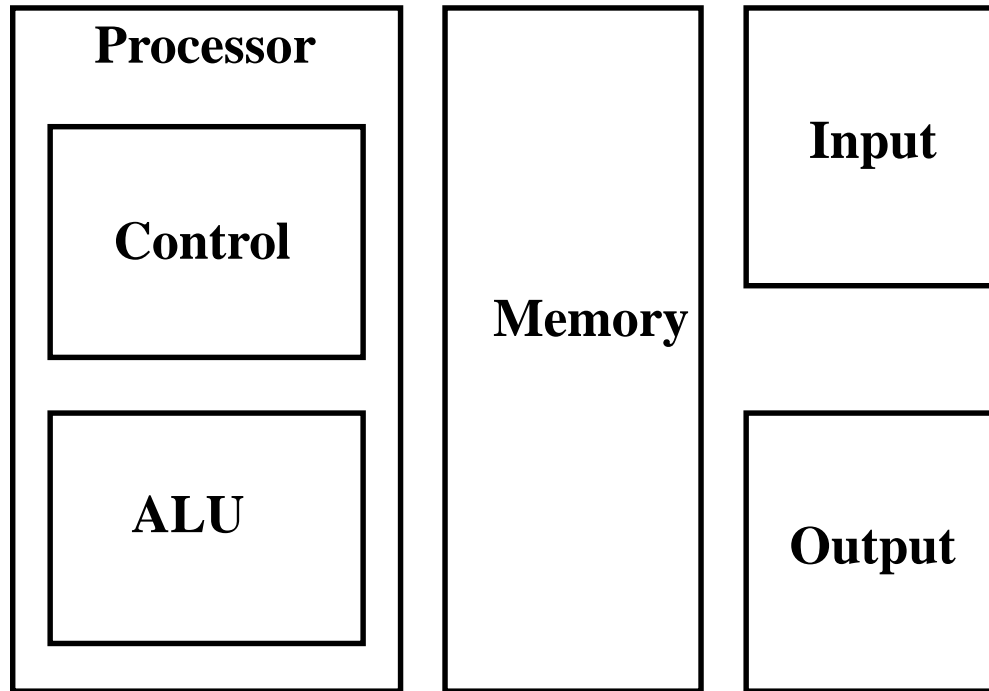


# **Functional Units of a Computer**

# FUNCTIONAL UNITS OF COMPUTER

- Input Unit
- Output Unit
- Central processing Unit (ALU and Control Units)
- Memory
- Bus Structure

# The Big Picture



Since 1946 all computers have had 5 components!!!

# Function

IMPORTANT  
SLIDE !

- ALL computer functions are:

- Data PROCESSING

- Data STORAGE

- Data MOVEMENT

- CONTROL

Data = Information

Coordinates How  
Information is Used

- NOTHING ELSE!

## **INPUT UNIT:**

- Converts the external world data to a binary format, which can be understood by CPU
- Eg: Keyboard, Mouse, Joystick etc

## **OUTPUT UNIT:**

- Converts the binary format data to a format that a common man can understand
- Eg: Monitor, Printer, LCD, LED etc

# CPU

- The “brain” of the machine
- Responsible for carrying out computational task
- Contains ALU, CU, Registers
- ALU Performs Arithmetic and logical operations
- CU Provides control signals in accordance with some timings which in turn controls the execution process
- Register Stores data and result and speeds up the operation

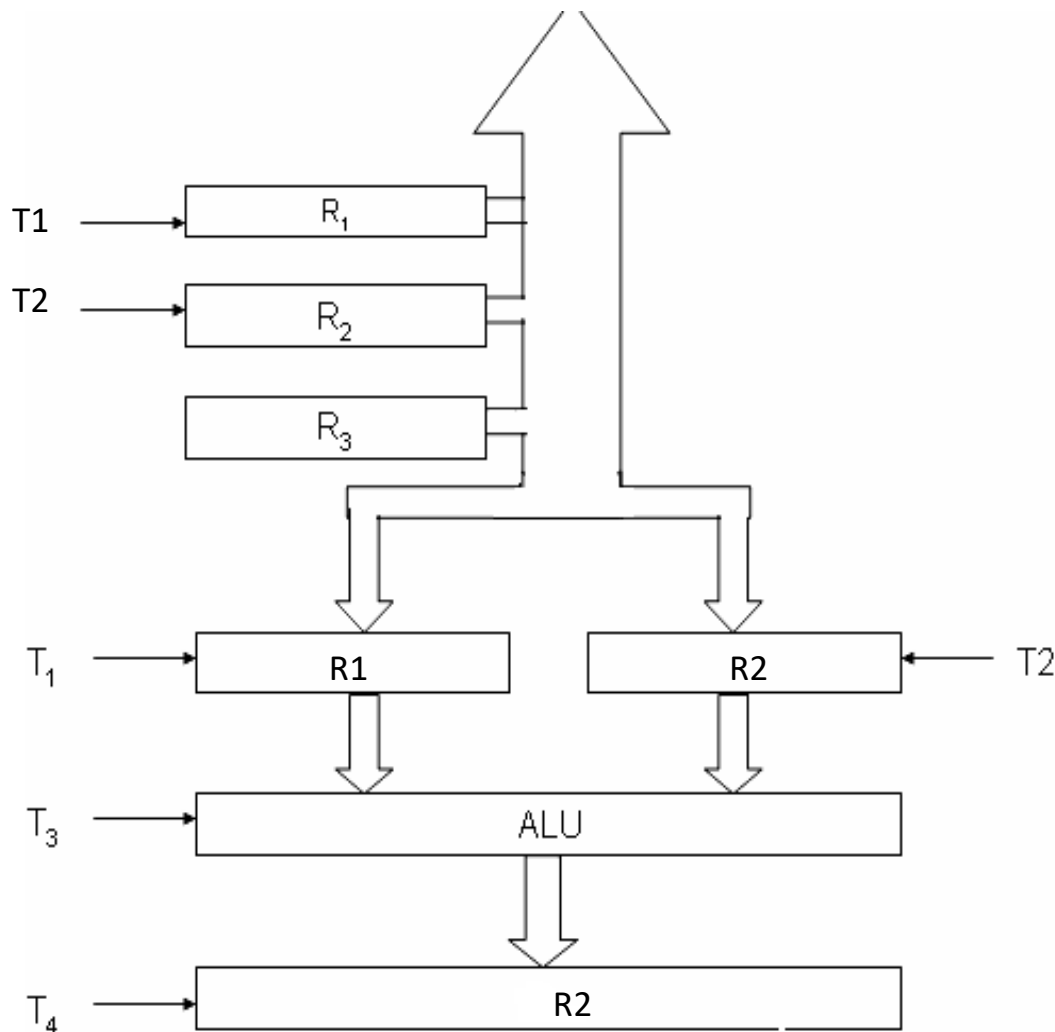
Example  
Add R1, R2

T1      → Enable R1

T2      →    Enable R2    ◀

T3      → Enable ALU for addition operation

T4      → Enable out put of ALU to store result of the  
          operation



- Control unit works with a reference signal called processor clock

- Processor divides the operations into basic steps

- Each basic step is executed in one clock cycle



# MEMORY

- Stores data, results, programs
- Two class of storage  
(i) Primary (ii) Secondary
- Two types are RAM or R/W memory and ROM read only memory
- ROM is used to store data and program which is not going to change.
- Secondary storage is used for bulk storage or mass storage

# Basic Operational Concepts

## Basic Function of Computer

- To Execute a given task as per the appropriate program
- Program consists of list of instructions stored in memory

# Review

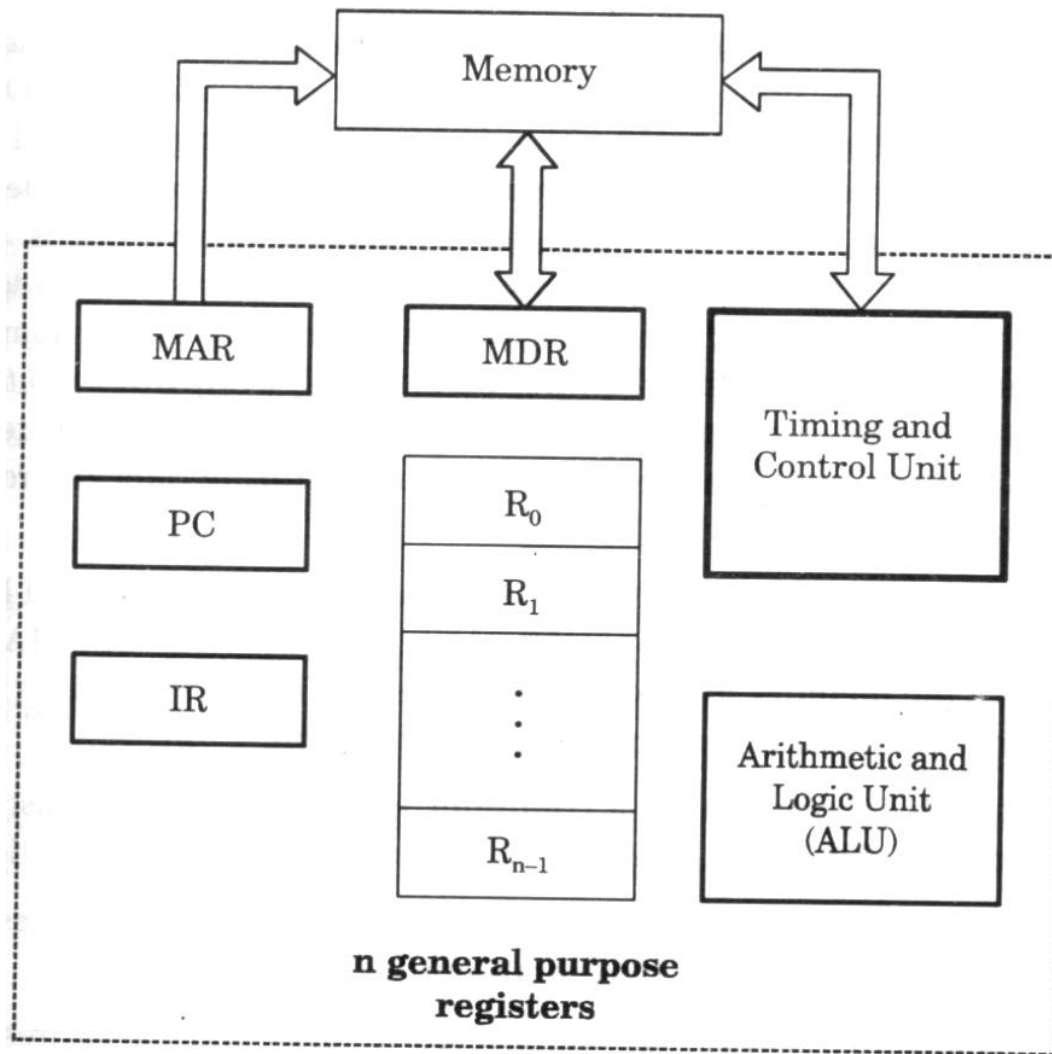
- Activity in a computer is governed by instructions.
- To perform a task, an appropriate program consisting of a list of instructions is stored in the memory.
- Individual instructions are brought from the memory into the processor, which executes the specified operations.
- Data to be used as operands are also stored in the memory.

