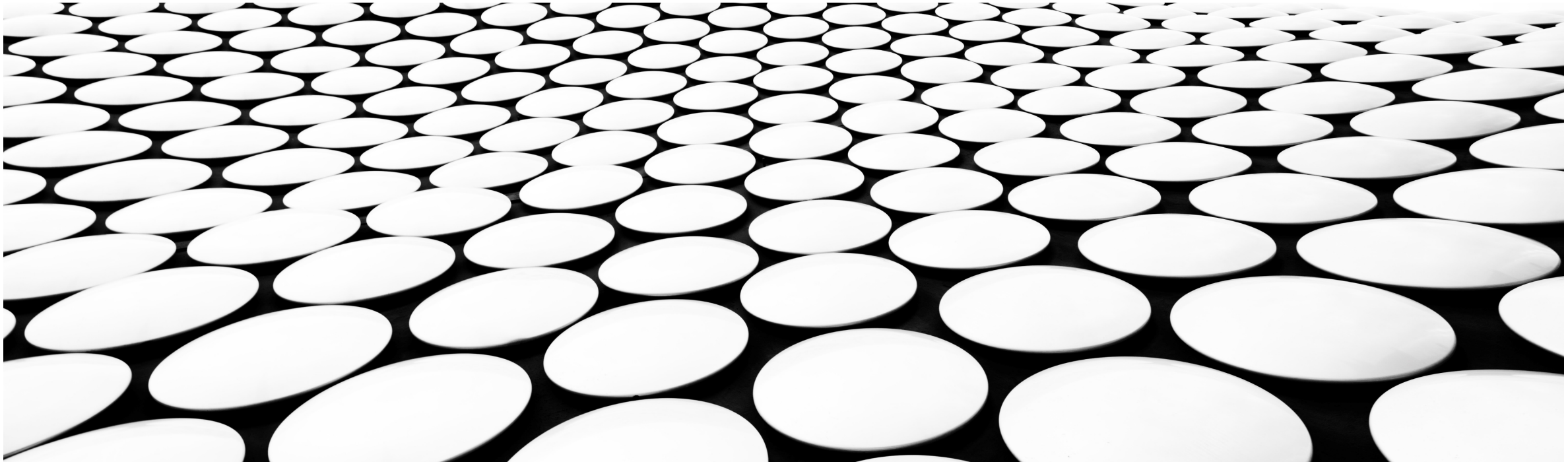
CENTRAL PROCESSING UNIT

UNIT - IV



CONTENT

1. General register organization
2. Stack organization
3. Instruction format
4. Addressing modes
5. Data transfer and manipulation instructions
6. Program control
7. RISC, and CISC

1. GENERAL REGISTER ORGANIZATION

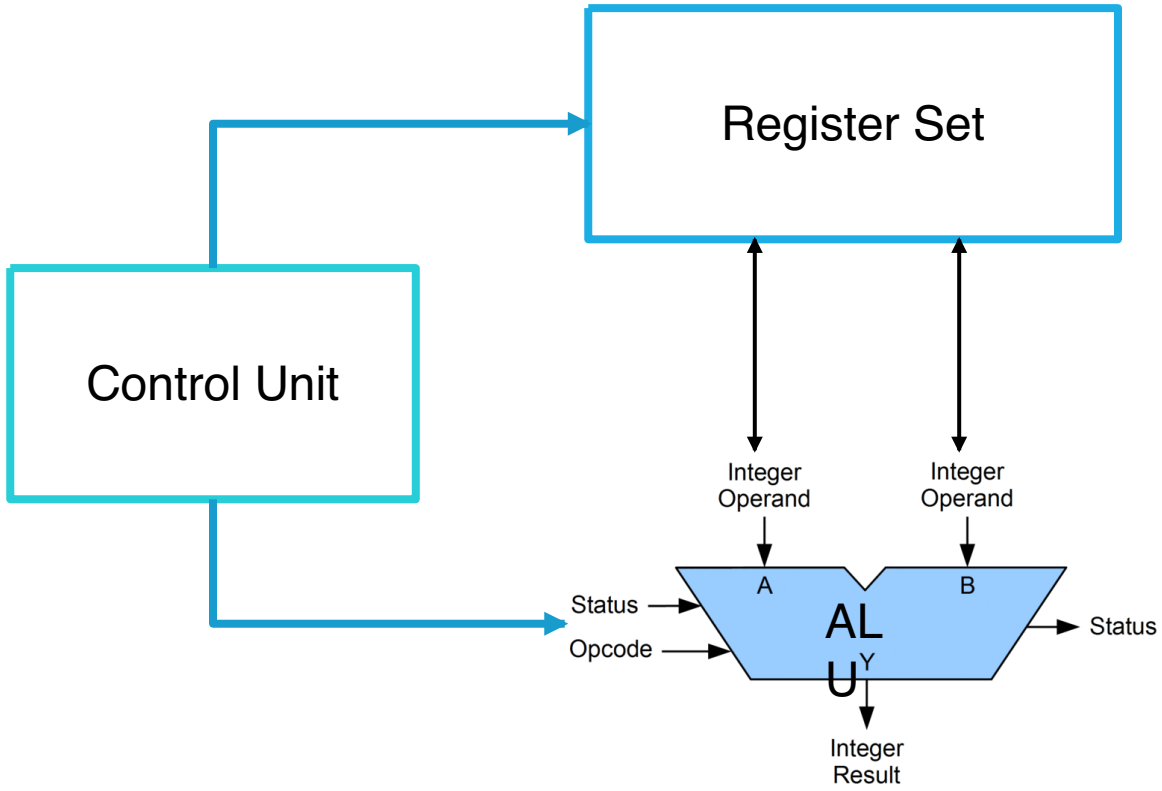
INTRODUCTION

The central processing unit is where all the calculations and logic operations take place.

CPU performs data-processing operations.

