UNIT-II:

Central Processing Unit: General Register Organization, Instruction Formats, Addressing modes, Data Transfer and Manipulation

Microprogrammed Control: Control memory, Address sequencing, micro program example, design of control unit.

CO-2: Understand the micro programming, instruction formats and addressing modes

Central Processing Unit

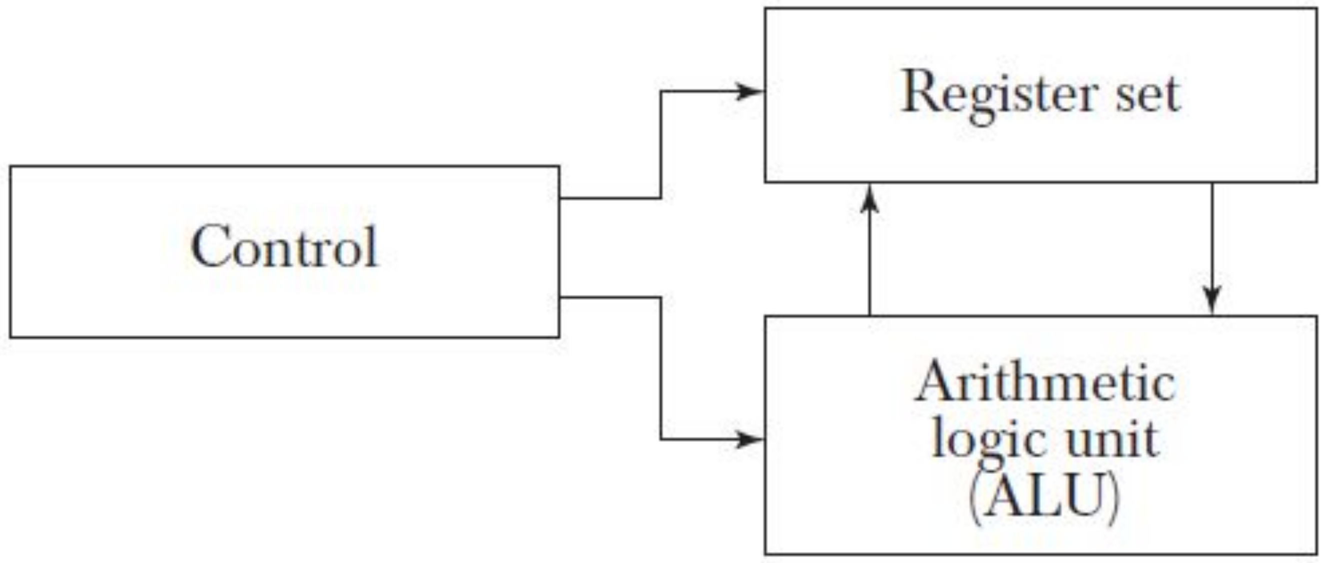
Objective:

To describe the organization and architecture of the CPU with an emphasis on the user’s view of the computer

Introduction

- The part of the computer that performs the bulk of data-processing operations is called the central processing unit and is referred to as the CPU.
- The CPU is made up of three major parts

Major components of CPU



Introduction

- The register set stores intermediate data used during the execution of the instructions.
- The arithmetic logic unit (ALU) performs the required microoperations for executing the instructions.
- The control unit supervises the transfer of information among the registers and instructs the ALU as to which operation to perform.
- The CPU performs a variety of functions dictated by the type of instructions that art Incorporated in the computer.

Computer Architecture

- Computer architecture is sometimes defined as the computer structure and behavior as seen by the programmer that uses machine language instructions.
- This includes the instruction formats, addressing modes, the instruction set, and the general organization of the CPU registers leading to two computer architectures as **reduced instruction set computer (RISC)** and **complex instruction set computer (CISC).**

Computer Architecture

- Based on memory usage for programs and data, two architectures, namely nonembedded and embedded are evolved.
- **Nonembedded computer architectures** are basically stored program computer (SPC) architectures in which programs and data reside in the same memory system.
- **Embedded architectures** are basically Harvard computer architectures in which programs and data reside in different memory systems leading to doubling the memory bandwidth.
- Example of nonembedded computers are all desktop systems such as personal computers.
- Example of embedded computers are microcontroller-based systems and digital signal processor-based (DSP) systems.

Computer designer and Computer Programmer

- One boundary where the computer designer and the computer programmer see the same machine is the part of the CPU associated with the instruction set.
- From the designer’s point of view, the computer instruction set provides the specifications for the design of the CPU.
- The design of a CPU is a task that in large part involves choosing the hardware for implementing the machine instructions.
- The user who programs the computer in machine or assembly language must be aware of the register set, the memory structure, the type of data supported by the instructions, and the function that each instruction performs.

General Register Organization

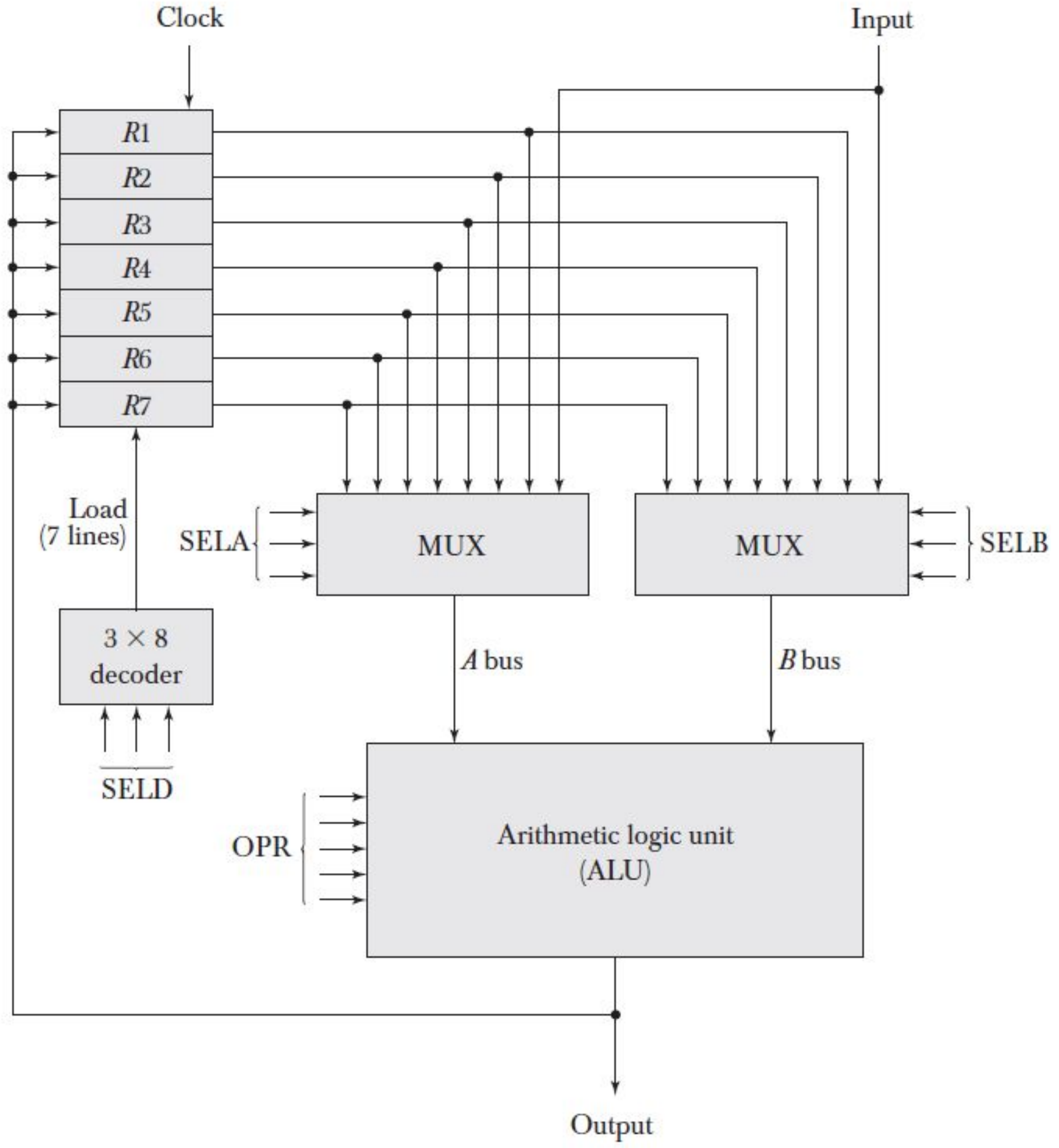
General Register Organization

- In programming memory locations are needed for storing pointers, counters, return addresses, temporary results, and partial products during multiplication.
- Having to refer to memory locations for such applications is time consuming because **memory access is the most time-consuming operation** in a computer.
- It is more convenient and more efficient to store these intermediate values in processor registers.

General Register Organization

- When a large number of registers are included in the CPU, it is most efficient to connect them through a common bus system.
- The registers communicate with each other not only for direct data transfers, but also while performing various microoperations.
- Hence it is necessary to provide a common unit that can perform all the arithmetic, logic, and shift microoperations in the processor.

Register set with common ALU-Block Diagram



Register set with common ALU

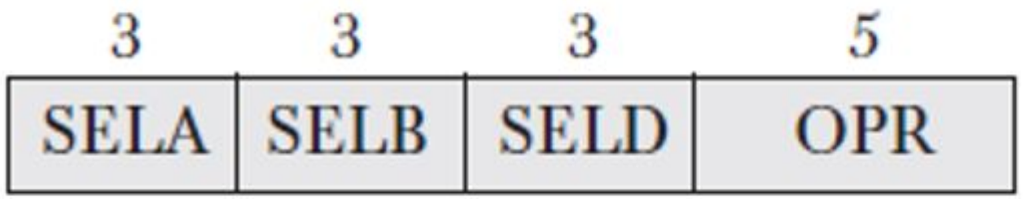
For example, to perform the operation

$$R1 \leftarrow R2 + R3$$

the control must provide binary selection variables to the following selector inputs:

1. MUX A selector (SELA): to place the content of $R2$ into bus A.
2. MUX B selector (SELB): to place the content of $R3$ into bus B.
3. ALU operation selector (OPR): to provide the arithmetic addition $A + B$.
4. Decoder destination selector (SELD): to transfer the content of the output bus into $R1$.

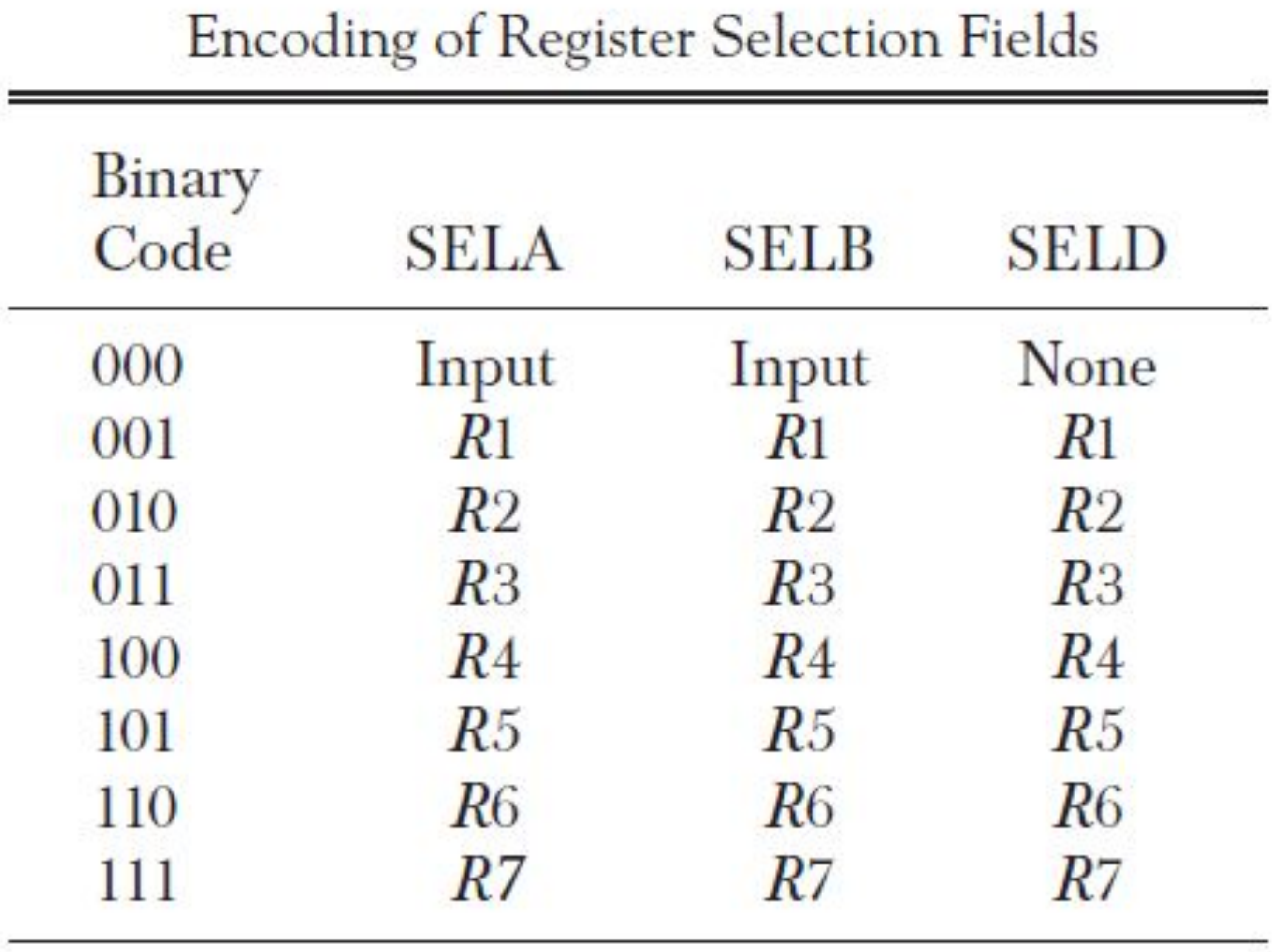
Register set with common ALU- “Control word”



3	3	3	5
SELA	SELB	SELD	OPR

The 14-bit control word when applied to the selection inputs specify a particular microoperation.

- The four control selection variables are generated in the control unit and must be available at the beginning of a clock cycle.
- The data from the two source registers propagate through the gates in the multiplexers and the ALU, to the output bus, and into the inputs of the destination register, all during the clock cycle interval.
- Then, when the next clock transition occurs, the binary information from the output bus is transferred into $R1$.
- To achieve a fast response time, the ALU is constructed with high-speed circuits.



Encoding of Register Selection Fields

Binary Code	SELA	SELB	SELD
000	Input	Input	None
001	*R*1	*R*1	*R*1
010	*R*2	*R*2	*R*2
011	*R*3	*R*3	*R*3
100	*R*4	*R*4	*R*4
101	*R*5	*R*5	*R*5
110	*R*6	*R*6	*R*6
111	*R*7	*R*7	*R*7

- The register selected by fields SELA, SELB, and SELD is the one whose decimal number is equivalent to the binary number in the code.
- When SELA or SELB is 000, the corresponding multiplexer selects the external input data.
- When SELD 000, no destination register is selected but the contents of the output bus are available in the external output.

Encoding of ALU Operations

OPR Select	Operation	Symbol
00000	Transfer *A*	TSFA
00001	Increment *A*	INCA
00010	Add $A + B$	ADD
00101	Subtract $A - B$	SUB
00110	Decrement *A*	DECA
01000	AND *A* and *B*	AND
01010	OR *A* and *B*	OR
01100	XOR *A* and *B*	XOR
01110	Complement *A*	COMA
10000	Shift right *A*	SHRA
11000	Shift left A	SHLA

Examples of Microoperations

- A control word of 14 bits is needed to specify a microoperation in the CPU.
- The control word for a given microoperation can be derived from the selection variables.