# A Typical Instruction

- **Add LOCA, R0**
- Add the operand at memory location LOCA to the operand in a register R0 in the processor.
- Place the sum into register R0.
- The original contents of LOCA are preserved.
- The original contents of R0 is overwritten.
- Instruction is fetched from the memory into the processor – the operand at LOCA is fetched and added to the contents of R0 – the resulting sum is stored in register R0.

# Separate Memory Access and ALU Operation

- Load LOCA, R1
- Add R1, R0
- Whose contents will be overwritten?



Interconnection between Processor and Memory

# Registers




Registers are fast stand-alone storage locations that hold data temporarily. Multiple registers are needed to facilitate the operation of the CPU. Some of these registers are

- ❑ **Two registers-MAR (Memory Address Register) and MDR (Memory Data Register) :** To handle the data transfer between main memory and processor. MAR-Holds addresses, MDR-Holds data
- ❑ **Instruction register (IR) :** Hold the Instructions that is currently being executed
- ❑ **Program counter:** Points to the next instructions that is to be fetched from memory

- (PC) → (MAR)( the contents of PC transferred to MAR)
- (MAR) → (Address bus) Select a particular memory location
- Issues RD control signals
- Reads instruction present in memory and loaded into MDR
- Will be placed in IR (Contents transferred from MDR to IR)



- Instruction present in IR will be decoded by which processor understand what operation it has to perform
- Increments the contents of PC by 1, so that it points to the next instruction address
- If data required for operation is available in register, it performs the operation
- If data is present in memory following sequence is performed

- Address of the data            MAR
- MAR       select memory location where is issued RD signal
- Reads data via data bus            MDR
- From MDR data can be directly routed to ALU or it can be placed in register and then operation can be performed
- Results of the operation can be directed towards output device, memory or register
- Normal execution preempted (interrupt)

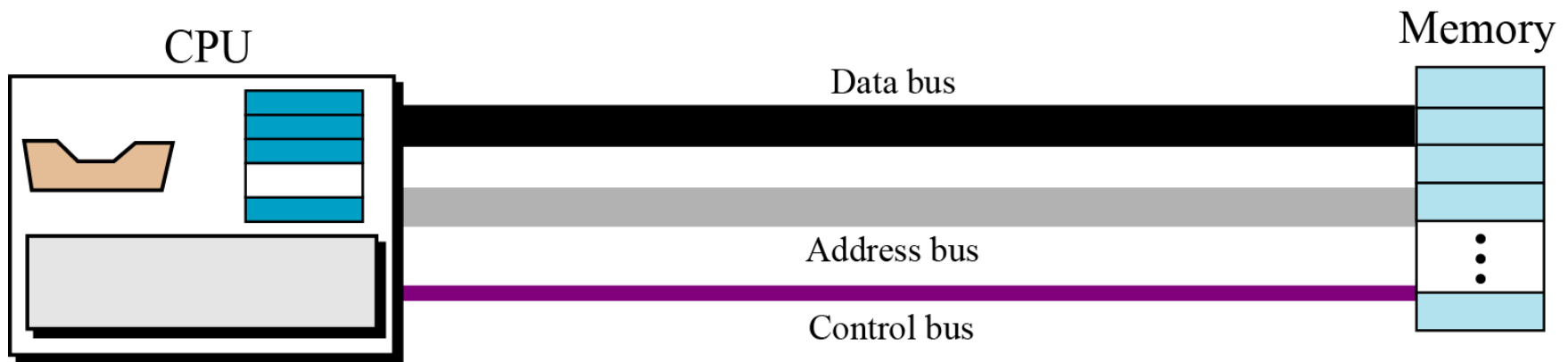
# Interrupt

- An interrupt is a request from I/O device for service by processor
- Processor provides requested service by executing interrupt service routine (ISR)
- Contents of PC, general registers, and some control information are stored in memory .
- When ISR completed, processor restored, so that interrupted program may continue

# BUS STRUCTURE

## Connecting CPU and memory

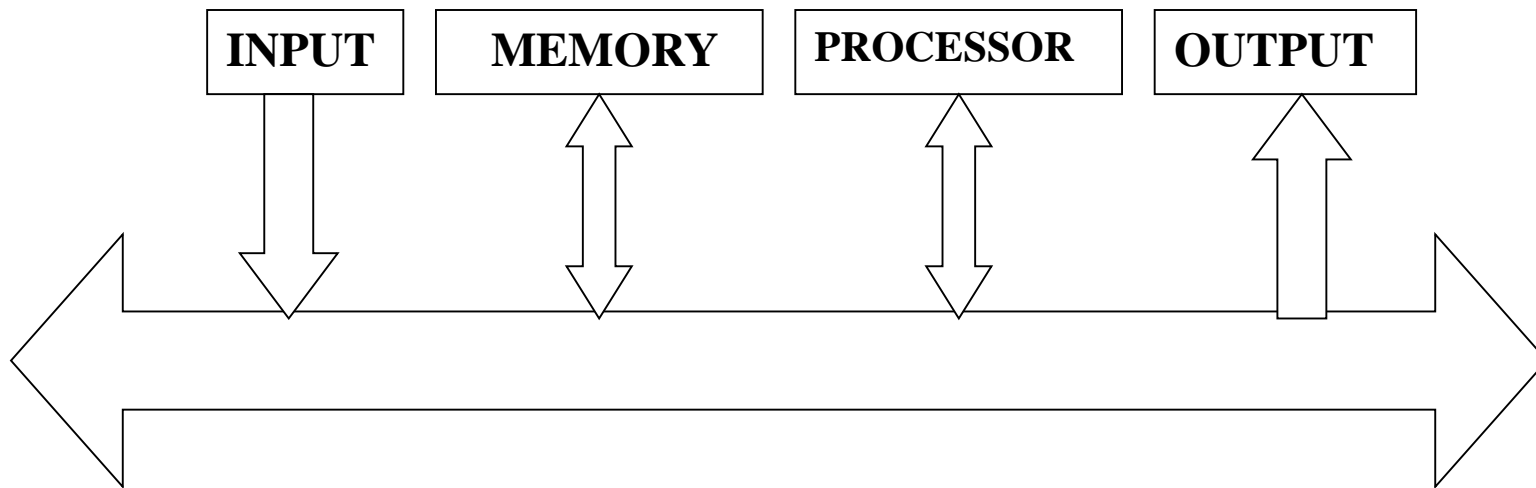
The CPU and memory are normally connected by three groups of connections, each called a **bus**: *data bus*, *address bus* and *control bus*



Connecting CPU and memory using three buses

## BUS STRUCTURE

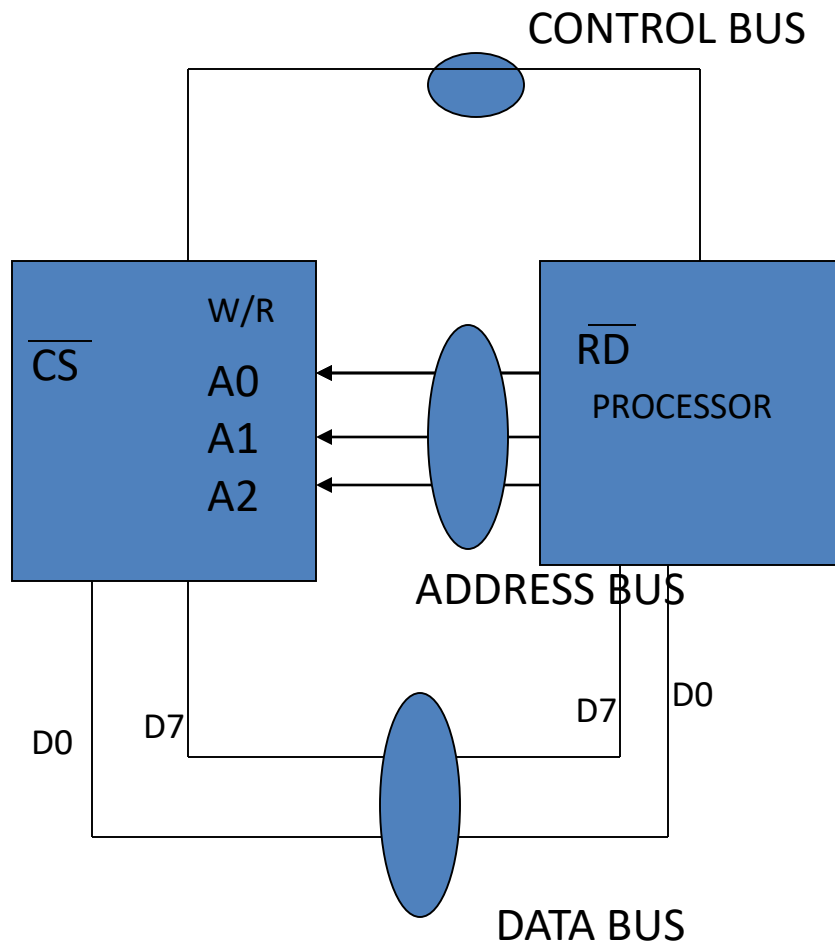
- Group of wires which carries information from CPU to peripherals or vice versa – versa
- **Single bus structure:** Common bus used to communicate between peripherals and microprocessor



**SINGLE BUS STRUCTURE**

Continued:-

- To improve performance **multibus** structure can be used
- In two – bus structure : One bus can be used to fetch instruction other can be used to fetch data, required for execution.
- Thus improving the performance ,but cost increases



$A_2$	$A_1$	$A_0$	Selected location
0	0	0	0 <sup>th</sup> Location
0	0	1	1 <sup>st</sup> Location
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