Main parts of the CPU are:

- Arithmetic Logic Unit (ALU)
- Control Unit (CU)
- Registers

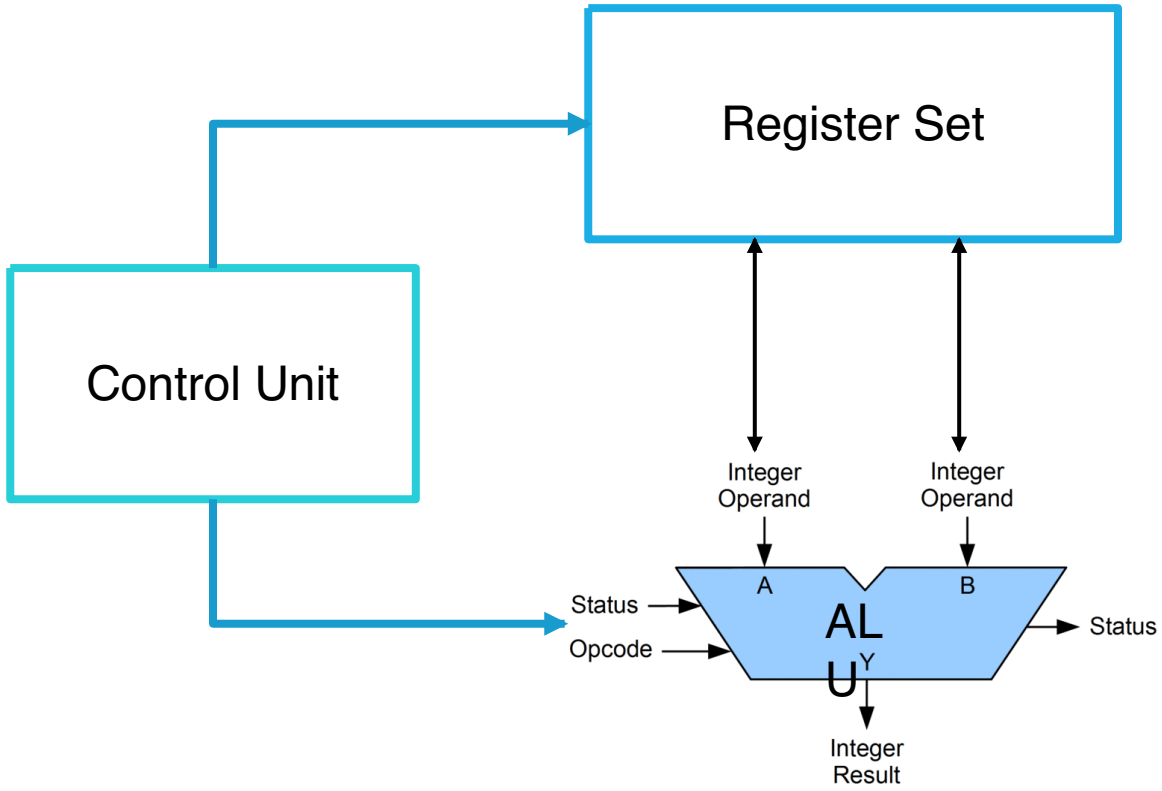


INTRODUCTION

The **registers set** stores intermediate data used during the execution of the instructions.

The **ALU** performs the required micro-operations 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.



GENERAL REGISTER ORGANIZATION

Registers are more convenient and efficient to **store pointers, return addresses, counters, temporary results**, etc.

In Basic Computer, there is only one general-purpose register, the Accumulator (A), but in modern CPUs, there are many general-purpose registers.

It is advantageous to have many registers:

- Transfer between registers within the processor is relatively fast.
- Going “off the processor” to access memory is much slower.

GENERAL REGISTER ORGANIZATION

When a **large number of registers** are included in the CPU it is efficient to connect them through a **common system bus**.

Because registers communicate with each other not only for direct data transfers but also while performing various microoperations.

A BUS ORGANIZATION FOR SEVEN CPU REGISTERS

The control unit directs the information flow through ALU by

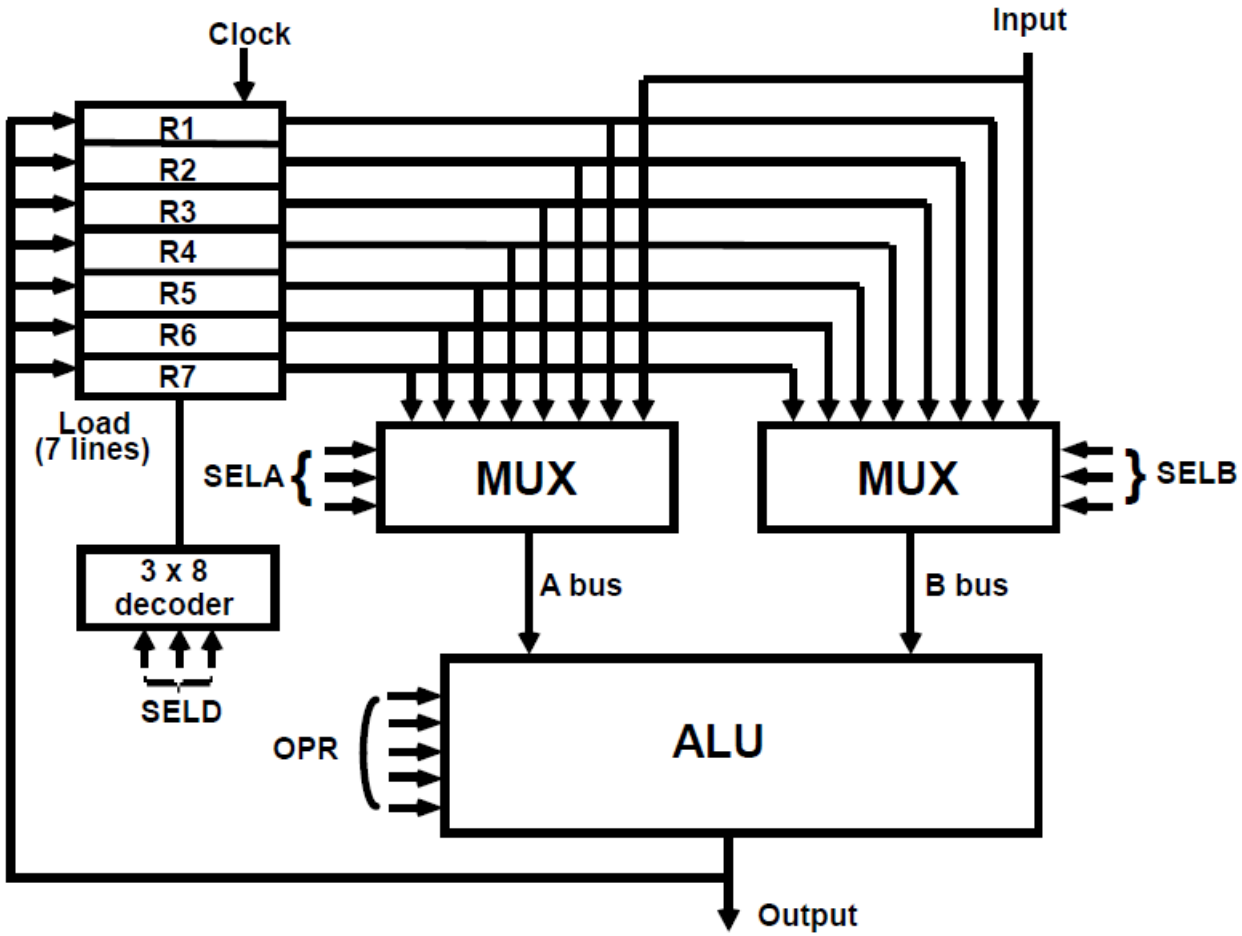
Selecting various components in the system.