For example, the subtract microoperation given by the statement

$$R1 \leftarrow R2 - R3$$

Field:	SELA	SELB	SELD	OPR
Symbol:	R2	R3	R1	SUB
Control word:	010	011	001	00101

3	3	3	5
SELA	SELB	SELD	OPR

Examples of Microoperations for the CPU

	Symbolic Designation				
Microoperation	SELA	SELB	SELD	OPR	Control Word
$R1 \leftarrow R2 - R3$	R2	R3	R1	SUB	010 011 001 00101
$R4 \leftarrow R4 \vee R5$	R4	R5	R4	OR	100 101 100 01010
$R6 \leftarrow R6 + 1$	R6	–	R6	INCA	110 000 110 00001
$R7 \leftarrow R1$	R1	–	R7	TSFA	001 000 111 00000
Output $\leftarrow R2$	R2	–	None	TSFA	010 000 000 00000
Output $\leftarrow$ Input	Input	–	None	TSFA	000 000 000 00000
$R4 \leftarrow$ sh1 $R4$	R4	–	R4	SHLA	100 000 100 11000
$R5 \leftarrow 0$	R5	R5	R5	XOR	101 101 101 01100

Microprogrammed Control

- The most **efficient way to generate control words with a large number of bits** is to store them in a memory unit.
- A memory unit that stores control words is referred to as a **control memory.**
- **By reading consecutive control words from memory, it is possible to initiate the desired sequence of microoperations for the CPU.**
- This type of control is referred to as **microprogrammed control.**

Stack Organization

LIFO

- A useful feature that is included in the CPU of most computers is a stack or last-in, first-out (LIFO) list.
- A stack is a storage device that stores information in such a manner that the item stored last is the first item retrieved.

Stack Pointer (SP)

- The stack in digital computers is essentially a memory unit with an address register that can count only (after an initial value is loaded into it).
- The register that holds the address for the stack is called a stack pointer (SP) because its value always points at the top item in the stack.

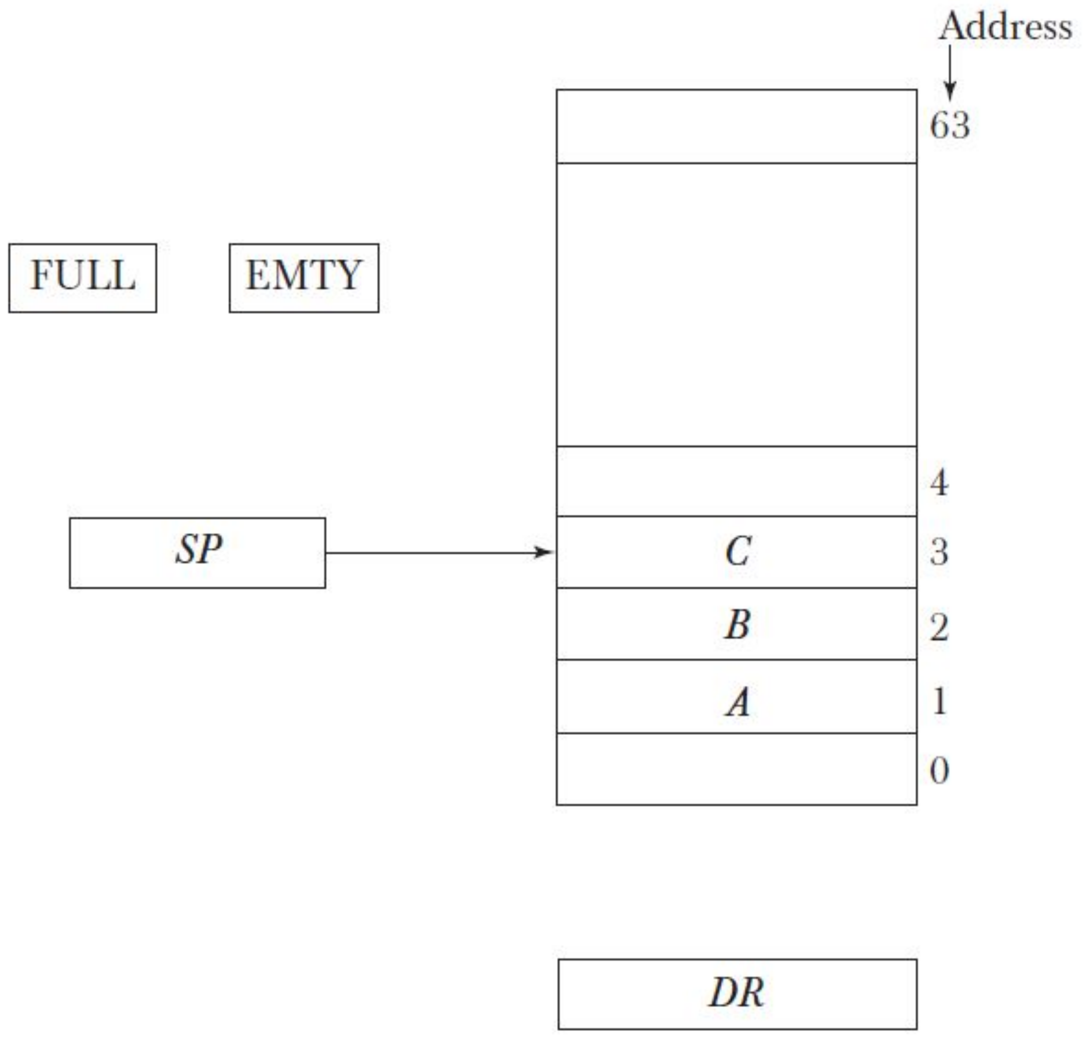
Two operations of a stack

- The two operations of a stack are the insertion and deletion of items.
- The operation of insertion is called push (or push-down) because it can be thought of as the result of pushing a new item on top.
- The operation of deletion is called pop (or pop-up) because it can be thought of as the result of removing one item so that the stack pops up.

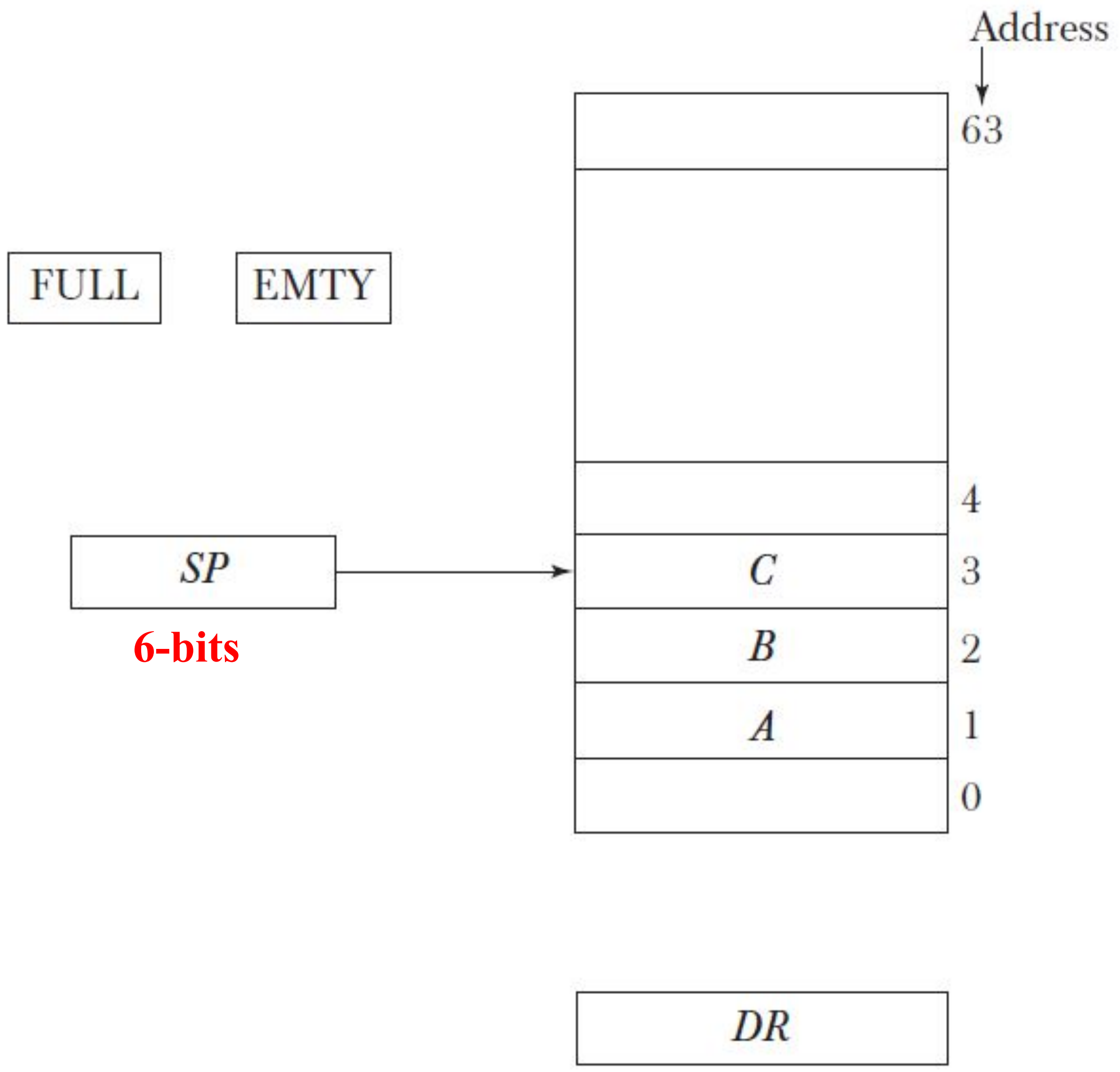
Register Stack

- A stack can be organized as a collection of a finite number of registers.

Block diagram of a 64-word stack



- The stack pointer register SP contains a binary number whose value is equal to the address of the word that is currently on top of the stack.
- To remove the top item, the stack is **popped** by **reading the memory word at top of stack and decrementing the content of SP.**
- To insert a new item, the stack is **pushed** by **incrementing SP and writing a word in the next-higher location in the stack.**



When 63 is incremented by 1, the result is 0 since 111111 + 1= 1000000 in binary, but *SP* can accommodate only the six least significant bits.
Similarly, when 000000 is decremented by 1, the result is 111111.
The one-bit register *FULL* is set to 1 when the stack is full, and the one-bit register *EMTY* is
set to 1 when the stack is empty of items.

DR is the data register that holds the binary data to be written into or

PUSH

$SP \leftarrow SP + 1$	Increment stack pointer
$M[SP] \leftarrow DR$	Write item on top of the stack
If $(SP = 0)$ then $(FULL \leftarrow 1)$	Check if stack is full
$EMTY \leftarrow 0$	Mark the stack not empty

POP

$DR \leftarrow M[SP]$	Read item from the top of stack
$SP \leftarrow SP - 1$	Decrement stack pointer
If $(SP = 0)$ then $(EMTY \leftarrow 1)$	Check if stack is empty
$FULL \leftarrow 0$	Mark the stack not full

Note

- Note also that an erroneous operation will result if the stack is pushed when FULL = 1 or popped when EMTY = 1.

Advantage of Stack

- The advantage of a memory stack is that the CPU can refer to it without having to specify an address, since the address is always available and automatically updated in the stack pointer.

Instruction formats

Instruction formats

- A computer will usually have a variety of instruction code formats.
- It is the function of the control unit within the CPU to interpret each instruction code and provide the necessary control functions needed to process the instruction.

Instruction formats

- 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 found in instruction formats are:

1. An **operation code** field that specifies the operation to be performed.
2. An **address field** that designates a memory address or a processor register.
3. A **mode field** that specifies the way the operand or the effective address is determined.

Note:

Other special fields are sometimes employed under certain circumstances, as for example a field that gives the number of shifts in a shift-type instruction.

In this section we are concerned with the address field of an instruction format and consider the effect of including multiple address fields in an instruction.

Memory Address and Register Address

- Operations specified by computer instructions are executed on some data stored in memory or processor registers.
- Operands residing in memory are specified by their **memory address.**
- Operands residing in processor registers are specified with a **register address.**
- A register address is a binary number of k bits that defines one of 2^K registers in the CPU.
- Thus a CPU with 16 processor registers R0 through R15 will have a register address field of four bits.
- The binary number 0101, for example, will designate register R5.

Internal organization of registers

- Computers may have instructions of several different lengths containing varying number of addresses.
- The number of address fields in the instruction format of a computer depends on the internal organization of its registers.
- Most computers fall into one of three types of CPU organizations:
    1. Single accumulator organization.
    2. General register organization.
    3. Stack organization.

Single accumulator organization

- All operations are performed with an implied accumulator register.
- The instruction format in this type of computer uses one address field.
- For example, the instruction that specifies an arithmetic addition is defined by an assembly language instruction as

$$\boxed{\text{ADD X}} \qquad AC \leftarrow AC + M[X]$$

where X is the address of the operand

AC is the accumulator register and

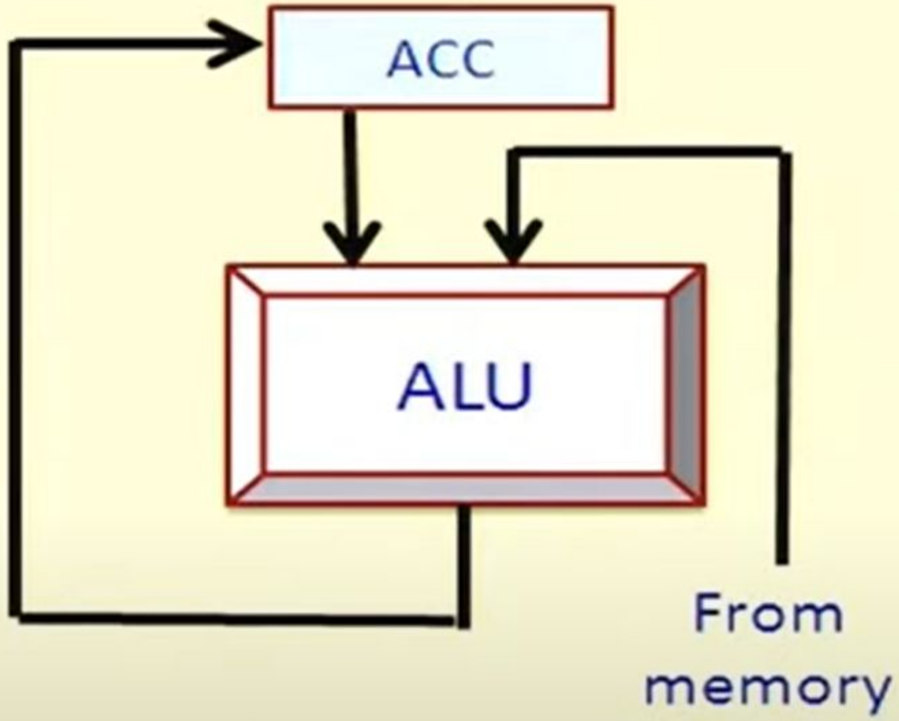
$M[X]$ symbolizes the memory word located at address X.

