data transfer is allowed.

# System-Level Single Bus Architecture



# System-Level Two-Bus Architecture

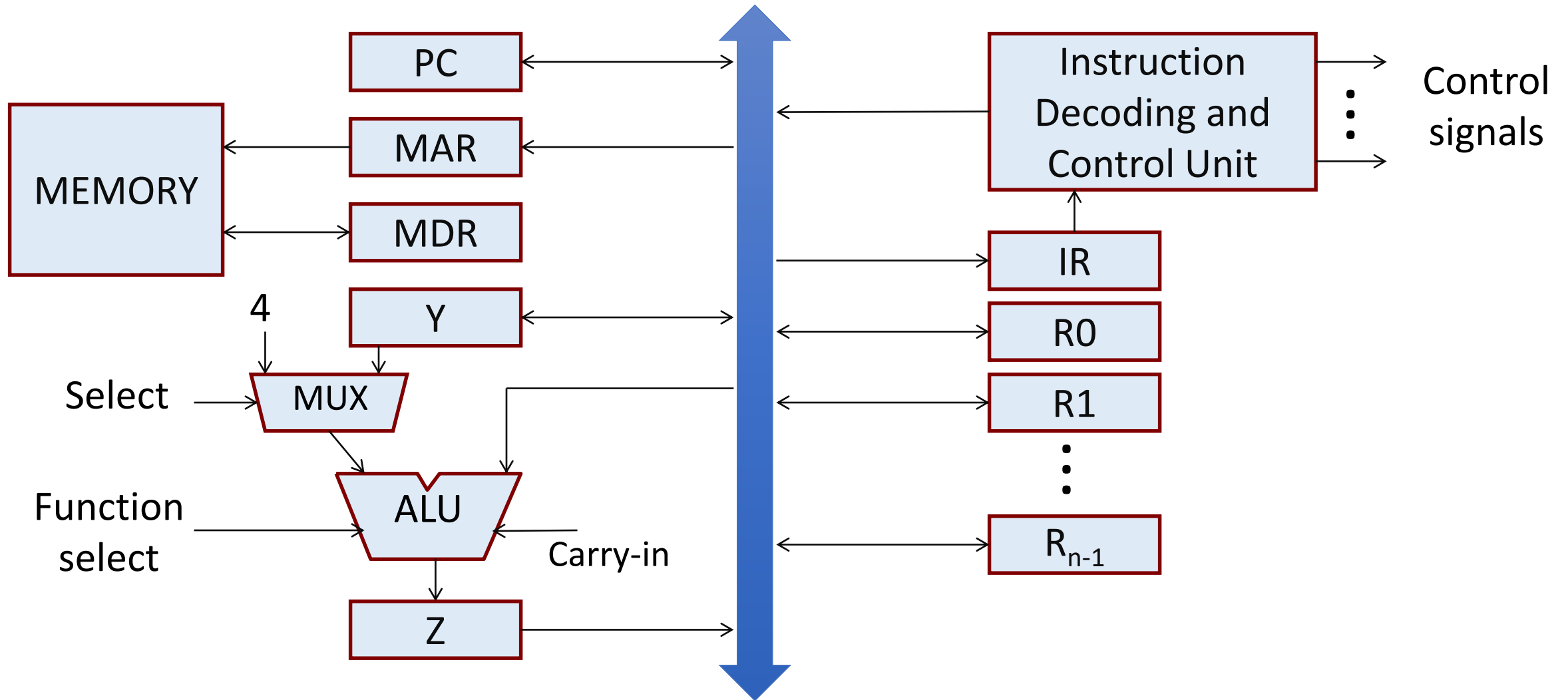




# Single-Bus Architecture Inside the Processor

- There is a single bus inside the processor.
  - ALU and the registers are all connected via the single bus.
  - This bus is internal to the processor and should not be confused with the external bus that connects the processor to the memory and I/O devices.
- A typical single-bus processor architecture is shown on the next slide.
  - Two temporary registers *Y* and *Z* are also included.
  - Register *Y* temporarily holds one of the operands of the ALU.
  - Register *Z* temporarily holds the result of the ALU operation.
  - The multiplexer selects a constant operand 4 during execution of the micro-operation:  $PC \leftarrow PC + 4$ .

## Internal Processor Bus



# Multi-Bus Architectures

- Modern processors have multiple buses that connect the registers and other functional units.
  - Allows multiple data transfer micro-operations to be executed in the same clock cycle.
  - Results in overall faster instruction execution.
- Also advantageous to have multiple shorter buses rather than a single long bus.
  - Smaller parasitic capacitance, and hence smaller delay.

# **Memory Addressing and Languages**

# Overview of Memory Organization

- Memory is one of the most important sub-systems of a computer that determines the overall performance.
- Conceptual view of memory:
  - Array of storage locations, with each storage location having a unique address.
  - Each storage location can hold a fixed amount of information (multiple of bits, which is the basic unit of data storage).
- A memory system with  $M$  locations and  $N$  bits per location, is referred to as an  $M \times N$  memory.
  - Both  $M$  and  $N$  are typically some powers of 2.
  - Example: 1024 x 8, 65536 x 32, etc.

# Some Terminologies

- Bit: A single binary digit (0 or 1).
- Nibble: Collection of 4 bits.
- Byte: Collection of 8 bits.
- Word: Does not have a unique definition.
  - Varies from one computer to another; typically 32 or 64 bits.

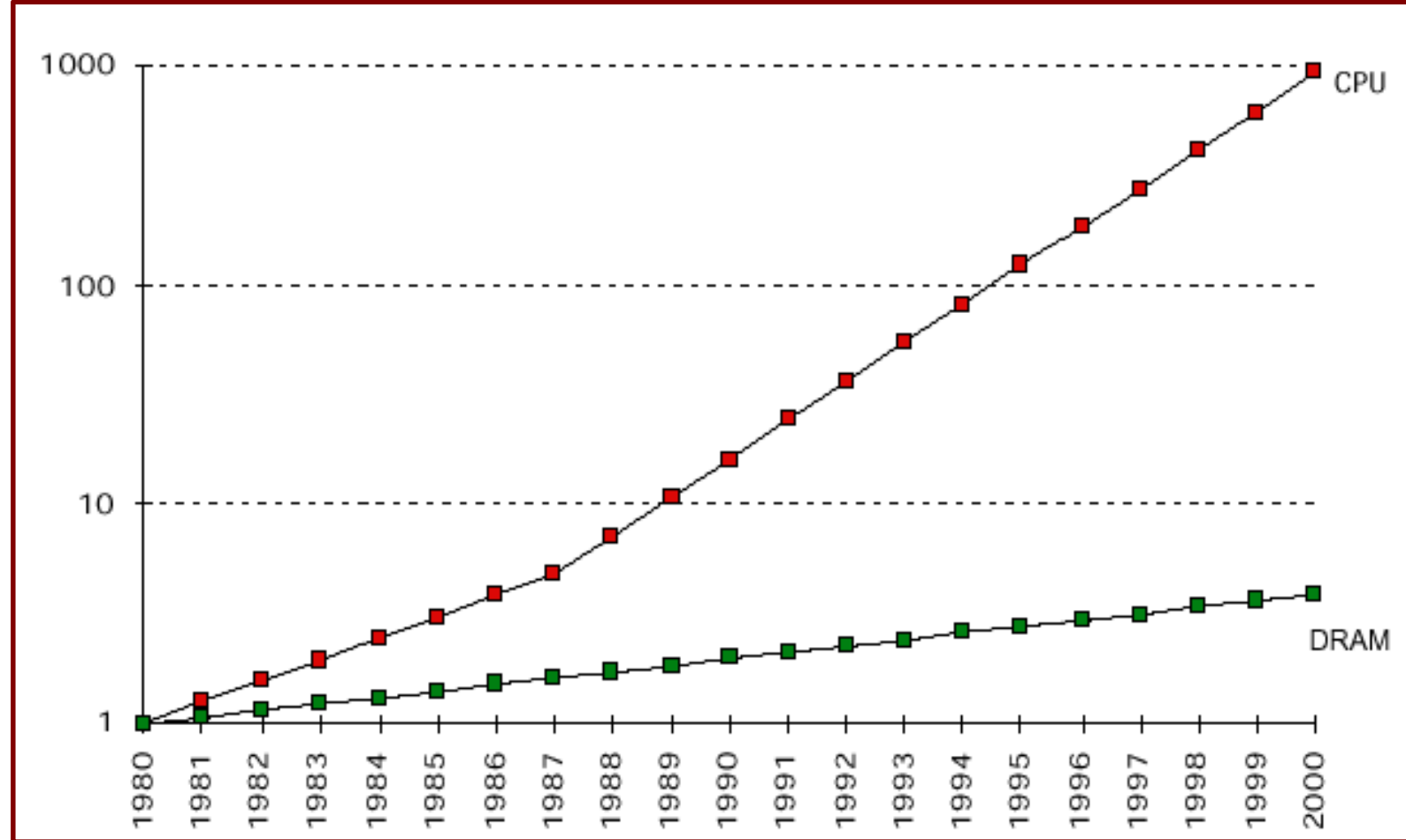


# How is Memory Organized?

- Memory is *byte or word organized*.
  - Every byte or w of the memory has a unique address.
- Multiple bytes of data can be accessed by an instruction.
  - Example: *Half-word* (2 bytes), *Word* (4 bytes), *Long Word* (8 bytes).
- For higher data transfer rate, memory is often organized such that multiple bytes can be read or written simultaneously.
  - Necessary to bridge the processor-memory speed gap.
  - Shall be discussed later in detail.

## Processor-Memory Performance Gap

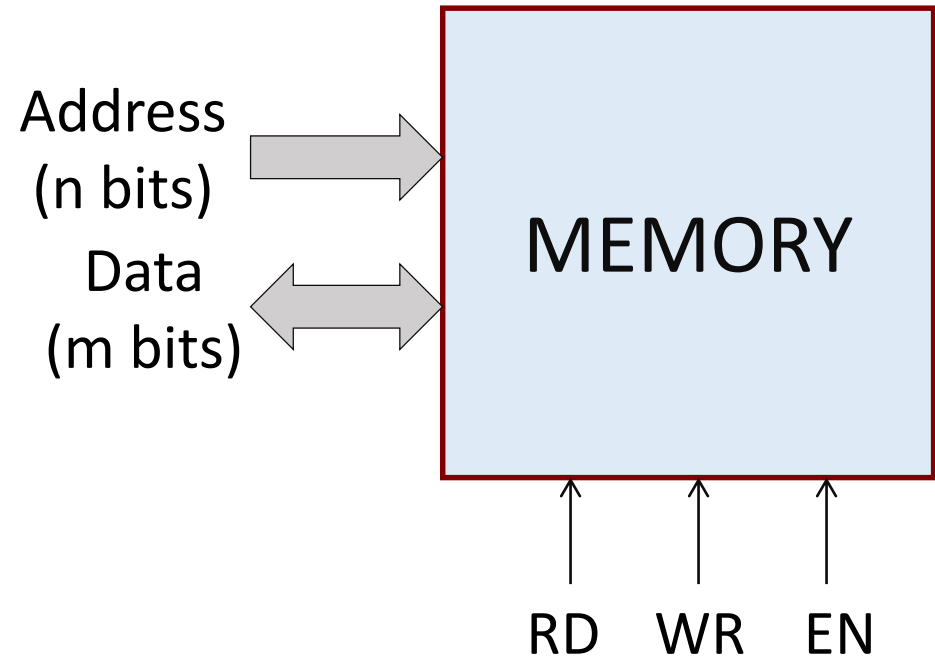
- With technological advancements, both processor and memory are becoming faster.
- However, the speed gap is steadily increasing.
- Special techniques are needed to bridge this gap.
  - Cache memory
  - Memory interleaving



# How do we Specify Memory Sizes?

Unit	Bytes	In Decimal
8 bits (B)	1 or $2^0$	$10^0$
Kilobyte (KB)	1024 or $2^{10}$	$10^3$
Megabyte (MB)	1,048,576 or $2^{20}$	$10^6$
Gigabyte (GB)	1,073,741,824 or $2^{30}$	$10^9$
Terabyte (TB)	1,099,511,627,776 or $2^{40}$	$10^{12}$
Petabyte (PB)	$2^{50}$	$10^{15}$
Exabyte (EB)	$2^{60}$	$10^{18}$
Zettabyte (ZB)	$2^{70}$	$10^{21}$

- If there are  $n$  bits in the address, the maximum number of storage locations can be  $2^n$ .
  - For  $n=8$ , 256 locations.
  - For  $n=16$ , 64K locations.
  - For  $n=20$ , 1M locations.
  - For  $n=32$ , 4G locations.
- Modern-day memory chips can store several Gigabits of data.
  - Dynamic RAM (DRAM).