Selecting the Function of ALU.

Example: R1 ← R2 + R3

1. MUX A selector (SELA): BUS A ← R2.
2. MUX B selector (SELB): BUS B ← R3.
3. ALU operation selector (OPR): ALU to ADD.
4. Decoder destination selector (SELD): R1←Out Bus

The entire information is passed through the **Control Word.**



CONTROL WORD FORMAT

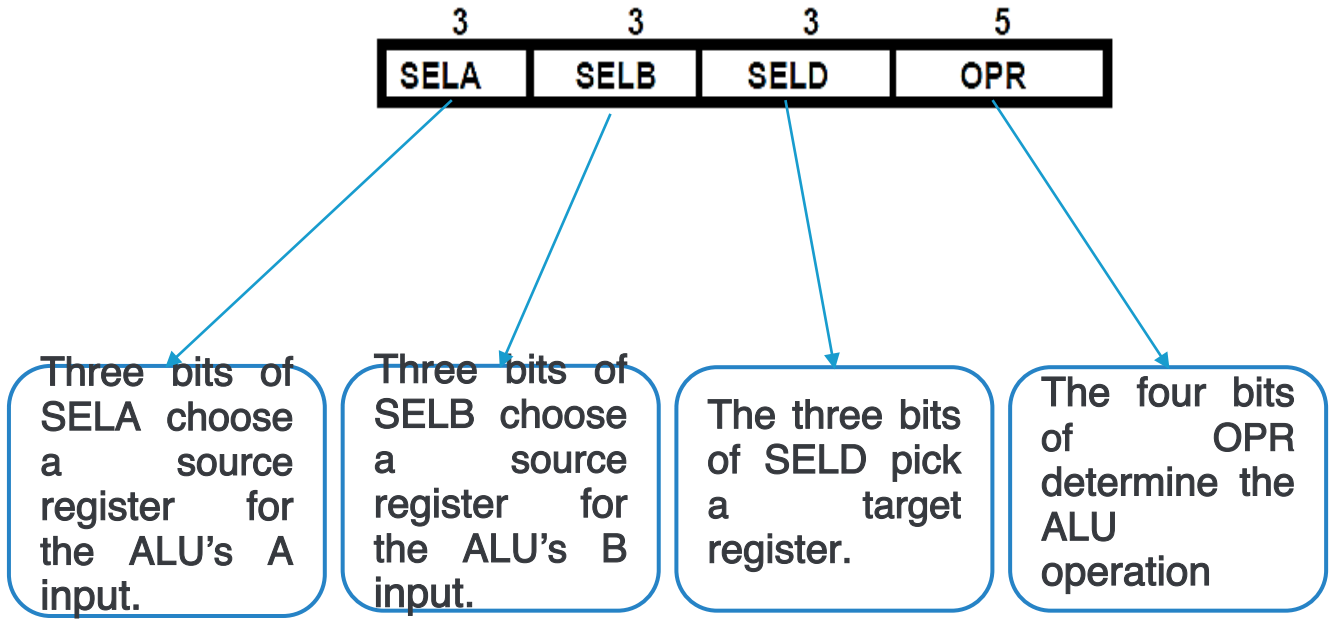
The control word is determined by the sum of the binary selection inputs.

It is divided into four sections.

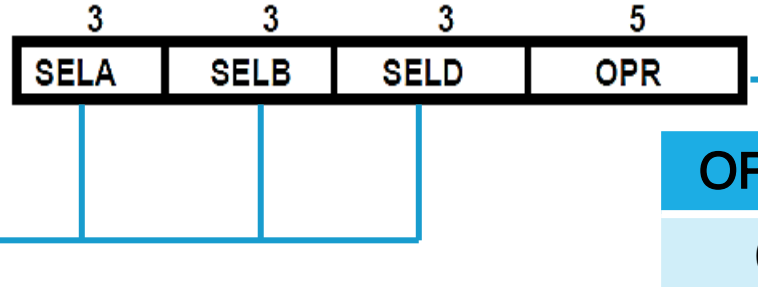
SELA, SELB, and SELD each have three bits,

the OPR field has five bits.

For **a total of 14 bits** in the control word.



CONTROL WORD FORMAT



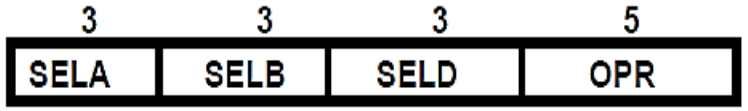
ALU operations encoding

Register Selection Field

Binary Code	SELA	SELB	SELD
000	Input	Input	None
001	R1	R1	R1
010	R2	R2	R2
011	R3	R3	R3
100	R4	R4	R4
101	R5	R5	R5
110	R6	R6	R6
111	R7	R7	R7

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

MICRO-OPERATIONS AT ALU



Control Word Format

Micro-operation	SEL A	SEL B	SELD	OPR	Control Word
R1 ← R2 – R3	R2	R3	R1	SUB	010 011 001 00101
R4 ← R4 ∨ R5	R4	R5	R4	OR	100 101 100 01010
R6 ← R6 + 1	R6	-	R6	INCA	110 000 110 00001
R7 ← R1	R1	-	R7	TSFA	001 000 111 00000
Output ← R2	R2	-	None	TSFA	010 000 000 00000
Output ← Input	Input	-	None	TSFA	000 000 000 00000
R4 ← shl R4	R4	-	R4	SHLA	100 000 100 11000
R5 ← 0	R5	R5	R5	XOR	101 101 101 01100

2. STACK ORGANIZATION

STACK ORGANIZATION

A **stack is an ordered linear list** in which all insertions and deletions are made at one end, called the top.