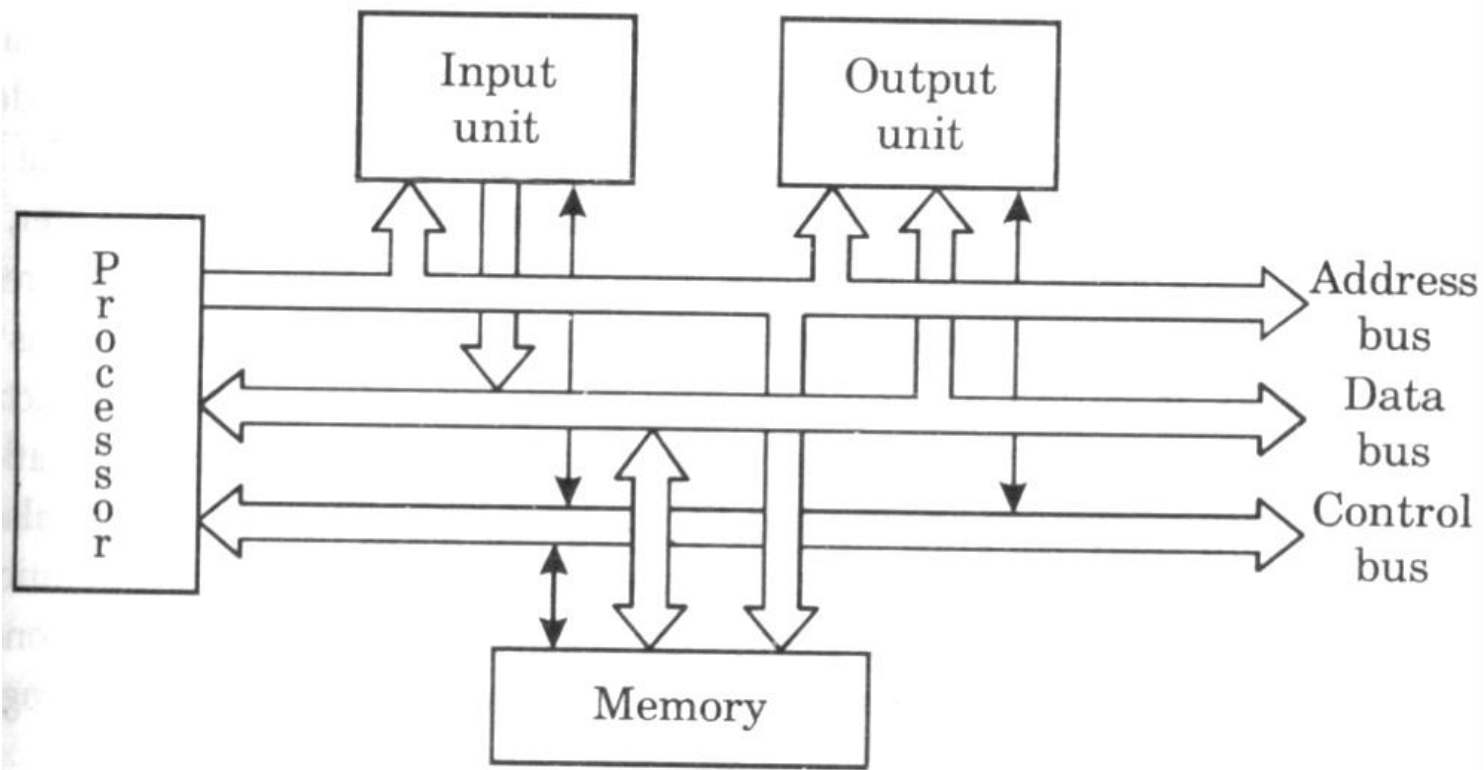
## Cont:-

- $2^3 = 8$  i.e. 3 address line is required to select 8 location
- In general  $2^x = n$  where x number of address lines (address bit) and n is number of location
- **Address bus** : unidirectional : group of wires which carries address information bits from processor to peripherals (16,20,24 or more parallel signal lines)



## Cont:-

- **Databus:** bidirectional : group of wires which carries data information bit form processor to peripherals and vice – versa
- **Controlbus:** bidirectional: group of wires which carries control signals form processor to peripherals and vice – versa
- Figure below shows address, data and control bus and their connection with peripheral and microprocessor



Single bus structure showing the details of connection

# Memory Locations, Addresses, and Operations

# Memory Location, Addresses, and Operation

- Memory consists of many millions of storage cells, each of which can store 1 bit.
- Data is usually accessed in  $n$ -bit groups.  $n$  is called word length.

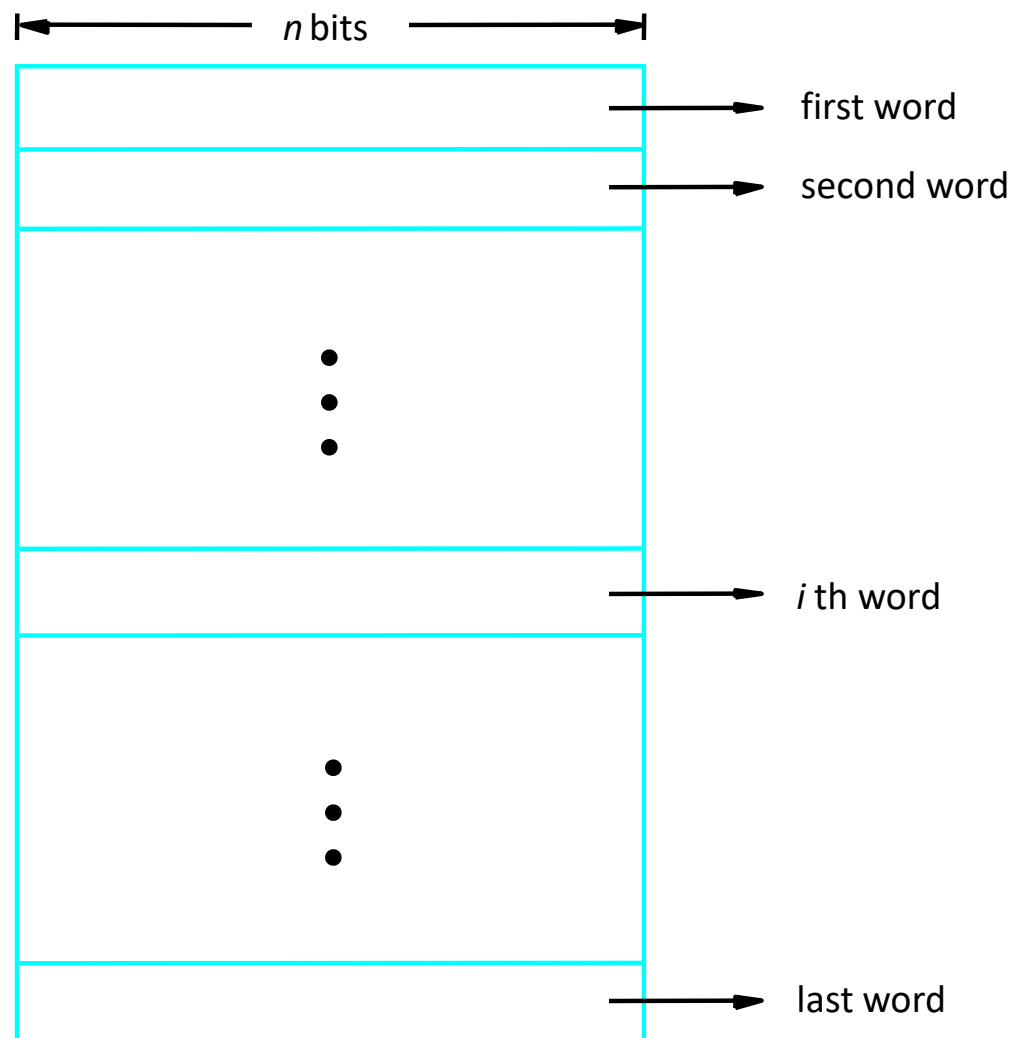
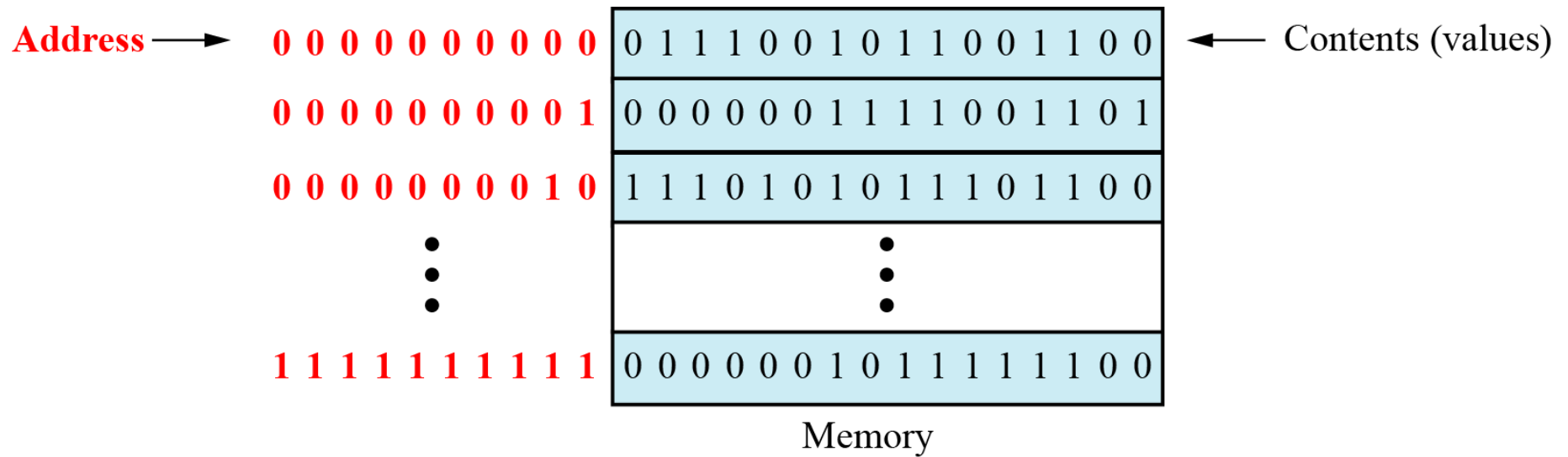


Figure 2.5. Memory words.

# MEMORY LOCATIONS AND ADDRESSES

- **Main memory** is the second major subsystem in a computer. It consists of a collection of storage locations, each with a unique identifier, called an **address**.
- Data is transferred to and from memory in groups of bits called **words**. A word can be a group of 8 bits, 16 bits, 32 bits or 64 bits (and growing).
- If the word is 8 bits, it is referred to as a **byte**. The term “byte” is so common in computer science that sometimes a 16-bit word is referred to as a 2-byte word, or a 32-bit word is referred to as a 4-byte word.