Address	Contents
0000 0000	0000 0000 0000 0001
0000 0001	0000 0100 0101 0000
0000 0010	1010 1000 0000 0000
⋮	⋮
1111 1111	1011 0000 0000 1010

An example:  $2^8 \times 16$  memory

# Some Examples

1. A computer has 64 MB (megabytes) of byte-addressable memory. How many bits are needed in the memory address?
  - Address Space = 64 MB =  $2^6 \times 2^{20}$  B =  $2^{26}$  B
  - If the memory is byte addressable, we need 26 bits of address.
2. A computer has 1 GB of memory. Each word in this computer is 32 bits. How many bits are needed to address any single word in memory?
  - Address Space = 1 GB =  $2^{30}$  B
  - 1 word = 32 bits = 4 B
  - We have  $2^{30} / 4 = 2^{28}$  words
  - Thus, we require 28 bits to address each word.



# Byte Ordering Conventions

- Many data items require multiple bytes for storage.
- Different computers use different data ordering conventions.
  - Low-order byte first
  - High-order byte first
- Thus a 16-bit number **11001100 00101010** can be stored as either:

**11001100 00101010** or **00101010 11001100**

Data Type	Size (in Bytes)
Character	1
Integer	4
Long integer	8
Floating-point	4
Double-precision	8

Typical data sizes

- The two conventions have been named as:

- a) Little Endian**

- The least significant byte is stored at lower address followed by the most significant byte.  
Examples: Intel processors, DEC alpha, etc.
    - Same concept followed for arbitrary multi-byte data.

- b) Big Endian**

- The most significant byte is stored at lower address followed by the least significant byte.  
Examples: IBM's 370 mainframes, Motorola microprocessors, TCP/IP, etc.
    - Same concept followed for arbitrary multi-byte data.

# An Example

- Represent the following 32-bit number in both Little-Endian and Big-Endian in memory from address 2000 onwards:

01010101 00110011 00001111 11000011

Little Endian	
Address	Data
2000	11000011
2001	00001111
2002	00110011
2003	01010101

Big Endian	
Address	Data
2000	01010101
2001	00110011
2002	00001111
2003	11000011

# Memory Access by Instructions

- The program instructions and data are stored in memory.
  - In *von-Neumann architecture*, they are stored in the same memory.
  - In *Harvard architecture*, they are stored in different memories.
- For executing the program, two basic operations are required.
  - a) Load:** The contents of a specified memory location is read into a processor register.  
*LOAD R1, 2000*
  - b) Store:** The contents of a processor register is written into a specified memory location.  
*STORE 2020, R3*

# An Example

- Compute  $S = (A + B) - (C - D)$

LOAD R1,A

LOAD R2,B

ADD R3,R1,R2 //  $R3 = A + B$

LOAD R1,C

LOAD R2,D

SUB R4,R1,R2 //  $R4 = C - D$

SUB R3,R3,R4 //  $R3 = R3 - R4$

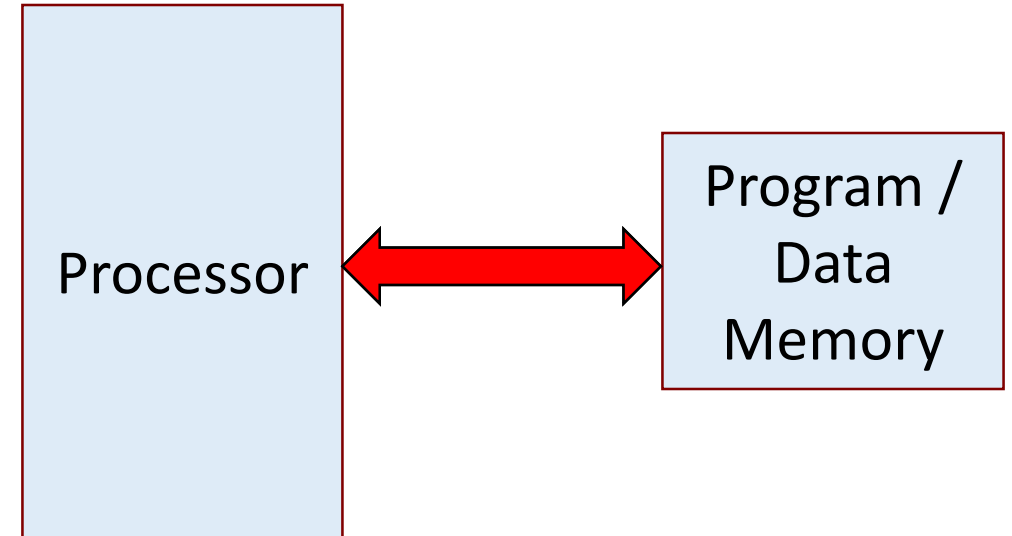
STORE S,R3

# Classification of Computer Architecture

- Broadly can be classified into two types:
  - a) Von-Neumann architecture
  - b) Harvard architecture
- How is a computer different from a calculator?
  - They have similar circuitry inside (e.g. for doing arithmetic).
  - In a calculator, user has to interactively give the sequence of commands.
  - In contrast, a computer works using the *stored-program* concept.
  - Write a program, store it in memory, and run it in one go.

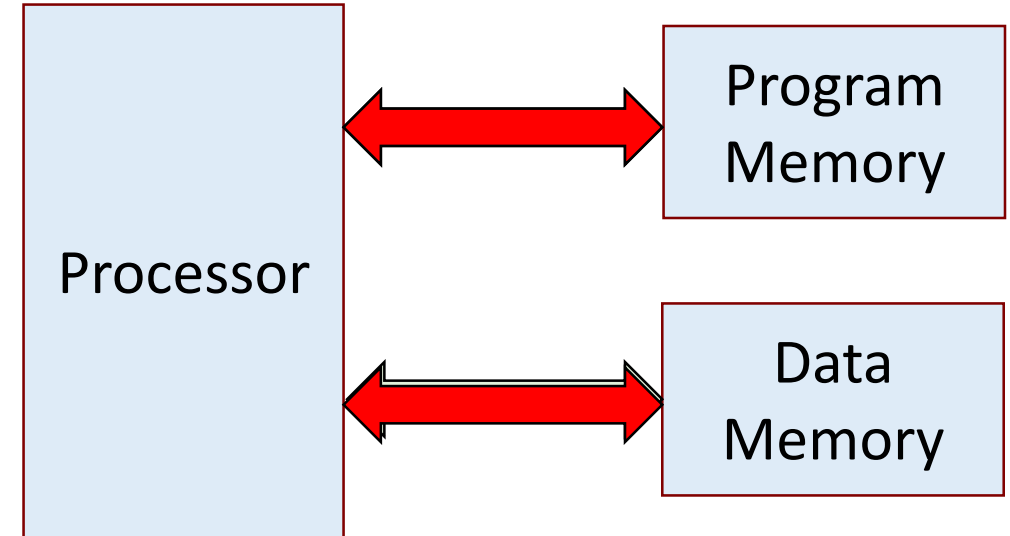
# von-Neumann Architecture

- Instructions and data are stored in the *same* memory module.
- More flexible and easier to implement.
- Suitable for most of the general purpose processors.
- General disadvantage:
  - The processor-memory bus acts as the bottleneck.
  - All instructions and data are moved back and forth through the pipe.



# Harvard Architecture

- Separate memory for program and data.
  - Instructions are stored in program memory and data are stored in data memory.
- Instruction and data accesses can be done in parallel.
- Some microcontrollers and pipelines with separate instruction and data caches follow this concept.
- The processor-memory bottleneck remains.





# Pipeline in Executing Instructions

- Instruction execution is typically divided into 5 stages:
  - Instruction Fetch (IF)
  - Instruction Decode (ID)
  - ALU operation (EX)
  - Memory Access (MEM)
  - Write Back result to register file (WB)
- These five stage can be executed in an overlapped fashion in a pipeline architecture.
  - Results in significant speedup by overlapping instruction execution.

# Basic 5-stage Pipelining Diagram

Instruction	Pipeline Stage						
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM
5					IF	ID	EX
Clock Cycle	1	2	3	4	5	6	7

# How can Harvard Architecture Help?

- In clock cycle 4, instruction 4 is trying to fetch an instruction (IF), while instruction 1 may be trying to access data (MEM).
  - In von-Neumann architecture, one of these two operations will have to wait resulting in pipeline slowdown.
  - In Harvard architecture, the operations can go on without any speed penalty as the instruction and data memories are separate.

# **Instruction Set Architecture**

# Introduction

- Instruction Set Architecture (ISA)
  - Serves as an interface between software and hardware.
  - Typically consists of information regarding the programmer's view of the architecture (i.e. the registers, address and data buses, etc.).
  - Also consists of the instruction set.
- Many ISA's are not specific to a particular computer architecture.
  - They survive across generations.
  - Classic examples: IBM 360 series, Intel x86 series, etc.

# Instruction Set Design Issues

- Number of explicit operands:
  - 0, 1, 2 or 3.
- Location of the operands:
  - Registers, accumulator, memory.
- Specification of operand locations:
  - **Addressing modes**: register, immediate, indirect, relative, etc.
- Sizes of operands supported:
  - Byte (8-bits), Half-word (16-bits), Word (32-bits), Double (64-bits), etc.
- Supported operations:
  - ADD, SUB, MUL, AND, OR, CMP, MOVE, JMP, etc.