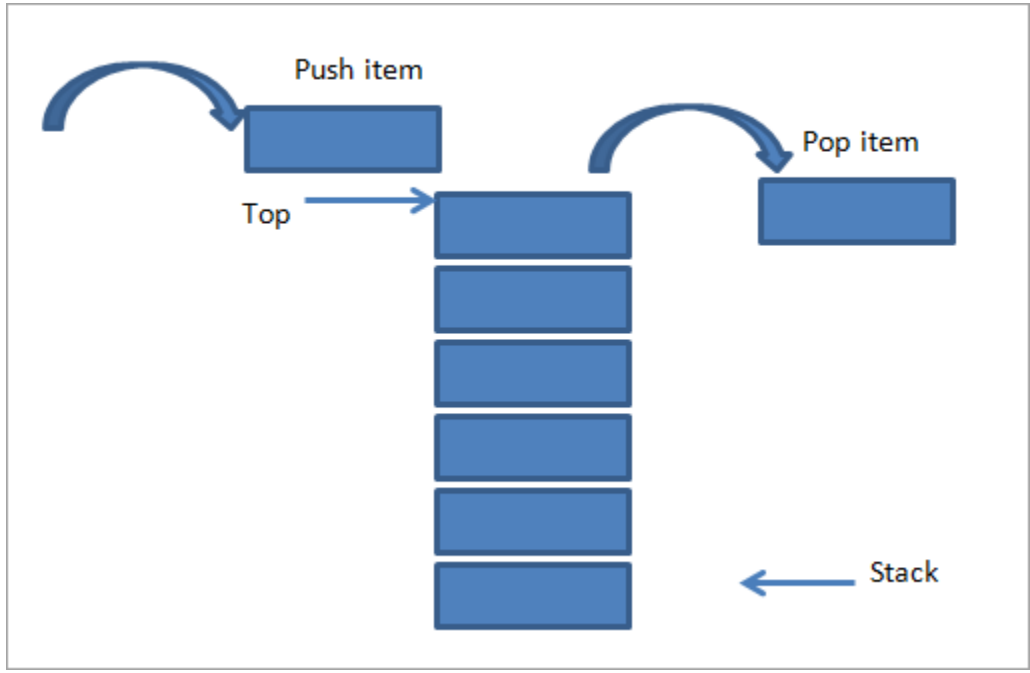
It uses the Last In First Out (LIFO) access method which is the most popular access method in most of the CPUs.

A **register is used to store the address** of the topmost element of the stack which is known as a **Stack Pointer(SP)** because its value always points at the top of the stack.

The main two operations that are performed on the operands of the stack are:

Push Operation: The operation of **inserting** an item onto a stack is called push operation.

Pop Operation: The operation of **deleting** an item onto a stack is called pop operation.



STACK IN COMPUTER ARCHITECTURE

There are two types of stack organization that are used in computer architecture:

Register stack: It is built using a register.

Memory stack: It is the logical part of memory allocated as the stack. The logically partitioned part of RAM is used to implement the stack.

REGISTER STACK ORGANIZATION

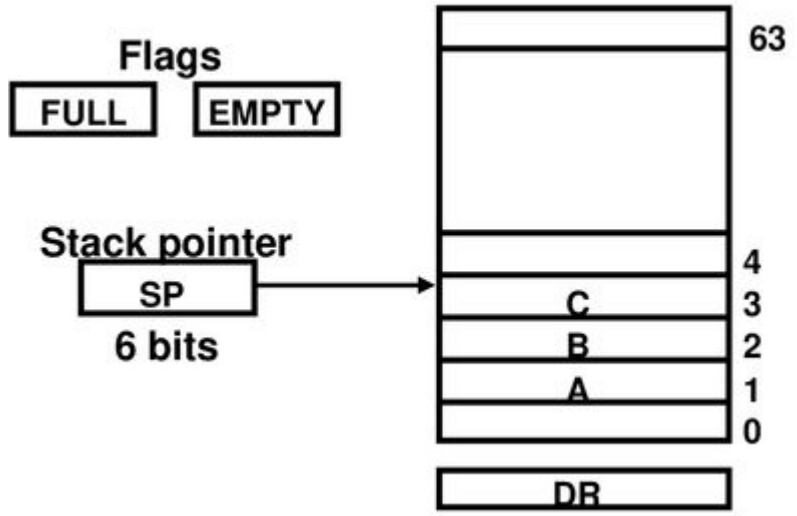
A stack can be organized as a collection of a finite number of memory words or **registers**.

In the figure, there are 64 registers used to make a register stack. The numbers 0,1,2,3,..... 63 denote the address of different registers (Binary address representation is 000000 … 111111).

SP is a pointer that points to the top of the stack i.e. it currently points to the item at the top (in this figure SP contains a 6-bit address).

In a 64-word stack, the stack pointer contains 6 bits because 2^6=64.

Since SP has only 6 bits, it cannot exceed a number greater than 63(111111 in binary). When 63 is incremented by 1 the result is 0 since 111111+1=1000000, but SP can accommodate only the six least significant bits → SP points to the 000000 address register which implies the stack is full.



REGISTER STACK ORGANIZATION

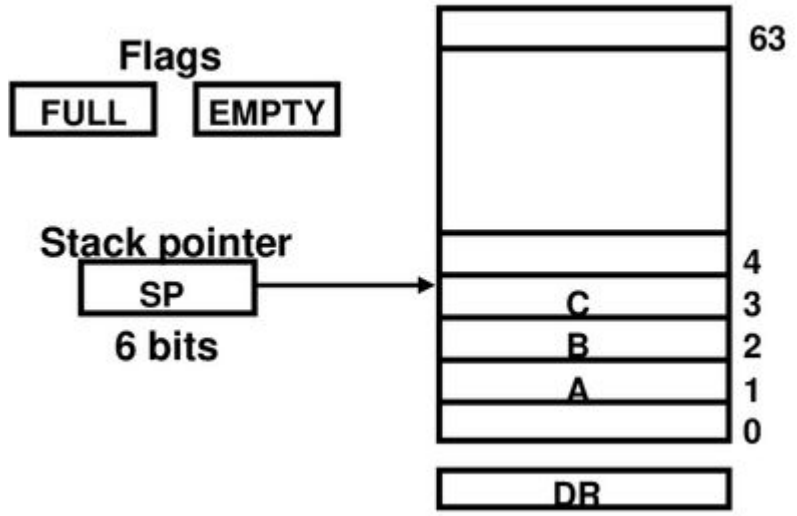
The two more registers called **FULL** and **EMPTY** are used. These are made up of flip-flops also known as flags. It indicates whether the stack is full or not.

If FULL = 1, then EMPTY = 0 → stack is full.

If FULL= 0, then EMPTY =1 → stack is empty.

DR is the data register through which data is transferred to and from the stack.

Zero address instructions are used in registers stack organization i.e. the instruction that does not contain the address of the operands.



REGISTER STACK ORGANIZATION

PUSH Operation

For the PUSH operation initially,

SP ← 0

Full ← 0

EMTY ← 1

SP ← SP+1 Increment stack pointer

M[SP] ← DR Write on top of the stack

If (SP = 63) then (FULL ← 1) Check if the stack is full

EMTY← 0 Stack is not empty

POP Operation

DR ← M[SP] Read from the top of the stack

SP← SP-1 Decrement the stack pointer

If (SP=0) then (EMTY←1) Check if the stack is empty

FULL ← 0 Stack is not full

MEMORY STACK ORGANIZATION

A stack may be implemented in a computer's random access memory (RAM).

