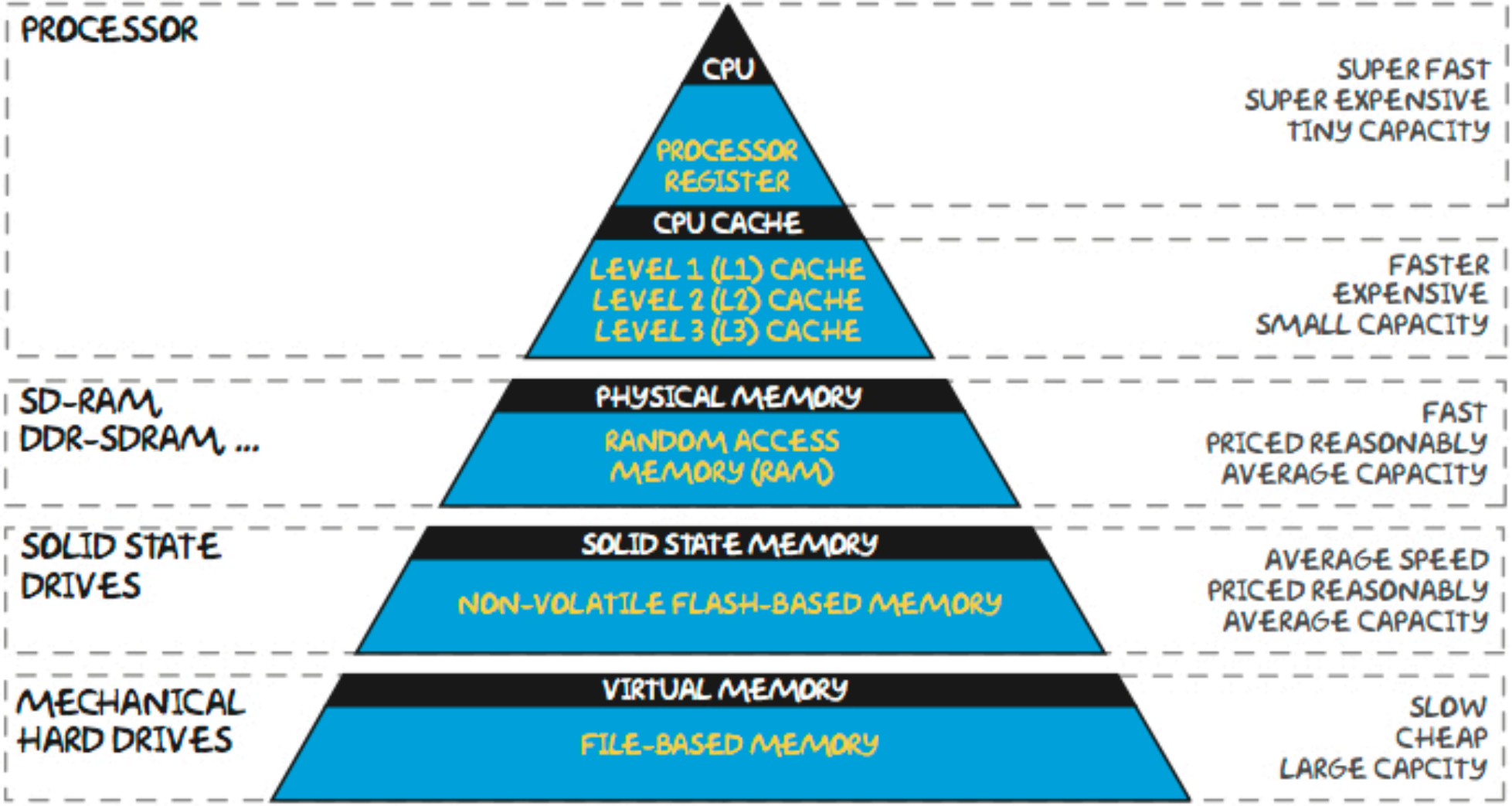
THE MEMORY HIERARCHY



MEMORY HIERARCHY

As one goes down the hierarchy, the following occur:

- ✓ Decreasing cost per bit
- ✓ Increasing capacity
- ✓ Increasing access time
- ✓ Decreasing frequency of access of the memory by the processor

During the design of computer system, smaller, more expensive, faster memories are supplemented by larger, cheaper, slower memories, hence balancing the associated tradeoffs

MEMORY ACCESS TIME

The performance of the memory is determined by the average memory access time.

The average memory access time (AMAT) is the average time required to access data from the memory.

The memory access time in affected by the presence of cache memory.

We shall consider the following terminologies before proceeding to the calculation.

MEMORY ACCESS TIME

Cache hit : cache hit is said to occur when data requested is found in the cache

Hit time: time taken to access data in the cache

Cache miss: when data is not found in the cache

Hit rate : the fraction of accesses that result in a hit

Miss rate: the fraction of access that results in a miss

Hit rate+ miss rate=1 (100%)

Miss penalty : The time taken to bring block of data from the main memory due to cache miss.

*AMAT=hit_rate*hit_time+miss_rate*miss_penalty*

MEMORY ACCESS TIME

AMAT(ns or clocks)=hit_time*hit_rate+(miss_rate*miss_penalty)

Given h=hit rate, c=hit time, M=miss penalty

AMAT=hC+(1-h)M for 1 level cache memory

AMAT=$h_1C_1+(1-h_1)(h_2C_2+(1-h_2)M)$ for 2 level cache memory

MEMORY ACCESS TIME

Problem 1

If a direct mapped cache has a hit rate of 95%, a hit time of 4 ns, and a miss penalty of 100 ns, what is the AMAT?

Problem 2

All modern microprocessors have an on-chip cache which is called the L1 cache and another larger off chip cache called the L2 cache. Assume a L1 hit time of 1, a L2 hit time of 5 and a L2 miss penalty of 17. Given a L1 hit rate of 98% and a L2 hit rate of 98% the average access time is ??????

MEMORY ACCESS TIME

Improve cache performance by:

In order to improve the performance of the cache, we could do the following

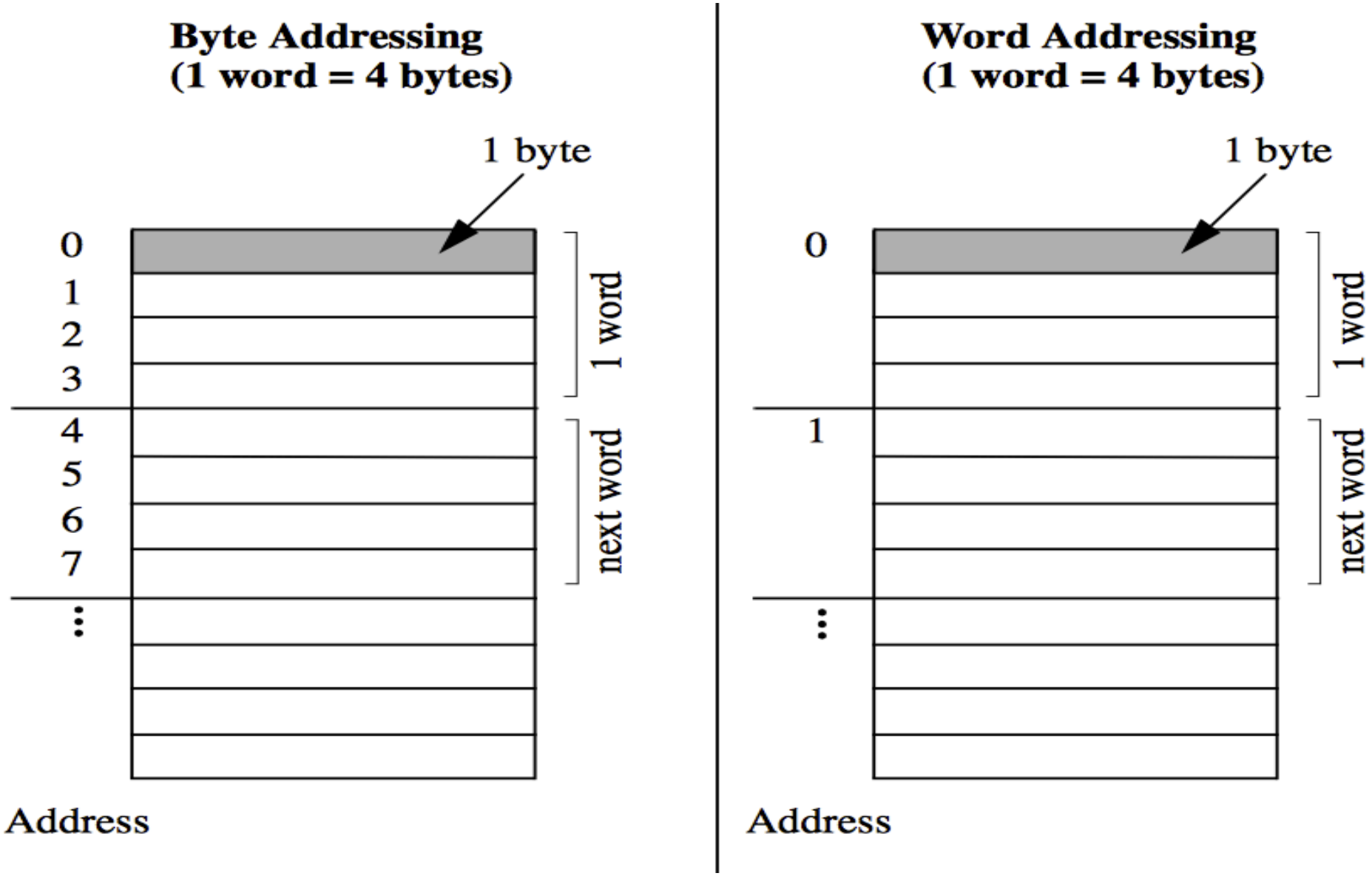
- Reduce the miss rate
- Reduce the miss penalty, or
- Reduce the time to hit in the cache.