Z = X + Y

- **<u>Accumulator based machine:</u>**

```
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.



General register organization

```
ADD    R1, R2, R3
```
$R1 \leftarrow R2 + R3$

The number of address fields in the instruction can be reduced from three to two if the destination register is the same as one of the source registers.
Thus the instruction

```
ADD    R1, R2
```
$R1 \leftarrow R1 + R2$

General register organization

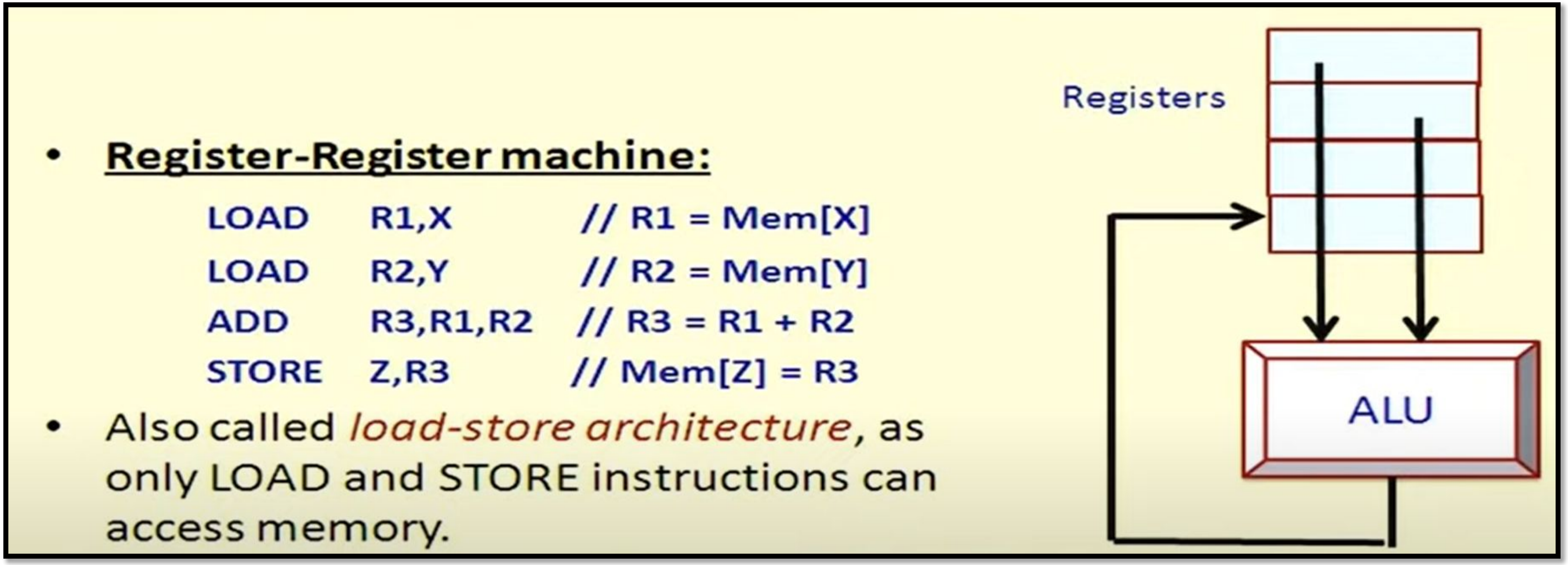
- General register-type computers employ two or three address fields in their instruction format.
- Each address field may specify a processor register or a memory word.

- An instruction symbolized by

```
ADD     R1,  X
```

$$R1 \leftarrow R1 + M[X]$$

$Z = X + Y$



- **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
```

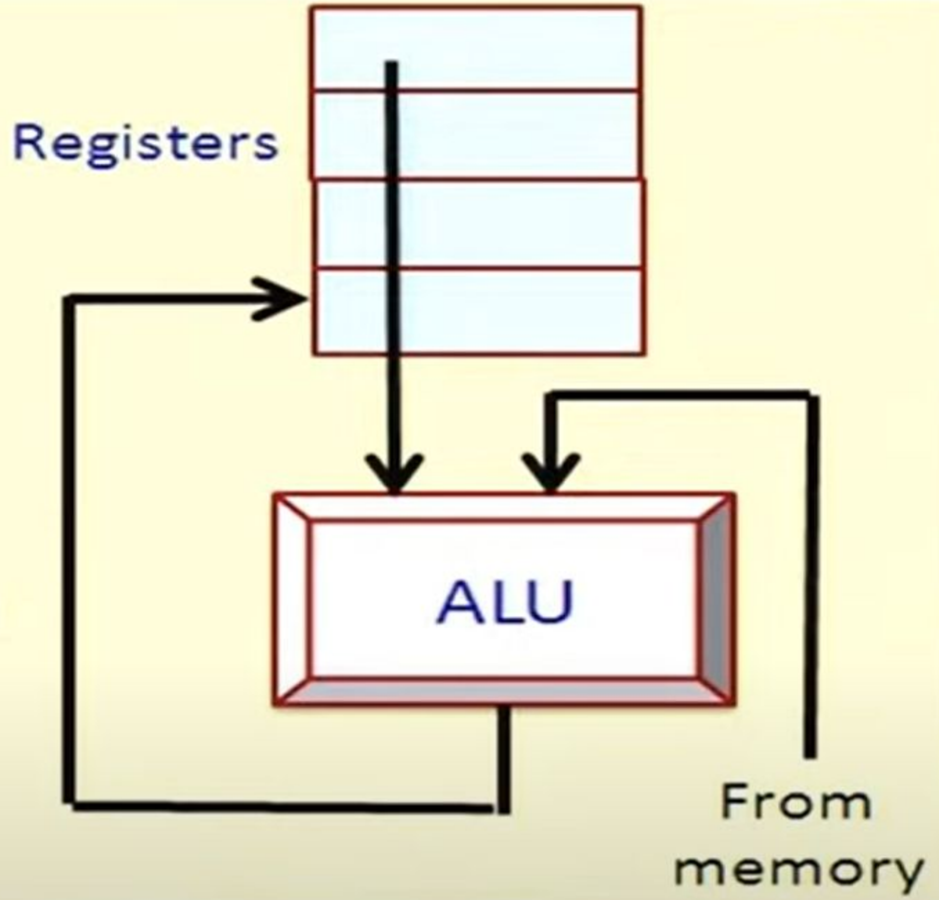
- Also called *load-store architecture*, as only LOAD and STORE instructions can access memory.

$Z = X + Y$

- **Register-Memory machine:**

```
LOAD    R2,X     // R2 = Mem[X]
ADD     R2,Y     // R2 = R2 + Mem[Y]
STORE   Z,R2     // Mem[Z] = R2
```

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



Stack organization

- Computers with stack organization would have PUSH and POP instructions which require an address field.
- Thus the instruction

```
PUSH      X
```

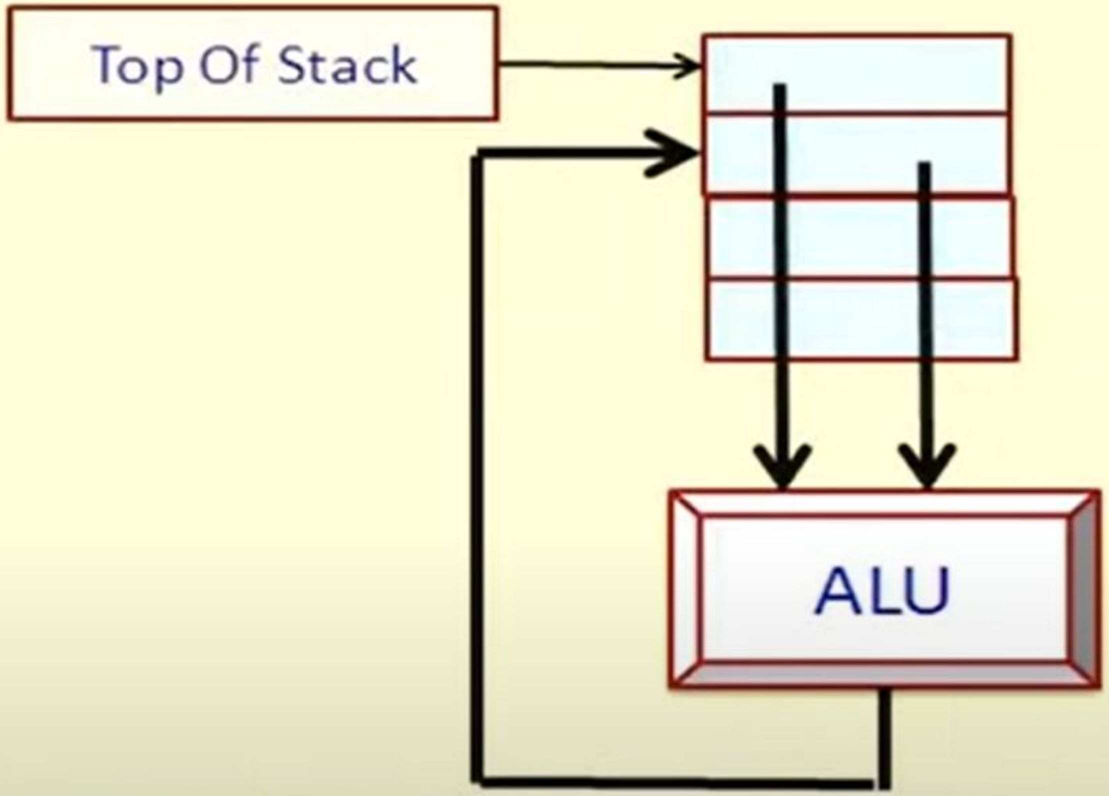
will push the word at address *X* to the top of the stack.
The stack pointer is updated automatically.

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.



Stack organization

- Operation-type instructions do not need an address field in stack-organized computers.
- This is because the operation is performed on the two items that are on top of the stack.
- The instruction

ADD

in a stack computer consists of an operation code only with no address field.
This operation has the effect of popping the two top numbers from the stack, adding the numbers, and pushing the sum into the stack.
There is no need to specify operands with an address field since all operands are implied to be in the stack.

Note

- Most computers fall into one of the three types of organizations that have just been described.
- Some computers combine features from more than one organizational structure.
- **For example,** the Intel 8080 microprocessor has seven CPU registers, one of which is an accumulator register.
- As a consequence, the processor has some of the characteristics of a general register type and some of the characteristics of an accumulator type.

Note

- All arithmetic and logic instructions, as well as the load and store instructions, use the accumulator register, so these instructions have only one address field.
- On the other hand, instructions that transfer data among the seven processor registers have a format that contains two register address fields.
- Moreover, the Intel 8080 processor has a stack pointer and instructions to push and pop from a memory stack.
- The processor, however, does not have the zero-address-type instructions which are characteristic of a stack-organized CPU.

To illustrate the **influence of the number of addresses on computer programs**, we will evaluate the arithmetic statement using zero, one, two, or three address instructions.

$$X = (A + B) * (C + D)$$

We will use the symbols
ADD, SUB, MUL, and DIV for the four arithmetic operations;
MOV for the transfer-type operation; and
LOAD and STORE for transfers to and from memory and *AC* register.
We will assume that the operands are in memory addresses *A*, *B*, *C*, and *D*, and the result must be stored in memory at address *X*

Three-Address Instructions

Computers with three-address instruction formats can use each address field to specify either a processor register or a memory operand.

$$X = (A + B) * (C + D)$$

```
ADD     R1,  A,  B     R1 ← M[A]+M[B]
ADD     R2,  C,  D     R2 ← M[C]+M[D]
MUL     X,  R1,  R2    M[X]  ←  R1*R2
```

It is assumed that the computer has two processor registers, R1 and R2.
The symbol M[A] denotes the operand at memory address symbolized by A.

Advantage and Disadvantage

- The advantage of the three-address format is that it results in short programs when evaluating arithmetic expressions.
- The disadvantage is that the binary-coded instructions require too many bits to specify three addresses.
- An example of a commercial computer that uses three-address instructions is the **Cyber 170.**
- The instruction formats in the Cyber computer are restricted to either three register address fields or two register address fields and one memory address field.

Two-Address Instructions

- Two-address instructions are the most common in commercial computers.
- Here again each address field can specify either a processor register or a memory word.

$$X = (A + B) * (C + D)$$

```
MOV    R1, A     R1 ← M[A]
ADD    R1, B     R1 ← R1 + M[B]
MOV    R2, C     R2 ← M[C]
ADD    R2, D     R2 ← R2 + M[D]
MUL    R1, R2    R1 ← R1*R2
MOV    X, R1     M[X] ← R1
```

One-Address Instructions

- One-address instructions use an implied accumulator (AC) register for all data manipulation.
- For multiplication and division there is a need for a second register.
- However, here we will neglect the second register and assume that the AC contains the result of all operations.

$$X = (A + B) * (C + D)$$

```
LOAD    A     AC ← M[A]
ADD     B     AC ← AC + M[B]
STORE   T     M[T] ← AC
LOAD    C     AC ← M[C]
ADD     D     AC ← AC + M[D]
MUL     T     AC ← AC * M[T]
STORE   X     M[X] ← AC
```

All operations are done between the AC register and a memory operand.
T is the address of a temporary memory location required for storing the intermediate result.

Zero-Address Instructions

- A stack-organized computer does not use an address field for the instructions ADD and MUL.
- The PUSH and POP instructions, however, need an address field to specify the operand that communicates with the stack.

$$X = (A + B) * (C + D)$$

- To evaluate arithmetic expressions in a stack computer, it is necessary to convert the expression into reverse Polish notation.
- The name “zero-address” is given to this type of computer because of the absence of an address field in the computational instructions.

```
PUSH    A       TOS ← A
PUSH    B       TOS ← B
ADD             TOS ← (A + B)
PUSH    C       TOS ← C
PUSH    D       TOS ← D
ADD             TOS ← (C + D)
MUL             TOS ← (C + D) * (A + B)
POP     X       M[X] ← TOS
```

TOS stands for top of stack

RISC Instructions

- The instruction set of a typical RISC (reduced instruction set computer) processor is restricted to the use of load and store instructions when communicating between memory and CPU.
- All other instructions are executed within the registers of the CPU without referring to memory.
- A program for a RISC-type CPU consists of LOAD and STORE instructions that have one memory and one register address, and computational-type instructions that have three addresses with all three specifying processor registers.

RISC Instructions

$$X = (A + B) * (C + D)$$

```
LOAD     R1, A          R1 ← M[A]
LOAD     R2, B          R2 ← M[B]
LOAD     R3, C          R3 ← M[C]
LOAD     R4, D          R4 ← M[D]
ADD      R1, R1, R2     R1 ← R1 + R2
ADD      R3, R3, R2     R3 ← R3 + R4
MUL      R1, R1, R3     R1 ← R1 * R3
STORE    X, R1          M[X] ← R1
```

The load instructions transfer the operands from memory to CPU registers.
The add and multiply operations are executed with data in the registers without accessing memory.
The result of the computations is then stored in memory with a store instruction.

Addressing Modes

Introduction

- 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.
- Computers use addressing mode techniques for the purpose of accommodating one or both of the following provisions:

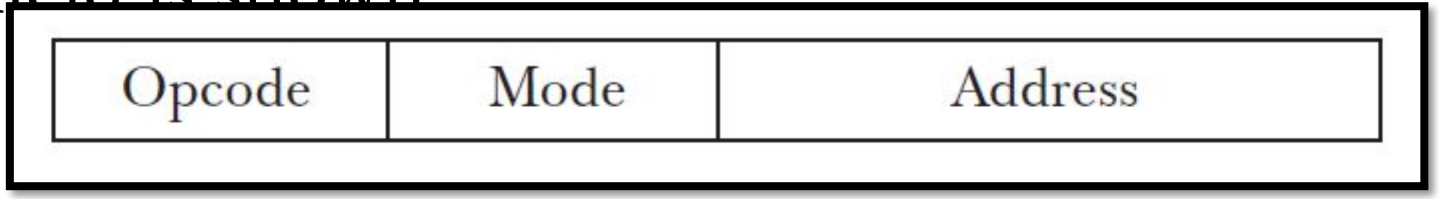
1. To give programming versatility to the user by providing such facilities as pointers to memory, counters for loop control, indexing of data, and program relocation.

2. To reduce the number of bits in the addressing field of the instruction.

The availability of the addressing modes gives the experienced assembly language programmer flexibility for writing programs that are more efficient with respect to the number of instructions and execution time.