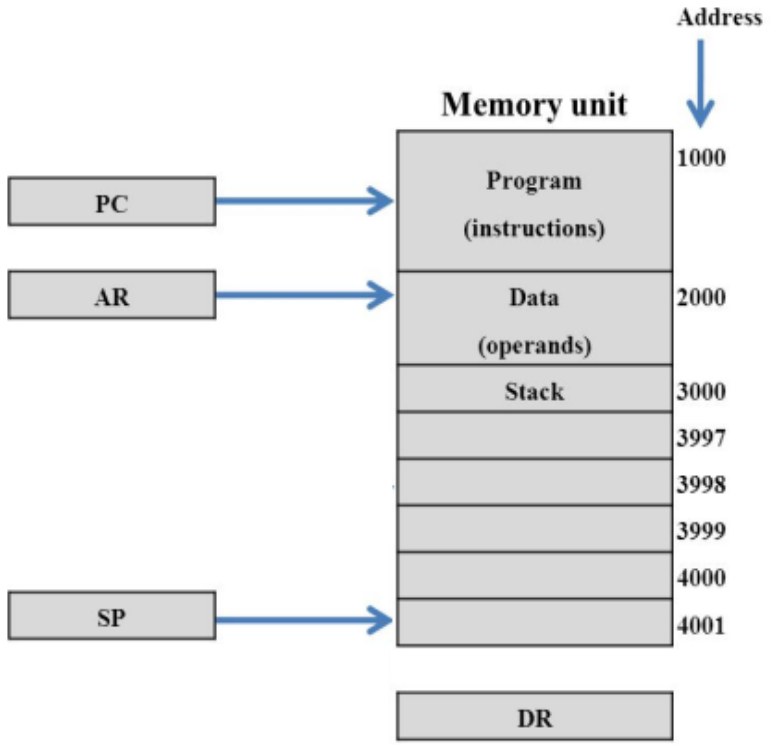
A stack is implemented in the CPU by allocating a portion of memory to a stack operation and utilizing a processor register as a stack pointer. The stack pointer is a CPU register that specifies the stack's initial memory address.

The RAM is divided into three logical parts:

Program: The logical part of RAM where programs are stored.

Data: It is the logical part of the RAM where data(operands) are stored.

Stack: It is the part of RAM used to implement stack.



MEMORY STACK ORGANIZATION

PUSH Operation

SP ← SP -1 Decrement stack pointer

M[SP] ← DR Write on top of the stack

POP Operation

DR ← M[SP] Read from the top of the stack

SP← SP + 1 Increment the stack pointer

The upper limit register and lower limit register are used to check the stack's overflow (Full) and underflow (Empty).

Most computers do not provide hardware to check stack overflow (full stack) or underflow (empty stack). It must be done in the software

REVERSE POLISH NOTATION

Introduction

The stack organization is very effective in evaluating the arithmetic expression.

Computers found difficulties in evaluating common arithmetic mathematical expressions because they are represented in "Infix notation" (Eg. A+B*C+D).

The Polish mathematician Lukasiewicz demonstrated that arithmetic expressions can be represented in prefix or postfix notations.

Types of Notations

Infix notation (Eg. A+B)

Prefix notation or Polish notation (Eg. +AB)

Used by general register organization

Postfix notations or Reverse Polish Notation (Eg. AB+)

Used by stack organization

REVERSE POLISH NOTATION

Reverse Polish notation (RPN) is a method for conveying mathematical expressions without the use of separators such as brackets and parentheses.

In this notation, the operators follow their operands, hence removing the need for the brackets to define evaluation priority.

The operation is read from left to right but execution is done every time an operator is reached.

This notation is suited for computers and calculators since there are fewer characters to track and fewer operations to execute.

Reverse Polish notation is also known as postfix notation

Example:

(A+B)*(C*(D+E)+F)

RPN is AB+CDE+*F+*

EVALUATION OF ARITHMETIC EXPRESSION

Let us consider an arithmetic expression: (3*4) + (5*6).

In Reverse Polish Notation: 34*56*+

Rules

In RPN, the numbers and operators are listed one after another, and an operator always acts on the most recent number in the list.

The numbers can be thought of as forming a stack, and the most recent number goes on the top of the stack.

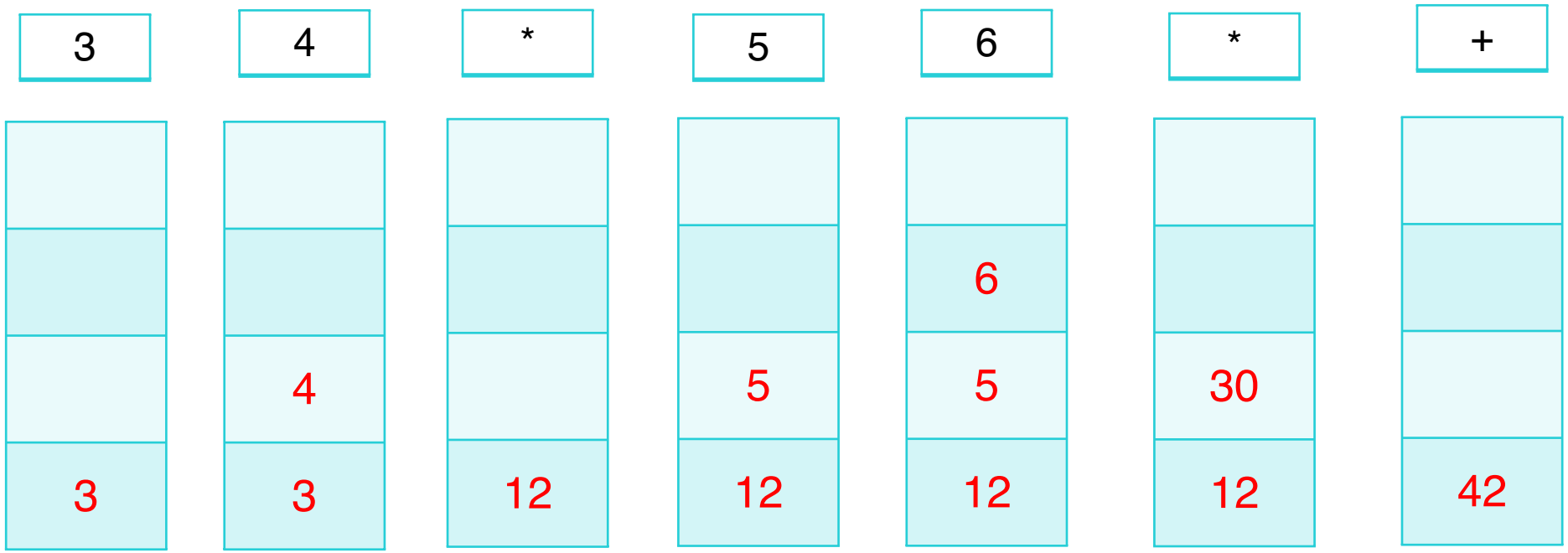
An operator takes the appropriate number of arguments from the top of the stack and replaces them with the result of the operation.