BYTE ADDRESSABILITY

- There are three basic information quantities to deal with: bit, byte, and word.
- A byte is always 8 bits but the word length typically ranges from 16 to 64 bits.
- It is impractical to assign distinct addresses to individual bit locations in the memory.
- The most practical assignment is to have successive addresses refer to successive byte locations in the memory.
- binary address always points to a single byte only.
- A word is just a group of bytes – 2, 4, 8 depending upon the data bus size of the CPU.
- In reality memory is only byte addressable

BYTE ADDRESSABILITY

- Most computers today have memories that are *byte-addressable*; thus each byte in the memory has a unique address that can be used to address it.
- Under this addressing scheme, a word corresponds to a number of addresses.
  - A 16–bit word at address Z contains bytes at addresses Z and Z + 1.
  - A 32–bit word at address Z contains bytes at addresses Z, Z + 1, Z + 2, and Z + 3.
- In many computers with byte addressing, there are constraints on word addresses.
  - A 16–bit word must have an even address(multiple of 2)
  - A 32–bit word must have an address of multiple of 4.



BYTE ADDRESSING VS. WORD ADDRESSING

BYTE ADDRESSABILITY

- Suppose a byte-addressable computer with a 32-bit address space.
- The highest byte address is $2^{32} - 1$.
- From this fact and the address allocation to multi-byte words, we conclude that;
  - the highest address for a 16-bit word is $(2^{32} - 2)$, and
  - the highest address for a 32-bit word is $(2^{32} - 4)$,

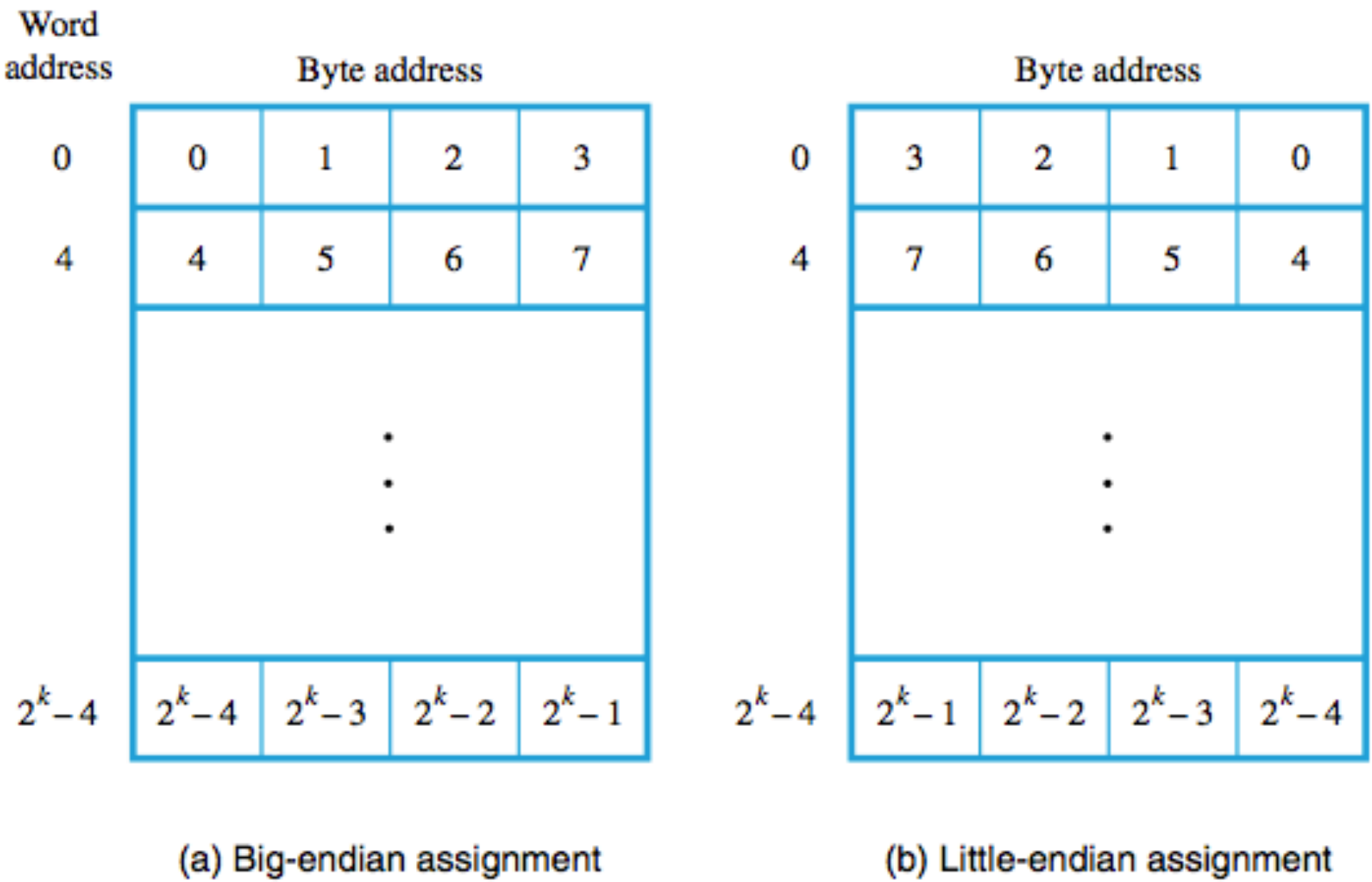
BIG AND LITTLE ENDIAN

There are two ways that byte addresses can be assigned across words; *Big-endian and Little-endian*

Big-endian and *little-endian* are terms that describe the order in which a sequence of bytes are stored in computer memory.

The name *big-endian* is used when lower byte addresses are used for the more significant bytes (the leftmost bytes) of the word

The name *little-endian* is used for the opposite ordering, where the lower byte addresses are used for the less significant bytes (the rightmost bytes) of the word.


Word address
Byte address
0
4
2^k-4
0 1 2 3
4 5 6 7
2^k-4 2^k-3 2^k-2 2^k-1
(a) Big-endian assignment
Byte address
0
4
2^k-4
3 2 1 0
7 6 5 4
2^k-1 2^k-2 2^k-3 2^k-4
(b) Little-endian assignment

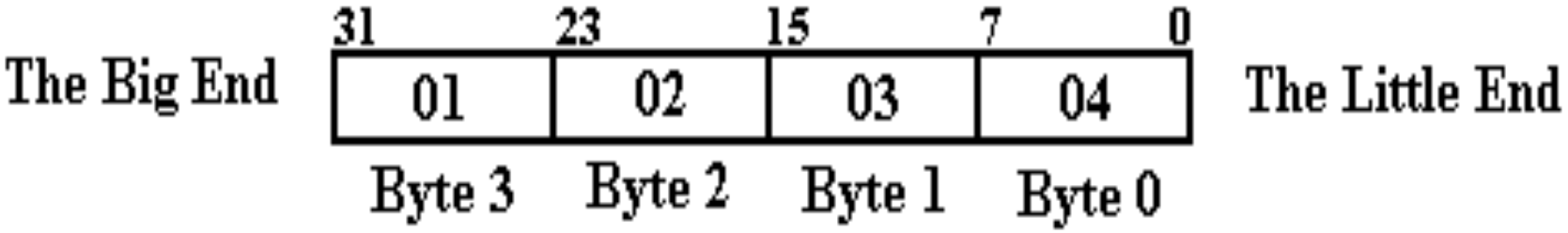
BIG AND LITTLE ENDIAN

Consider the 32–bit number represented by the eight–digit hexadecimal number 0x01020304, stored at location Z in memory.

In all byte-addressable memory locations, this number will be stored in the four consecutive addresses

Z, (Z + 1), (Z + 2), and (Z + 3).