Instruction format with mode field

- An example of an instruction format with a distinct addressing mode field is shown



- The operation code specifies the operation to be performed.
- The mode field is used to locate the operands needed for the operation.
- There may or may not be an address field in the instruction.
- If there is an address field, it may designate a memory address or a processor register.

(REFER NOTES)

Effective Address

- The address field of an instruction is used by the control unit in the CPU to obtain the operand from memory.
- Sometimes the value given in the address field is the address of the operand, but sometimes it is just an address from which the address of the operand is calculated.
- To differentiate among the various addressing modes, it is necessary to distinguish between the address part of the instruction and the effective address used by the control when executing the instruction.
- **The *effective address* is defined to be the memory address obtained from the computation dictated by the given addressing mode.**
- The effective address is the address of the operand in a computational-type instruction. It is the address where control branches in response to a branch-type instruction.

Implied Mode

- In this mode the operands are specified implicitly in the definition of the instruction.
- For example, the instruction “complement accumulator” is an implied-mode instruction because the operand in the accumulator register is implied in the definition of the instruction.
- In fact, all register reference instructions that use an accumulator are implied-mode instructions.
- Zero-address instructions in a stack-organized computer are implied-mode instructions since the operands are implied to be on top of the stack.

Immediate Mode

- In this mode the operand is specified in the instruction itself.
- In other words, an immediate-mode instruction has an operand field rather than an address field.
- The operand field contains the actual operand to be used in conjunction with the operation specified in the instruction.
- Immediate-mode instructions are useful for initializing registers to a constant value.

Register Mode

- In this mode the operands are in registers that reside within the CPU.
- The particular register is selected from a register field in the instruction.
- A *k*-bit field can specify any one of 2^k registers.

Register Indirect Mode

- In this mode the instruction specifies a register in the CPU whose contents give the address of the operand in memory.
- In other words, the selected register contains the address of the operand rather than the operand itself.
- Before using a register indirect mode instruction, the programmer must ensure that the memory address of the operand is placed in the processor register with a previous instruction.
- A reference to the register is then equivalent to specifying a memory address.
- The advantage of a register indirect mode instruction is that the address field of the instruction uses fewer bits to select a register than would have been required to specify a memory address directly.

Autoincrement or Autodecrement Mode

- This is similar to the register indirect mode except that the register is incremented or decremented after (or before) its value is used to access memory.
- When the address stored in the register refers to a table of data in memory, it is necessary to increment or decrement the register after every access to the table.
- This can be achieved by using the increment or decrement instruction.
- However, because it is such a common requirement, some computers incorporate a special mode that automatically increments or decrements the content of the register after data access.

Direct Address Mode

- In this mode the effective address is equal to the address part of the instruction.
- The operand resides in memory and its address is given directly by the address field of the instruction.
- In a branch-type instruction the address field specifies the actual branch address.

Indirect Address Mode

- In this mode the address field of the instruction gives the address where the effective address is stored in memory.
- Control fetches the instruction from memory and uses its address part to access memory again to read the effective address.

NOTE

- A few addressing modes require that the address field of the instruction be added to the content of a specific register in the CPU.
- The effective address in these modes is obtained from the following computation:
- effective address =address part of instruction +content of CPU register
- The CPU register used in the computation may be the program counter, an index register, or a base register.
- In either case we have a different addressing mode which is used for a different application.

Relative Address Mode

- In this mode the content of the program counter is added to the address part of the instruction in order to obtain the effective address.
- The address part of the instruction is usually a signed number (in 2's complement representation) which can be either positive or negative.
- When this number is added to the content of the program counter, the result produces an effective address whose position in memory is relative to the address of the next instruction.
- To clarify with an example, assume that the program counter contains the number 825 and the address part of the instruction contains the number 24.
- The instruction at location 825 is read from memory during the fetch phase and the program counter is then incremented by one to 826.
- The effective address computation for the relative address mode is 826 + 24 =850.
- This is 24 memory locations forward from the address of the next instruction.
- Relative addressing is often used with branch-type instructions when the branch address is in the area surrounding the instruction word itself.

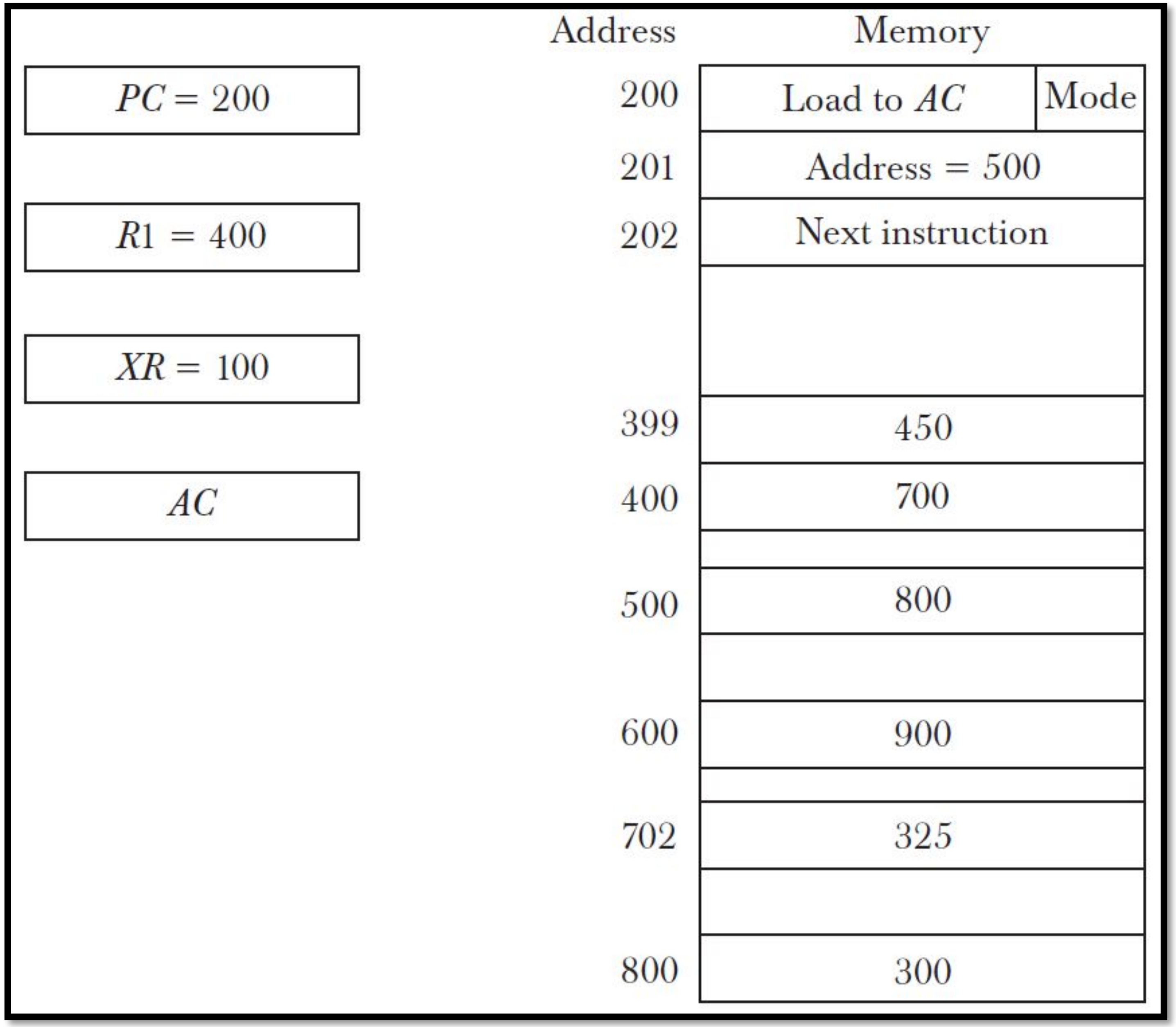
Indexed Addressing Mode

- In this mode the content of an index register is added to the address part of the instruction to obtain the effective address.
- The index register is a special CPU register that contains an index value.
- The address field of the instruction defines the beginning address of a data array in memory.
- Each operand in the array is stored in memory relative to the beginning address.
- The distance between the beginning address and the address of the operand is the index value stored in the index register.
- Any operand in the array can be accessed with the same instruction provided that the index register contains the correct index value.
- The index register can be incremented to facilitate access to consecutive operands.
- Note that if an index-type instruction does not include an address field in its format, the instruction converts to the register indirect mode of operation.
- Some computers dedicate one CPU register to function solely as an index register.
- This register is involved implicitly when, the index-mode instruction is used.
- In computers with many processor registers, any one of the CPU registers can contain the index number.

Base Register Addressing Mode

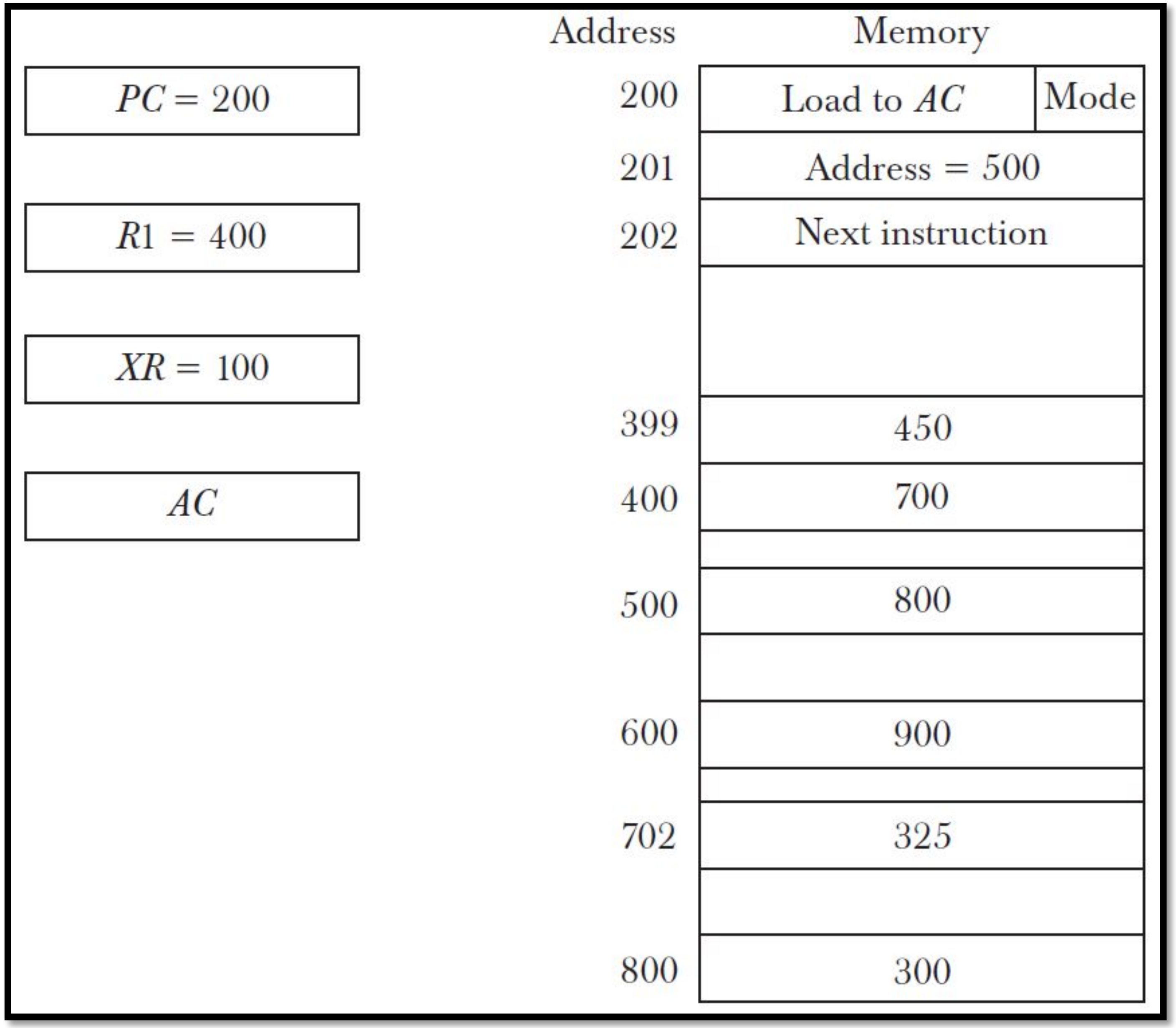
- In this mode the content of a base register is added to the address part of the instruction to obtain the effective address.
- This is similar to the indexed addressing mode except that the register is now called a base register instead of an index register.
- The difference between the two modes is in the way they are used rather than in the way that they are computed.
- An index register is assumed to hold an index number that is relative to the address part of the instruction.
- A base register is assumed to hold a base address and the address field of the instruction gives a displacement relative to this base address.
- The base register addressing mode is used in computers to facilitate the relocation of programs in memory.
- When programs and data are moved from one segment of memory to another, as required in multiprogramming systems, the address values of instructions must reflect this change of position.
- With a base register, the displacement values of instructions do not have to change.
- Only the value of the base register requires updating to reflect the beginning of a new memory segment.

Numerical example for addressing modes



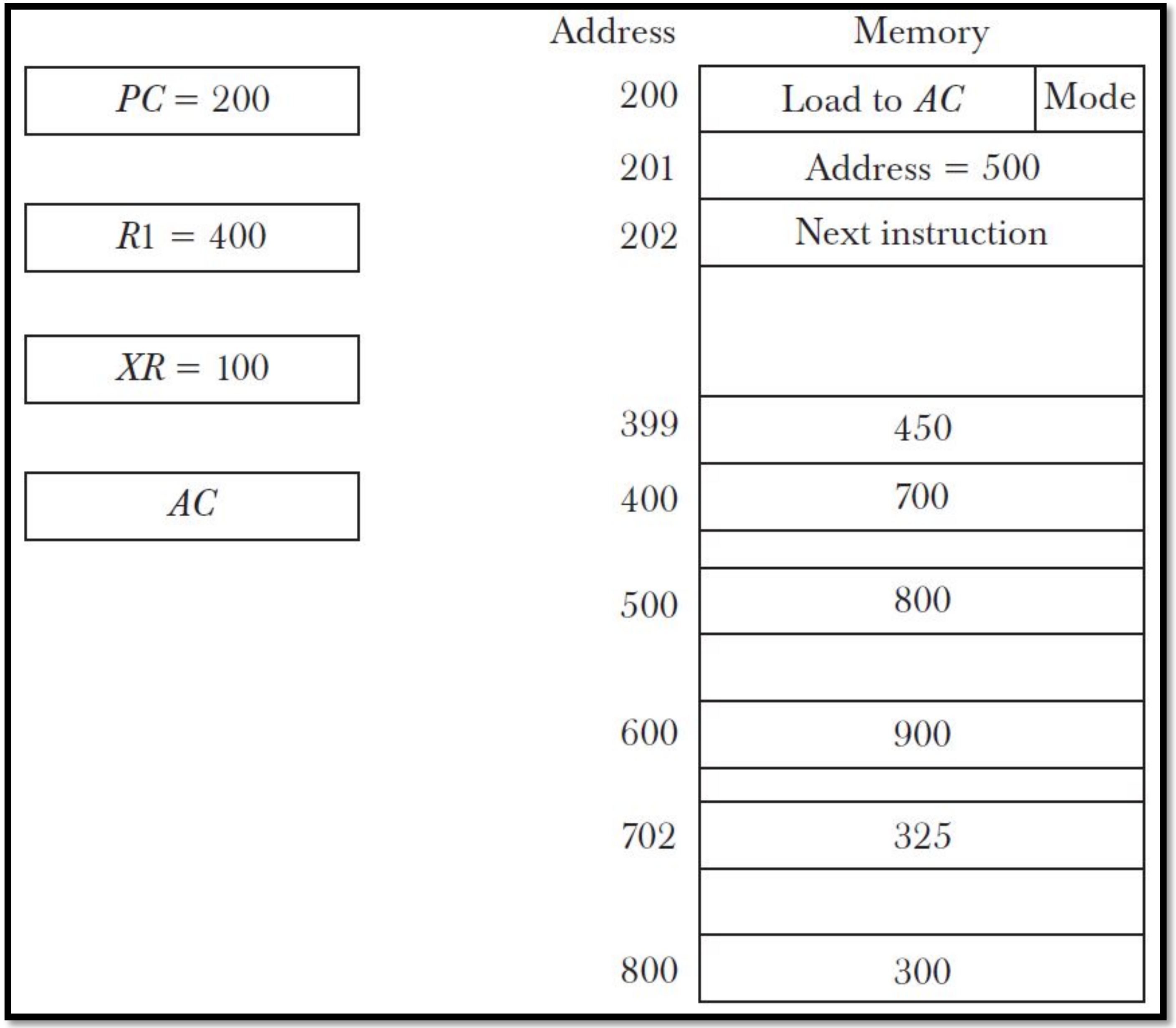
(REFER NOTES)