**Figure 5.3** Main memory

# Address space

- To access a word in memory requires an identifier. Although programmers use a name to identify a word (or a collection of words), at the hardware level each word is identified by an address.
- The total number of uniquely identifiable locations in memory is called the **address space**.
- For example, a memory with 64 kilobytes (16 address lines required) and a word size of 1 byte has an address space that ranges from 0 to 65,535.

**Table 5.1**    Memory units

<i>Unit</i>	<i>Exact Number of Bytes</i>	<i>Approximation</i>
kilobyte	$2^{10}$ (1024) bytes	$10^3$ bytes
megabyte	$2^{20}$ (1,048,576) bytes	$10^6$ bytes
gigabyte	$2^{30}$ (1,073,741,824) bytes	$10^9$ bytes
terabyte	$2^{40}$ bytes	$10^{12}$ bytes



**Memory addresses are defined using unsigned binary integers.**



## Example 1

A computer has 32 MB (megabytes) of memory. How many bits are needed to address any single byte in memory?

### Solution

The memory address space is 32 MB, or  $2^{25}$  ( $2^5 \times 2^{20}$ ). This means that we need  $\log_2 2^{25}$ , or **25 bits**, to address each byte.

## Example 2

A computer has 128 MB of memory. Each word in this computer is eight bytes. How many bits are needed to address any single word in memory?

### Solution

The memory address space is 128 MB, which means  $2^{27}$ . However, each word is eight ( $2^3$ ) bytes, which means that we have  $2^{24}$  words. This means that we need  $\log_2 2^{24}$ , or **24 bits**, to address each word.

# MEMORY OPERATIONS

- Today, **general-purpose computers** use a set of instructions called a **program** to process data.
- A computer executes the program to create output data from input data
- Both program instructions and data operands are stored in memory
- Two basic operations requires in memory access
  - Load operation (Read or Fetch)-Contents of specified memory location are read by processor
  - Store operation (Write)- Data from the processor is stored in specified memory location

# Assignment of Byte Address

- **Big-endian** and **little-endian** are terms that describe the order in which a sequence of bytes are stored in computer **memory**. **Big-endian** is an order in which the "**bigend**" (most significant value in the sequence) is stored first (at the lowest storage address).
-

# Assignment of byte addresses

- Little Endian (e.g., in DEC, Intel)
  - » low order byte stored at lowest address
  - » byte0 byte1 byte2 byte3

- Eg: 46,78,96,54 (32 bit data)

- H BYTE ← L BYTE

- 8000
- 8001
- 8002
- 8003
- 8004

54
96
78
46

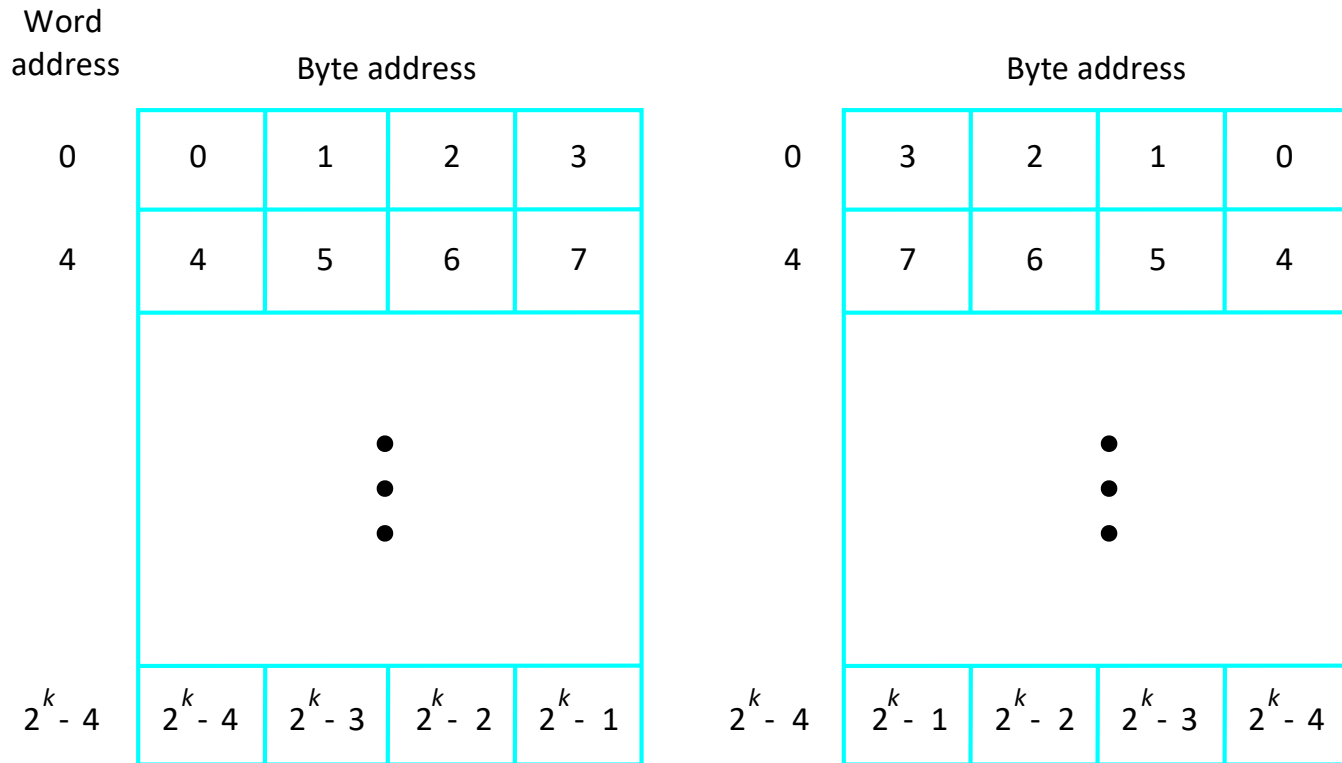
# Big Endian

- Big Endian (e.g., in IBM, Motorola, Sun, HP)
  - » high order byte stored at lowest address
  - » byte3 byte2 byte1 byte0
- Programmers/protocols should be careful when transferring binary data between Big Endian and Little Endian machines

# Big-Endian and Little-Endian Assignments

Big-Endian: lower byte addresses are used for the most significant bytes of the word

Little-Endian: opposite ordering. lower byte addresses are used for the less significant bytes of the word



(a) Big-endian assignment

(b) Little-endian assignment

Figure 2.7. Byte and word addressing.

- In case of 16 bit data, aligned words begin at byte addresses of 0,2,4,.....
- In case of 32 bit data, aligned words begin at byte address of 0,4,8,.....
- In case of 64 bit data, aligned words begin at byte addresses of 0,8,16,.....
- In some cases words can start at an arbitrary byte address also then, we say that word locations are **unaligned**

# Instruction and Instruction Sequencing



- INSTRUCTION SET ARCHITECTURE:-Complete instruction set of the processor
- BASIC 4 TYPES OF OPERATION:-
  - Data transfer between memory and processor register
  - Arithmetic and logic operation
  - Program sequencing and control
  - I/O transfer

# Register transfer notation (RTN)

Transfer between processor registers & memory, between processor register & I/O devices

Memory locations, registers and I/O register names are identified by a symbolic name in uppercase alphabets

- LOC, PLACE, MEM are the address of memory location
- R1 , R2,... are processor registers
- DATA\_IN, DATA\_OUT are I/O registers

- Contents of location is indicated by using square brackets [ ]
- RHS of RTN always denotes a values, and is called **Source**
- LHS of RTN always denotes a symbolic name where **value** is to **be stored** and is called **destination**
- Source contents are not modified
- Destination contents are overwritten

# Examples of RTN statements

- $R2 \leftarrow [LOCN]$
- $R4 \leftarrow [R3] + [R2]$

# ASSEMBLY LANGUAGE NOTATION (ALN)

- RTN is easy to understand and but cannot be used to represent machine instructions
- Mnemonics can be converted to machine language, which processor understands using assembler

Eg:

1. `MOVE LOCN, R2`
2. `ADD R3, R2, R4`

# TYPES OF INSTRUCTION

## ➤ Three address instruction

- Syntax: **Operation source 1, source 2, destination**
- Eg: ADD D,E,F     where D,E,F are memory location
- Advantage: Single instruction can perform the complete operation
- Disadvantage : Instruction code will be too large to fit in one word location in memory

# TWO ADDRESS INSTRUCTION

- Syntax : Operation source, destination
- Eg:      MOVE    E,F                                  MOVE D,F  
             ADD     D,F           OR                 ADD E,F

## Perform ADD A,B,C using 2 instructions

MOVE B,C

ADD A,C

- ❖ Disadvantage: Single instruction is not sufficient to perform the entire operation.

# ONE ADDRESS INSTRUCTION

- Syntax- **Operation source/destination**
- In this type either a source or destination operand is mentioned in the instruction
- Other operand is implied to be a processor register called **Accumulator**
- Eg: ADD B (general)
- Load D;            ACC       ← [memlocation \_D]
- ADD E;            ACC       ← (ACC)+(E)
- STORE F;        memlocation\_F    (ACC )



# Zero address instruction

- Location of all operands are defined implicitly
- Operands are stored in a structure called pushdown stack

# Continued

- If processor supports ALU operations one data in memory and other in register then the instruction sequence is
  - MOVE    D, Ri
  - ADD     E, Ri
  - MOVE   Ri, F
- If processor supports ALU operations only with registers then one has to follow the instruction sequence given below
  - LOAD    D, Ri
  - LOAD    E, Rj
  - ADD     Ri, Rj
  - MOVE    Rj, F

# Instruction Formats

Example: Evaluate  $(A+B) * (C+D)$

- Three-Address

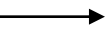
- |        |           |                               |
|--------|-----------|-------------------------------|
| 1. ADD | R1, A, B  | ; $R1 \leftarrow M[A] + M[B]$ |
| 2. ADD | R2, C, D  | ; $R2 \leftarrow M[C] + M[D]$ |
| 3. MUL | X, R1, R2 | ; $M[X] \leftarrow R1 * R2$   |

# Instruction Formats

Example: Evaluate  $(A+B) * (C+D)$

- Two-Address

- |        |        |                             |
|--------|--------|-----------------------------|
| 1. MOV | R1, A  | ; $R1 \leftarrow M[A]$      |
| 2. ADD | R1, B  | ; $R1 \leftarrow R1 + M[B]$ |
| 3. MOV | R2, C  | ; $R2 \leftarrow M[C]$      |
| 4. ADD | R2, D  | ; $R2 \leftarrow R2 + M[D]$ |
| 5. MUL | R1, R2 | ; $R1 \leftarrow R1 * R2$   |
| 6. MOV | X, R1  | ; $M[X] \leftarrow R1$      |