Let 0x01 represents bits 31 – 24, 0x02 represents bits 23 – 16,0x03 represents bits 15 – 8, and 0x04 represents bits 7 – 0 of the word. We have the representation below

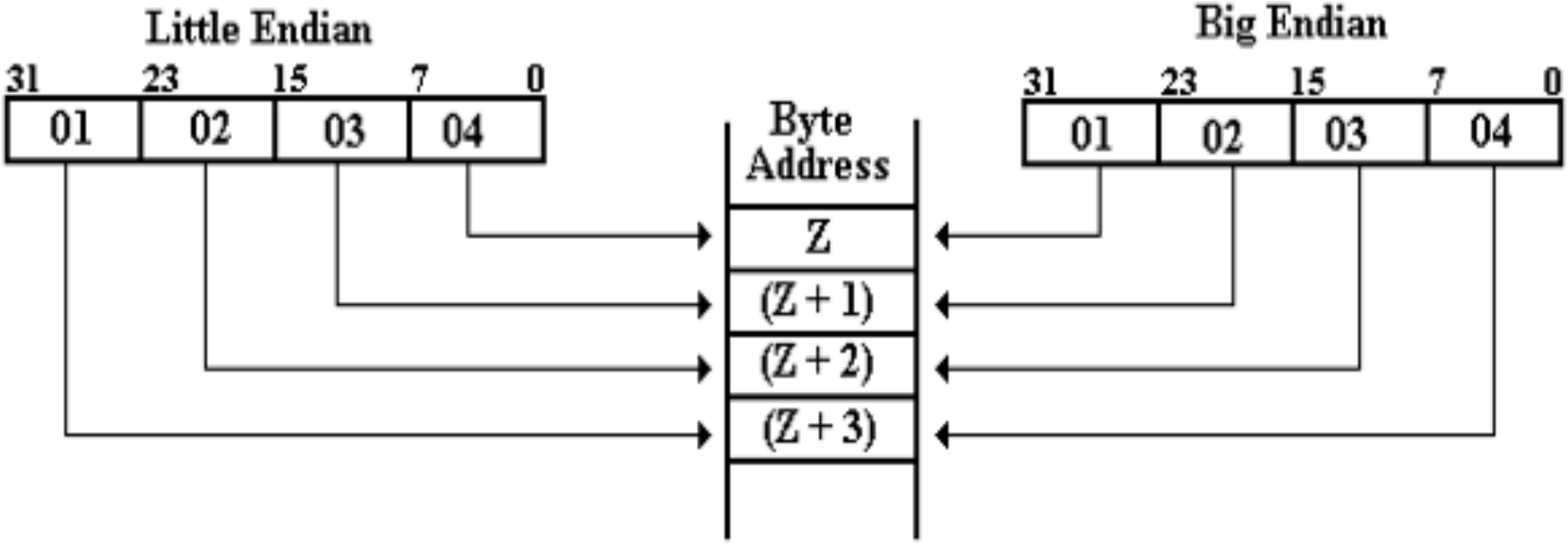


BIG AND LITTLE ENDIAN

The “big end” contains the most significant digits of the number and the “little end” contains the least significant digits of the number.

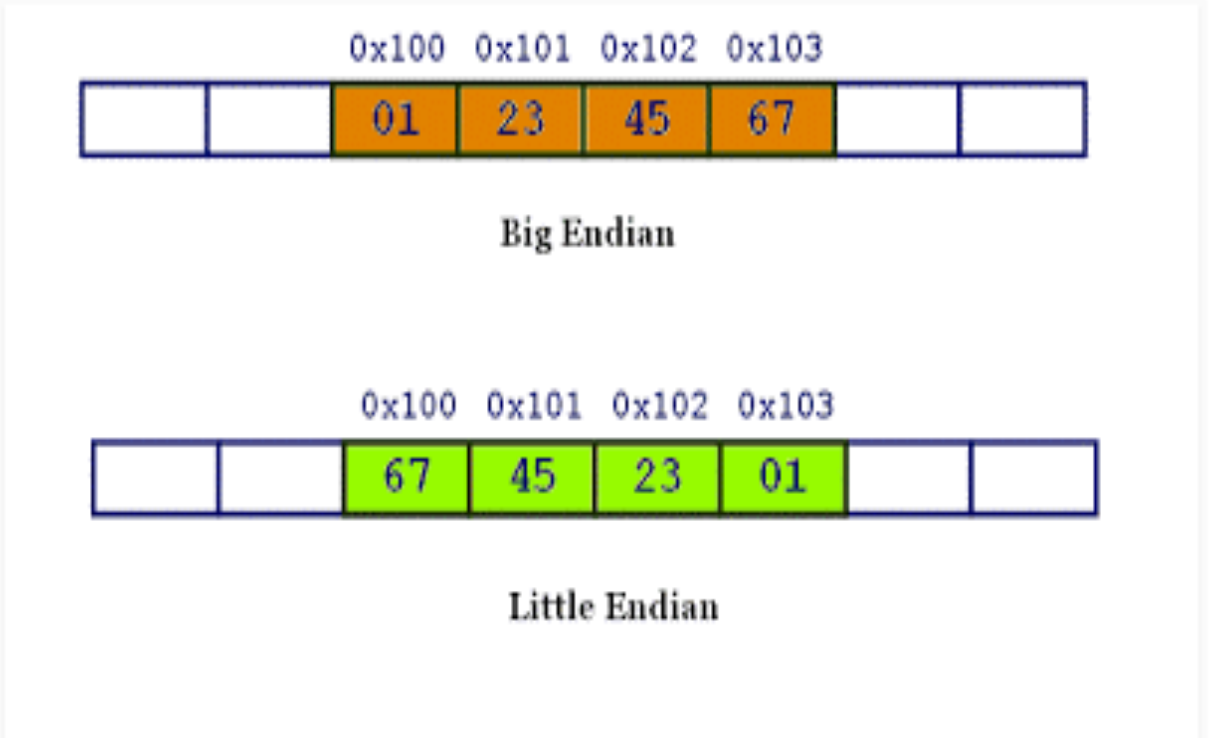
The hexadecimal values stored in these four byte addresses are shown below.

Address	Big-Endian	Little-Endian
Z	01	04
Z + 1	02	03
Z + 2	03	02
Z + 3	04	01



BIG-ENDIAN VS. LITTLE ENDIAN

Suppose integer is stored as 4 bytes (For those who are using DOS based compilers such as C++ 3.0 , integer is 2 bytes) then a variable x with value 0x01234567 will be stored as following.



BIG-ENDIAN VS. LITTLE ENDIAN

BIG AND LITTLE ENDIAN

- ✓ Big-endian computers include the IBM 360 series, Motorola 68xxx, and SPARC by Sun.
- ✓ Little-endian computers include the Intel Pentium and related computers.

What are bi-endians?

Bi-endian processors can run in both modes little and big endian.

EXERCISE

Consider the 16–bit number represented by the four hex digits 0A0B.

Suppose that the 16-bit word is at location W; show does the number will be stored in memory using Big-endian and little-endian

WORD ALIGNMENT

- We say that the word locations have *aligned* addresses if they begin at a byte address that is a multiple of the number of bytes in a word.
- the number of bytes in a word is a power of 2.
- if the word length is 16 (2 bytes), aligned words begin at byte addresses 0, 2, 4, . . . ,
- if the word length is 32 (4 bytes), aligned words begin at byte addresses 0, 4, 8, . . . ,
- if a word length is 64 (8 bytes), aligned words begin at byte addresses 0,8,16,....

INSTRUCTION SET ARCHITECTURE:ISA

- *Instruction set architecture (ISA)* or Instruction set is the set of all possible instructions that the CPU understands.
- It is the part of computer architecture that is visible to programmers.
- Every CPU has its own ISA which might not be understood by another CPU
- ISA serves as the starting point for the design of a new machine or modification of an existing one.
- The instructions executed by the processor are referred to as *machine instructions* or *computer instructions.*

INSTRUCTION SET ARCHITECTURE:ISA

Elements of a Machine Instruction

Each instruction must contain the information required by the processor for execution.

These elements are as follows:

Operation code: Specifies the operation to be performed (e.g., ADD, I/O). The operation is specified by a binary code, known as the operation code, or **opcode.**

Source operand reference: The operation may involve one or more source operands, that is, operands that are inputs for the operation.

Result operand reference: The operation may produce a result.

Next instruction reference: This tells the processor where to fetch the next instruction after the execution of this instruction is complete.

INSTRUCTION SET ARCHITECTURE:ISA

- Opcodes are represented by abbreviations, called *mnemonics,* that indicate the operation.

Common examples include

- ADD Add
- SUB Subtract
- MUL Multiply
- DIV Divide
- LOAD Load data from memory
- STORE Store data to memory

- Operands are also represented symbolically with alphabets or letters
- Operands could either be located in registers or memory locations

INSTRUCTION SET ARCHITECTURE

Key ISA decisions

When crafting an ISA, the following issues are considered

-Data types/size

What data types are supported /size allocated for each data

-Registers

(# of special purpose & General Purpose)

-*Addressing mode*

how to access operand from register/memory

-Instruction format

How are instructions represented/instruction length

-Memory architecture

Memory representation/byte addressing/word addressing

-types of operations

INSTRUCTION SET ARCHITECTURE

Classification of ISA

ISA can be classified based on the internal storage of the CPU:

Thus: operands specification(implicit or explicit)

- Stack Architecture
- Accumulator Architecture
- General Purpose Register Architecture

Stack Architecture

- Operands are not explicit. No registers /Memory
- They are on the top of the stack.
- For example, a binary operation pops the top two elements of the stack, applies the operation and pushes the result back on the stack.