# Evolution of Instruction Sets

- |                             |              |                         |
|-----------------------------|--------------|-------------------------|
| 1. Accumulator based:       | 1960's       | (EDSAC, IBM 1130)       |
| 2. Stack based:             | 1960-70      | (Burroughs 5000)        |
| 3. Memory-Memory based:     | 1970-80      | (IBM 360)               |
| 4. Register-Memory based:   | 1970-present | (Intel x86)             |
| 5. Register-Register based: | 1960-present | (MIPS, CDC 6600, SPARC) |

1: 1-address instructions:

$\text{ADD } X \rightarrow \text{ACC} = \text{ACC} + \text{Mem}[X]$

2: 0-address instructions:

$\text{ADD} \rightarrow \text{TOS} = \text{TOS} + \text{NEXT}$

3: 2- or 3-address instructions:

$\text{ADD } A, B \rightarrow \text{Mem}[A] = \text{Mem}[A] + \text{Mem}[B]$

$\text{ADD } A, B, C \rightarrow \text{Mem}[A] = \text{Mem}[B] + \text{Mem}[C]$

5: 3-address instructions:

$\text{ADD } R1, R2, R3 \rightarrow R1 = R2 + R3$

4: 2-address instructions:

$\text{LOAD } R1, X \rightarrow R1 = \text{Mem}[X]$

# Example Code Sequence for $Z = X + Y$

- Stack machine:

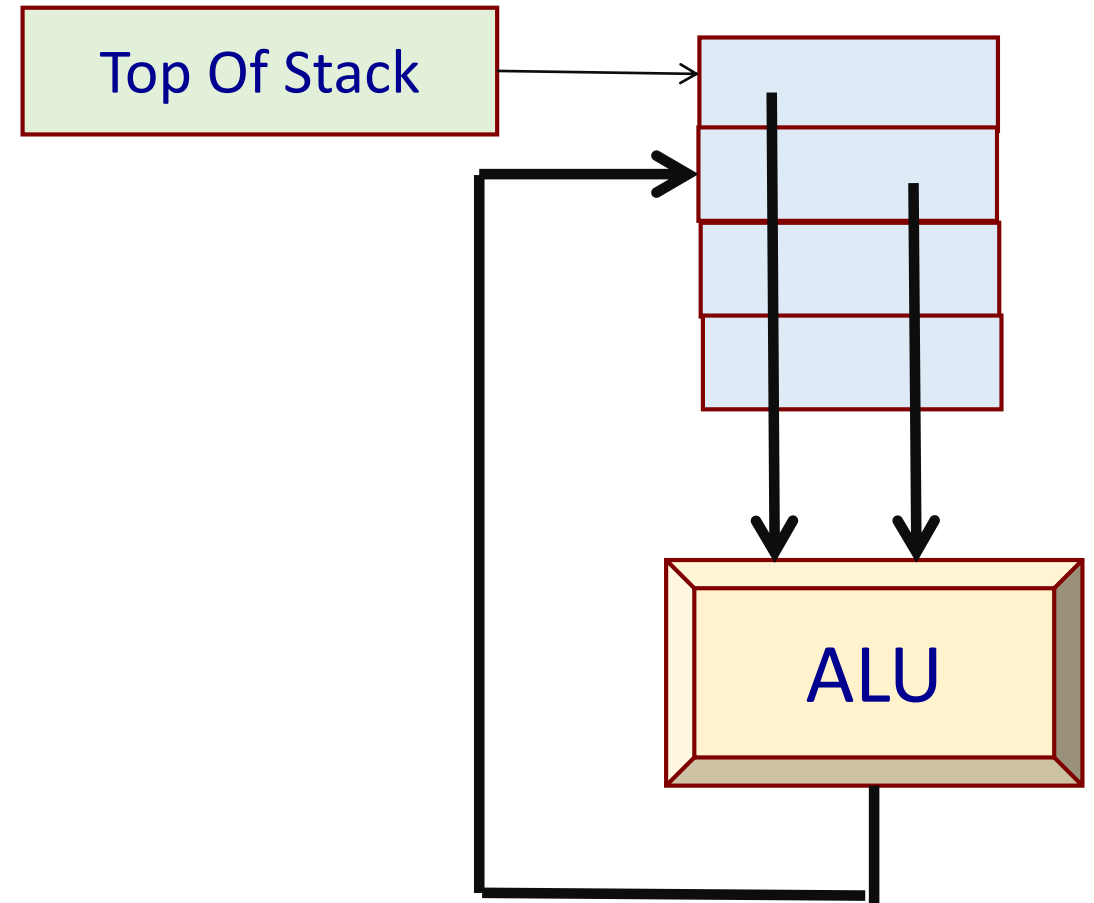
**PUSH X**

**PUSH Y**

**ADD**

**POP Z**

- The ADD instruction pops two elements from stack, adds them, and pushes back result.





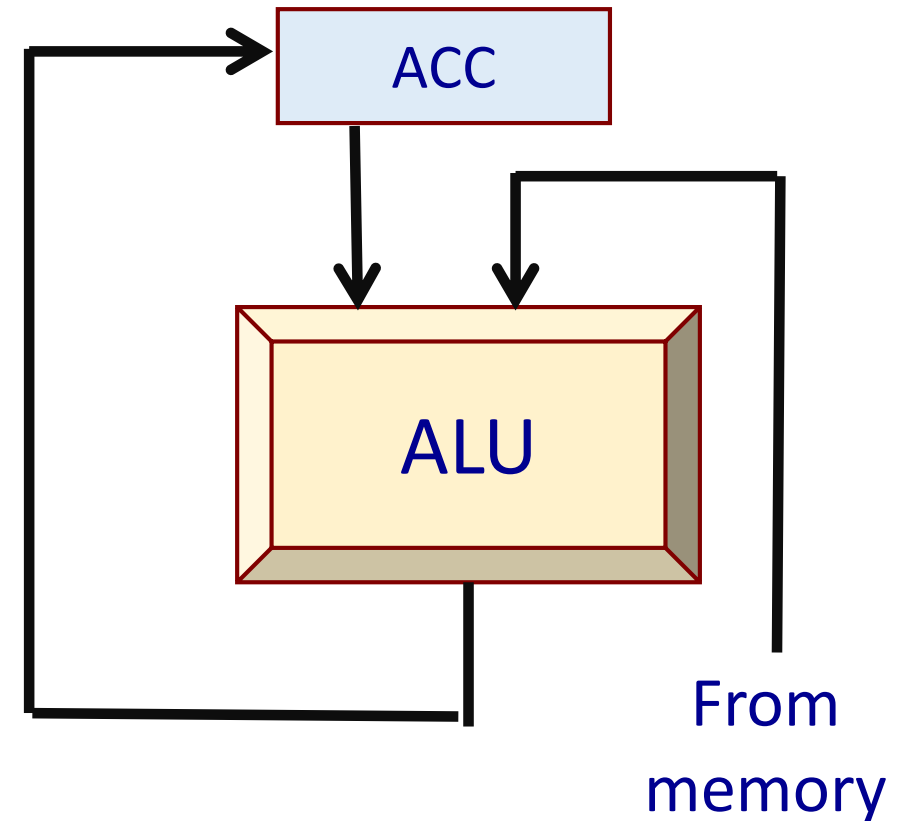
- Accumulator based machine:

LOAD X //  $ACC = Mem[X]$

ADD Y //  $ACC = ACC + Mem[Y]$

STORE Z //  $Mem[Z] = ACC$

- All instructions assume that one of the operands (and also the result) is in a special register called *accumulator*.



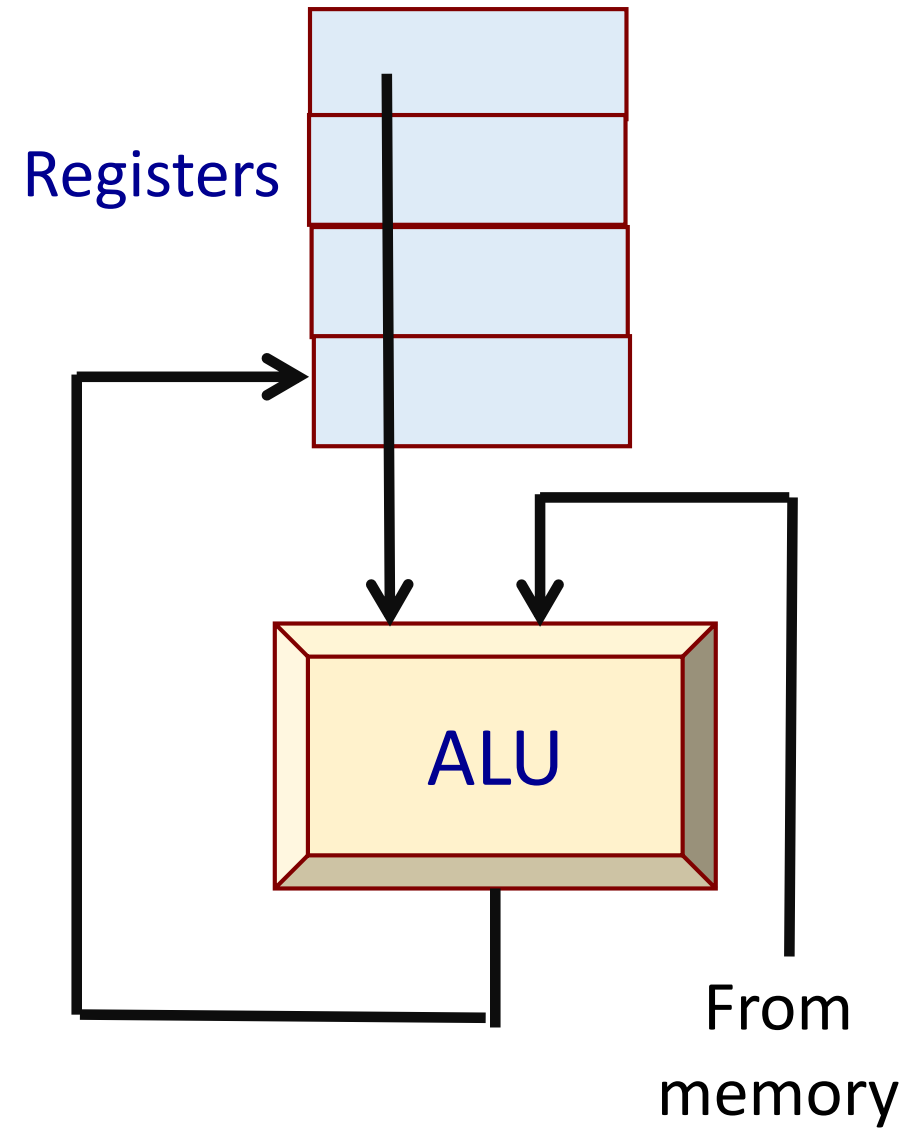
- Register-Memory machine:

**LOAD R2,X //  $R2 = \text{Mem}[X]$**

**ADD R2,Y //  $R2 = R2 + \text{Mem}[Y]$**

**STORE Z,R2 //  $\text{Mem}[Z] = R2$**

- One of the operands is assumed to be in register and another in memory.



- Register-Register machine:

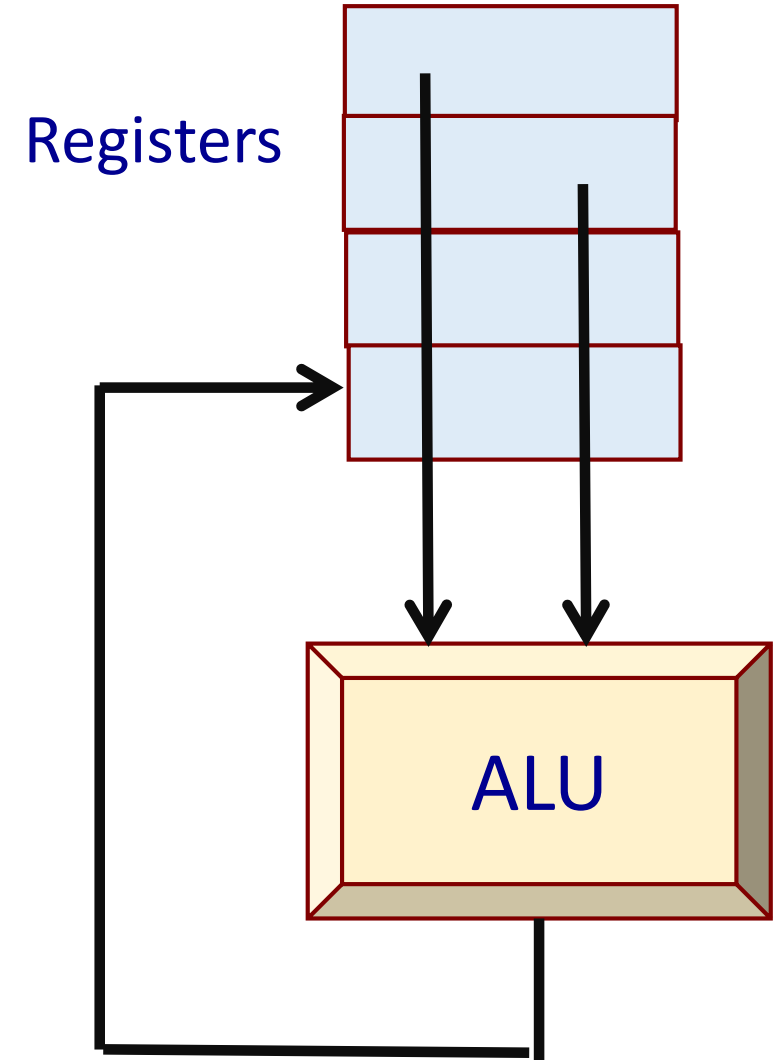
LOAD R1,X      // R1 = Mem[X]

LOAD R2,Y      // R2 = Mem[Y]

ADD R3,R1,R2   // R3 = R1 + R2

STORE Z,R3     // Mem[Z] = R3

- Also called *load-store architecture*, as only LOAD and STORE instructions can access memory.