# Instruction Formats

Example: Evaluate  $(A+B) * (C+D)$

- One-Address

1. LOAD	A	; $AC \leftarrow M[A]$
2. ADD	B	; $AC \leftarrow AC + M[B]$
3. STORE	T	; $M[T] \leftarrow AC$
4. LOAD	C	; $AC \leftarrow M[C]$
5. ADD	D	; $AC \leftarrow AC + M[D]$
6. MUL	T	; $AC \leftarrow AC * M[T]$
7. STORE	X	; $M[X] \leftarrow AC$

# Instruction Formats

Example: Evaluate  $(A+B) * (C+D)$

- Zero-Address

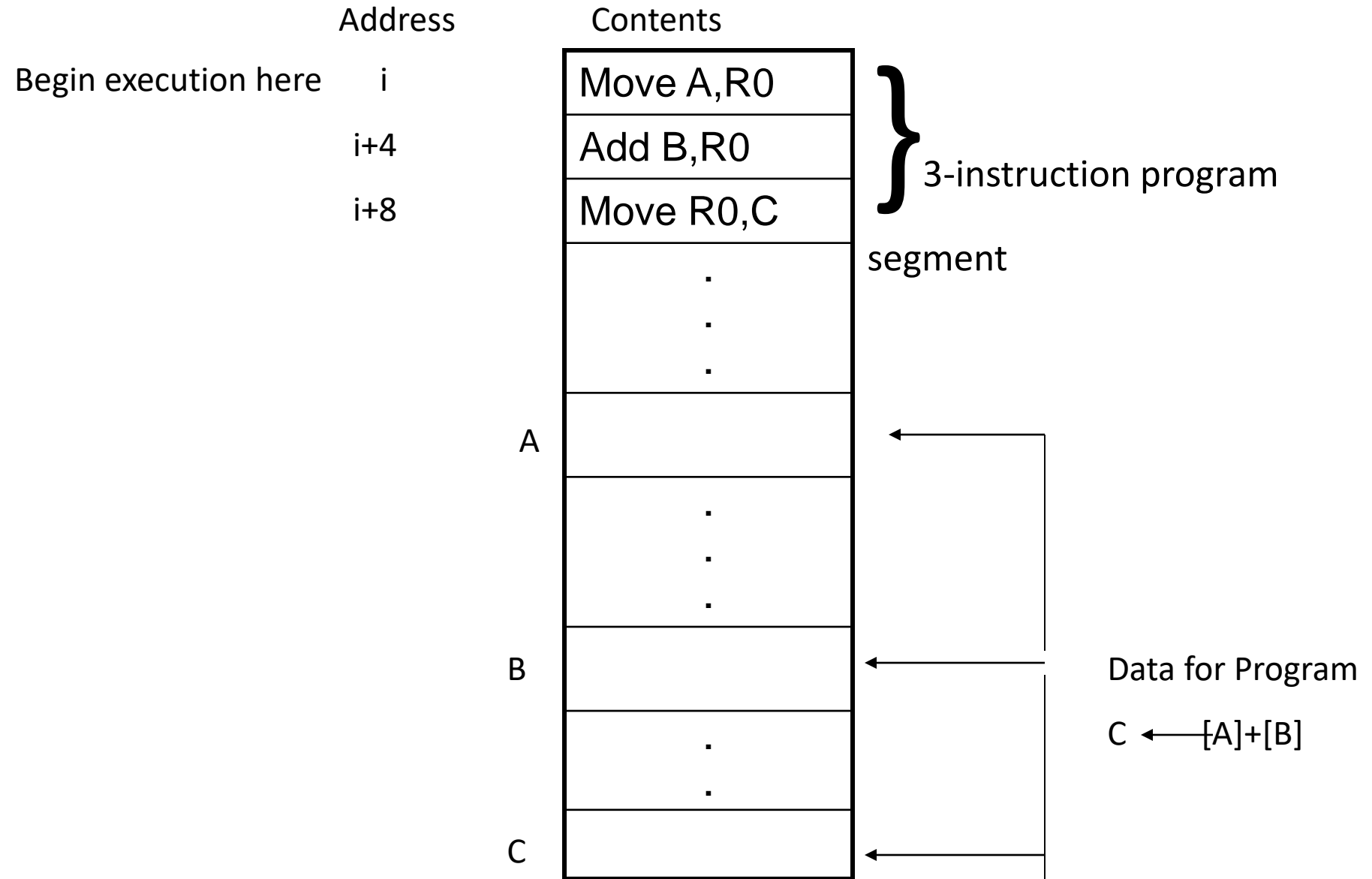
- |         |   |                                |
|---------|---|--------------------------------|
| 1. PUSH | A | ; TOS $\leftarrow$ A           |
| 2. PUSH | B | ; TOS $\leftarrow$ B           |
| 3. ADD  |   | ; TOS $\leftarrow$ (A + B)     |
| 4. PUSH | C | ; TOS $\leftarrow$ C           |
| 5. PUSH | D | ; TOS $\leftarrow$ D           |
| 6. ADD  |   | ; TOS $\leftarrow$ (C + D)     |
| 7. MUL  |   | ; TOS $\leftarrow$ (C+D)*(A+B) |
| 8. POP  | X | ; M[X] $\leftarrow$ TOS        |



# Basic Instruction Cycle

- Basic computer operation cycle
  - Fetch the instruction from memory into a control register (PC)
  - Decode the instruction
  - Locate the operands used by the instruction
  - Fetch operands from memory (if necessary)
  - Execute the operation in processor registers
  - Store the results in the proper place
  - Go back to step 1 to fetch the next instruction

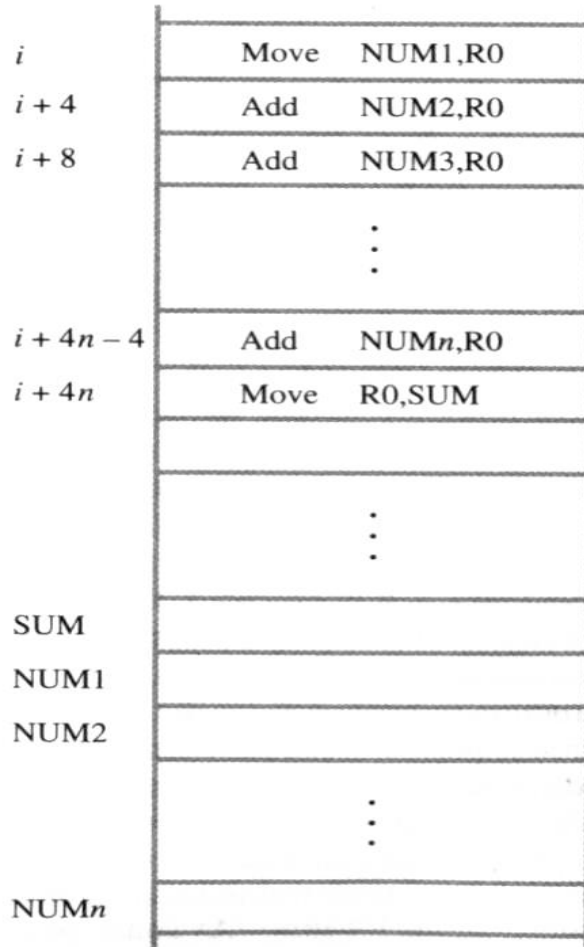
# INSTRUCTION EXECUTION & STRIAGHT LINE SEQUENCING



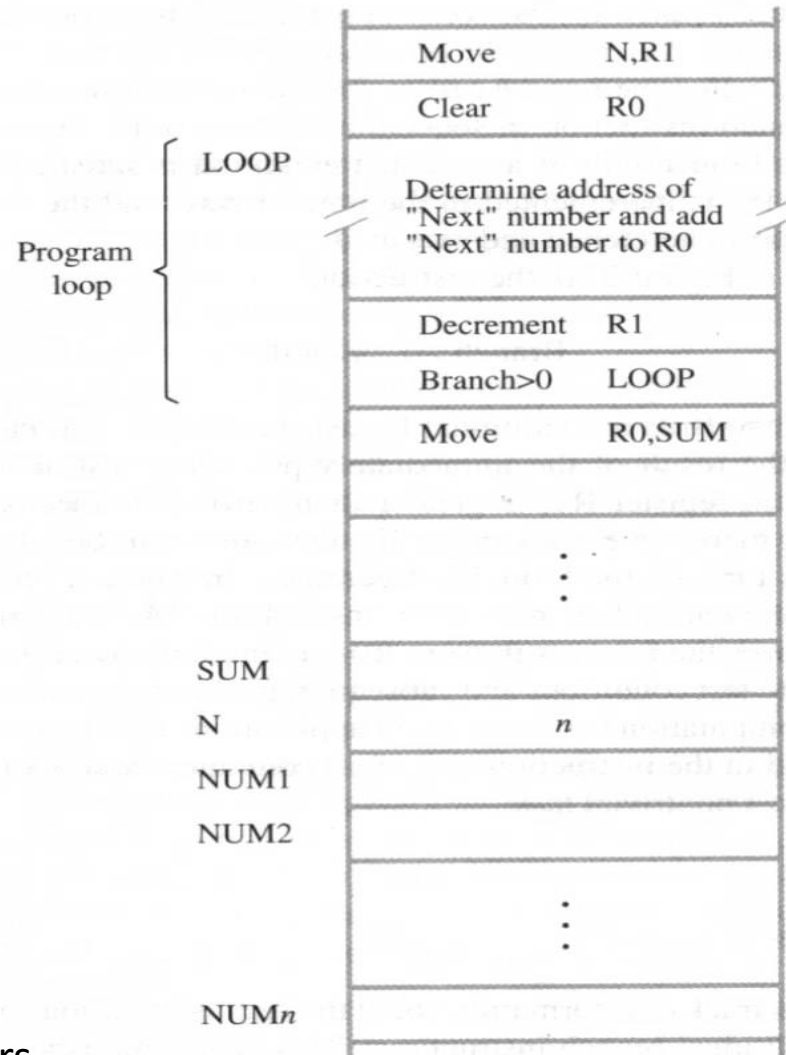


- **PC – Program counter**: hold the address of the next instruction to be executed
- **Straight line sequencing**: If fetching and executing of instructions is carried out one by one from successive addresses of memory, it is called straight line sequencing.
- Major two phase of instruction execution
- **Instruction fetch phase**: Instruction is fetched from memory and is placed in instruction register IR
- **Instruction execute phase**: Contents of IR is decoded and processor carries out the operation either by reading data from memory or registers.