EVALUATION OF ARITHMETIC EXPRESSION

Let us consider an arithmetic expression: (3*4) + (5*6).

In Reverse Polish Notation: 34*56*+

Stack representation



PROCESSOR ORGANIZATION

In general, most processors are organized in one of three ways:

- Single register (Accumulator) organization
  - Basic Computer is a good example
  - Accumulator is the only general-purpose register
- General register organization
  - Used by most modern computer processors
  - Any of the registers can be used as the source or destination for computer operations
- Stack organization
  - All operations are done using the hardware stack
  - For example, an OR instruction will pop the two top elements from the stack, do a logical OR on them, and push the result on the stack

3. INSTRUCTION FORMAT

INSTRUCTION FORMAT

In computer architecture, the instruction format is defined as a standard machine instruction format that can be directly decoded and executed by the central processing unit (CPU).

The instruction format is simply a sequence of bits (binary 0 or 1) contained in a machine instruction that defines the layout of the instruction.

The machine instruction contains the number of bits (patterns of 0 and 1). These bits are grouped together and called fields.

Each field of the machine instruction provides specific information to the CPU regarding the operation to be performed and the location of the data.

An instruction is of various lengths depending upon the number of addresses it contains. Generally, CPU organizations are of three types on the basis of the number of address fields:

1. Single Accumulator organization
2. General register organization`
3. Stack organization

CPU ORGANIZATION

1. Single Accumulator Organization

   All the operations on a system are performed with an implied accumulator register. The instruction format in this type of computer uses one address field.

   For example, the instruction for arithmetic addition is defined by an assembly language instruction ‘ADD.’ Where X is the operand’s address, the ADD instruction results in the operation.

   $AC \leftarrow AC + M[X]$.

   AC is the accumulator register, and M[X] symbolizes the memory word located at address X.

CPU ORGANIZATION

2. General Register Organization

The general register type computers employ two or three address fields in their instruction format. Each address field specifies a processor register or a memory.

An instruction symbolized by ADD R1, X specifies the operation R1 ← R1 + M [X].

This instruction has two address fields: register R1 and memory address X.

CPU ORGANIZATION

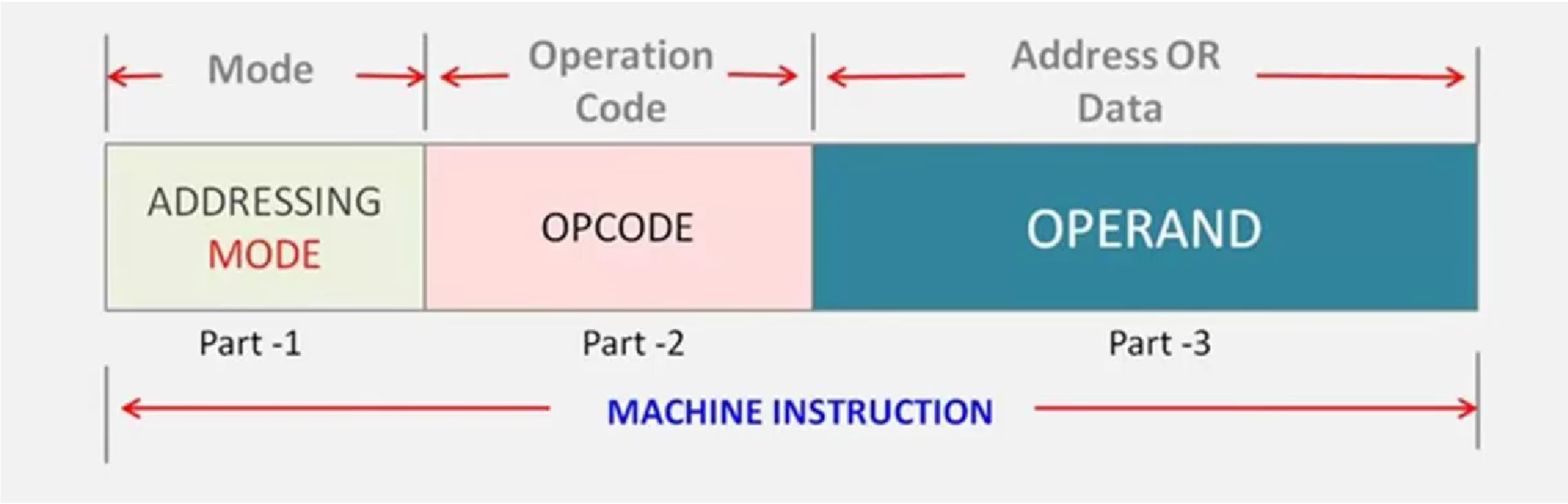
3. Stack Organization

A computer with a stack organization has PUSH and POP instructions that require an address field. Hence, the instruction PUSH X pushes the word at address X to the top of the stack. The stack pointer updates automatically.

In stack-organized computers, the operation type instructions don't require an address field as the operation is performed on the two items on the top of the stack.

INSTRUCTION FORMAT REPRESENTATION

Instruction format is a sequence of bits contained int the machine instruction that defines the layout of an instruction.



INSTRUCTION FORMAT TYPE

The set of instructions that manages the operation codes is called the format of instruction. The design of bits in instruction is supported by the format of instruction.

The length of instruction is generally preserved in multiples of character, which is 8 bits. Depending upon the number of addresses, the format of instruction is of variable length.

Types of instruction format include:

1. Zero(0) Address Instruction format
2. One(1) Address Instruction format
3. Two(2) Address Instruction format
4. Three(3) Address Instruction format

1. ZERO (0) ADDRESS INSTRUCTION FORMAT

This instruction does not have an operand field, and the location of operands is implicitly represented.

The stack-organized computer system supports these instructions.

To evaluate the arithmetic expression, it is required to convert it into reverse Polish notation.

- Example: Consider the below operations, which show how X = (A + B) ∗ (C + D) expression will be written for a stack-organized computer.

TOS: Top of the Stack

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

2. ONE (1) ADDRESS INSTRUCTION FORMAT

The instruction format in which the instruction uses only one address field is called the one address instruction format.

In this type of instruction format, one operand is in the accumulator and the other is in the memory location

It has only one operand

It has two special instructions LOAD and STORE

MODE	OPCODE	OPERAND

Example: The program to evaluate X = (A + B) * (C + D) is as follows:

- LOAD: This is used to transfer the data to the accumulator.
- STORE: This is used to move the data from the accumulator to the memory.
  - M[ ] is any memory location.
  - M[T] addresses a temporary memory location for storing the intermediate result.