Direct Addressing Mode



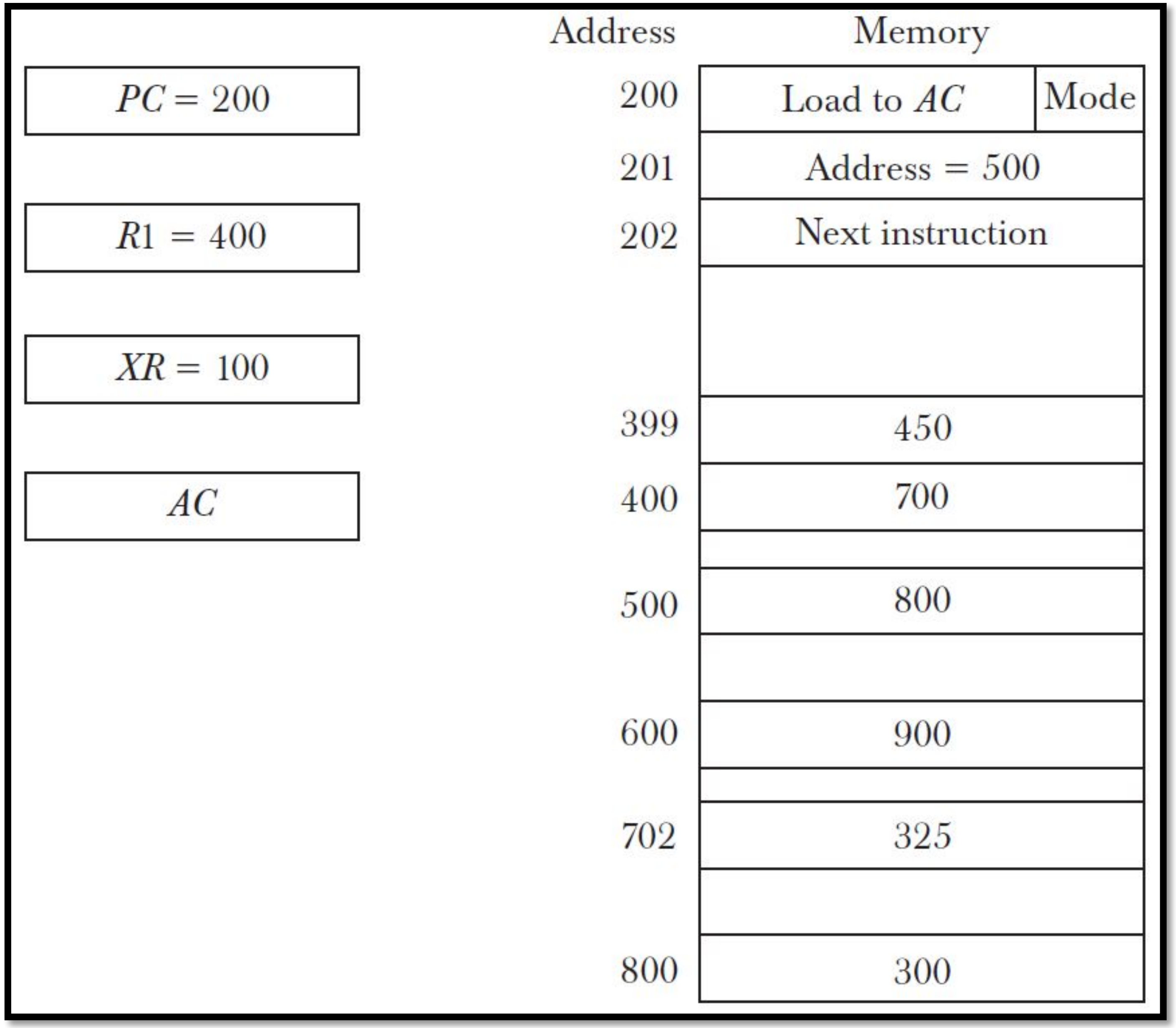
In the direct address mode, the effective address is the address part of the instruction 500 and the operand to be loaded into *AC* is 800.

Immediate Addressing Mode



In the immediate mode the second word of the instruction is taken as the operand rather than an address, so 500 is loaded into AC.
(The effective address in this case is 201.)

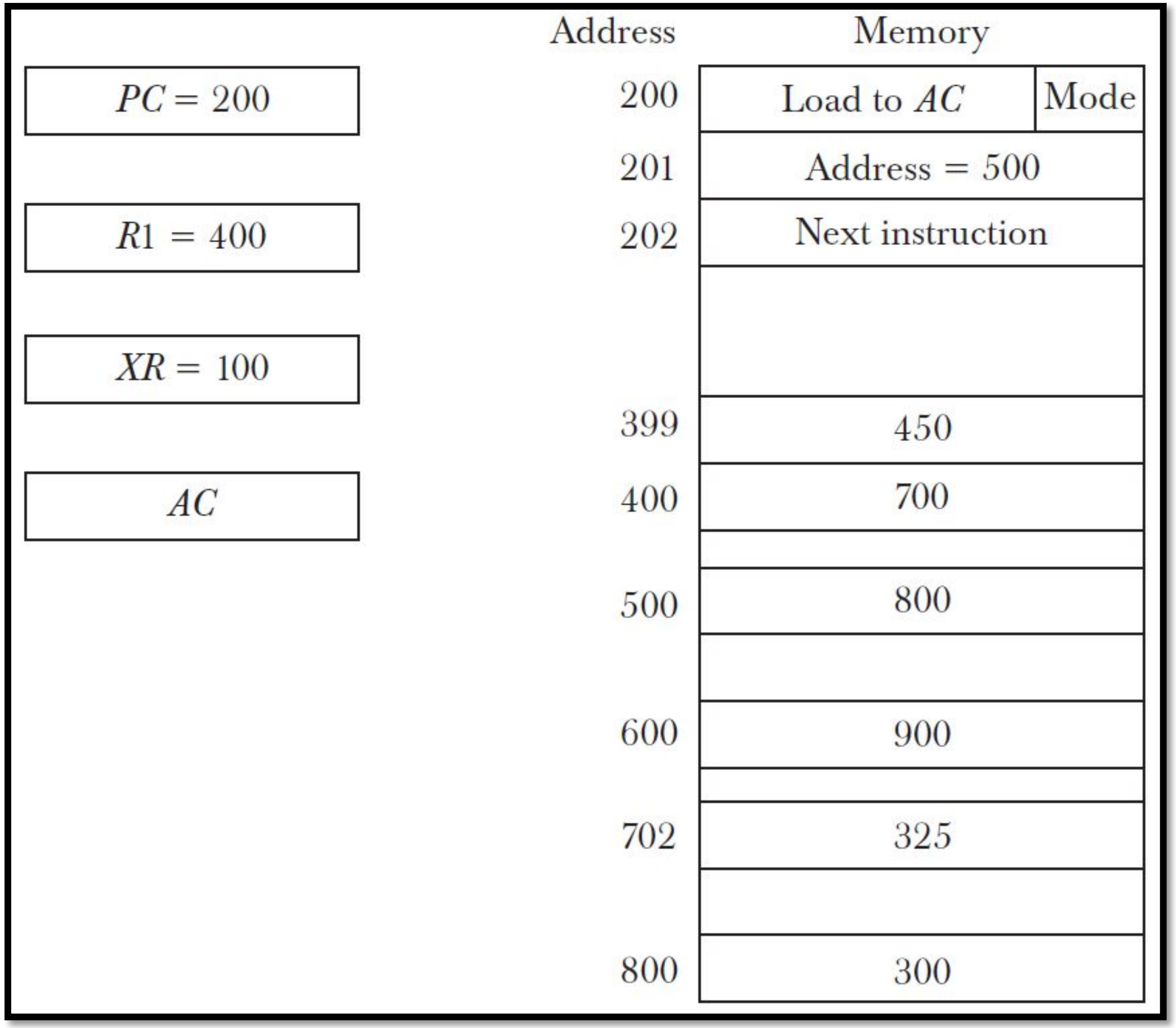
Indirect Addressing Mode



In the indirect mode the effective address is stored in memory at address 500.

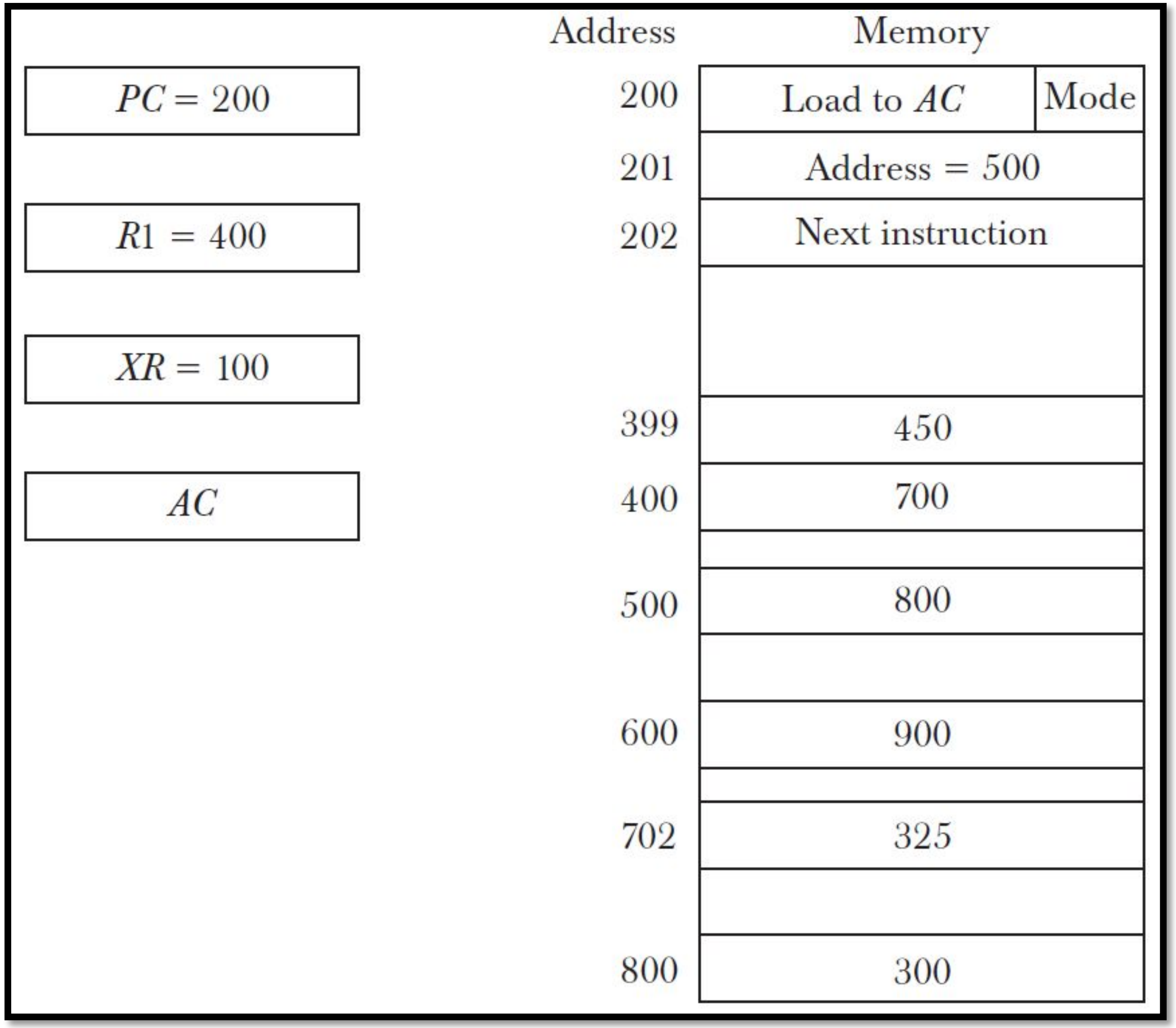
Therefore, the effective address is 800 and the operand is 300.

Relative Addressing Mode



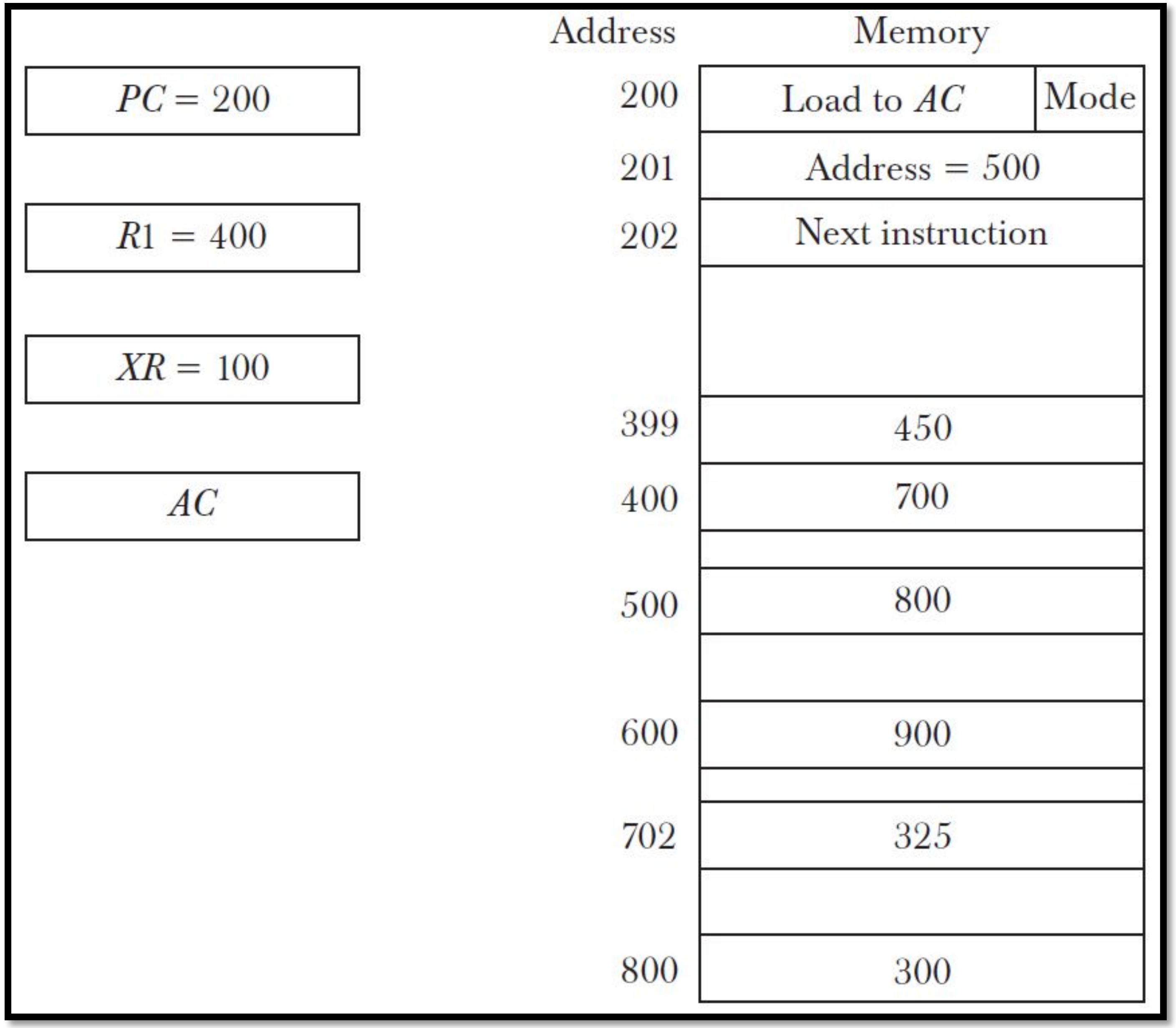
In the relative mode the effective address is 500 + 202 = 702 and the operand is 325.
(Note that the value in PC after the fetch phase and during the execute phase is 202.)