# BRANCHING



A straight line program for adding  $n$  numbers



Using a loop to add  $n$  numbers

# BRANCHING

- Branch instructions are those which change the normal sequence of execution.
- Sequence can be changed either conditionally or unconditionally.
- Accordingly we have **conditional** branch instructions and **unconditional branch instructions**.
- Conditional branch instructions change the sequence only when certain conditions are met.
- Unconditional branch instructions change the sequence of execution irrespective of the condition of the results.

# CONDITION CODES

- **CONDITIONAL CODE FLAGS:** The processor keeps track of information about the results of various operations for use by subsequent conditional branch instructions
- **N – Negative**      1 if results are Negative  
                             0 if results are Positive
  - **Z – Zero**            1 if results are Zero  
                             0 if results are Non zero
  - **V – Overflow**      1 if arithmetic overflow occurs  
                             0 non overflow occurs
  - **C – Carry**            1 if carry and from MSB bit  
                             0 if there is no carry from MSB bit

# Conditional Branch Instructions

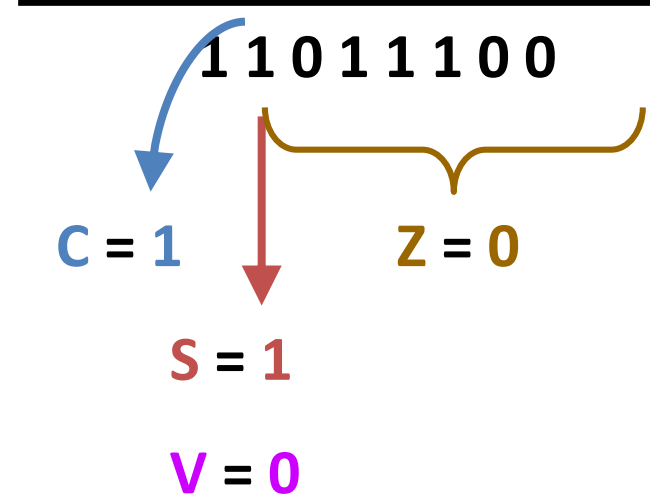
- Example:

— A: 1 1 1 1 0 0 0 0

— B: 0 0 0 1 0 1 0 0

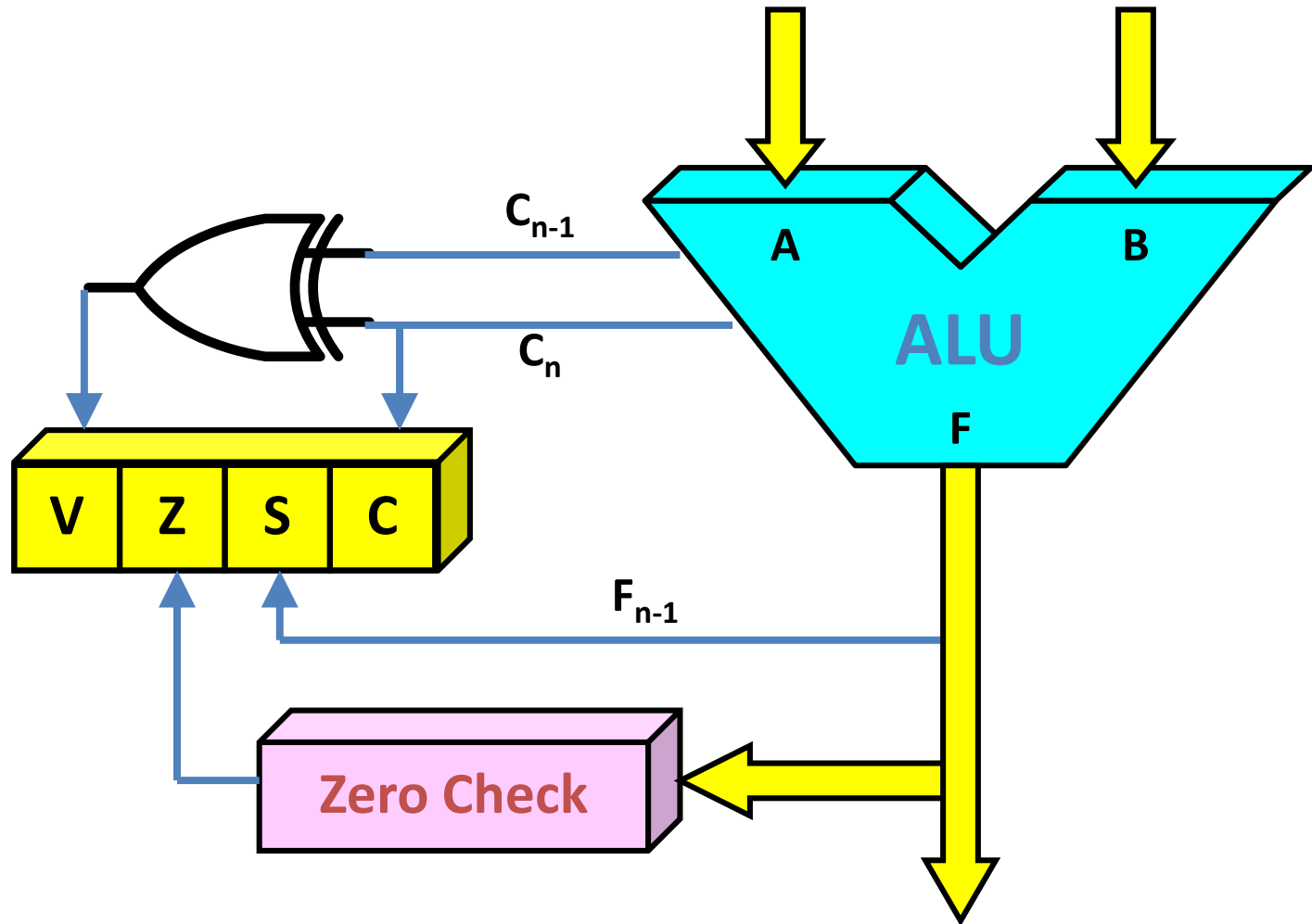
A: 1 1 1 1 0 0 0 0

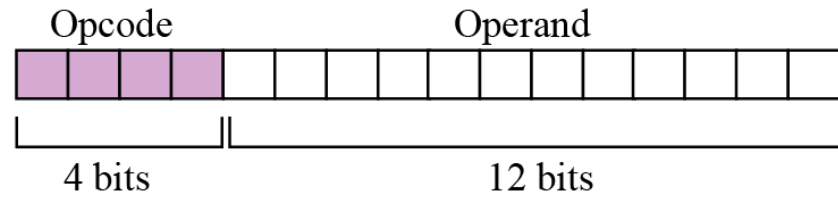
+(-B): 1 1 1 0 1 1 0 0



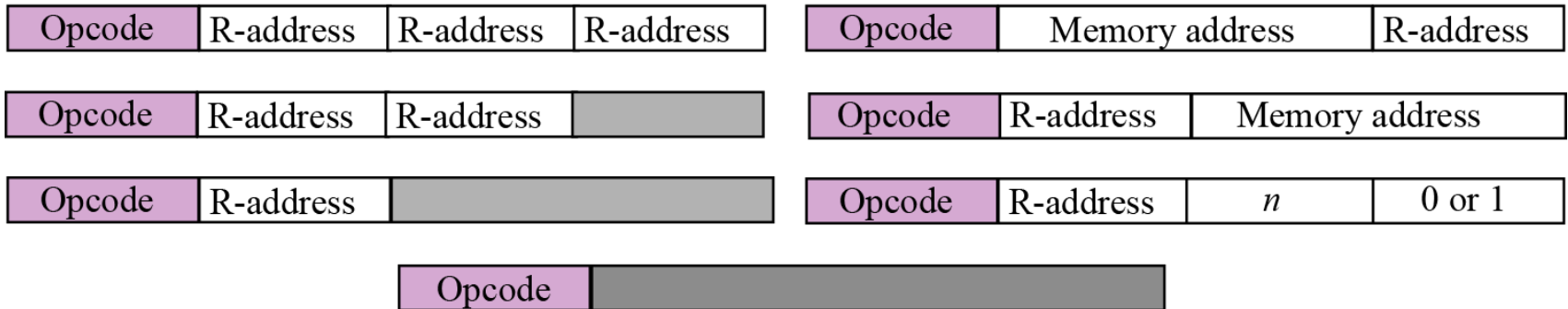
**Overflow** occurs when the magnitude of a number exceeds the range allowed by the size of the bit field

# Status Bits





a. Instruction format



b. Instruction types

**Figure** Format and different instruction types



# Processing the instructions

Simple computer, like most computers, uses machine cycles.

A cycle is made of **three phases: fetch, decode and execute**.

During the **fetch phase**, the instruction whose address is determined by the PC is obtained from the memory and loaded into the IR. The PC is then incremented to point to the next instruction.

During the **decode phase**, the instruction in IR is decoded and the required operands are fetched from the register or from memory.

During the **execute phase**, the instruction is executed and the results are placed in the appropriate memory location or the register.

Once the third phase is completed, the control unit starts the cycle again, but now the PC is pointing to the next instruction.

The process continues until the CPU reaches a HALT instruction.