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

3. TWO (2) ADDRESS INSTRUCTION FORMAT

The instruction format in which the instruction uses only two address fields is called the two-address instruction format

This type of instruction format is the most commonly used instruction format

As in one address instruction format, the result is stored in the accumulator only, but in the two addresses instruction format the result can be stored in different locations

This type of instruction format has two operands

It requires shorter assembly language instructions

MODE	OPCODE	OPERAND 1	OPERAND 2

- Example: The program to evaluate X = (A + B) * (C + D) is as follows:
  - The MOV instruction transfers the operands to the memory from the processor registers. R1, R2 registers.

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

4. THREE (3) ADDRESS INSTRUCTION FORMAT

The instruction format in which the instruction uses the three address fields is called the three-address instruction format.

The format of a three-address instruction requires three operand fields. These three fields can be either memory addresses or registers.

It requires shorter assembly language instructions

MODE	OPCODE	OPERAND 1	OPERAND 2	OPERAND 3

- Example: The program to evaluate X = (A + B) * (C + D) is as follows:
  - Two processor registers, R1 and R2.
  - The symbol M [A] denotes the operand at memory address symbolized by A. The operand1 and operand2 contain the data or address that the CPU will operate. Operand 3 contains the result's address.

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

4. ADDRESSING MODES

Review

* Instruction :-

I_1 → FC → Instruction (MM → IR)

→ EC → Instⁿ Decode → Operand fetch → Perform Opⁿ → Write Back

Instⁿ

LOAD	1000
OPcode	Operand/Address

Instⁿ

OPcode	Instⁿ	MM Address
		Register

Addressing modes

Ø	cy
1	

(a) Implied/Implicit AM :-

* Operand is available inside the OPcode.

Ex: CMA/CLA

CMC → Compliment Carry

(2) Immediate AM:-

* The operand is available inside the instr?

Ex:

LDI 1000 → immediate value

ADI	2000

→ Immediate value

LOAD #1000 ≡ LDI 1000

3000

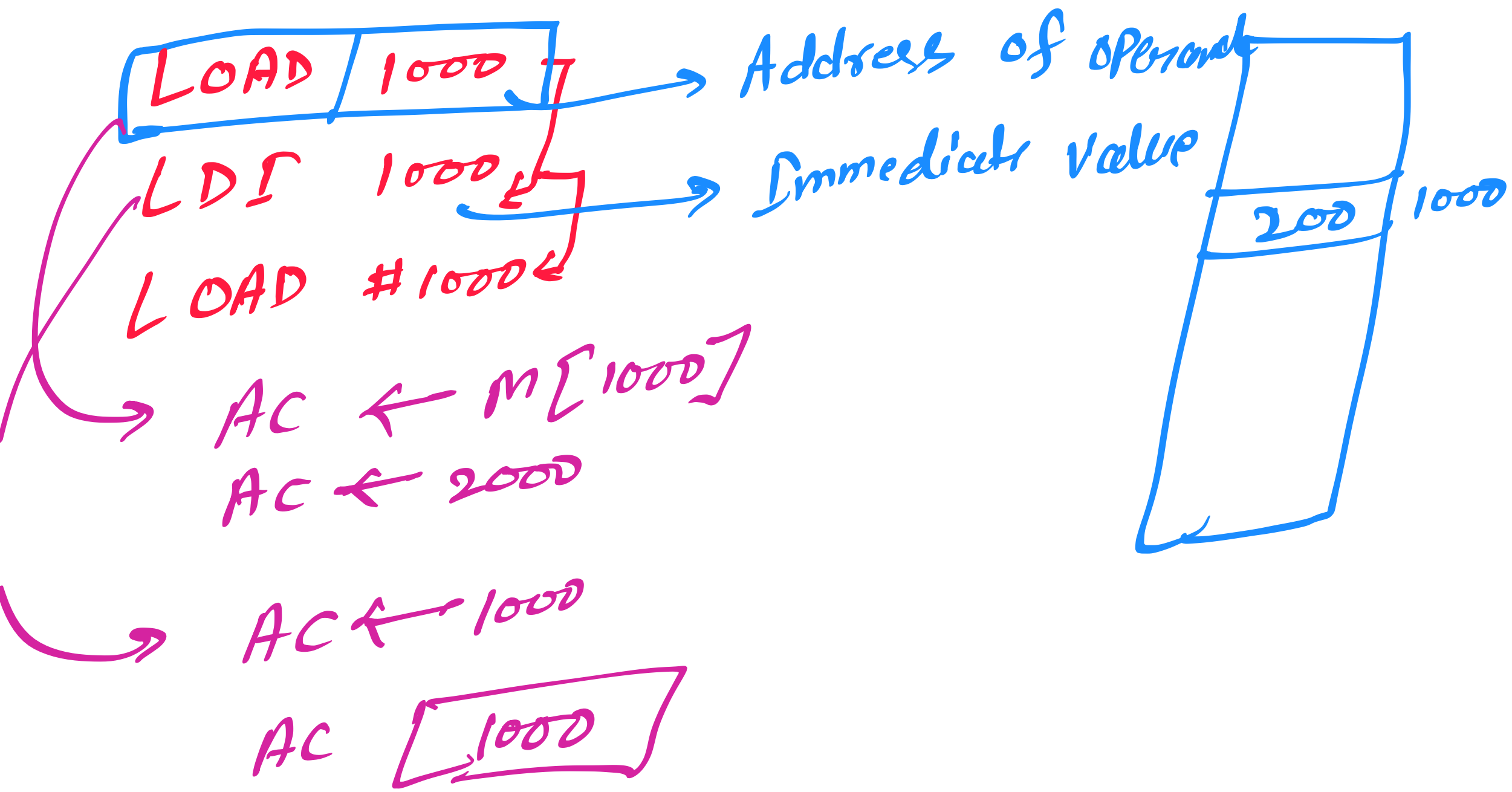
1000

AC

AC ← 1000

AC ← AC + 2000

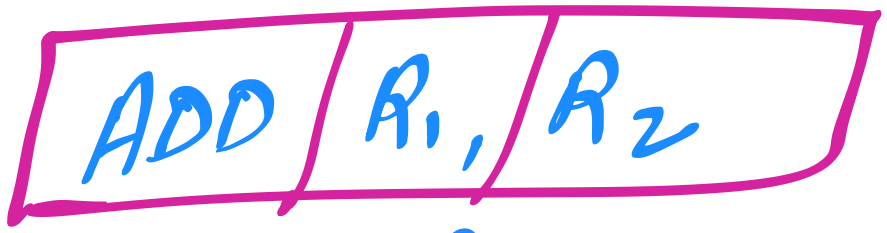
AC ← 1000 + 2000


LOAD 1000
Address of operand
LDI 1000
Immediate value
LOAD #1000
AC ← M[1000]
AC ← 2000
AC ← 1000
AC 1000
2000
1000