INSTRUCTION SET ARCHITECTURE

Stack Architecture

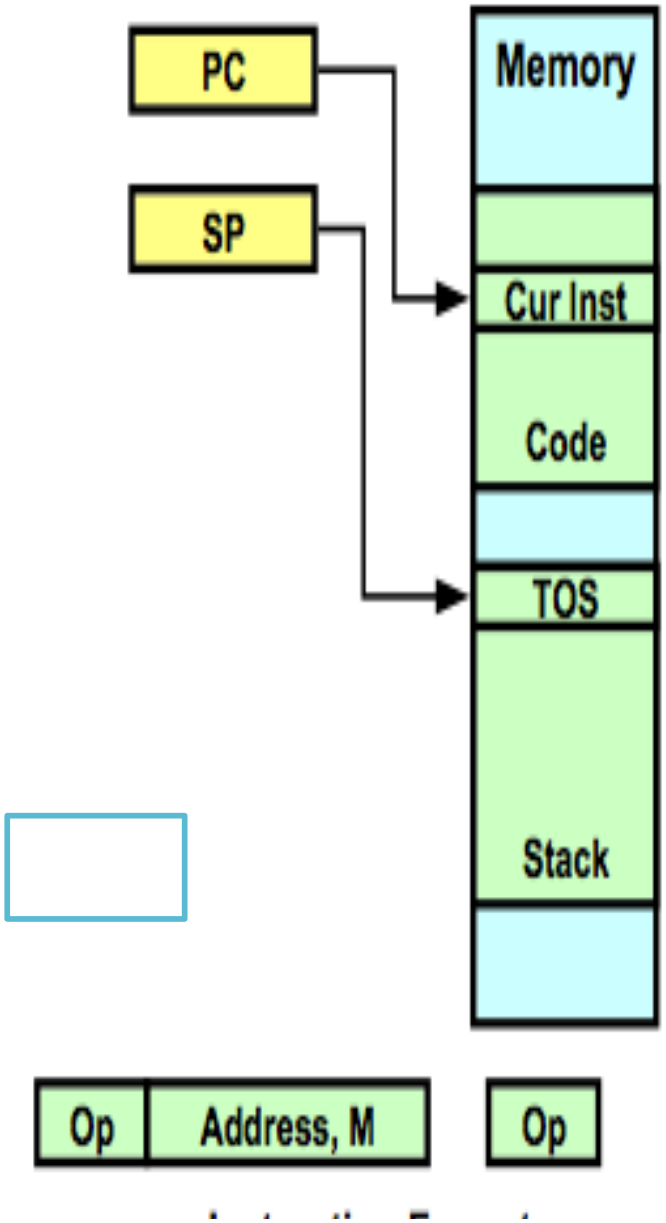
Example: a=b + c:

```
push b
push c
add
pop a
```

Attributes

- ✧ Short instructions possible
- ✧ Compiler is easy to write
- ✧ Inefficient code
- ✧ Burroughs B5500/6500,

HP 3000/70, Java VM

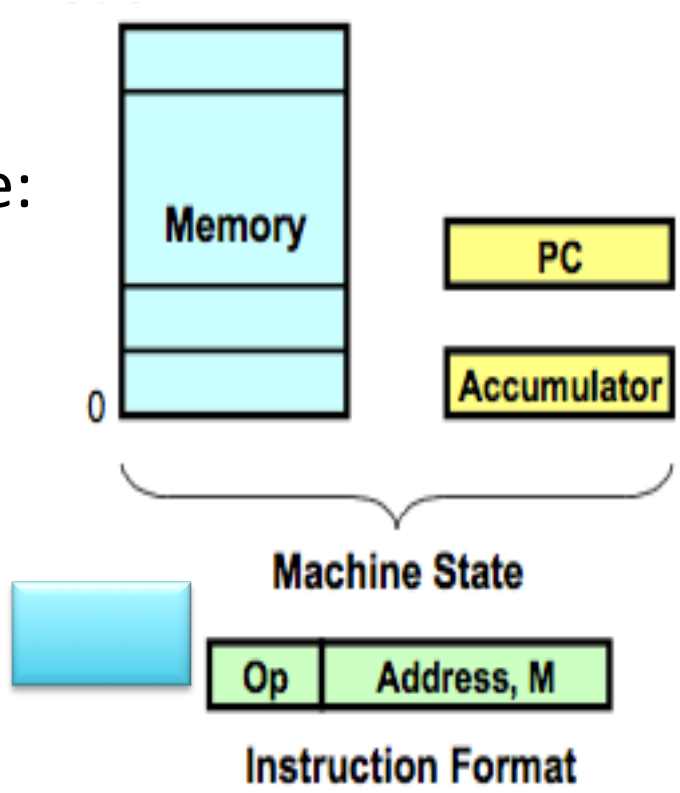


INSTRUCTION SET ARCHITECTURE

Accumulator Architecture

- ✧ One of the source operands is implicit, the accumulator.
- ✧ Single register M_p
- ✧ The destination operand is assumed to be the accumulator.
- ✧ Short instructions possible
- ✧ The most popular early architecture:

IBM 7090, DEC PDP-8, MOS 6502



Example:

AC←AC * M_p

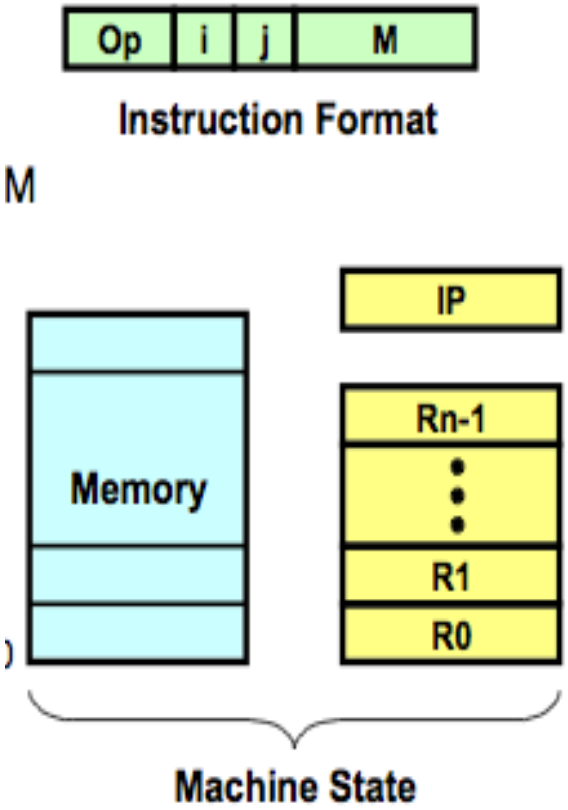
INSTRUCTION SET ARCHITECTURE

General Purpose Register Architecture(GPRs)

- Operands are explicit: memory operands or register operands.
- The dominant architecture: CDC 6600, IBM 360/370 (RX),PDP-11, 68000, all RISC machines, etc.
- longer instructions

Three classes exist

- Register -Memory Architecture
- Register-Register/Load-Store Architecture
- Memory-Memory Architecture



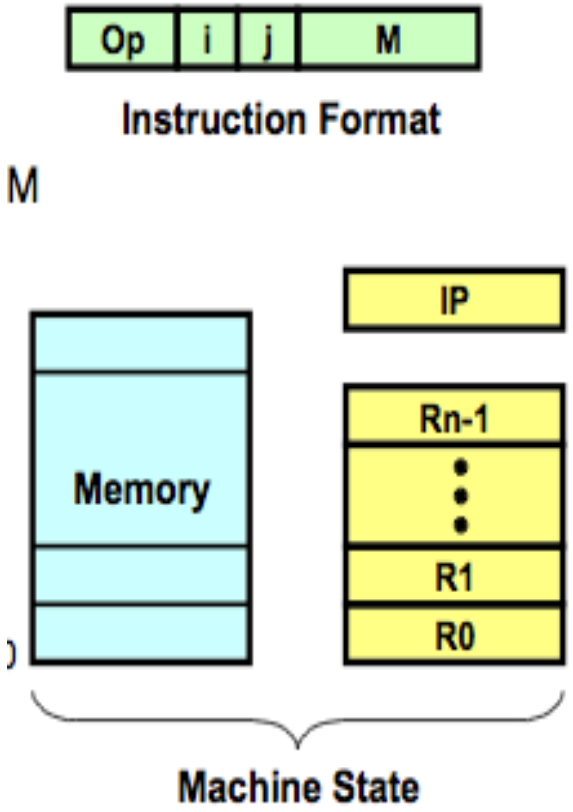
INSTRUCTION SET ARCHITECTURE

Register –Memory Architecture

- ✧ Any instruction can access memory or registers
- ✧ Longer instructions
- ✧ Dominant architecture: IBM 360/370
- ✧ High instruction count

EXAMPLE:

```
LOAD R1<--M(A)
ADD R1<--R1+M(B)
STORE M(C)<-- R1
```



INSTRUCTION SET ARCHITECTURE

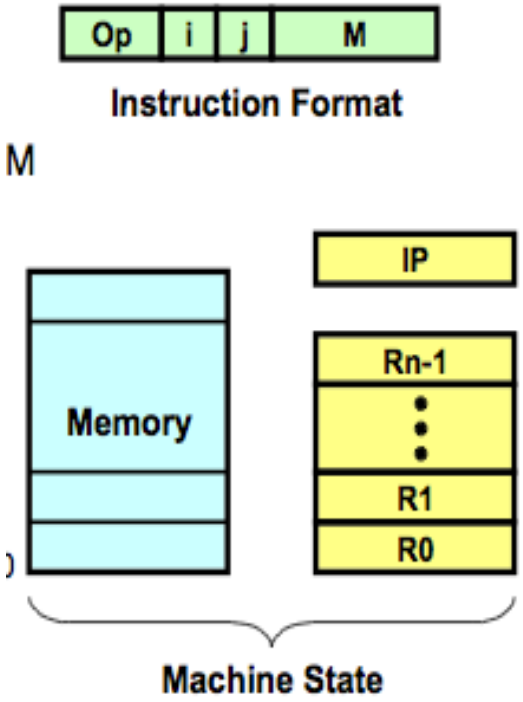
Load/Store (Register-Register)Architecture:

- ✧ Only load/store instructions can access memory.
- ✧ All other instructions use registers.
- ✧ Simple fixed-length instruction
- ✧ High instruction count

Dominant architecture:CDC6600, CRAY-1, most RISCs

Example:

```
LOAD R1 <-- M(A)
LOAD R2 <-- M(B)
ADD R3 <-- R1 + R2
STORE M(C)<-- R3
```



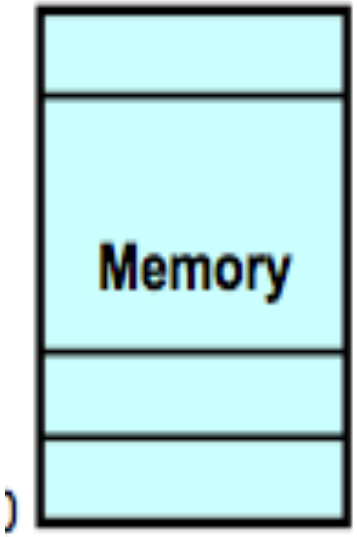
INSTRUCTION SET ARCHITECTURE

Memory-Memory Architecture

- ✧ All operands are explicitly accessed in memory
- ✧ No register are used
- ✧ Causes memory traffic

Example:

ADD M(C) <-- M(A)+M(B)



INSTRUCTION FORMAT

An instruction format defines the layout of the bits of an instruction, in terms of its constituent parts .

The format of an instruction is usually depicted in a rectangular box symbolizing the bits of the instruction as they appear in memory words or in a control register.

The bits of the instruction are divided into groups

called fields.

The most common fields are:

- An operation code that specifies the operation to be Performed.
- An address field that specifies a memory address or register.
- A mode field that specifies the way the operand or the effective address is determined.

INSTRUCTION FORMAT

Opcode	Mode	Address

Instruction Fields

- The operation code field of an instruction is a group of bits that define various processor operations such as add, subtract, etc.
- Address fields contain either a memory address operand field or a register address operand field.
- Mode fields offer a variety of ways in which an operand is chosen.
- Various types of instruction formats exist
  - Three address instruction format
  - Two address instruction format
  - One address instruction format
  - One and Half address instruction format
  - Zero address instruction format

INSTRUCTION FORMAT

- The operand field can encode *source field or, destination field*
- The source field represents the source operand(s).
- The source operand can be a constant, a value stored in a register, or a value stored in the memory.
- The destination field represents the place where the result of the operation is to be stored, for example, a register or a memory location.

3-ADDRESS INSTRUCTION USING REGISTERS

- A three-address instruction takes the form

  Operation oprd1, oprd2, oprd3.
- Where *oprd1, oprd2, and oprd3* refers to a registers.
- Consider, for example, the instruction *ADD R1,R2,R3.*

  From the instruction above, operation (opcode) is *addition* (ADD)

  `Source operands:R2 and R3`

  `Destination operand:R1`

  values to be added are those stored in registers *R2* and *R3* the results should be stored in register *R1*.

3-ADDRESSING MODES USING MEMORY LOCATION

Three-address instruction that refers to memory locations

Example:

ADD M[A], M[B], M[C]

`Source operand: Memory address B and C`

`Destination operand: Memory address A`

The instruction adds the contents of memory location C to the contents of memory location B and stores the result in memory location A.

2-ADDRESS INSTRUCTION USING REGISTERS

A two-address instruction takes the form

Operation oprd1, oprd2.

Where *oprd1 and oprd2* refers to a register

Example*: ADD R1,R2.*

Source operand: Registers R1 and R2

Destination operand: Register R1

This instruction adds the contents of register R1 to the contents of register R2 and stores the results in register R1.

This instruction is equivalent to a three-address instruction of the form

ADD R1,R1,R2.

2-ADDRESS INSTRUCTION USING MEMORY LOCATION

A similar instruction that uses memory locations instead of registers can take the form

Operation oprd1, oprd2.

Example

ADD M[A],M[B].

source operand: Memory addresses A and B

Destination operand: Memory address A

contents of memory location A is added to memory location B and the result is used to override the original contents of memory location A.

The three-address instruction `ADD M[A],M[B],M[C]`

can be performed by the two two-address instructions

`MOVE M[B],M[A] and ADD M[A],M[C].`

1-ADDRESS INSTRUCTION USING REGISTERS

A one-address instruction takes the form

operation oprd

In this case the instruction implicitly refers to a register, called the Accumulator R_{acc}

Example:

The instruction *ADD R1* The contents of the accumulator is added to the contents of the register R1 and the results are stored back into the accumulator $R_{acc.}$

The above instruction is equivalent to the three-address instruction $ADD\ R_{acc}, R_{acc,}\ R1$ or to the two-address instruction $ADD\ R_{acc}, R1$

DWUMFOUR ABDULLAI ABDUL-AZIZ

1-ADDRESS INSTRUCTION USING MEMORY LOCATION

If a memory location is used instead of a register then instruction is of the form

operation oprd

Example:

ADD M[B].

the instruction adds the content of the accumulator R_{acc} to the content of memory location B and stores the result back into the accumulator R_{acc}

1 AND HALF ADDRESS INSTRUCTION

Between the two- and the one-address instruction, there can be a one-and-half address instruction.

Example: `ADD R1,M[B]`

In this case, the instruction adds the contents of register R1 to the contents of memory location B and stores the result in register R1.

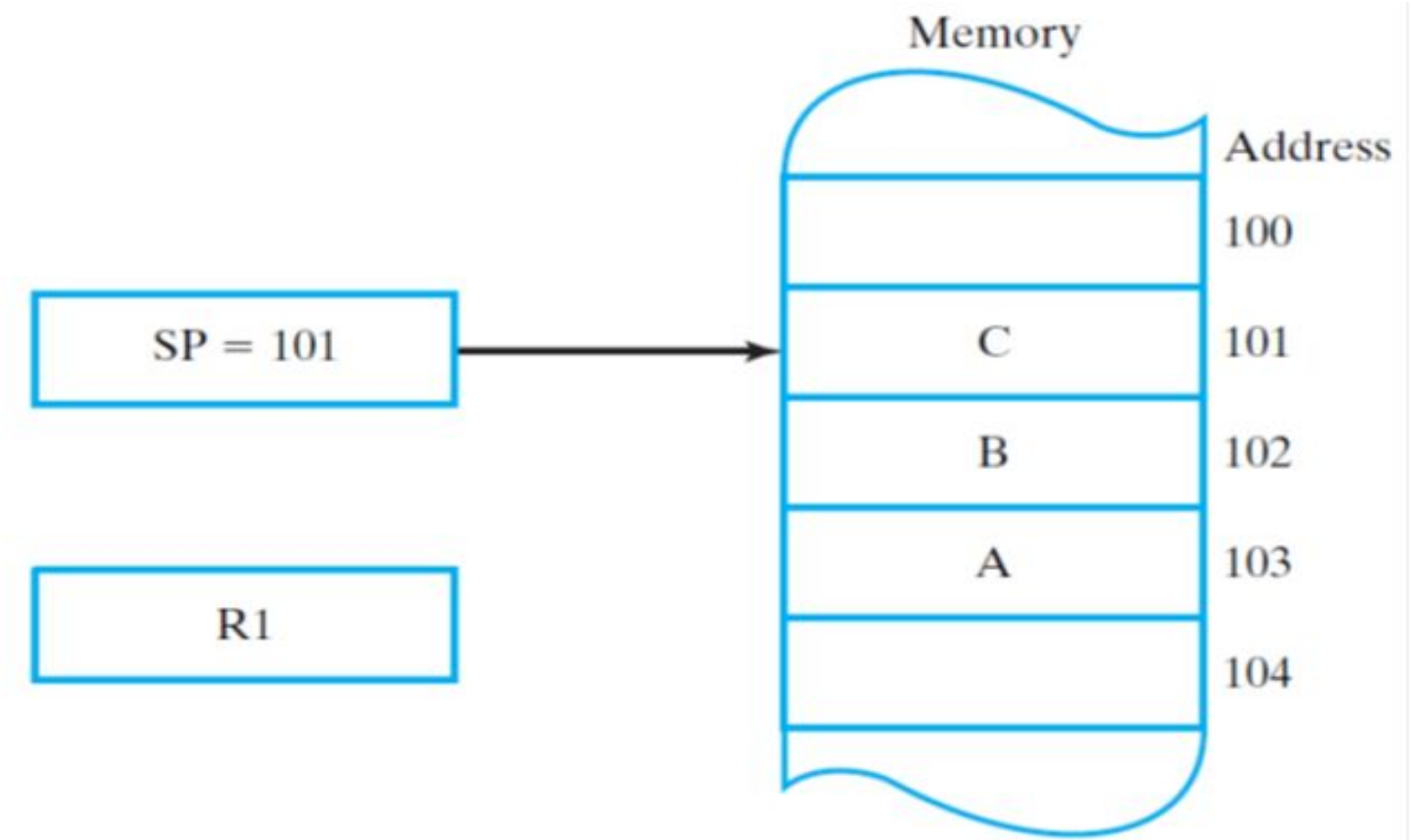
Owing to the fact that the instruction uses two types of addressing, that is, a register and a memory location, it is called a one-and-half-address instruction.