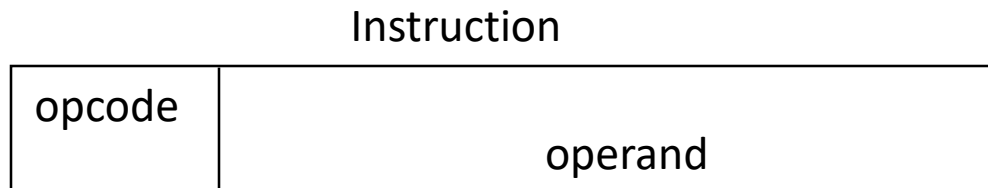
# Types of Addressing Modes

The different ways in which the location of the operand is specified in an instruction are referred to as addressing modes

- Immediate Addressing
- Direct Addressing
- Indirect Addressing
- Register Addressing
- Register Indirect Addressing
- Relative Addressing
- Indexed Addressing

# Immediate Addressing

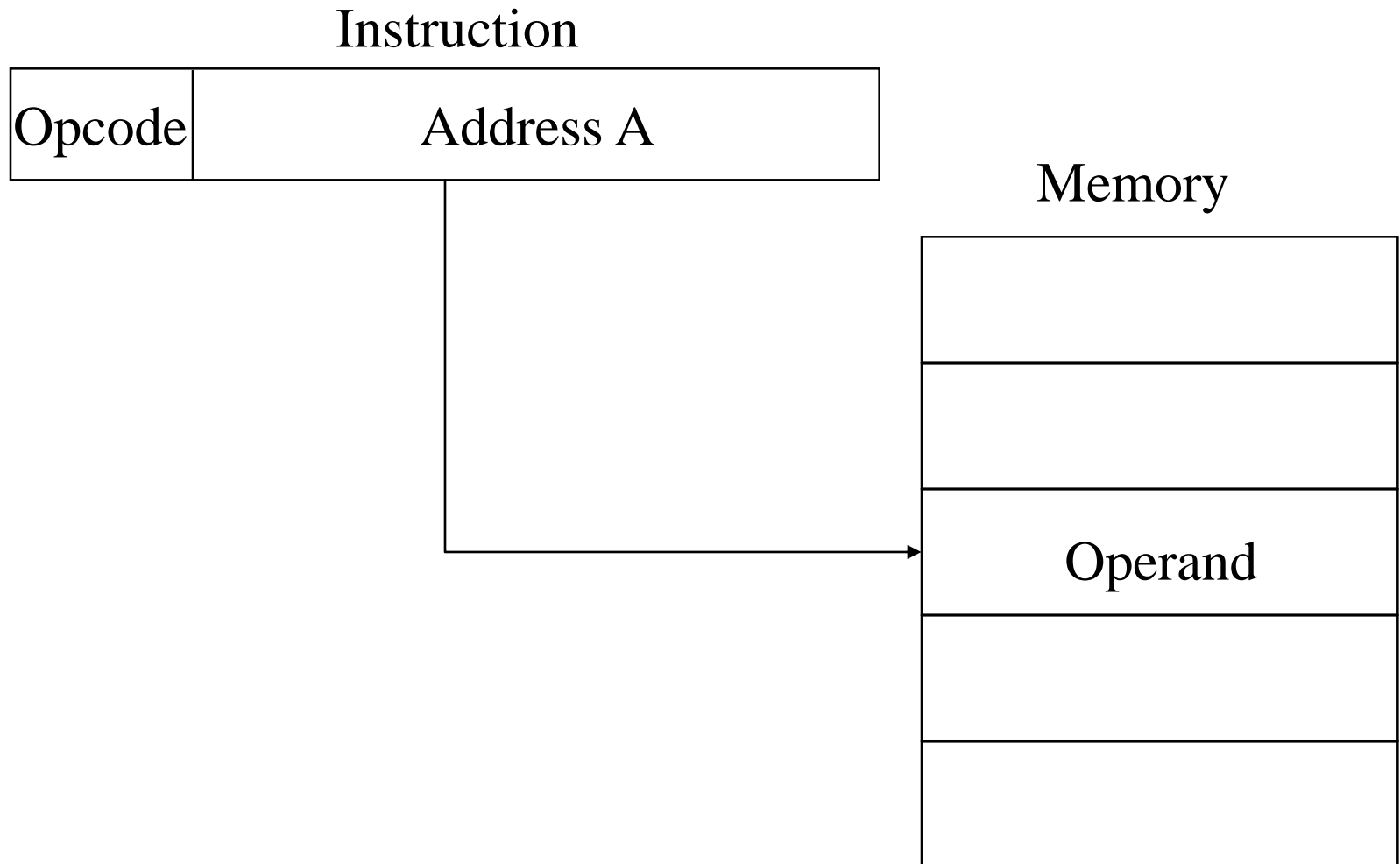
- Operand is given explicitly in the instruction
- Operand = Value
- e.g. ADD 5
  - Add 5 to contents of accumulator
  - 5 is operand
- No memory reference to fetch data
- Fast
- Limited range



# Direct Addressing

- Address field contains address of operand
- Effective address (EA) = address field (A)
- e.g. ADD A
  - Add contents of cell A to accumulator
  - Look in memory at address A for operand
- Single memory reference to access data
- No additional calculations to work out effective address
- Limited address space

# Direct Addressing Diagram



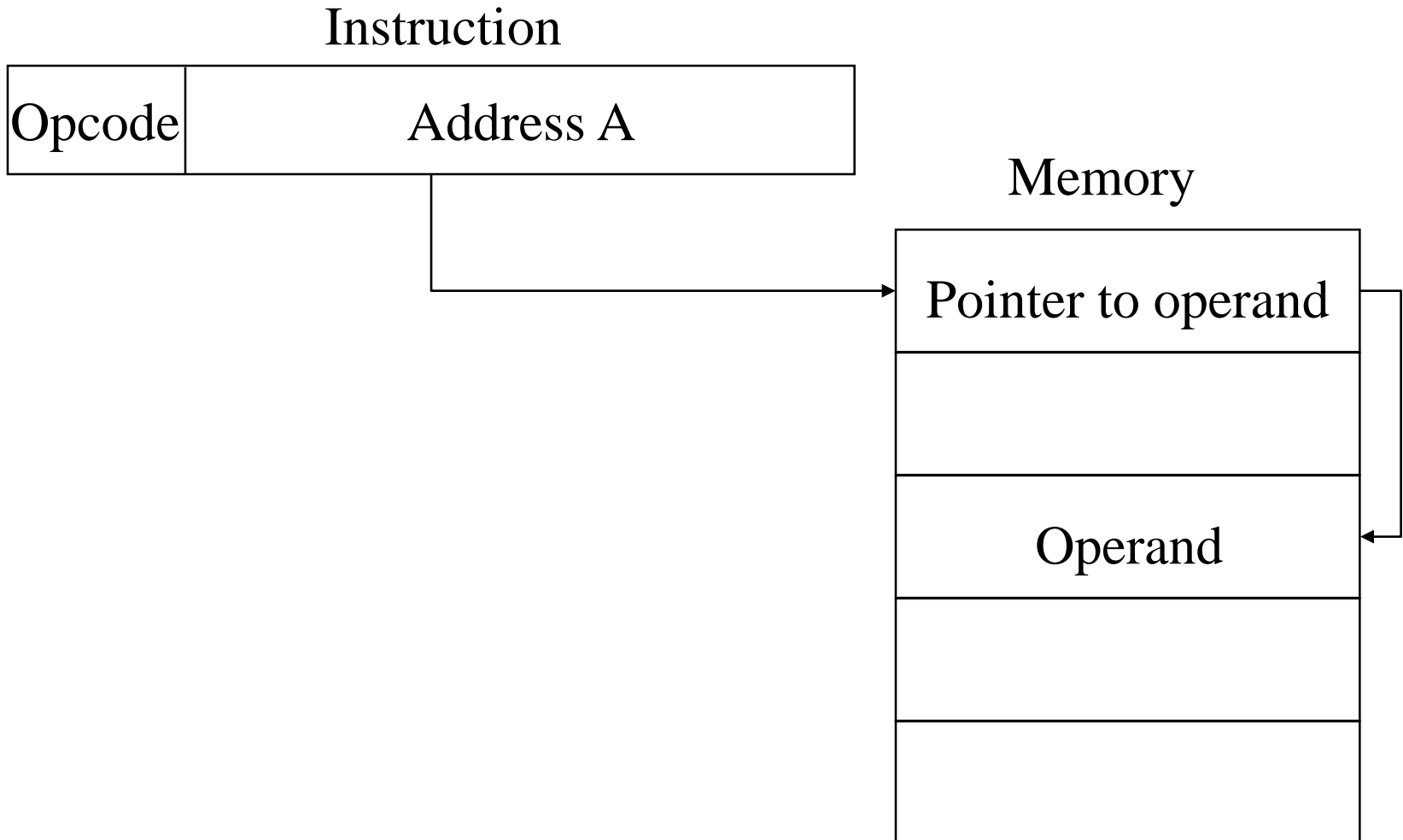
# Indirect Addressing (1)

- Memory cell pointed to by address field contains the address of (pointer to) the operand
- $EA = [A]$ 
  - Look in A, find address (A) and look there for operand
- e.g. ADD (A)
  - Add contents of cell pointed to by contents of A to accumulator

# Indirect Addressing (2)

- Large address space
- $2^n$  where  $n$  = word length
- May be nested, multilevel, cascaded
  - e.g.  $EA = (((A)))$ 
    - Draw the diagram yourself
- Multiple memory accesses to find operand
- Hence slower

# Indirect Addressing Diagram





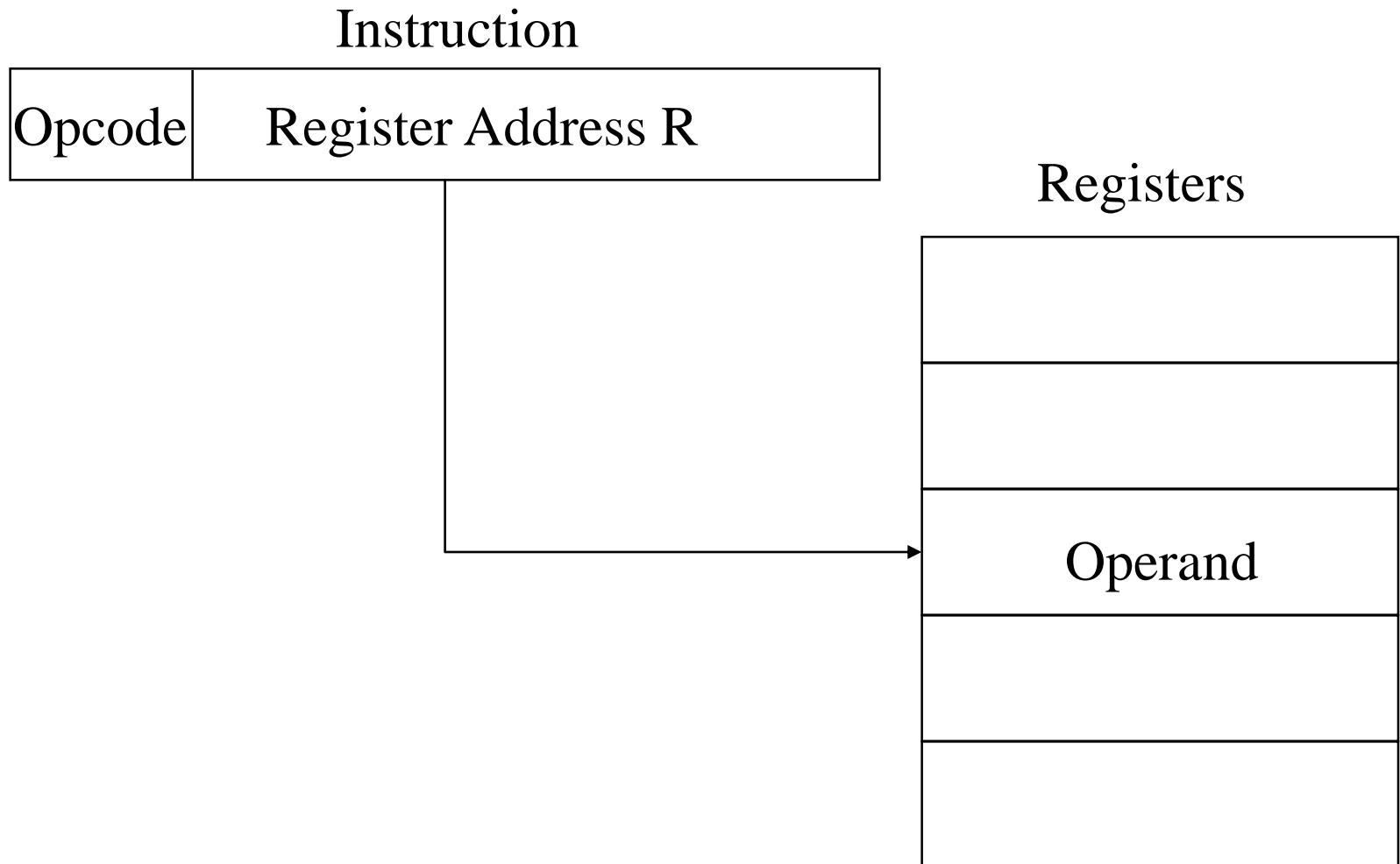
# Register Addressing (1)