Index Addressing Mode



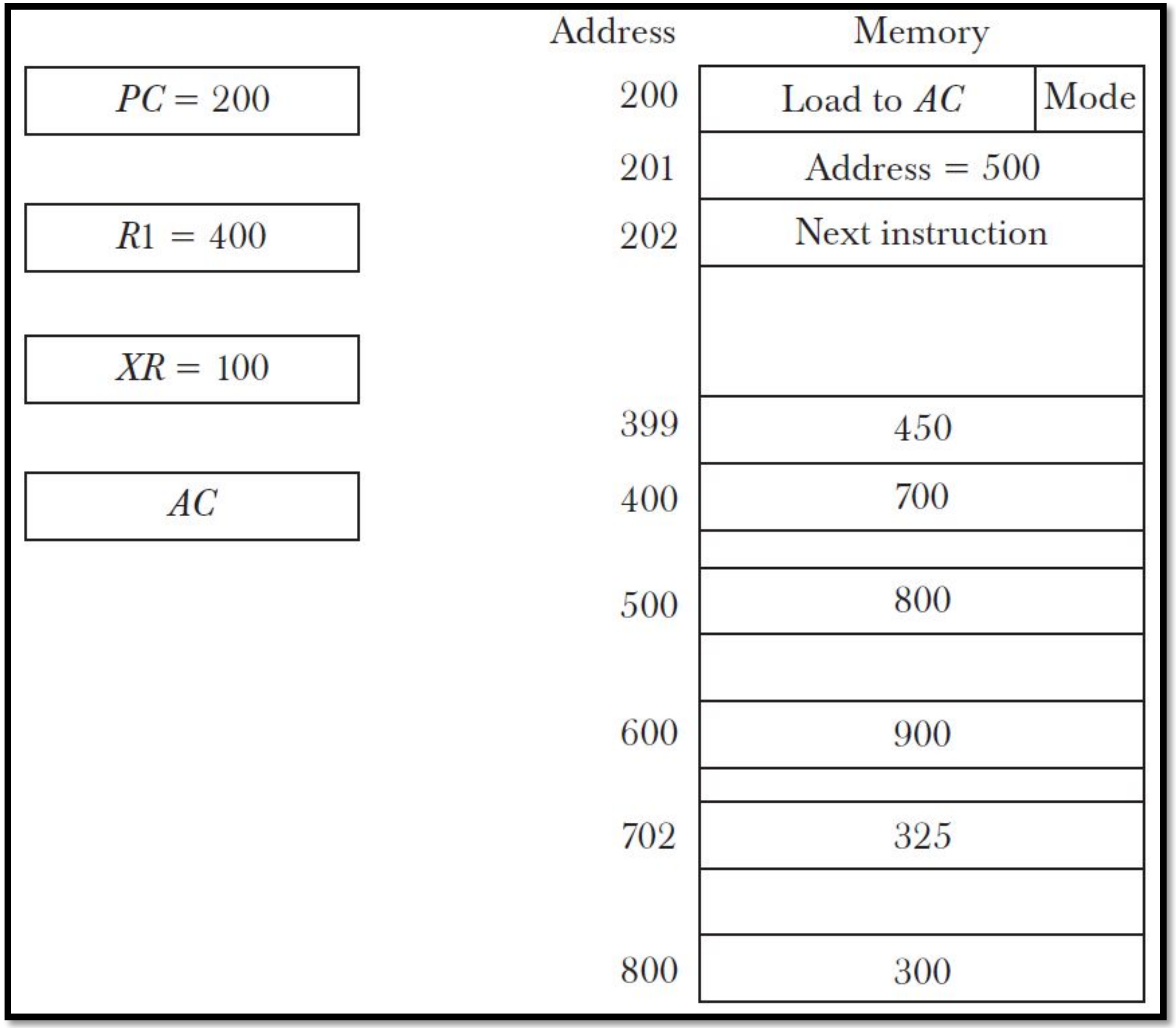
In the index mode the effective address is XR + 500 = 100 + 500 = 600 and the operand is 900.

Register Addressing Mode



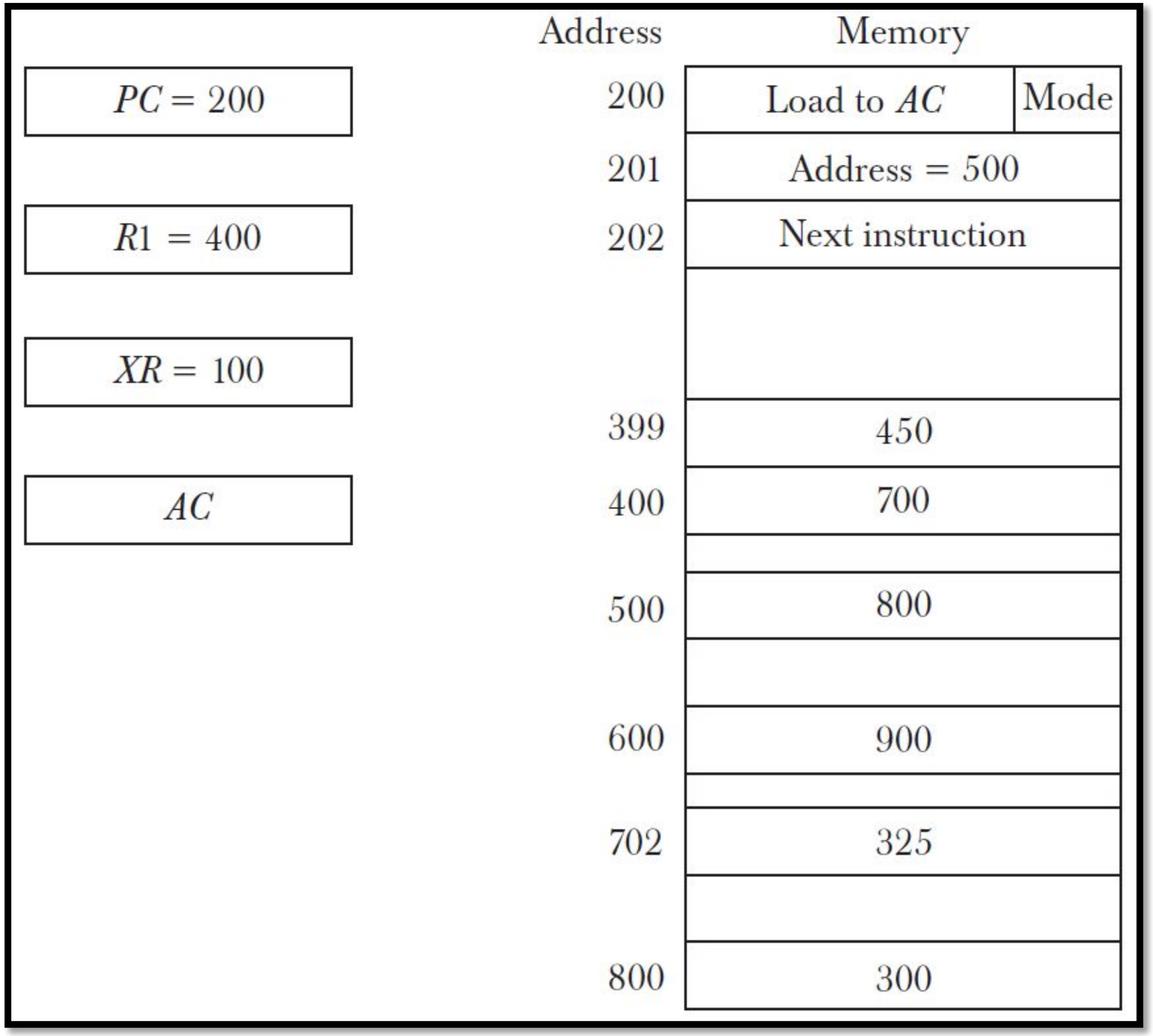
In the register mode the operand is in R1 and 400 is loaded into AC. (There is no effective address in this case.)

Register Indirect Addressing Mode



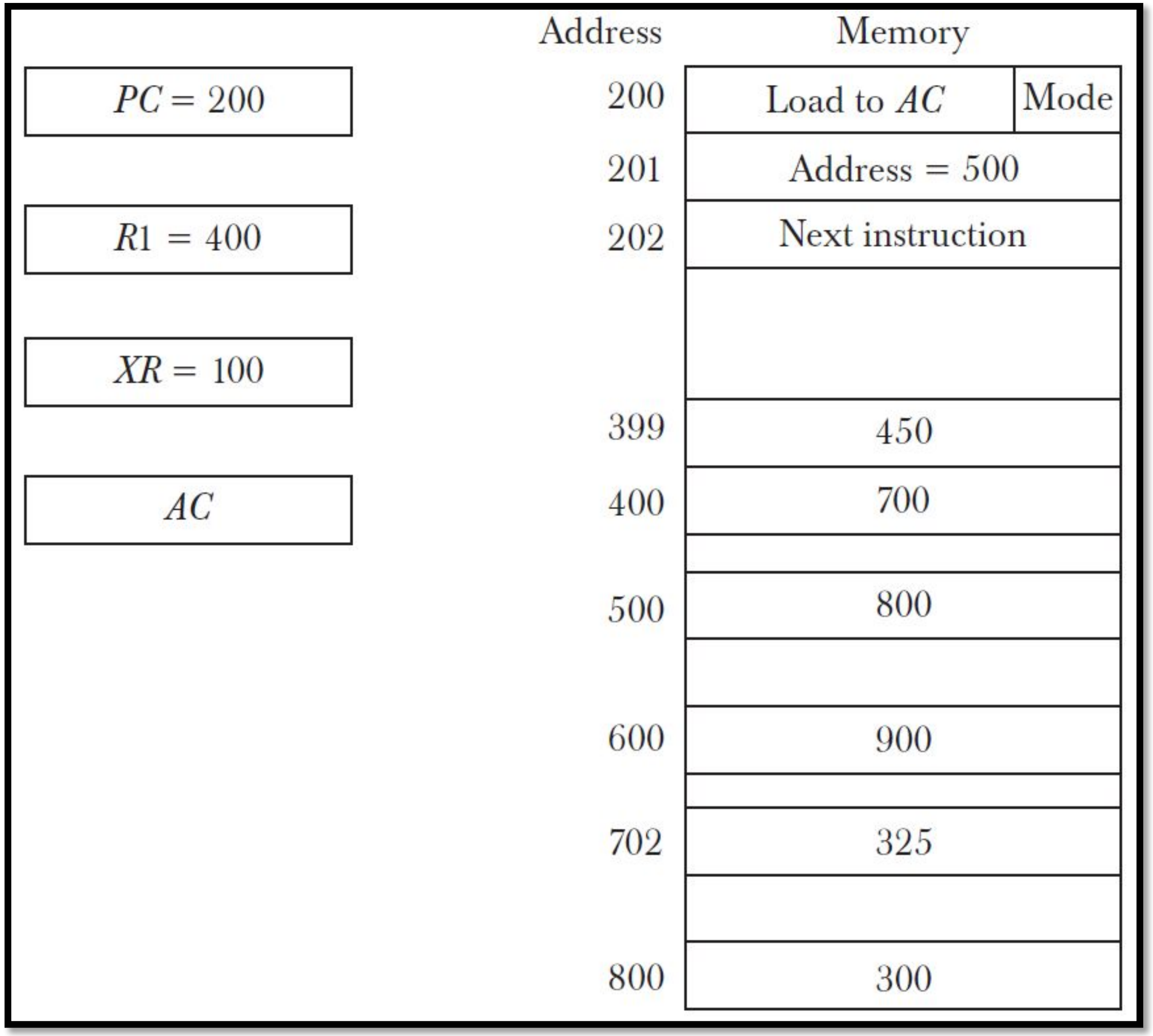
In the register indirect mode the effective address is 400, equal to the content of R1 and the operand loaded into AC is 700.

Auto-Increment Addressing Mode



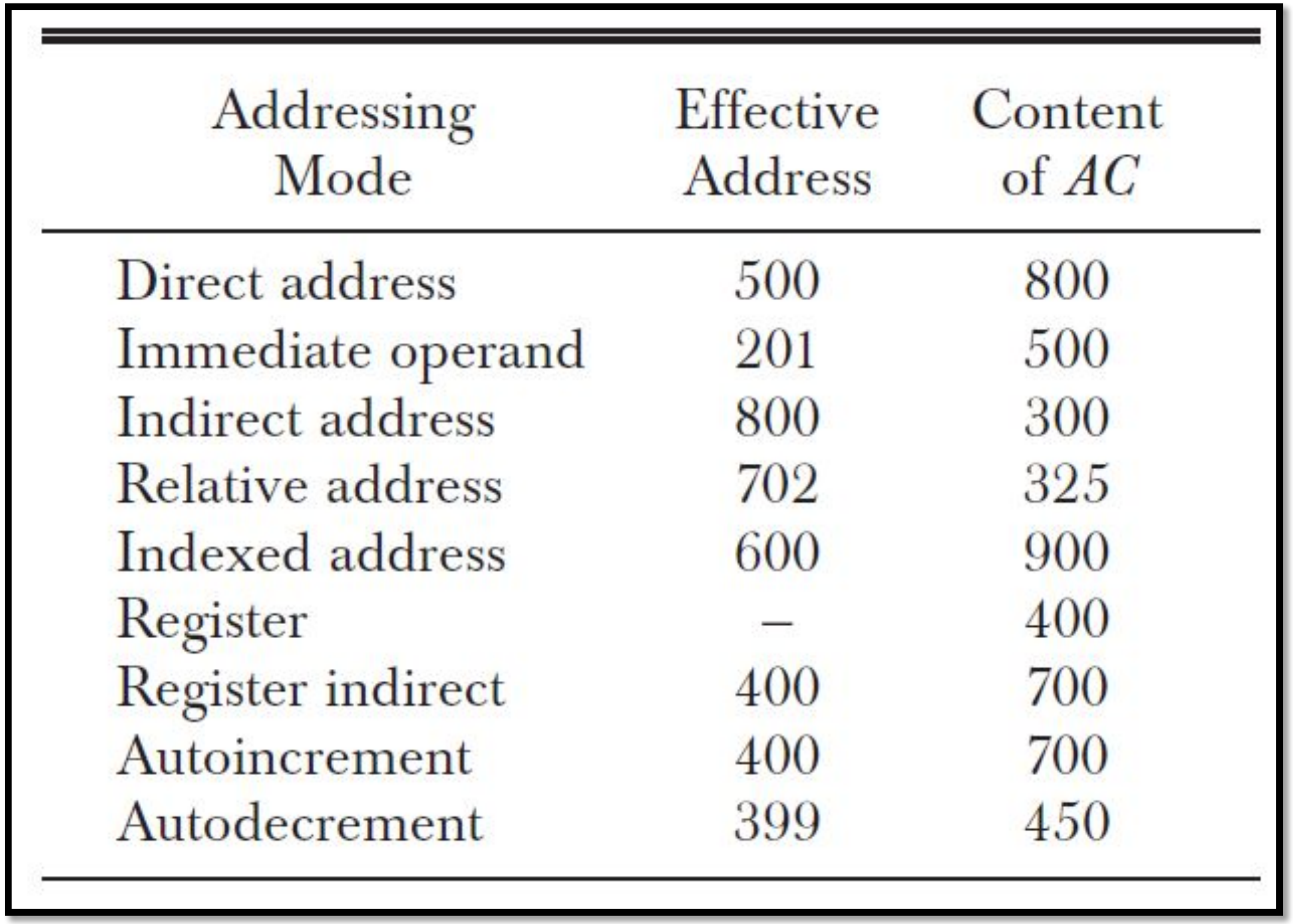
The autoincrement mode is the same as the register indirect mode except that R1 is incremented to 401 after the execution of the instruction.