ZERO ADDRESS INSTRUCTION

- It is interesting to indicate that there exist zero-address instructions.
- These are the instructions that use stack operation.
- A stack is a data organization mechanism in which the last data item stored is the first data item retrieved (LIFO)
- Two specific operations can be performed on a stack. These are the push and the pop operations
- specific register, called the stack pointer (SP), is used to indicate the stack location that can be addressed.
- In the stack push operation, the SP value is used to indicate the location (called the top of the stack) in which the value is to be stored.

STACK

- The SP(stack Pointer) point to the location that stores the current data. This location is referred to as the top of the stack



STACK PUSH/POP OPERATION

- The important operations of the STACK are push and pop
- The push insert new data on top of the stack
- The pop removes the current data on top of the stack
- During push operation the SP is incremented to point to a new location to push the new data
- During pop operation the current data on top of the stack is removed and the SP is decremented to point to the previous location of the stack.

INSTRUCTION FORMAT

An instruction format defines the layout of the bits of an instruction, in terms of its constituent parts .

The format of an instruction is usually depicted in a rectangular box symbolizing the bits of the instruction as they appear in memory words or in a control register.

The bits of the instruction are divided into groups

called fields.

The most common fields are:

- An operation code that specifies the operation to be Performed.
- An address field that specifies a memory address or register.
- A mode field that specifies the way the operand or the effective address is determined.

INSTRUCTION FORMAT

Opcode	Mode	Address

Instruction Fields

- The operation code field of an instruction is a group of bits that define various processor operations such as add, subtract, etc.
- Address fields contain either a memory address operand field or a register address operand field.
- Mode fields offer a variety of ways in which an operand is chosen
- Various types of instruction formats exist
  - Three address instruction format
  - Two address instruction format
  - One address instruction format
  - One and Half address instruction format
  - Zero address instruction format

INSTRUCTION FORMAT

- The operand field can encode *source field or, destination field*
- The source field represents the source operand(s).
- The source operand can be a constant, a value stored in a register, or a value stored in the memory.
- The destination field represents the place where the result of the operation is to be stored, for example, a register or a memory location.

3-ADDRESS INSTRUCTION USING REGISTERS

- A three-address instruction takes the form

  Operation oprd1, oprd2, oprd3.
- Where *oprd1, oprd2, and oprd3* refers to a registers.
- Consider, for example, the instruction *ADD R1,R2,R3.*

  From the instruction above, operation (opcode) is *addition* (ADD)

  `Source operands:R2 and R3`

  `Destination operand:R1`

  values to be added are those stored in registers *R2* and *R3* the results should be stored in register *R1*.

3-ADDRESSING MODES USING MEMORY LOCATION

Three-address instruction that refers to memory locations

Example:

ADD M[A], M[B], M[C]

`Source operand: Memory address B and C`

`Destination operand: Memory address A`

The instruction adds the contents of memory location C to the contents of memory location B and stores the result in memory location A.

2-ADDRESS INSTRUCTION USING REGISTERS

A two-address instruction takes the form

Operation oprd1, oprd2.

Where *oprd1 and oprd2* refers to a register

Example*: ADD R1,R2.*

Source operand: Registers R1 and R2

Destination operand: Register R1

This instruction adds the contents of register R1 to the contents of register R2 and stores the results in register R1.

This instruction is equivalent to a three-address instruction of the form

ADD R1,R1,R2.

2-ADDRESS INSTRUCTION USING MEMORY LOCATION

A similar instruction that uses memory locations instead of registers can take the form

Operation oprd1, oprd2.

Example

ADD M[A],M[B].

source operand: Memory addresses A and B

Destination operand: Memory address A

contents of memory location A is added to memory location B and the result is used to override the original contents of memory location A.

The three-address instruction `ADD M[A],M[B],M[C]`

can be performed by the two two-address instructions

`MOVE M[B],M[A] and ADD M[A],M[C].`

1-ADDRESS INSTRUCTION USING REGISTERS

A one-address instruction takes the form

operation oprd

In this case the instruction implicitly refers to a register, called the Accumulator R_{acc}

Example:

The instruction *ADD R1* The contents of the accumulator is added to the contents of the register R1 and the results are stored back into the accumulator $R_{acc.}$

The above instruction is equivalent to the three-address instruction $ADD\ R_{acc}, R_{acc,}\ R1$ or to the two-address instruction $ADD\ R_{acc}, R1$

1-ADDRESS INSTRUCTION USING MEMORY LOCATION

If a memory location is used instead of a register then instruction is of the form

operation oprd

Example:

ADD M[B].

the instruction adds the content of the accumulator R_{acc} to the content of memory location B and stores the result back into the accumulator R_{acc}

1 AND HALF ADDRESS INSTRUCTION

Between the two- and the one-address instruction, there can be a one-and-half address instruction.

Example: *ADD R1,M[B]*

In this case, the instruction adds the contents of register R1 to the contents of memory location B and stores the result in register R1.

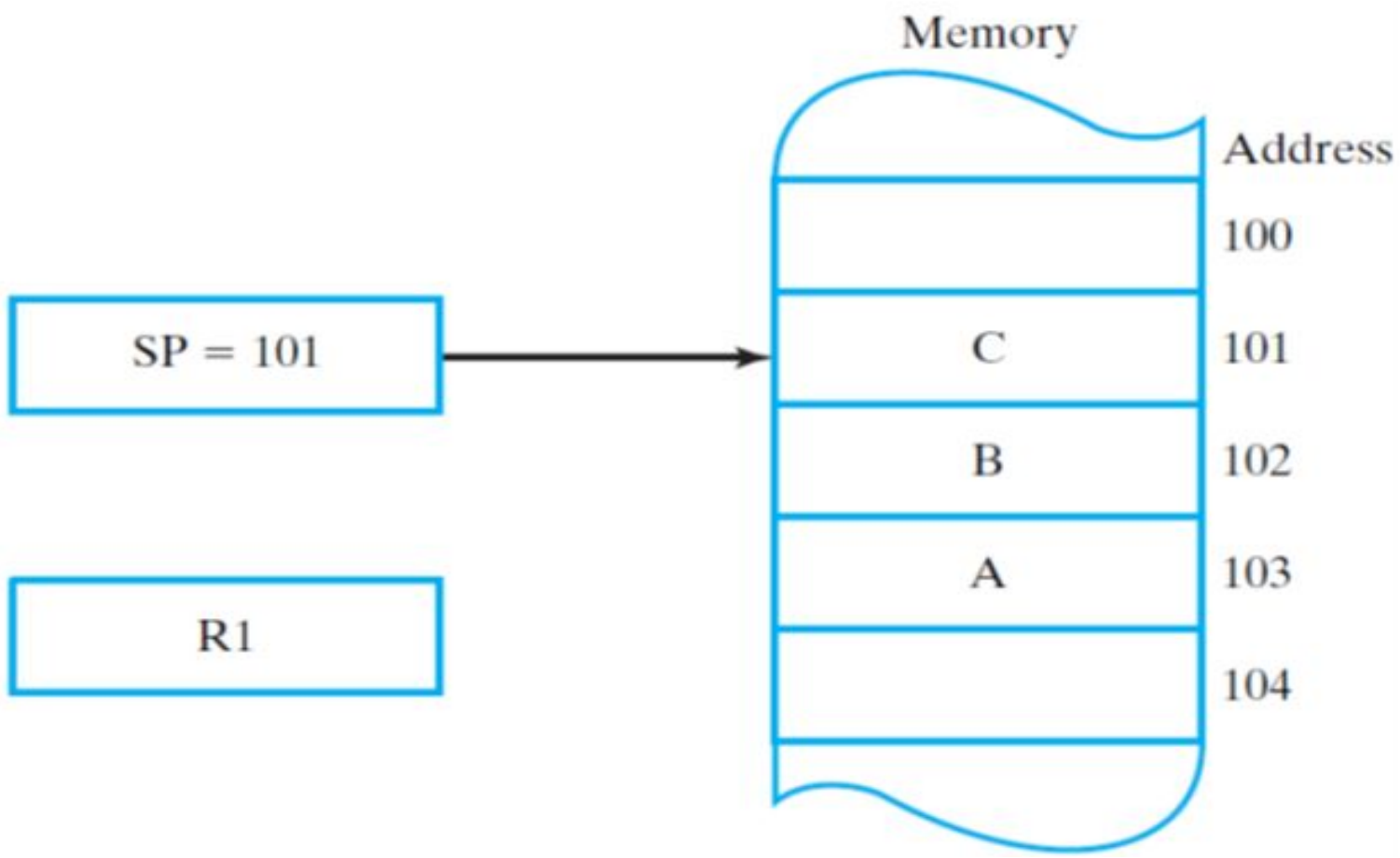
Owing to the fact that the instruction uses two types of addressing, that is, a register and a memory location, it is called a one-and-half-address instruction.

ZERO ADDRESS INSTRUCTION

- It is interesting to indicate that there exist zero-address instructions.
- These are the instructions that use stack operation.
- A stack is a data organization mechanism in which the last data item stored is the first data item retrieved (LIFO)
- Two specific operations can be performed on a stack. These are the push and the pop operations
- specific register, called the stack pointer (SP), is used to indicate the stack location that can be addressed.
- In the stack push operation, the SP value is used to indicate the location (called the top of the stack) in which the value is to be stored.