- Operand is held in register named in address field
- $EA = R$
- Limited number of registers
- Very small address field needed
  - Shorter instructions
  - Faster instruction fetch

# Register Addressing (2)

- No memory access
- Very fast execution
- Very limited address space
- Multiple registers helps performance
  - Requires good assembly programming or compiler writing

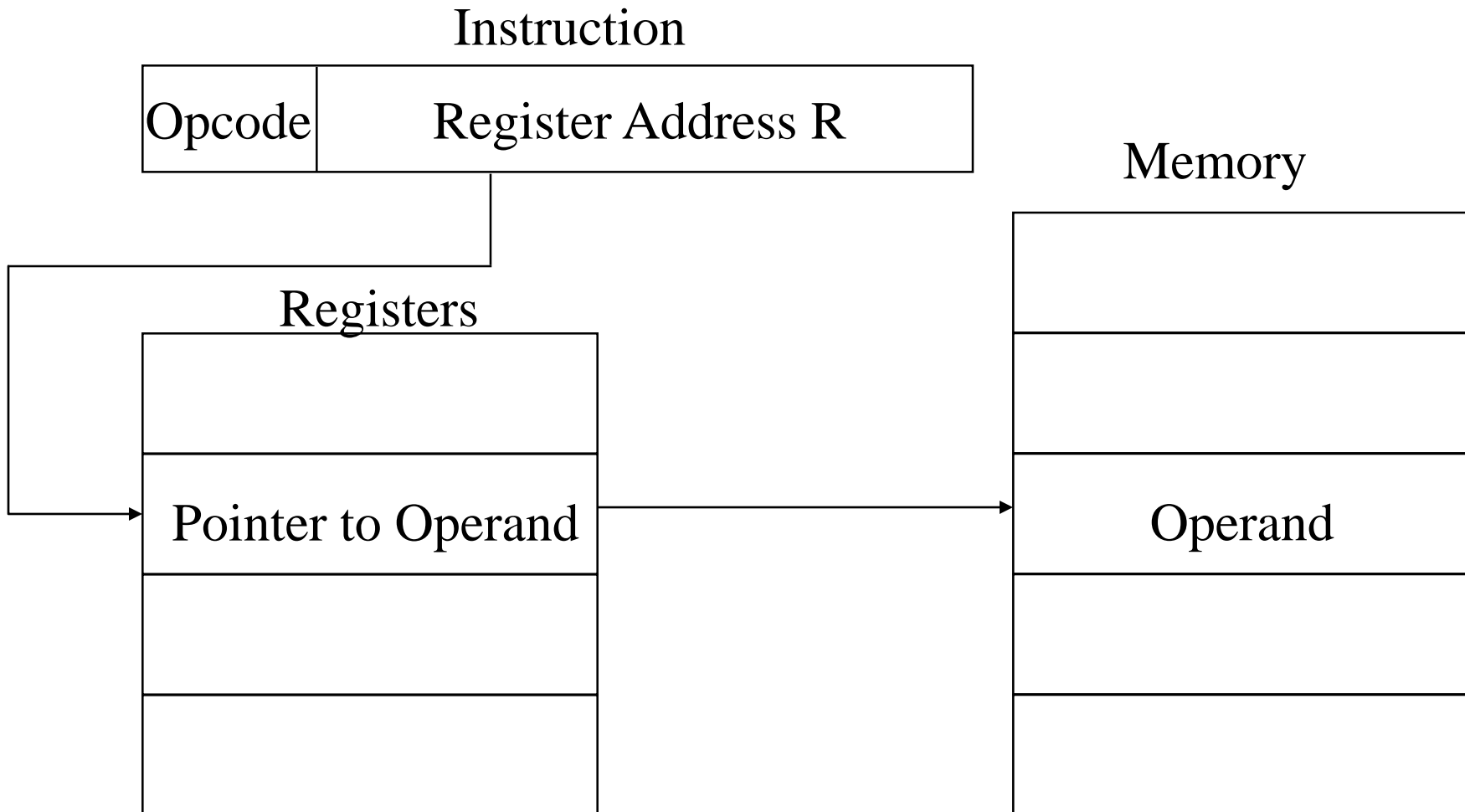
# Register Addressing Diagram



# Register Indirect Addressing

- C.f. indirect addressing
- $EA = [R]$
- Operand is in memory cell pointed to by contents of register R
- Large address space ( $2^n$ )
- One fewer memory access than indirect addressing

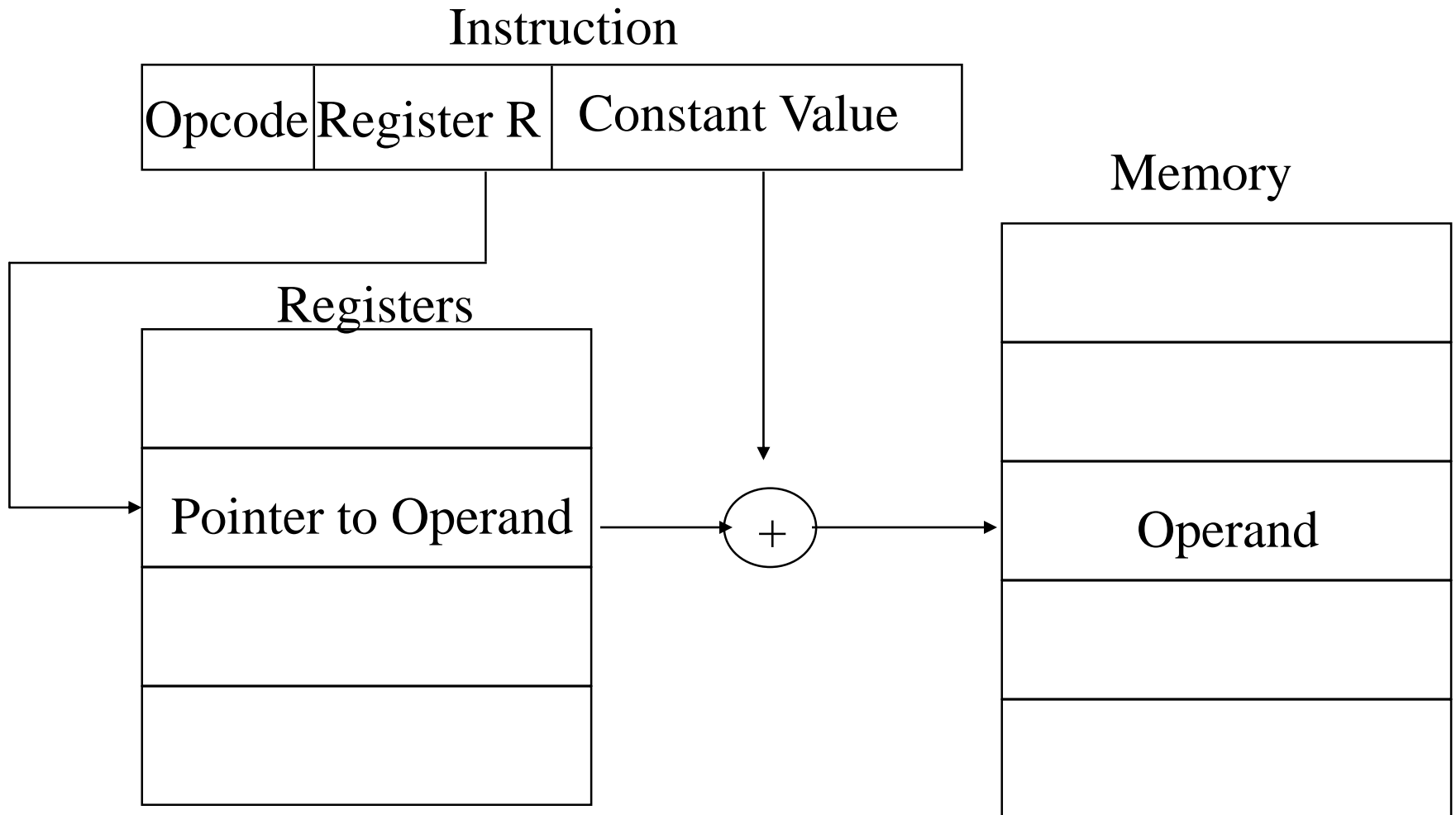
# Register Indirect Addressing Diagram



# Indexed Addressing

- $EA = X + [R]$
- Address field hold two values
  - $X$  = constant value (offset)
  - $R$  = register that holds address of memory locations
  - or vice versa
- (Offset given as constant or in the index register)  
Add 20(R1),R2 or Add 1000(R1),R2

# Indexed Addressing Diagram



# Relative Addressing

- A version of displacement addressing
- $R = \text{Program counter, PC}$
- $EA = X + (PC)$
- i.e. get operand from X bytes away from current location pointed to by PC
- c.f locality of reference & cache usage



# Auto increment mode

- The effective address of the operand is the contents of a register specified in the instruction.
- After accessing the operand, the contents of this register are automatically incremented to point to the next item in the list
- $EA=[Ri]; \text{ Increment } Ri \quad \text{----} (Ri)+$

Eg:     Add (R2)+,R0

# Auto decrement mode

- The contents of a register specified in the instruction are first automatically decremented and are then used as the effective address of the operand
- Decrement  $R_i$ ;  $EA = [R_i] - 1$  -----  $-(R_i)$