# About General Purpose Registers (GPRs)

- Older architectures had a large number of special purpose registers.
  - Program counter, stack pointer, index register, flag register, accumulator, etc.
- Newer architectures, in contrast, have a large number of GPRs.
- Why?
  - Easy for the compiler to assign some variables to registers.
  - Registers are much faster than memory.
  - More compact instruction encoding as fewer bits are required to specify registers.
  - Many processors have 32 or more GPR's.

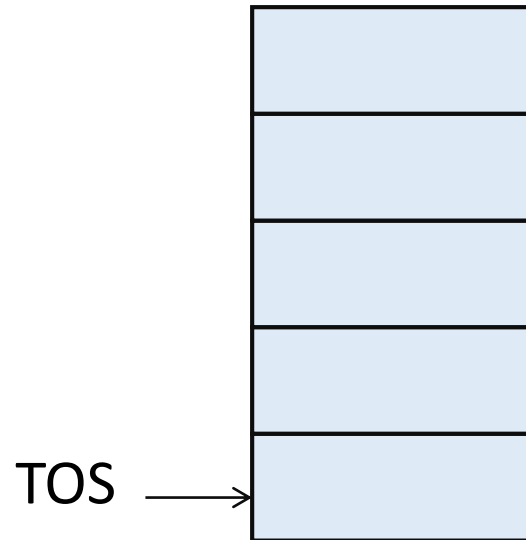
# **COMPARISON BETWEEN VARIOUS ARCHITECTURE TYPES**

# (a) Stack Architecture

- Typical instructions:

PUSH X, POP X

ADD, SUB, MUL, DIV



- **Example:**  $Y = A / B - (A - C * B)$

PUSH A

PUSH B

DIV

PUSH A

PUSH C

PUSH B

MUL

SUB

SUB

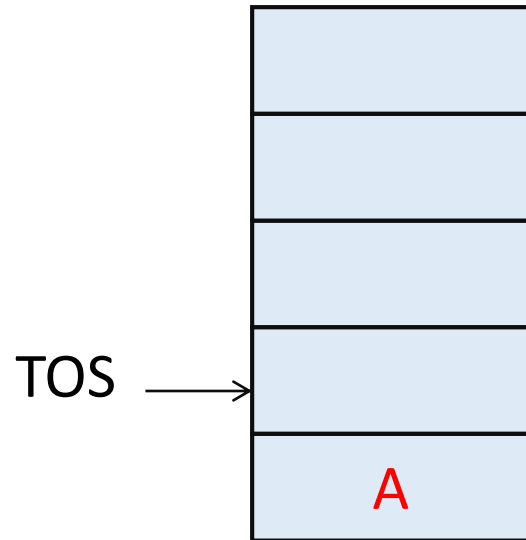
POP Y

# (a) Stack Architecture

- Typical instructions:

PUSH X, POP X

ADD, SUB, MUL, DIV



- **Example:**  $Y = A / B - (A - C * B)$

PUSH A

PUSH B

DIV

PUSH A

PUSH C

PUSH B

MUL

SUB

SUB

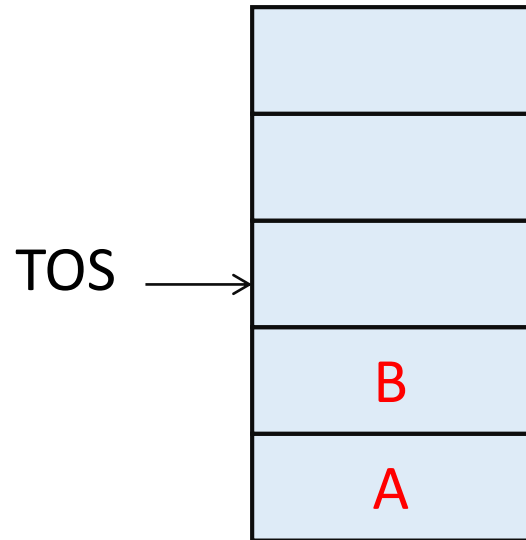
POP Y

# (a) Stack Architecture

- Typical instructions:

PUSH X, POP X

ADD, SUB, MUL, DIV



- **Example:**  $Y = A / B - (A - C * B)$

PUSH A

**PUSH B**

DIV

PUSH A

PUSH C

PUSH B

MUL

SUB

SUB

POP Y

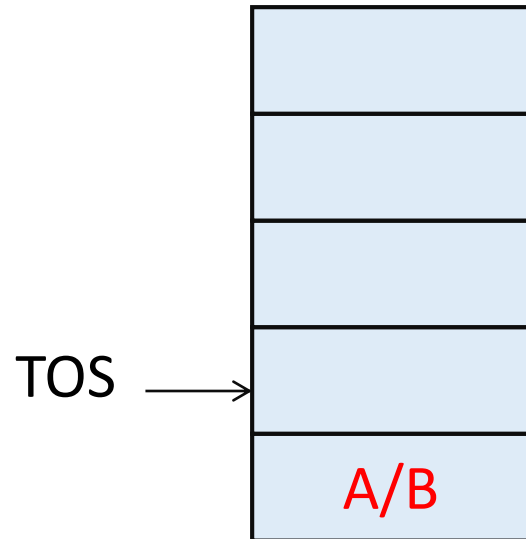


# (a) Stack Architecture

- Typical instructions:

PUSH X, POP X

ADD, SUB, MUL, DIV



- **Example:**  $Y = A / B - (A - C * B)$

PUSH A

PUSH B

**DIV**

PUSH A

PUSH C

PUSH B

MUL

SUB

SUB

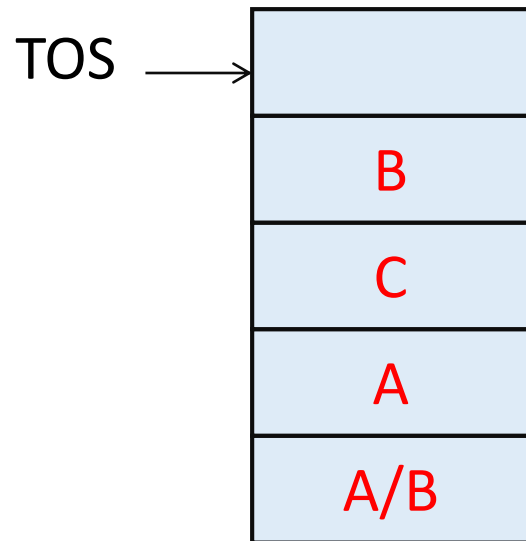
POP Y

# (a) Stack Architecture

- Typical instructions:

PUSH X, POP X

ADD, SUB, MUL, DIV



- **Example:**  $Y = A / B - (A - C * B)$

PUSH A

PUSH B

DIV

PUSH A

PUSH C

PUSH B

MUL

SUB

SUB

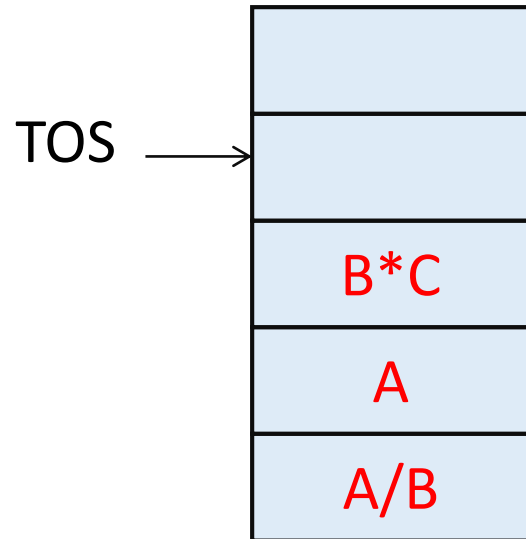
POP Y

# (a) Stack Architecture

- Typical instructions:

PUSH X, POP X

ADD, SUB, MUL, DIV



- **Example:**  $Y = A / B - (A - C * B)$

PUSH A

PUSH B

DIV

PUSH A

PUSH C

PUSH B

**MUL**

SUB

SUB

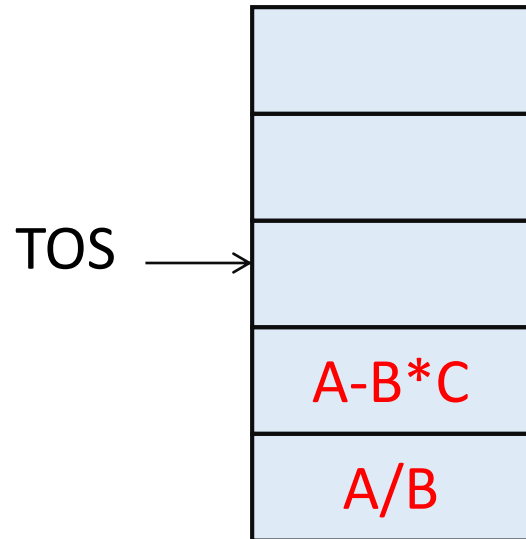
POP Y

# (a) Stack Architecture

- Typical instructions:

PUSH X, POP X

ADD, SUB, MUL, DIV



- **Example:**  $Y = A / B - (A - C * B)$

PUSH A

PUSH B

DIV

PUSH A

PUSH C

PUSH B

MUL

**SUB**

SUB

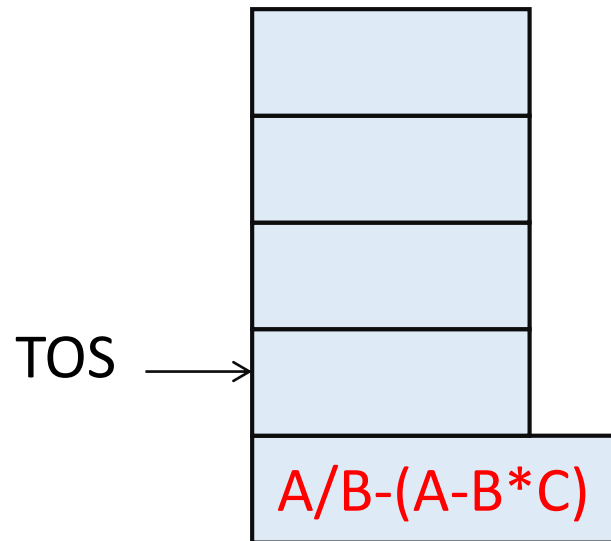
POP Y

# (a) Stack Architecture

- Typical instructions:

PUSH X, POP X

ADD, SUB, MUL, DIV



- **Example:**  $Y = A / B - (A - C * B)$

PUSH A

PUSH B

DIV

PUSH A

PUSH C

PUSH B

MUL

SUB

**SUB**

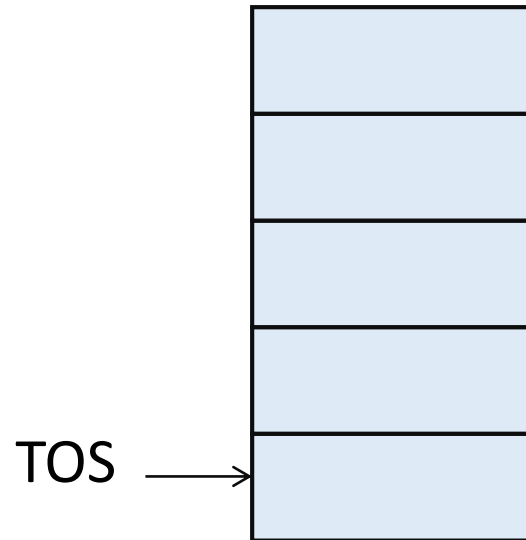
POP Y

# (a) Stack Architecture

- Typical instructions:

PUSH X, POP X

ADD, SUB, MUL, DIV



- **Example:**  $Y = A / B - (A - C * B)$

PUSH A

PUSH B

DIV

PUSH A

PUSH C

PUSH B

MUL

SUB

SUB

POP Y

Y = RESULT

## (b) Accumulator Architecture

- Typical instructions:

LOAD X, STORE X

ADD X, SUB X, MUL X, DIV X

**Example:**  $Y = A / B - (A - C * B)$

LOAD C

MUL B

STORE D //  $D = C * B$

LOAD A

SUB D

STORE D //  $D = A - C * B$

LOAD A

DIV B

SUB D

STORE Y

## (c) Memory-Memory Architecture

- Typical instructions (3 operands):

ADD X,Y,Z

SUB X,Y,Z

MUL X,Y,Z

- Typical instructions (2 operands):

MOV X,Y

ADD X,Y

SUB X,Y

MUL X,Y

**Example:**  $Y = A / B - (A - C * B)$

DIV D,A,B

MUL E,C,B

SUB E,A,E

SUB Y,D,E

MOV D,A

DIV D,B

MOV E,C

MUL E,B

SUB A,E

SUB D,A



## (d) Load-Store Architecture

- Typical instructions:

LOAD R1,X

STORE Y,R2

ADD R1,R2,R3

SUB R1,R2,R3

**Example:**  $Y = A / B - (A - C * B)$

LOAD R1,A

LOAD R2,B

LOAD R3,C

DIV R4,R1,R2

MUL R5,R3,R2

SUB R5,R1,R5

SUB R4,R4,R5

STORE Y,R4

# Registers: Pros and Cons

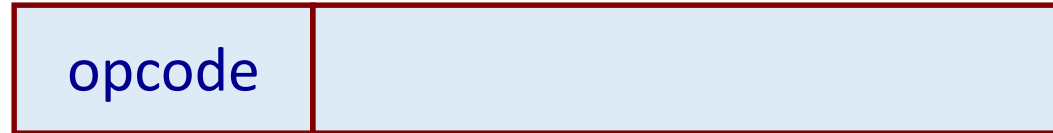
- The load-store architecture forms the basis of RISC ISA.
  - We shall explore one such RISC ISA, viz. MIPS.
- Helps in reducing memory traffic once the memory data are loaded into the registers.
- Compiler can generate very efficient code.
- Additional overhead for save/restore during procedure or interrupt calls and returns.
  - Many registers to save and restore.

# Instruction Format

- An instruction consists of two parts:-
  - *Operation Code or Opcode*
    - Specifies the operation to be performed by the instruction.
    - Various categories of instructions: data transfer, arithmetic and logical, control, I/O and special machine control.
  - *Operand(s)*
    - Specifies the source(s) and destination of the operation.
    - Source operand can be specified by an immediate data, by naming a register, or specifying the address of memory.
    - Destination can be specified by a register or memory address.

- Number of operands varies from instruction to instruction.
- Also for specifying an operand, various *addressing modes* are possible:
  - Immediate addressing
  - Direct addressing
  - Indirect addressing
  - Relative addressing
  - Indexed addressing, and many more.

# Instruction Format Examples



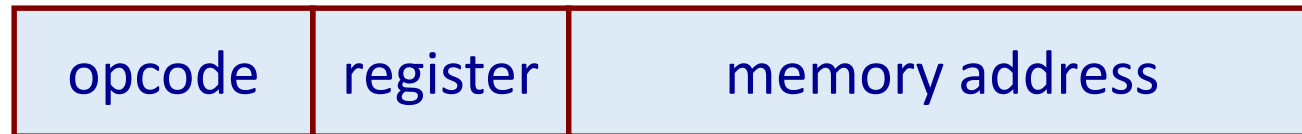
Implied addressing: NOP, HALT



1-address: ADD X, LOAD M



2-address: ADD X,Y



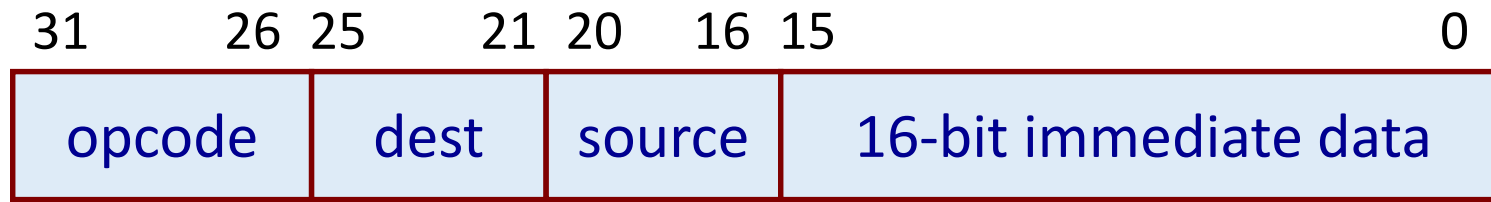
Register-memory: ADD R1,X



Register-register: ADD R1,R2,R3

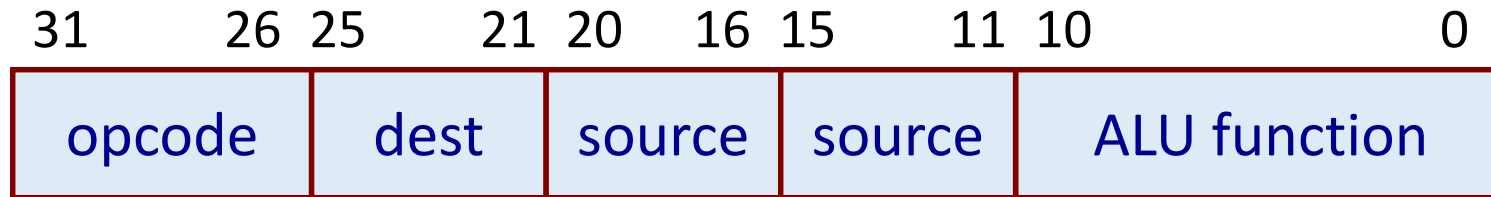
# A 32-bit Instruction Example

- Suppose we have an ISA with 32-bit instructions only.
  - Fixed size instructions make the decoding easier.
- Some instruction encoding examples are shown.
  - Assume that there are 32 registers R0 to R31, all of 32-bits.
  - 5-bits are required to specify a register.



LOAD R11,100(R2)  
 :: R11 = Mem(R2+100)

LOAD 01011 00010 0000000001100100



ADD R2,R5,R8  
 :: R2 = R5 + R8

ALU op 00010 00101 01000 ADD

# **ADDRESSING MODES**



# What are Addressing Modes?

- They specify the mechanism by which the operand data can be located.
- Some ISA's are quite complex and supports many addressing modes.
- ISA's based on load-store architecture are usually simple and support very limited number of addressing modes.
- Various addressing modes exist:
  - Immediate, Direct, Indirect, Register, Register Indirect, Indexed, Stack, Relative, Autoincrement, Autodecrement, Based, etc.
  - Not all processors support all addressing modes.
  - We shall briefly look at the common addressing modes and how they work.

# Immediate Addressing

- The operand is part of the instruction itself.
  - No memory reference is required to access the operand.
  - Fast but limited range (because a limited number of bits are provided to specify the immediate data).
- Examples:
  - ADD #25 // ACC = ACC + 25
  - ADDI R1,R2,42 // R1 = R2 + 42

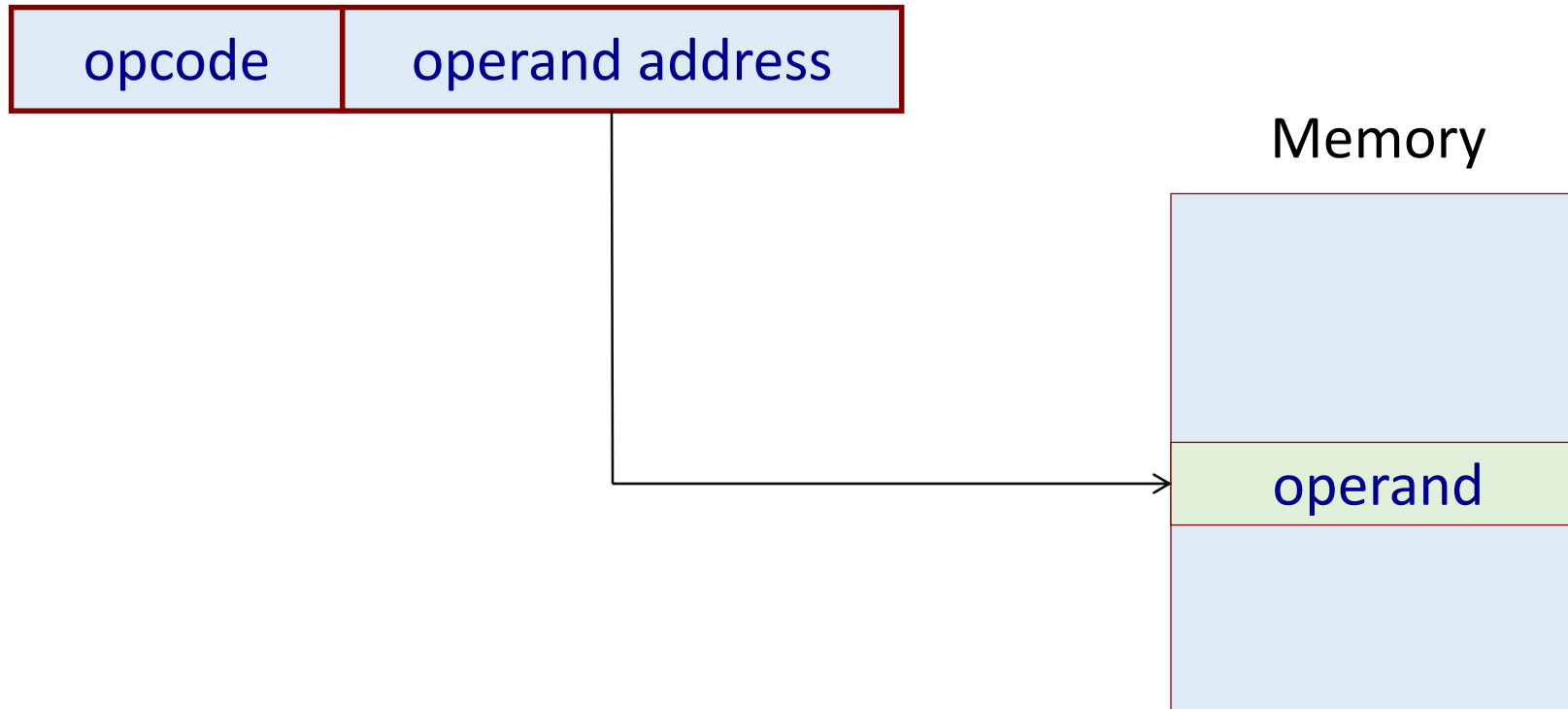


# Direct Addressing

- The instruction contains a field that holds the memory address of the operand.