STACK

- The SP(stack Pointer) point to the location that stores the current data. This location is referred to as the top of the stack



ADDRESSING MODE

- The term addressing modes refers to the way in which operand is specified in an instruction
- The operation field of an instruction specifies the operation to be performed.
- This operation must be executed on some data stored in computer registers or memory words.
- The way the operands are chosen during program execution is dependent on the addressing mode of the instruction.
- The addressing mode specifies a rule for interpreting or modifying the address field of the instruction before the operand is actually referenced

ADDRESSING MODE

The following addressing modes exist

- Immediate Addressing Mode
- Register Addressing Mode
- Direct Addressing Mode
- Indirect Addressing Mode
- Register Indirect Addressing Mode
- Displacement Addressing Mode
- Stack Addressing Mode

➢ *We shall use the following notations*

AC=Accumulator content

M[A] = Memory address A containing the operand

R = contents of the register containing the operand

EA = actual (effective) address containing the operand

(X) = contents of memory location X or register X

IMMEDIATE MODE

According to this addressing mode, the value of the operand is (immediately) available in the instruction itself. EA=Operand

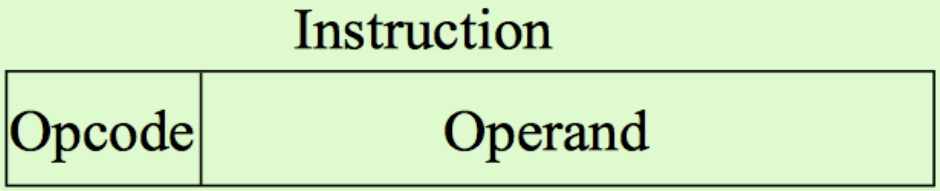
Example:

LOAD #1000, R1: load the value 1000 into register R1

ADD AC, #100: add the value 100 to the value of accumulator

Immediate values are preceded with #

Immediate Addressing Diagram



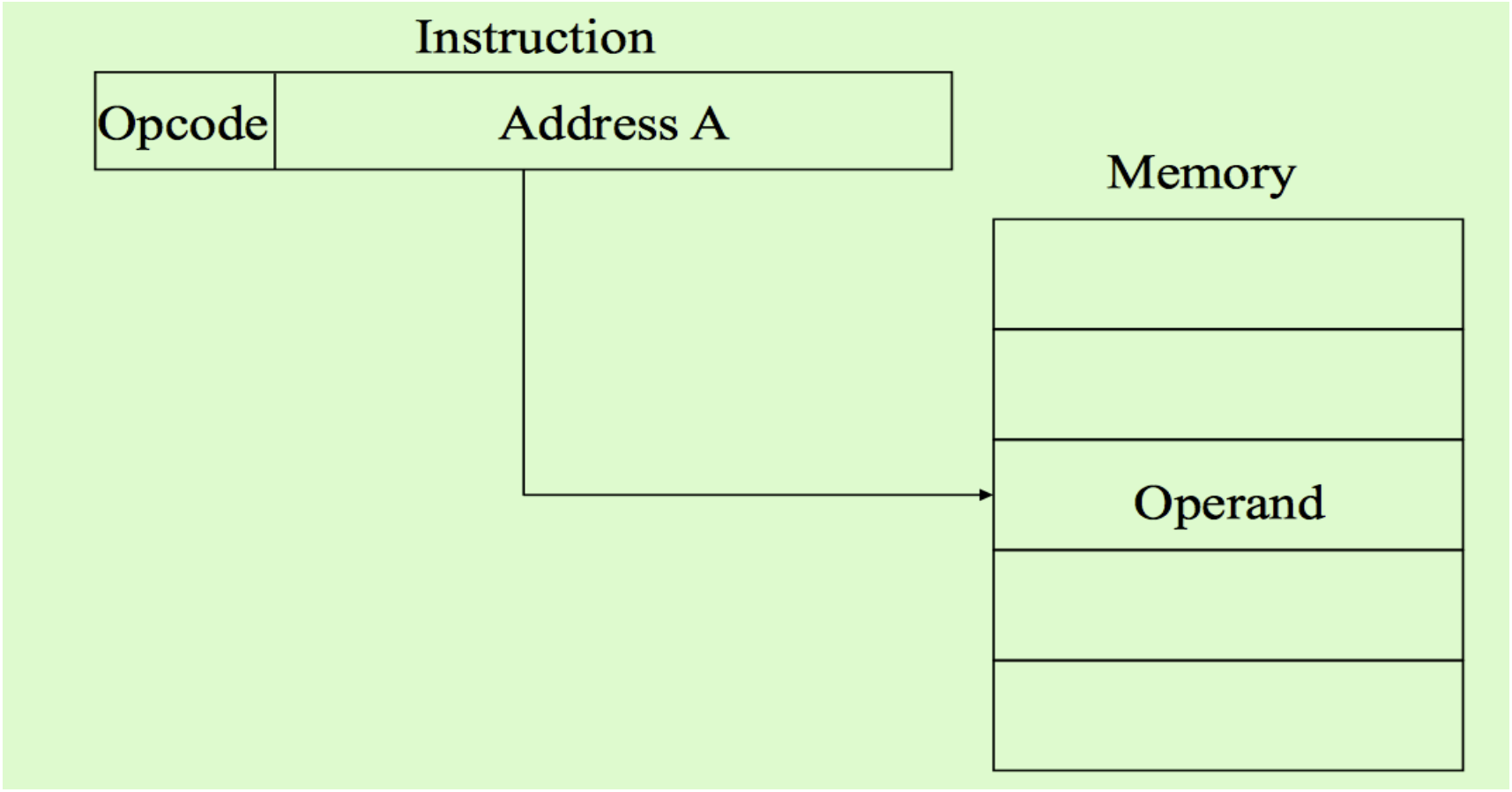
OPERAND IS PART OF THE INSTRUCTION

EA=OPERAND

DIRECT ADDRESSING MODE

In this mode, the effective address of the operand is the address field. EA=M[A]

ADD AC, M[A]: add the content of Address A to value of accumulator



EA=M[A]

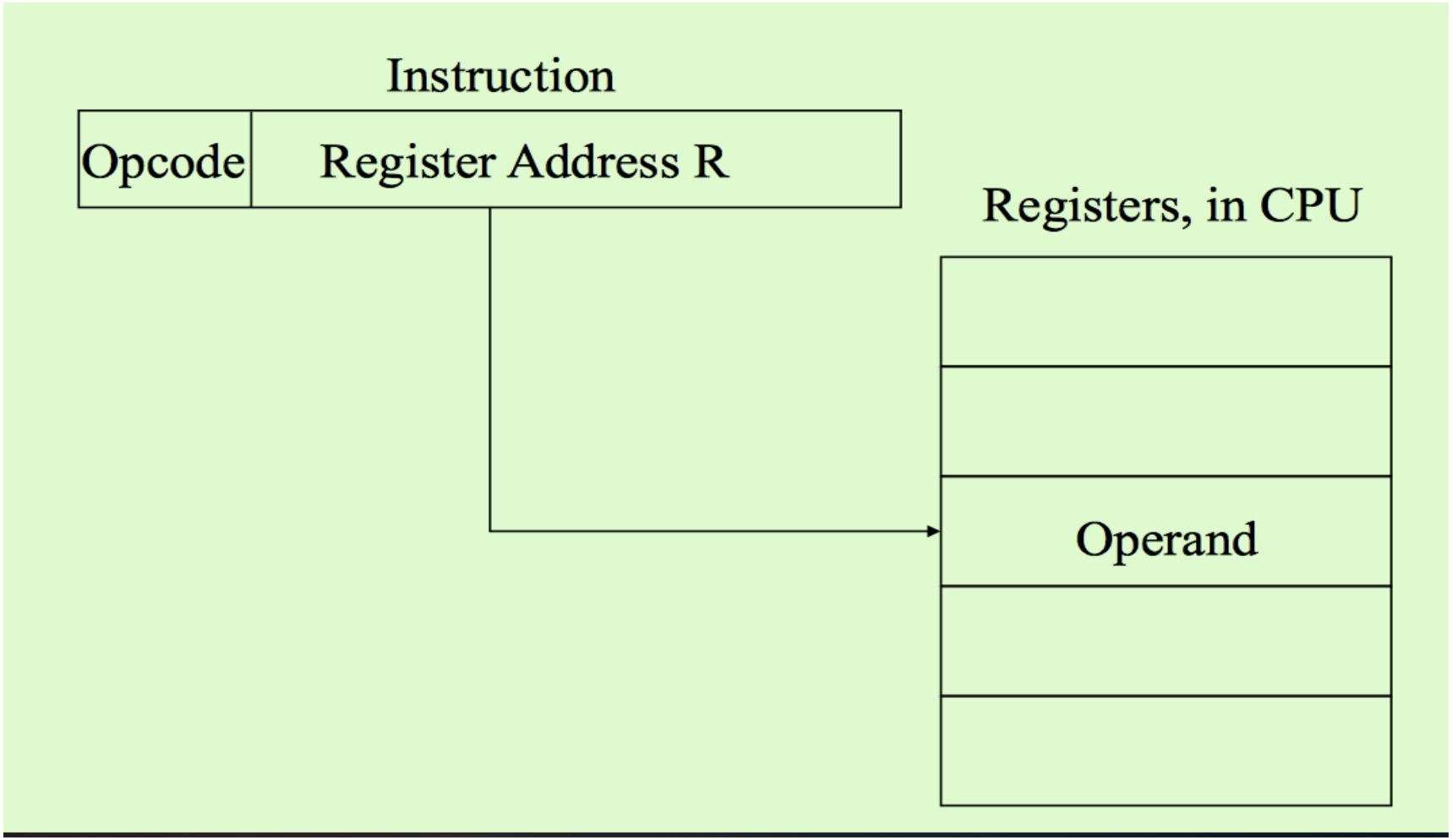
Direct addressing mode

REGISTER DIRECT ADDRESSING MODE

In this addressing mode, the source of data or operand is a Register. EA=R

The name of the register is given in the instruction where the data to be read or result is to be stored.