Auto-Decrement Addressing Mode



The autodecrement mode decrements R1 to 399 prior to the execution of the instruction.
The operand loaded into AC is now 450.

The values of the effective address and the operand loaded into AC for the nine addressing modes



Addressing Mode	Effective Address	Content of *AC*
Direct address	500	800
Immediate operand	201	500
Indirect address	800	300
Relative address	702	325
Indexed address	600	900
Register	–	400
Register indirect	400	700
Autoincrement	400	700
Autodecrement	399	450

Q:

A computer uses a memory unit with 256K words of 32 bits each. A binary instruction code is stored in 1 word of memory. The instruction has 4 parts : an indirect bit, an operation code, a register code part to specify one of 64 registers, and an address part ?

a) How many bits are There in The operation code, register code part and address part ?

b) Draw The instruction word format and indicate The no. of bits in each part ?

c) How many i/p's are There in the data & address i/p's of memory ?

A:

A: Size of memory = 256K × 32

$= 2^{18} \times 32$

a) No. of address bits = 18

No. of data bits = 32

$\Rightarrow$ Address part = 18 bits

Total size of instruction = 32 bits

No. of registers = 64 = 2^6

$\Rightarrow$ register code part = 6 bits

b)

1	7	6	18
I	opcode	Reg. Code	Address

$|\longleftarrow$ 32 bits $\longrightarrow|$

$\Rightarrow$ opcode part = $32 - [18 + 6 + 1]$

$= 32 - 25 = \underline{\underline{7 \text{ bits}}}$

c) No. of i/p's in data part of memory = 32 bits ($\because$ each location can hold 32 bit data)

No. of i/p's in address part of memory = 18 bits ($\because 2^{18}$ locations are available).

Q:

The memory unit of a computer has 256K words of 32 bits each. The computer has an instruction format with 4 fields: an operation code field, a mode field to specify one of the 7 addressing modes, a register field. to specify one of the 60 processor registers and a memory address. Specify the instruction format and the no. of bits in each field of the instruction is in 1 memory word.

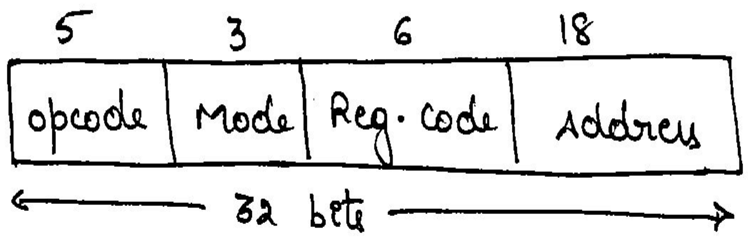
A:

A:

Since instruction is in 1 memory word and

each memory word has 32 bits

$\Rightarrow$ Instruction has 32 bits



7 addressing modes are supported $\Rightarrow$ 3 bits to specify mode

60 processor registers $\Rightarrow$ 6 bits to specify a processor register

Memory size = 256K x 32 $\Rightarrow 2^{10} \times 2^{8} \Rightarrow 2^{18} \Rightarrow$ 18 bits for memory address.

$\Rightarrow$ opcode bits = $32 - (3+6+18) = 32 - 27 = 5$

Q:

what is the difference between a direct and indirect address instruction? How many references to memory are needed for each type of instruction to bring an operand into a processor register?

A:

A:

A direct address instruction needs two references to memory

1) Read instruction

2) Read operand.

An indirect address instruction needs Three references to memory

1) Read instruction

2) Read effective address

3) Read operand.

Q:

A two-word instruction is stored in memory at location designated by W. The address field of the instruction (stored at W+1) is designated by the symbol Y. The operand used during the execution of the instruction is stored at an address symbolized by Z. An index register contains the value X. Show how Z is calculated from the other addresses if the addressing mode of the instruction is

a) direct

b) indirect

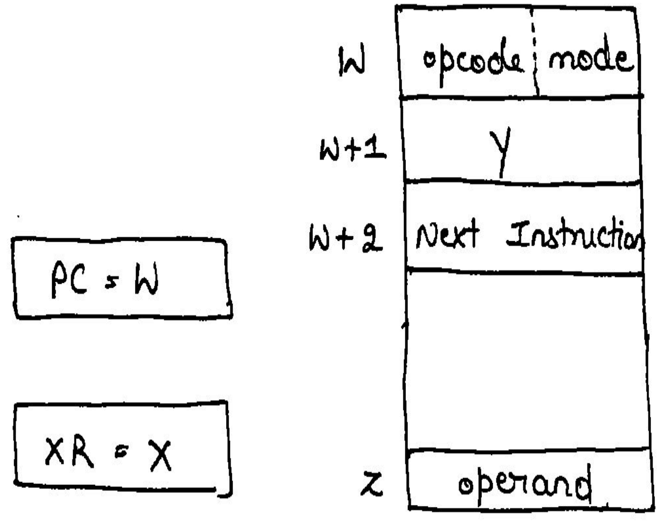
c) relative

d) indexed.

(Effective address)

A:

A:



Effective address of operand = Z

a) Direct: $Z = Y$

b) Indirect: $Z = M[Y]$

c) relative: $Z = Y + [W+2]$

since $PC = W+2$

d) Indexed: $Z = Y + X$ since $XR = X$

Q:

An instruction is stored at location 300 with its address field at location 301. The address field has the value 400. A processor register R_1 contains the number 200. Evaluate the EA if the addressing mode of the instruction is

a) direct

b) immediate

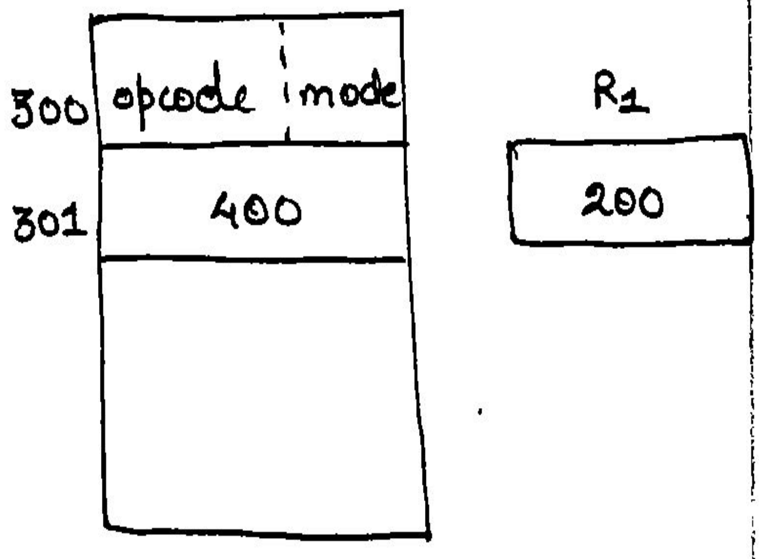
c) Relative

d) Register indirect

e) index with R_1 as index register

A:

A:



PC = 302 (300 + 2)

since two address inst.

a) Direct ⇒ EA = 400

b) Immediate ⇒ EA = 301

c) Relative ⇒ EA = 400 + PC = 400 + 302 = 702

d) Register indirect ⇒ EA = 200 = Content of R_1

e) Index with R_1 as index register ⇒ EA = 400 + 200 = 600

Q:

what must be The address field of an ~~opera~~ indexed addressing mode instruction be to make it The same as a register indirect mode instruction?

A:

A: ZERO

Q:

A relative mode branch type of instruction is stored in memory at an address equivalent to decimal 750. The branch is made to an address equivalent to decimal 500.

a) what should be the value of the relative address field of the instruction (in decimal)?

b) Determine the relative address value in binary using 12 bits. (why must be the no. in 2's complement?)

c) Determine the binary value in PC after the fetch phase and calculate the binary value of 500. Then show that the binary value of PC plus the relative address calculated in part (b) is equal to the binary value of 500?

A:

A:

a) PC = 751 when it is executing instruction at 750

After executing relative addressing mode instruction, PC = 500

⇒ 751 + Address field value = 500

⇒ Address field value = 500 - 751 = $\underline{\underline{-251}}$

A:

b) Address field value = −251 in binary

= 2's complement of 251 (∵ −ve nos. are represented in 2's comp)

$$\begin{array}{r|l l}
2 & 251 & \\
2 & 125 & 1 \\
2 & 62 & 1 \\
2 & 31 & 0 \\
2 & 15 & 1 \\
2 & 7 & 1 \\
2 & 3 & 1 \\
2 & 1 & 1 \\
 & 0 & 1 \uparrow
\end{array}$$

= 2's complement of (11111011)

= 2's complement of (0000 1111 1011)

↳ 12 bit representation

= $(1111\ 0000\ 0101)_2$

A:

c) 751 = PC = $(0010\ 1110\ 1111)_2$

$(500)_{10} = (0001\ 1111\ 0100)_2$

PC = 0010 1110 1111 (+700)

Addru = $\underline{1111\ 0000\ 0101}$ (-251)

$\boxed{1}$ 0001 1111 0100

discard = $(500)_{10}$

```
2|751          2|500
2|375  1       2|250  0
2|187  1       2|125  0
2|93   1       2|62   1
2|46   1       2|31   0
2|23   0       2|15   1
2|11   1       2|7    1
2|5    1       2|3    1
2|2    1       2|1    1  ↑
2|1    0  ↑      0    1
  0    1
```

Data Transfer and Manipulation

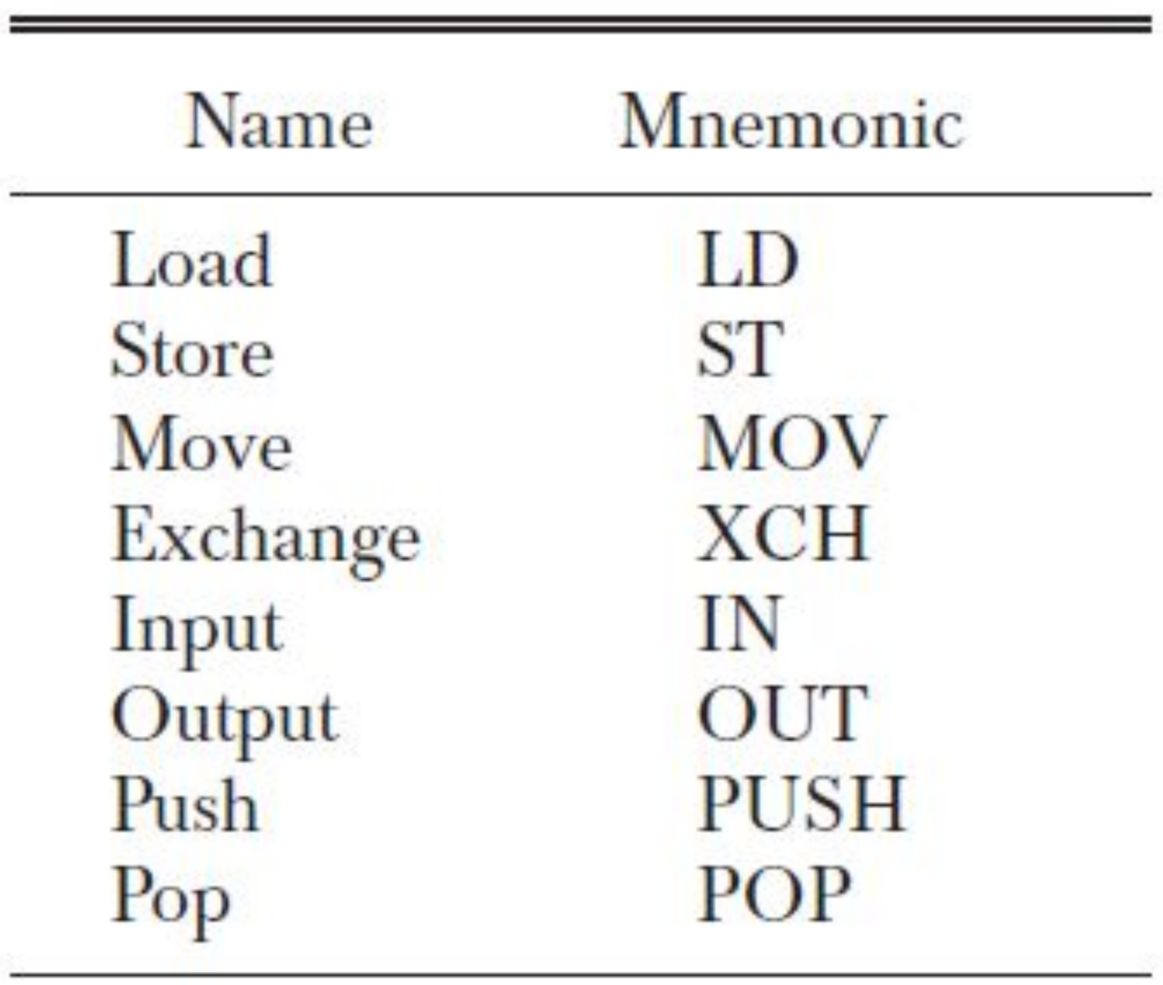
Data Transfer and Data Manipulation Instructions

- Data transfer instructions cause transfer of data from one location to another without changing the binary information content.
- Data manipulation instructions are those that perform arithmetic, logic, and shift operations.

Data Transfer Instructions

- Data transfer instructions move data from one place in the computer to another without changing the data content.
- The most common transfers are
    - between memory and processor registers,
    - between processor registers and input or output,
    - and between the processor registers themselves.

Typical Data Transfer Instructions



Name	Mnemonic
Load	LD
Store	ST
Move	MOV
Exchange	XCH
Input	IN
Output	OUT
Push	PUSH
Pop	POP

Description

- The load instruction has been used mostly to designate a transfer from memory to a processor register, usually an accumulator.
- The store instruction designates a transfer from a processor register into memory.
- The move instruction has been used in computers with multiple CPU registers to designate a transfer from one register to another. It has also been used for data transfers between CPU registers and memory or between two memory words.
- The exchange instruction swaps information between two registers or a register and a memory word.
- The input and output instructions transfer data among processor registers and input or output terminals.
- The push and pop instructions transfer data between processor registers and a memory stack.