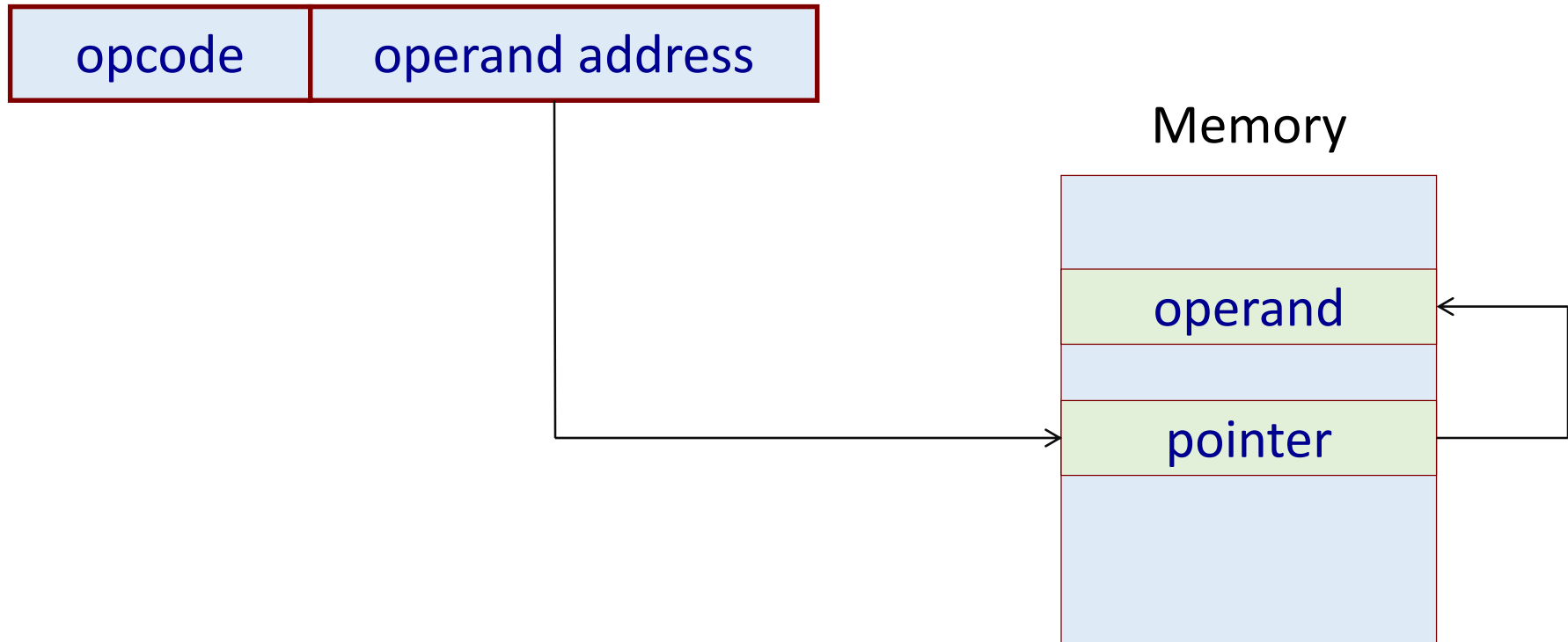


- Examples:
  - `ADD R1,20A6H` //  $R1 = R1 + \text{Mem}[20A6]$
- Single memory access is required to access the operand.
  - No additional calculations required to determine the operand address.
  - Limited address space (as number of bits is limited, say, 16 bits).



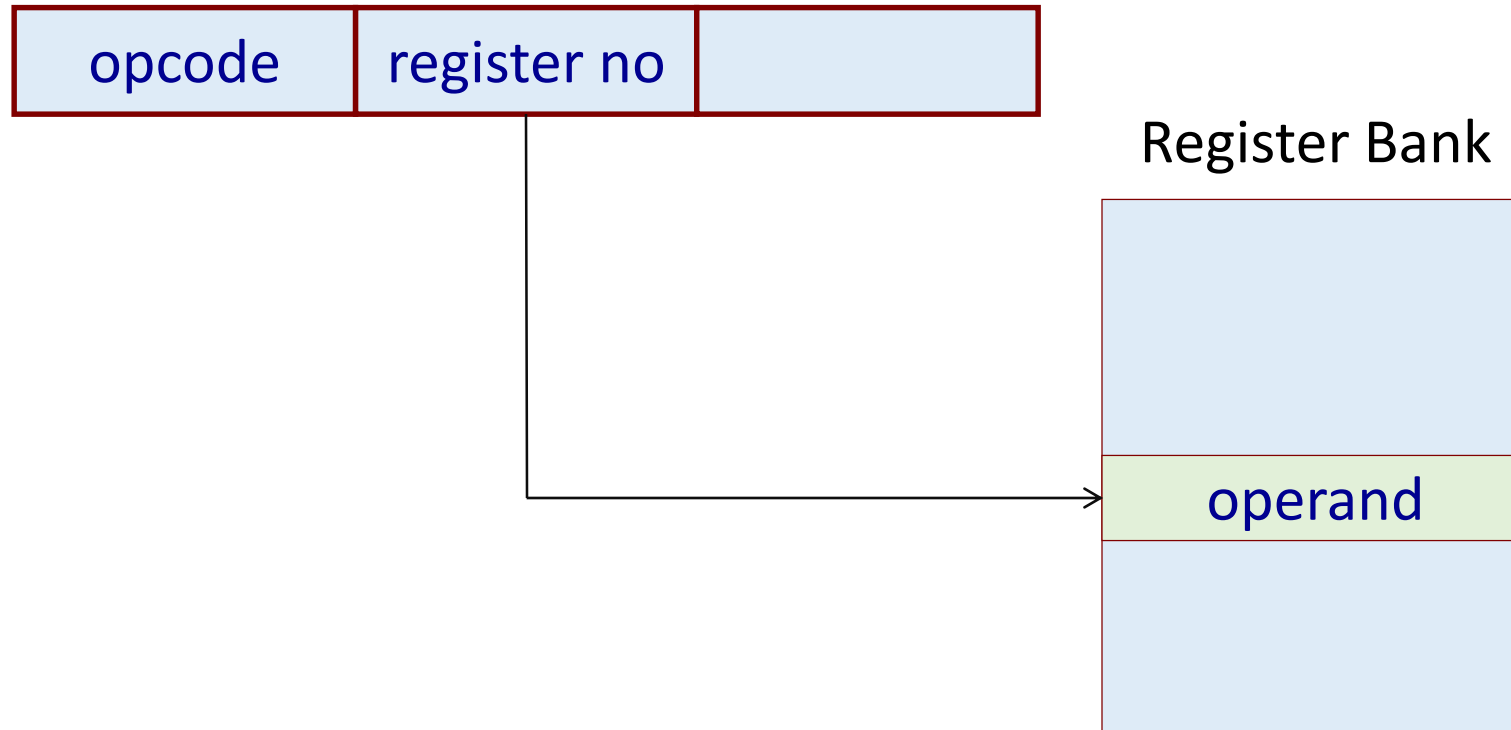
# Indirect Addressing

- The instruction contains a field that holds the memory address, which in turn holds the memory address of the operand.
- Two memory accesses are required to get the operand value.
- Slower but can access large address space.
  - Not limited by the number of bits in operand address like direct addressing.
- Examples:
  - `ADD R1,(20A6H)`      `// R1 = R1 + (Mem[20A6])`



# Register Addressing

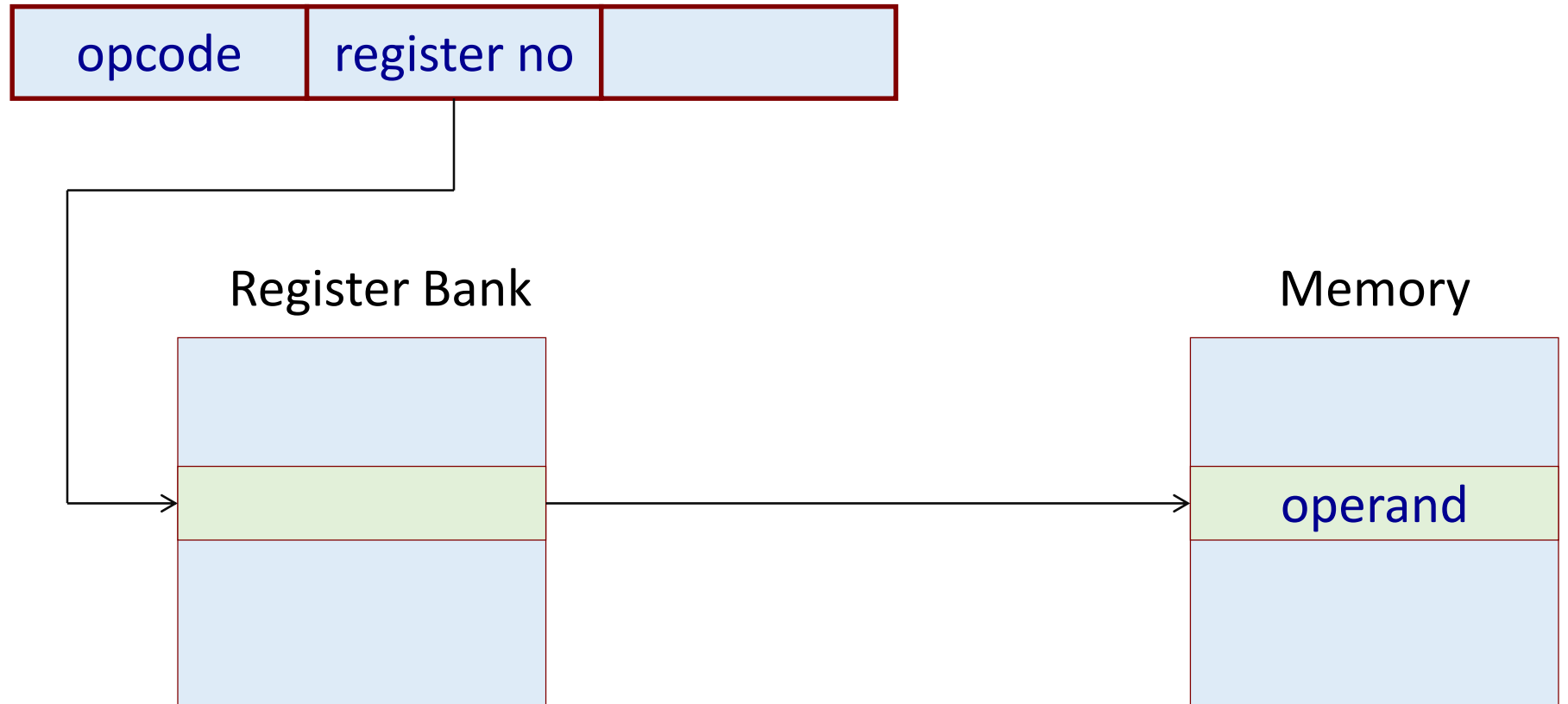
- The operand is held in a register, and the instruction specifies the register number.
  - Very few number of bits needed, as the number of registers is limited.
  - Faster execution, since no memory access is required for getting the operand.
- Modern load-store architectures support large number of registers.
- Examples:
  - `ADD R1,R2,R3` //  $R1 = R2 + R3$
  - `MOV R2,R5` //  $R2 = R5$