Example: *ADD AC,R*: add content of Register R to Accumulator



EA=R

REGISTER DIRECT

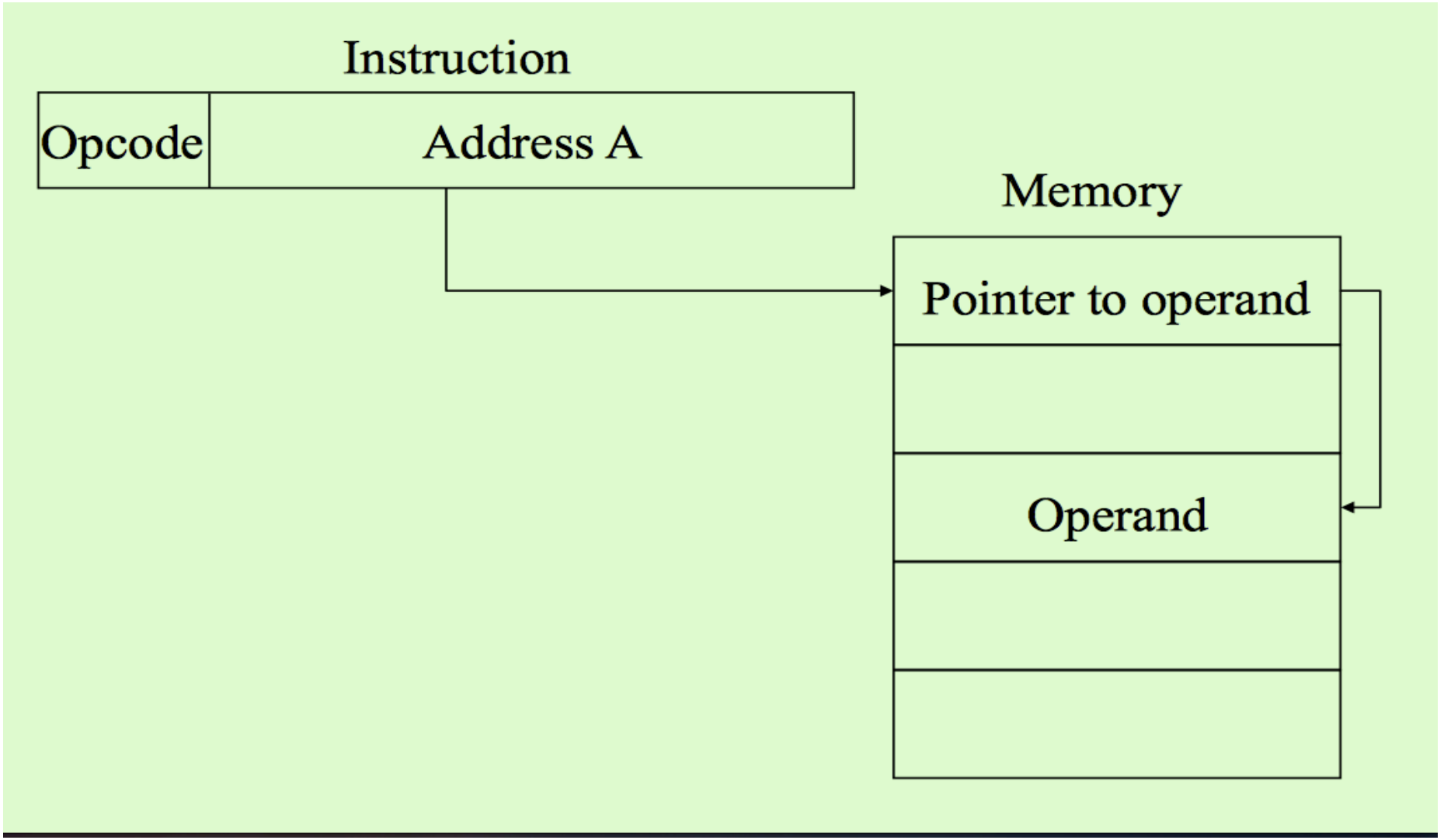
INDIRECT ADDRESSING

The address field in the instruction gives the address of the effective address where the operand is stored.

EA=(A)

Example: *ADD AC,(A):*

check the content of Address A and add the value of the Address found in Address A to AC



EA=M[M[A]]

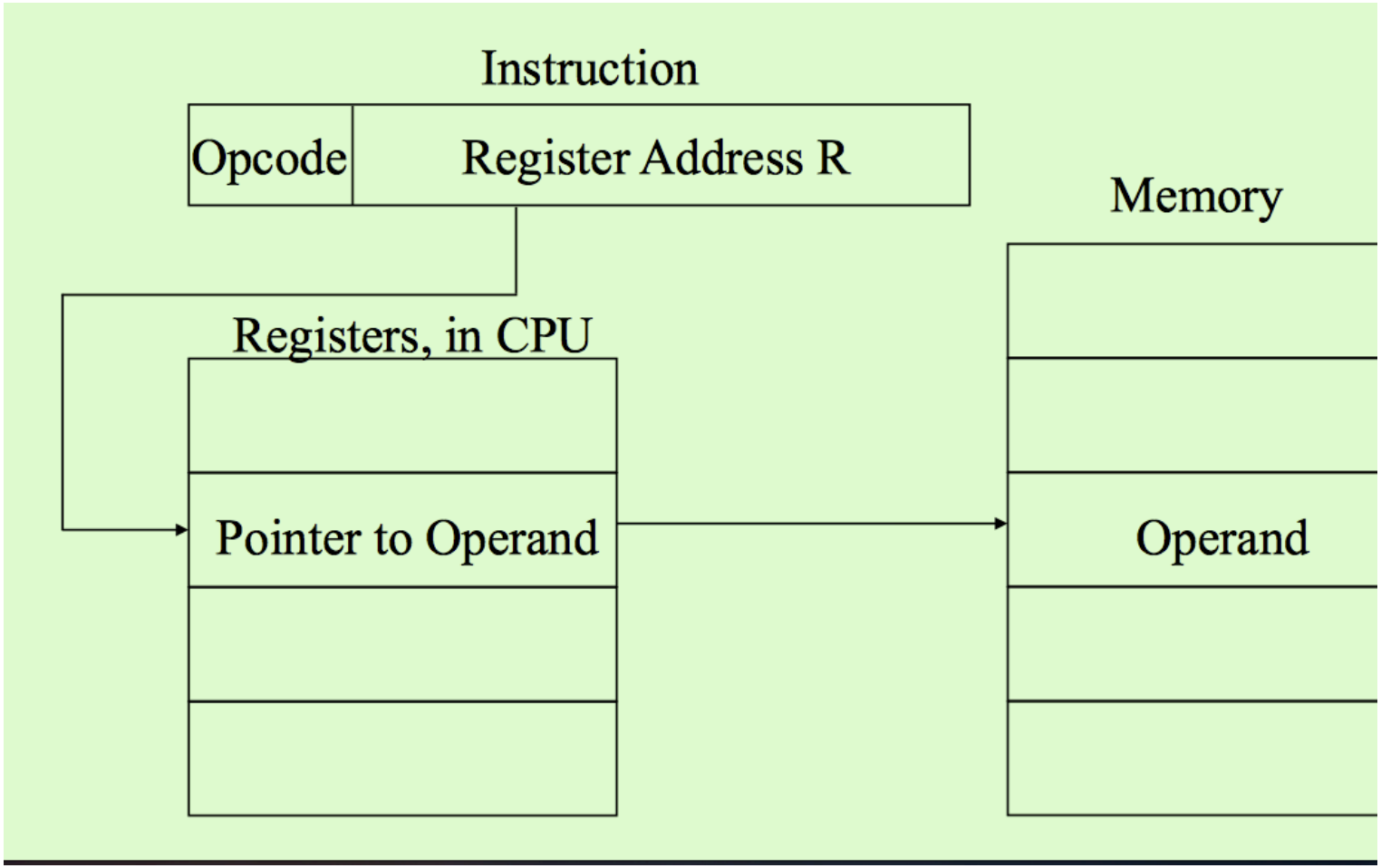
INDIRECT ADDRESSING MODE

REGISTER INDIRECT ADDRESSING

The Register field in the instruction gives the register which contains the effective address of the operand
EA=(R)

Example: *ADD AC,(R):*

check the content of Register R and add the content of the Address found in Register R to AC



EA=(R)

REGISTER INDIRECT ADDRESSING