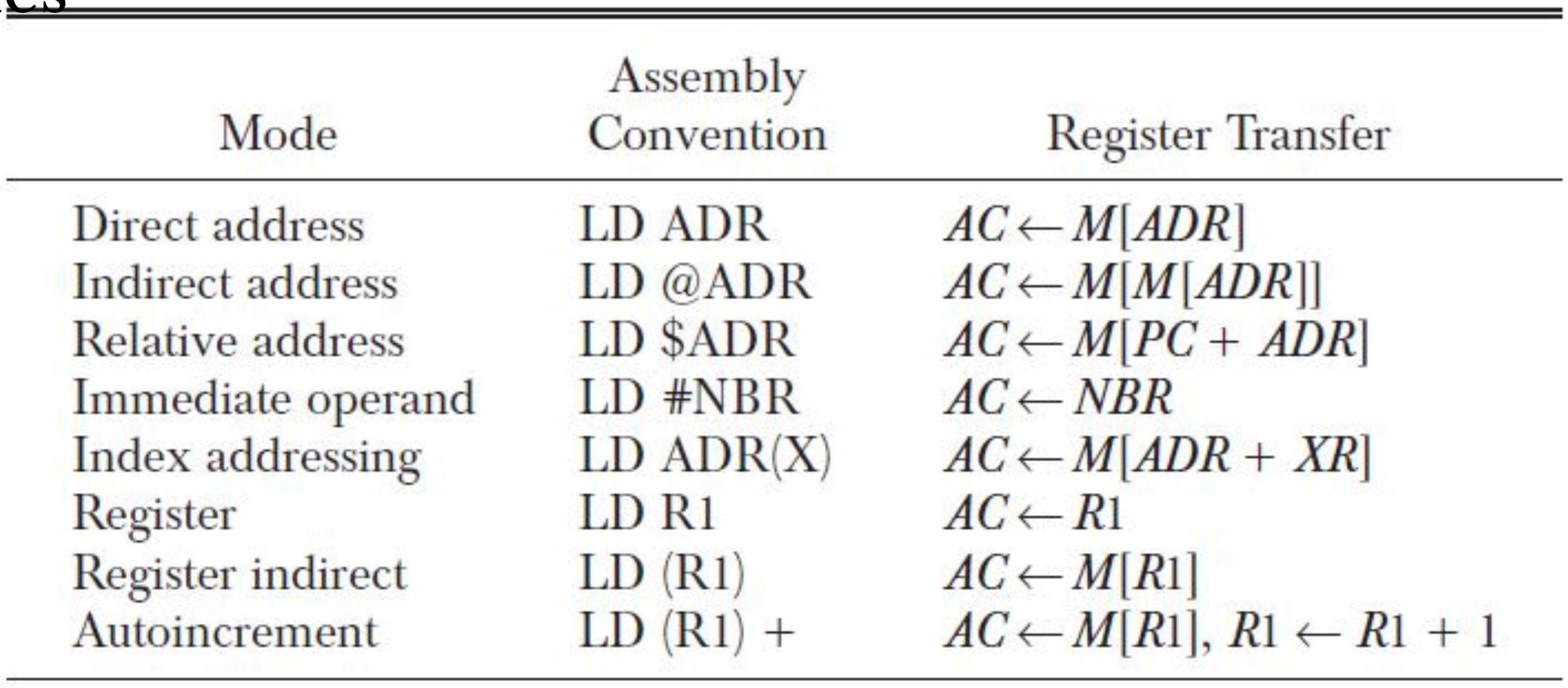
Addressing Mode

- It must be realized that the instructions are often associated with a variety of addressing modes.
- Some assembly language conventions modify the mnemonic symbol to differentiate between the different addressing modes.

“each instruction can occur with a variety of addressing modes.”

Eight Addressing Modes for the Load Instruction

As an example, consider the ***load to accumulator*** instruction when used with eight different addressing modes



Mode	Assembly Convention	Register Transfer
Direct address	LD ADR	$AC \leftarrow M[ADR]$
Indirect address	LD @ADR	$AC \leftarrow M[M[ADR]]$
Relative address	LD $ADR	$AC \leftarrow M[PC + ADR]$
Immediate operand	LD #NBR	$AC \leftarrow NBR$
Index addressing	LD ADR(X)	$AC \leftarrow M[ADR + XR]$
Register	LD R1	$AC \leftarrow R1$
Register indirect	LD (R1)	$AC \leftarrow M[R1]$
Autoincrement	LD (R1) +	$AC \leftarrow M[R1],\ R1 \leftarrow R1 + 1$

Description

- *ADR* stands for an address, *NBR* is a number or operand, *X* is an index register, *R*1 is a processor register, and *AC* is the accumulator register.
- The @ character symbolizes an indirect address.
- The $ character before an address makes the address relative to the program counter *PC*.
- The # character precedes the operand in an immediate-mode instruction.
- An indexed mode instruction is recognized by a register that is placed in parentheses after the symbolic address.
- The register mode is symbolized by giving the name of a processor register.
- In the register indirect mode, the name of the register that holds the memory address is enclosed in parentheses.
- The autoincrement mode is distinguished from the register indirect mode by placing a plus after the parenthesized register.
- The autodecrement mode would use a minus instead.

Note

- To be able to write assembly language programs for a computer, it is necessary to know the type of instructions available and also to be familiar with the addressing modes used in the particular computer.

Data Manipulation Instructions

- Data manipulation instructions perform operations on data and provide the computational capabilities for the computer.
- The data manipulation instructions in a typical computer are usually divided into three basic types:

1. Arithmetic instructions
2. Logical and bit manipulation instructions
3. Shift instructions

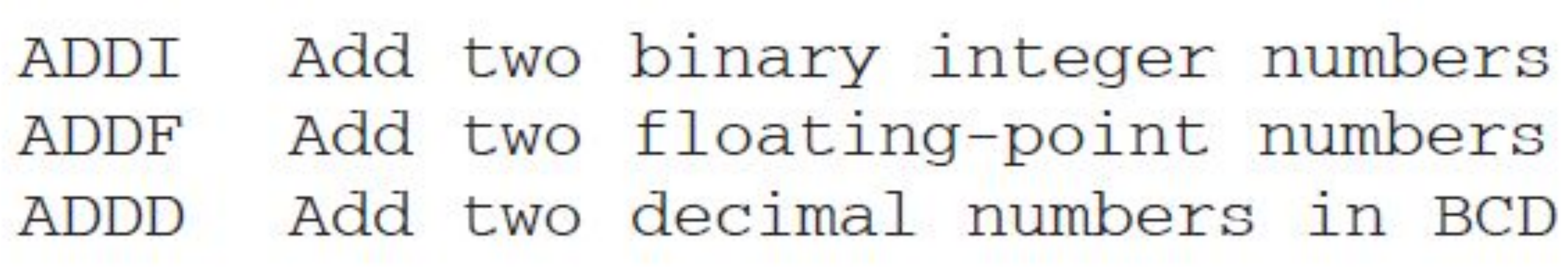
Arithmetic Instructions

- The four basic arithmetic operations are addition, subtraction, multiplication, and division.
- Most computers provide instructions for all four operations.
- Some small computers have only addition and possibly subtraction instructions.
- The multiplication and division must then be generated by means of software subroutines.
- The four basic arithmetic operations are sufficient for formulating solutions to scientific problems when expressed in terms of numerical analysis methods.

Typical Arithmetic Instructions

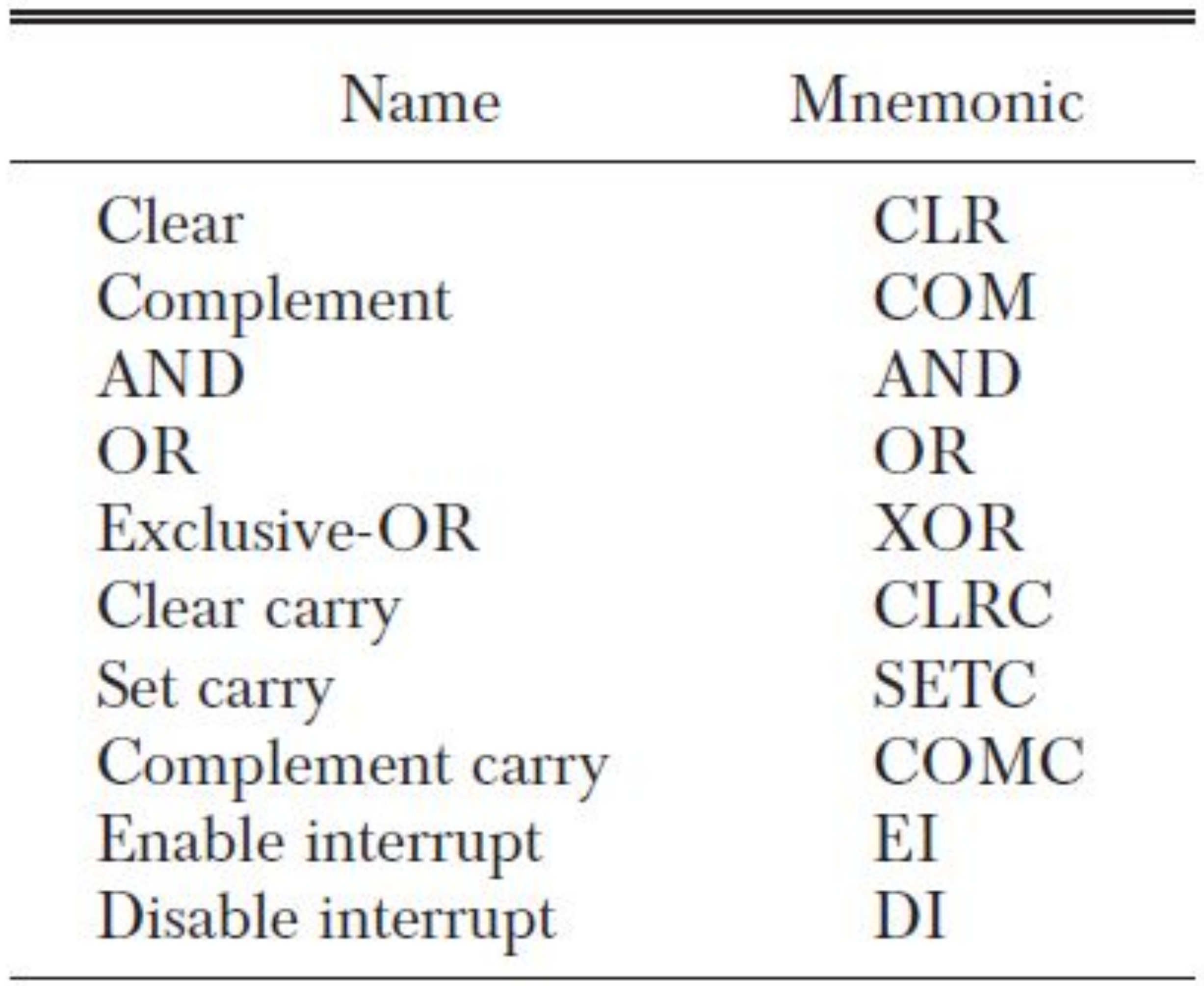
Name	Mnemonic
Increment	INC
Decrement	DEC
Add	ADD
Subtract	SUB
Multiply	MUL
Divide	DIV
Add with carry	ADDC
Subtract with borrow	SUBB
Negate (2's complement)	NEG

Data type



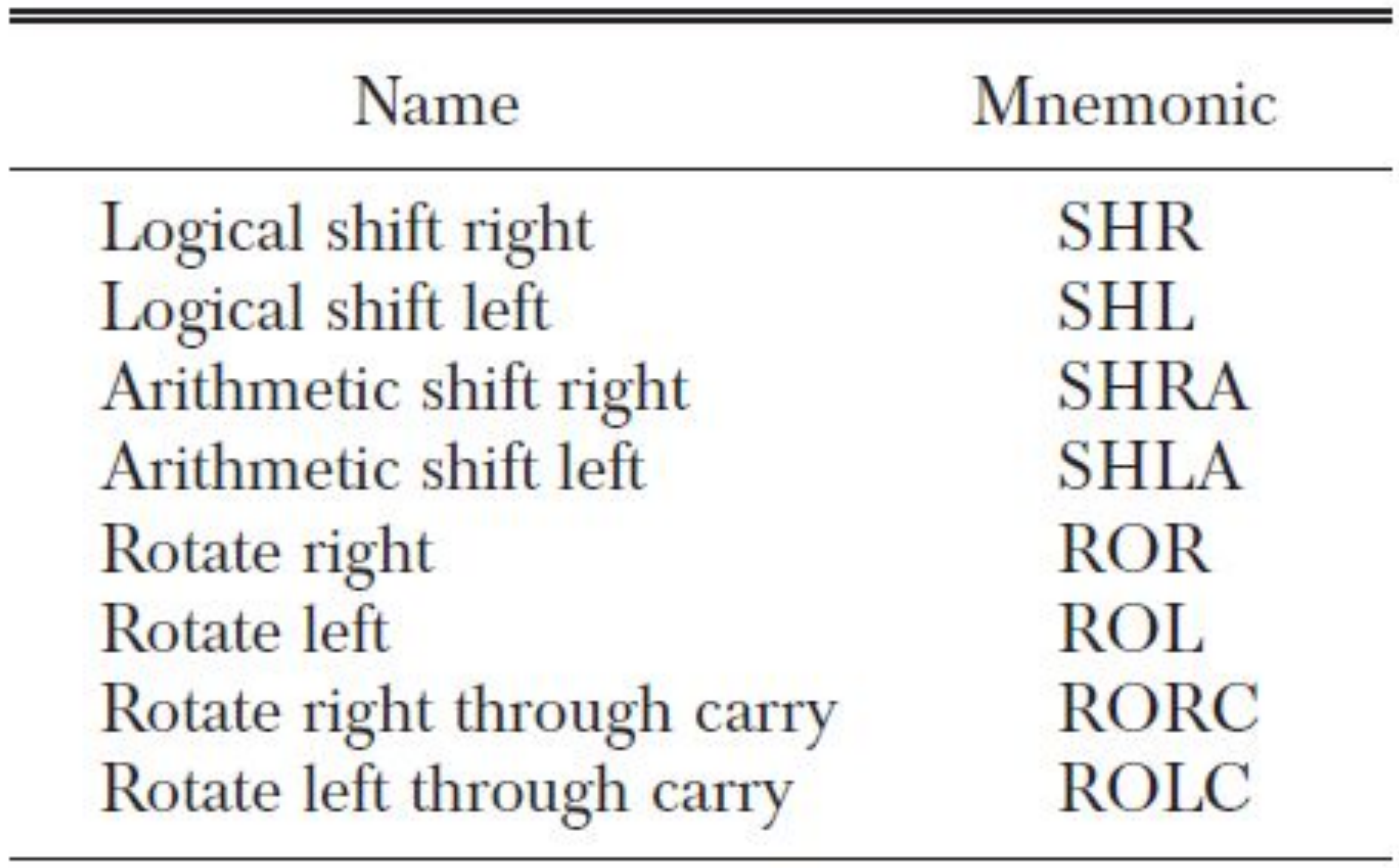
ADDI	Add two binary integer numbers
ADDF	Add two floating-point numbers
ADDD	Add two decimal numbers in BCD

Logical and Bit Manipulation Instructions



Name	Mnemonic
Clear	CLR
Complement	COM
AND	AND
OR	OR
Exclusive-OR	XOR
Clear carry	CLRC
Set carry	SETC
Complement carry	COMC
Enable interrupt	EI
Disable interrupt	DI

Shift Instructions



Name	Mnemonic
Logical shift right	SHR
Logical shift left	SHL
Arithmetic shift right	SHRA
Arithmetic shift left	SHLA
Rotate right	ROR
Rotate left	ROL
Rotate right through carry	RORC
Rotate left through carry	ROLC

A possible instruction code format of a shift instruction may include five fields as follows:

OP	REG	TYPE	RL	COUNT

Here OP is the operation code field; REG is a register address that specifies the location of the operand; TYPE is a 2-bit field specifying the four different types of shifts; RL is a 1-bit field specifying a shift right or left; and COUNT is a *k*-bit field specifying up to $2^k - 1$ shifts. With such a format, it is possible to specify the type of shift, the direction, and the number of shifts, all in one instruction.