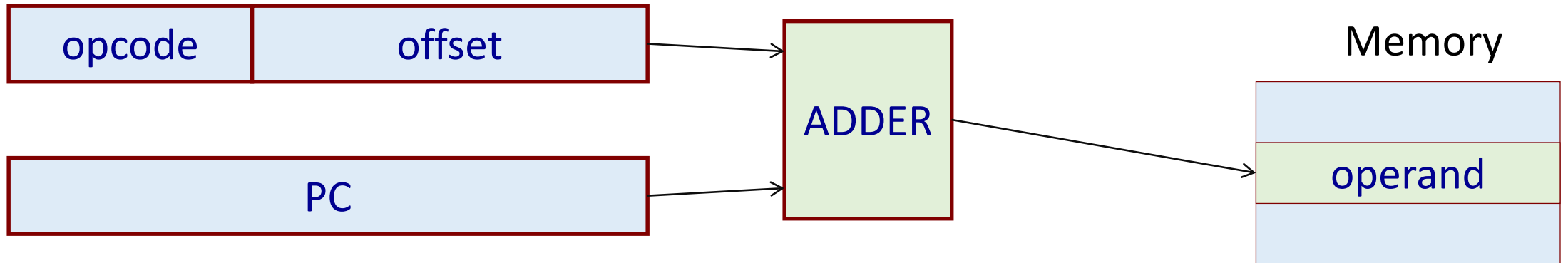
# Register Indirect Addressing

- The instruction specifies a register, and the register holds the memory address where the operand is stored.
  - Can access large address space.
  - One fewer memory access as compared to indirect addressing.
- Example:
  - `ADD R1,(R5)`    `// PC = R1 + Mem[R5]`



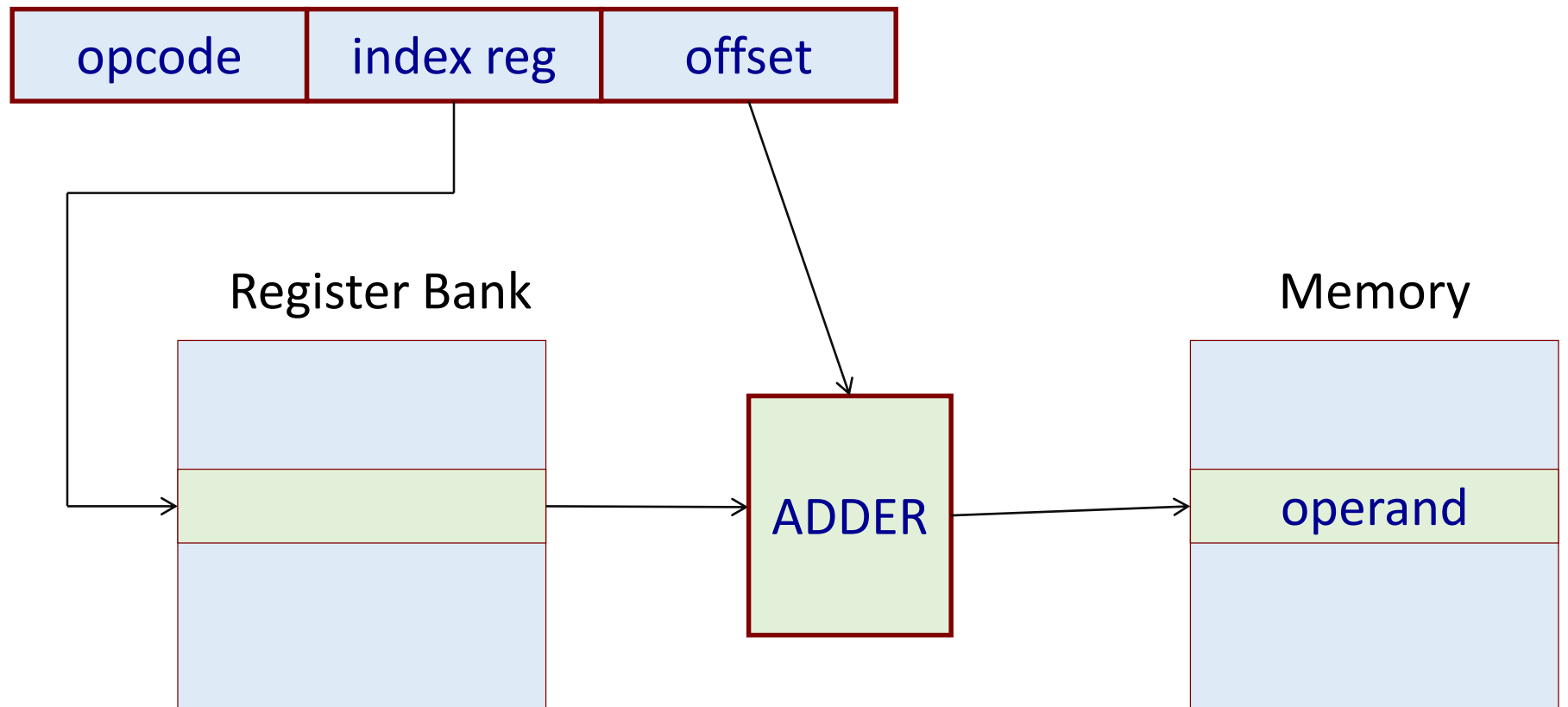
# Relative Addressing (PC Relative)

- The instruction specifies an offset of displacement, which is added to the program counter (PC) to get the effective address of the operand.
  - Since the number of bits to specify the offset is limited, the range of relative addressing is also limited.
  - If a 12-bit offset is specified, it can have values ranging from -2048 to +2047.



# Indexed Addressing

- Either a special-purpose register, or a general-purpose register, is used as *index register* in this addressing mode.
- The instruction specifies an offset of displacement, which is added to the index register to get the effective address of the operand.
- Example:
  - `LOAD R1,1050(R3)`     `// R1 = Mem[1050+R3]`
- Can be used to sequentially access the elements of an array.
  - Offset gives the starting address of the array, and the index register value specifies the array element to be used.



# Stack Addressing

- Operand is implicitly on top of the stack.
- Used in zero-address machines earlier.
- Examples:
  - ADD
  - PUSH X
  - POP X
- Many processors have a special register called the stack pointer (SP) that keeps track of the stack-top in memory.
  - PUSH, POP, CALL, RET instructions automatically modify SP.

# Some Other Addressing Modes

- **Base addressing**

- The processor has a special register called the *base register* or *segment register*.
- All operand addresses generated are added to the base register to get the final memory address.
- Allows easy movement of code and data in memory.

- **Autoincrement and Autodecrement**

- First introduced in the PDP-11 computer system.
- The register holding the operand address is automatically incremented or decremented after accessing the operand (like `a++` and `a--` in C).

# **RISC and CISC Architecture**



# Broad Classification

- Computer architectures have evolved over the years.
  - Features that were developed for mainframes and supercomputers in the 1960s and 1970s have started to appear on a regular basis on later generation microprocessors.
- Two broad classifications of ISA:
  - Complex Instruction Set Computer (CISC)
  - Reduced Instruction Set Computer (RISC)

# CISC versus RISC Architectures

- **Complex Instruction Set Computer (CISC)**

- More traditional approach.
- Main features:
  - Complex instruction set
  - Large number of addressing modes (R-R, R-M, M-M, indexed, indirect, etc.)
  - Special-purpose registers and Flags (sign, zero, carry, overflow, etc.)
  - Variable-length instructions / Complex instruction encoding
  - Ease of mapping high-level language statements to machine instructions
  - Instruction decoding / control unit design more complex
  - Pipeline implementation quite complex

- CISC Examples:

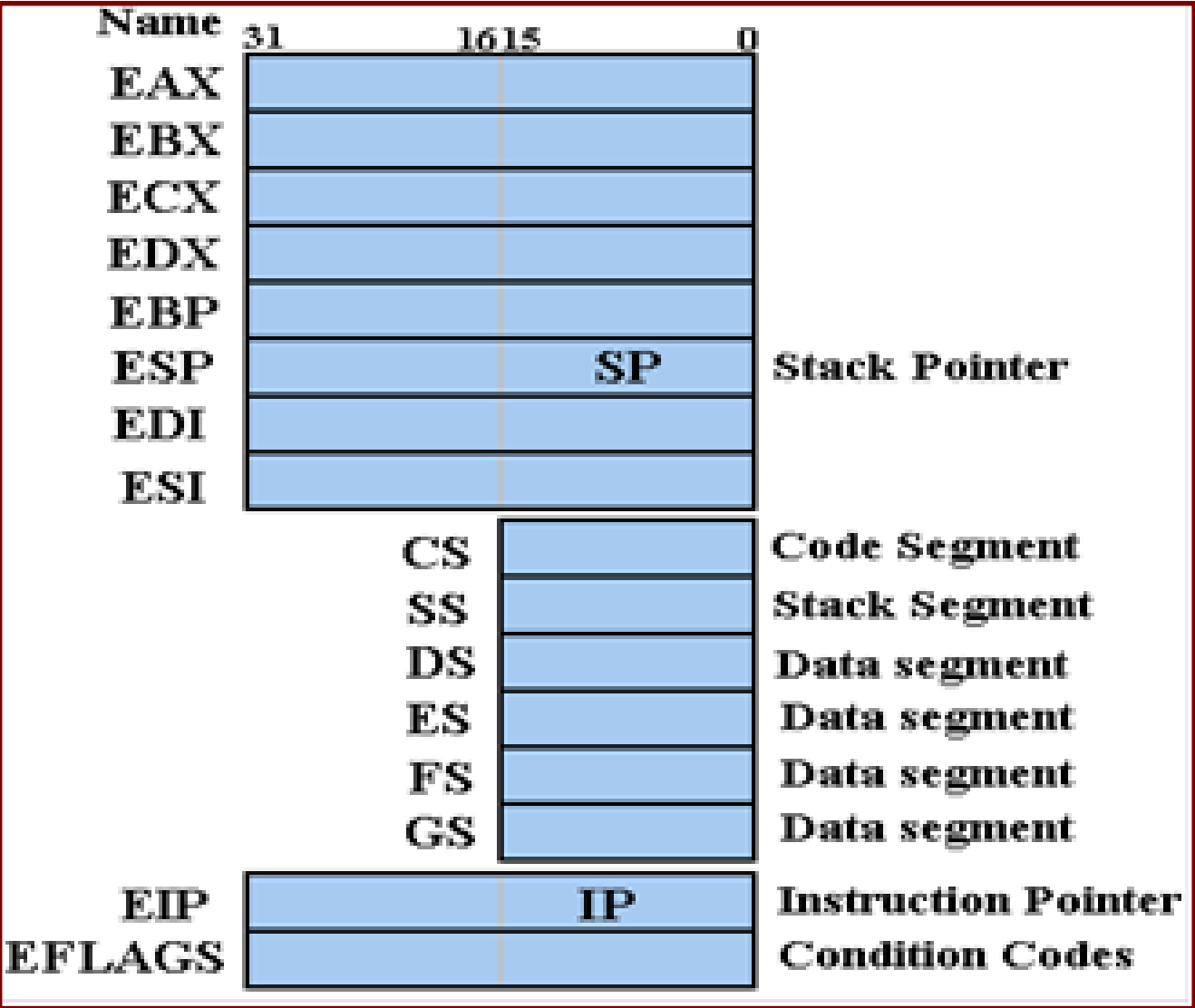
- IBM 360/370 (1960-70)
- VAX-11/780 (1970-80)
- Intel x86 / Pentium (1985-present)



Only CISC instruction set that survived over generations.

- Desktop PC's / Laptops use these.
- The volume of chips manufactured is so high that there is enough motivation to pay the extra design cost.
- Sufficient hardware resources available today to translate from CISC to RISC internally.

# Register Set in Pentium



# Addressing Modes in VAX

Addressing Mode	Example	Micro-operation
Register direct	ADD R1,R2	$R1 = R1 + R2$
Immediate	ADD R1,#15	$R1 = R1 + 15$
Displacement	ADD R1,220(R5)	$R1 = R1 + \text{Mem}[220+R5]$
Register indirect	ADD R1,(R3)	$R1 = R1 + \text{Mem}[R3]$
Indexed	ADD R1,(R2+R3)	$R1 = R1 + \text{Mem}[R2+R3]$
Direct	ADD R1, (1000)	$R1 = R1 + \text{Mem}[1000]$
Memory indirect	ADD R1,@(R4)	$R1 = R1 + \text{Mem}[\text{Mem}[R4]]$
Autoincrement	ADD R1,(R2)+	$R1 = R1 + \text{Mem}[R2]; R2++$
Autodecrement	ADD R1,(R2)-	$R1 = R1 + \text{Mem}[R2]; R2--$
Scaled	ADD R1,50(R2)[R3]	$R1 = R1 + \text{Mem}[50+R2+R3*d]$

- **Reduced Instruction Set Computer (RISC)**

- Very widely used among many manufacturers today.
- Also referred to as *Load-Store Architecture*.
  - Only LOAD and STORE instructions access memory.
  - All other instructions operate on processor registers.
- Main features:
  - Simple architecture for the sake of efficient pipelining.
  - Simple instruction set with very few addressing modes.
  - Large number of general-purpose registers; very few special-purpose.
  - Instruction length and encoding uniform for easy instruction decoding.
  - Compiler assisted scheduling of pipeline for improved performance.

- RISC Examples:
  - CDC 6600 (1964)
  - MIPS family (1980-90)
  - SPARC
  - ARM microcontroller family

- Almost all the computers today use a RISC based pipeline for efficient implementation.
  - RISC based computers use compilers to translate into RISC instructions.
  - CISC based computers (e.g. x86) use hardware to translate into RISC instructions.

# Results of a Comparative Study

- A quantitative comparison of VAX 8700 (a CISC machine) and MIPS M2000 (a RISC machine) with comparable organizations was carried out in 1991.
- Some findings:
  - MIPS required execution of about twice the number of instructions as compared to VAX.
  - Cycles Per Instructions (CPI) for VAX was about six times larger than that of MIPS.
  - Hence, MIPS had three times the performance of VAX.
  - Also, much less hardware is required to build MIPS as compared to VAX.



- **Conclusion:**

- Persisting with CISC architecture is too costly, both in terms of hardware cost and also performance.
- VAX was replaced by ALPHA (a RISC processor) by Digital Equipment Corporation (DEC).
- CISC architecture based on x86 is different.
  - Because of huge number of installed base, backward compatibility of machine code is very important from commercial point of view.
  - They have adopted a balanced view: (a) user's view is a CISC instruction set, (b) hardware translates every CISC instruction into an equivalent set of RISC instructions internally, (c) an instruction pipeline executes the RISC instructions efficiently.

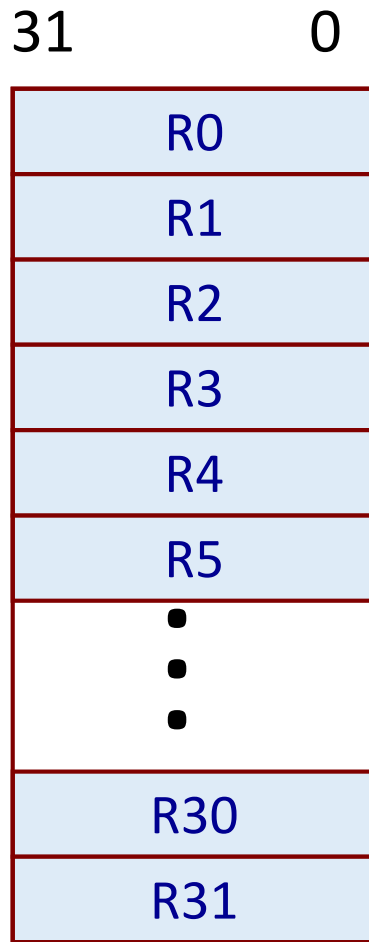
# MIPS32 Architecture: A Case Study

- As a case study of RISC ISA, we shall be considering the MIPS32 architecture.
  - Look into the instruction set and instruction encoding in detail.
  - Design the data path of the MIPS32 architecture, and also look into the control unit design issues.
  - Extend the basic data path of MIPS32 to a pipeline architecture, and discuss some of the issues therein.

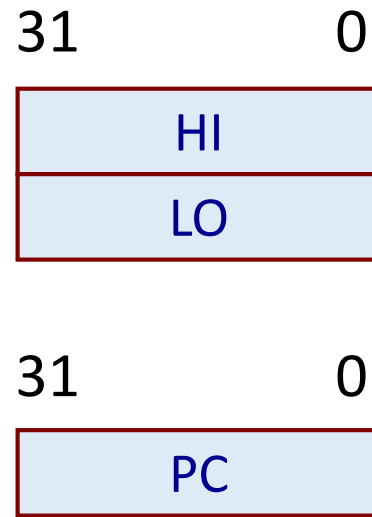
# MIPS32 CPU Registers

- The MIPS32 ISA defines the following CPU registers that are visible to the machine/assembly language programmer.
  - a) 32, 32-bit general purpose registers (GPRs), R0 to R31.
  - b) A special-purpose 32-bit program counter (PC).
    - Points to the next instruction in memory to be fetched and executed.
    - Not directly visible to the programmer.
    - Affected only indirectly by certain instructions (like branch, call, etc.)
  - c) A pair of 32-bit special-purpose registers HI and LO, which are used to hold the results of multiply, divide, and multiply-accumulate instructions.

- Some common registers are missing in MIPS32.
  - ***Stack Pointer*** (SP) register, which helps in maintaining a stack in main memory.
    - Any of the GPRs can be used as the stack pointer.
    - No separate PUSH, POP, CALL and RET instructions.
  - ***Index Register*** (IX), which helps in accessing memory words sequentially in memory.
    - Any of the GPRs can be used as an index register.
  - ***Flag registers*** (like ZERO, SIGN, CARRY, OVERFLOW) that keeps track of the results of arithmetic and logical operations.
    - Maintains flags in registers, to avoid problems in pipeline implementation.



**General Purpose Registers**



**Special Purpose Registers**

Two of the GPRs have assigned functions:

- a) R0 is hard-wired to a value of zero.
- Can be used as the target register for any instruction whose result is to be discarded.
  - Can also be used as a source when a zero value is needed.
- b) R31 is used to store the return address when a function call is made.
- Used by the jump-and-link and branch-and-link instructions like JAL, BLTZAL, BGEZAL, etc.
  - Can also be used as a normal register.

# Some Examples

```
LD    R4, 50(R3)    // R4 = Mem[50+R3]
ADD   R2, R1, R4     // R2 = R1 + R4
SD    54(R3), R2     // Mem[54+R3] = R2
```

```
ADD   R2, R5, R0     // R2 = R5
```

```
MAIN: ADDI   R1, R0, 35  // R1 = 35
      ADDI   R2, R0, 56  // R2 = 56
      JAL    GCD
      ....

GCD:   ....             // Find GCD of R1 & R2
      JR     R31
```

# How are the HI and LO registers used?

- During a multiply operation, the HI and LO registers store the product of an integer multiply.
  - HI denotes the high-order 32 bits, and LO denotes the low-order 32 bits.
- During a multiply-add or multiply-subtract operation, the HI and LO registers store the result of the integer multiply-add or multiply-subtract.
- During a division, the HI and LO registers store the quotient (in LO) and remainder (in HI) of integer divide.

# Some MIPS32 Assembly Language Conventions

- The integer registers of MIPS32 can be accessed as **R0..R31** or **r0..r31** in an assembly language program.
- Several assemblers and simulators are available in the public domain (like QtSPIM) that follow some specific conventions.
  - These conventions have become like a *de facto* standard when we write assembly language programs for MIPS32.
  - Basically some alternate names are used for the registers to indicate their intended usage.



Register name	Register number	Usage
\$zero	R0	Constant zero

Used to represent the constant zero value, wherever required in a program.

Register name	Register number	Usage
\$at	R1	Reserved for assembler

May be used as temporary register during macro expansion by assembler.

- Assembler provides an extension to the MIPS32 instruction set that are converted to standard MIPS32 instructions.

Example: Load Address instruction used to initialize pointers

```
la    R5, addr
```



```
lui   $at, Upper-16-bits-of-addr
```

```
ori   R5, $at, Lower-16-bits-of-addr
```

Register name	Register number	Usage
\$v0	R2	Result of function, or for expression evaluation
\$v1	R3	Result of function, or for expression evaluation

May be used for up to two function return values, and also as temporary registers during expression evaluation.

Register name	Register number	Usage
\$a0	R4	Argument 1
\$a1	R5	Argument 2
\$a2	R6	Argument 3
\$a3	R7	Argument 3

May be used to pass up to four arguments to functions.

Register name	Register number	Usage
\$t0	R8	Temporary (not preserved across call)
\$t1	R9	Temporary (not preserved across call)
\$t2	R10	Temporary (not preserved across call)
\$t3	R11	Temporary (not preserved across call)
\$t4	R12	May be used as temporary variables in programs. These registers might get modified when some functions are called (other than user-written functions).
\$t5	R13	
\$t6	R14	
\$t7	R15	
\$t8	R24	Temporary (not preserved across call)
\$t9	R25	Temporary (not preserved across call)

Register name	Register number	Usage
\$s0	R16	Temporary (preserved across call)
\$s1	R17	Temporary (preserved across call)
\$s2	R18	Temporary (preserved across call)
\$s3	R19	Temporary (preserved across call)
\$s4	R20	Temporary (preserved across call)
\$s5	R21	Temporary (preserved across call)
\$s6	R22	May be used as temporary variables in programs. These registers do not get modified across function calls.
\$s7	R23	

Register name	Register number	Usage
\$gp	R28	Pointer to global area
\$sp	R29	Stack pointer
\$fp	R30	Frame pointer
\$ra	R31	Return address (used by function call)

These registers are used for a variety of pointers:

- Global area: points to the memory address from where the global variables are allocated space.
- Stack pointer: points to the top of the stack in memory.
- Frame pointer: points to the activation record in stack.
- Return address: used while returning from a function.

Register name	Register number	Usage
\$k0	R26	Reserved for OS kernel
\$k1	R27	Reserved for OS kernel

These registers are supposed to be used by the OS kernel in a real computer system.

It is highly recommended not to use these registers.