3. Register Addressing mode:-

* Operands are available inside the Registers
  - Register is specified inside the instruction

Ex:



MOV R_1

$R_1 \leftarrow [R_1] + [R_2]$

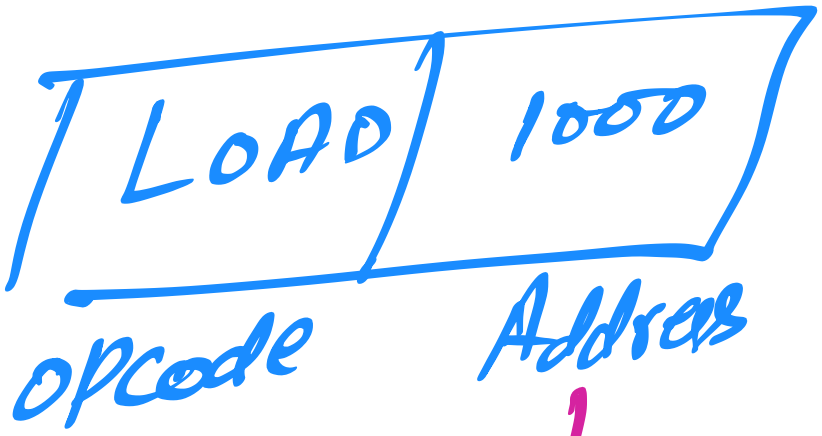
$R_1 \leftarrow 50 + 100$

→ (3) RA

(4) Direct / Absolute Am:-

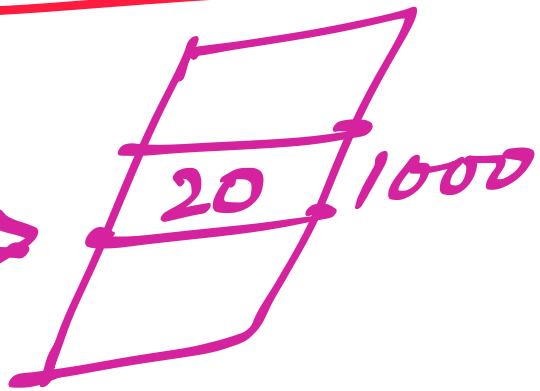
* operands are available in main Memory & the Address of operand (EA) is given in the address part of Instr.

Ex:



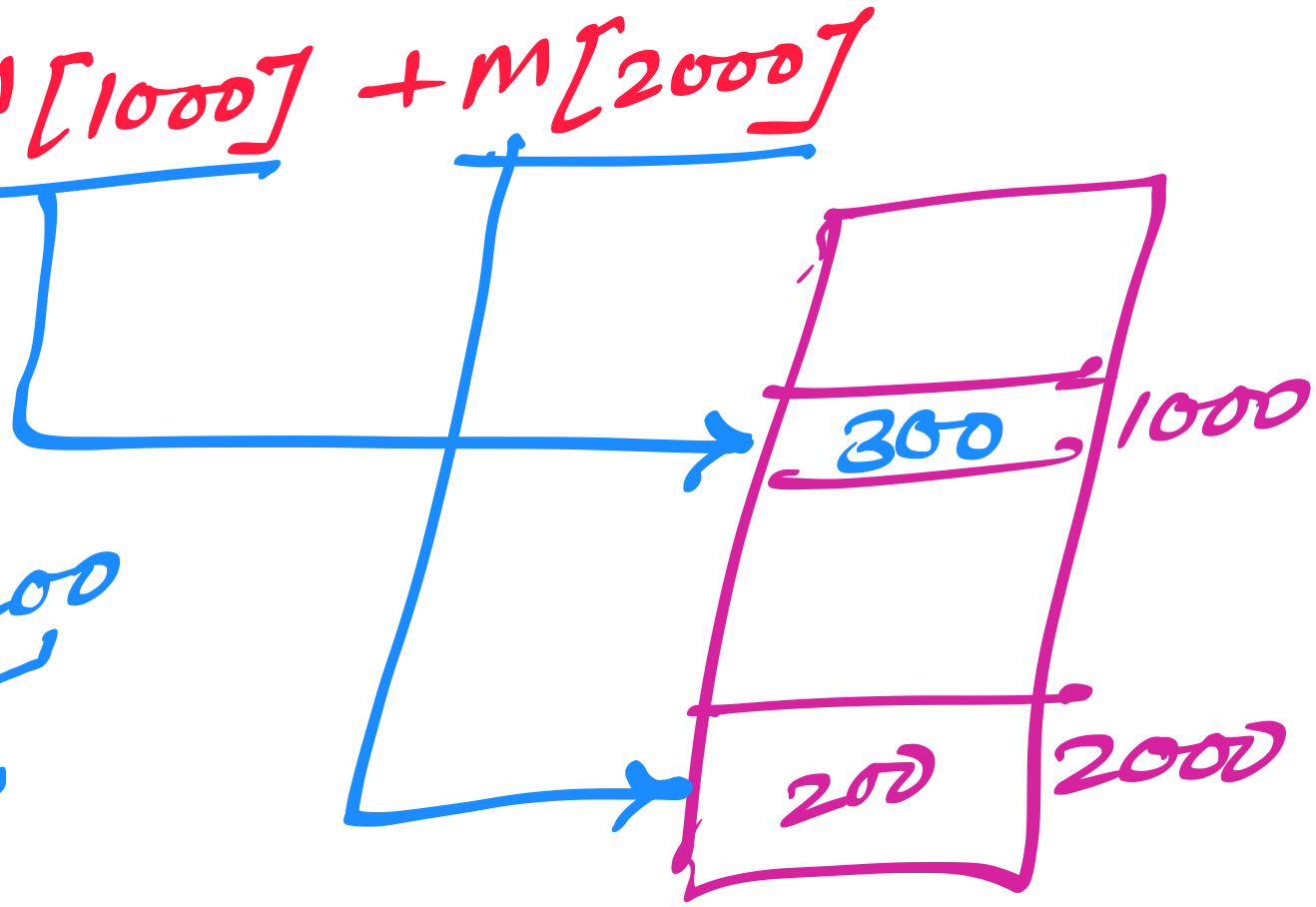
EA = Address [Instr]

AC ← M[1000] ⇒ AC ← 20



Ex. ADD 1000, 2000
S1, D S2

M[1000] ← M[1000] + M[2000]



M[1000] ← 100 + 200

M[1000] ← 300

(5) Indirect AM :-

a. Register Indirect AM

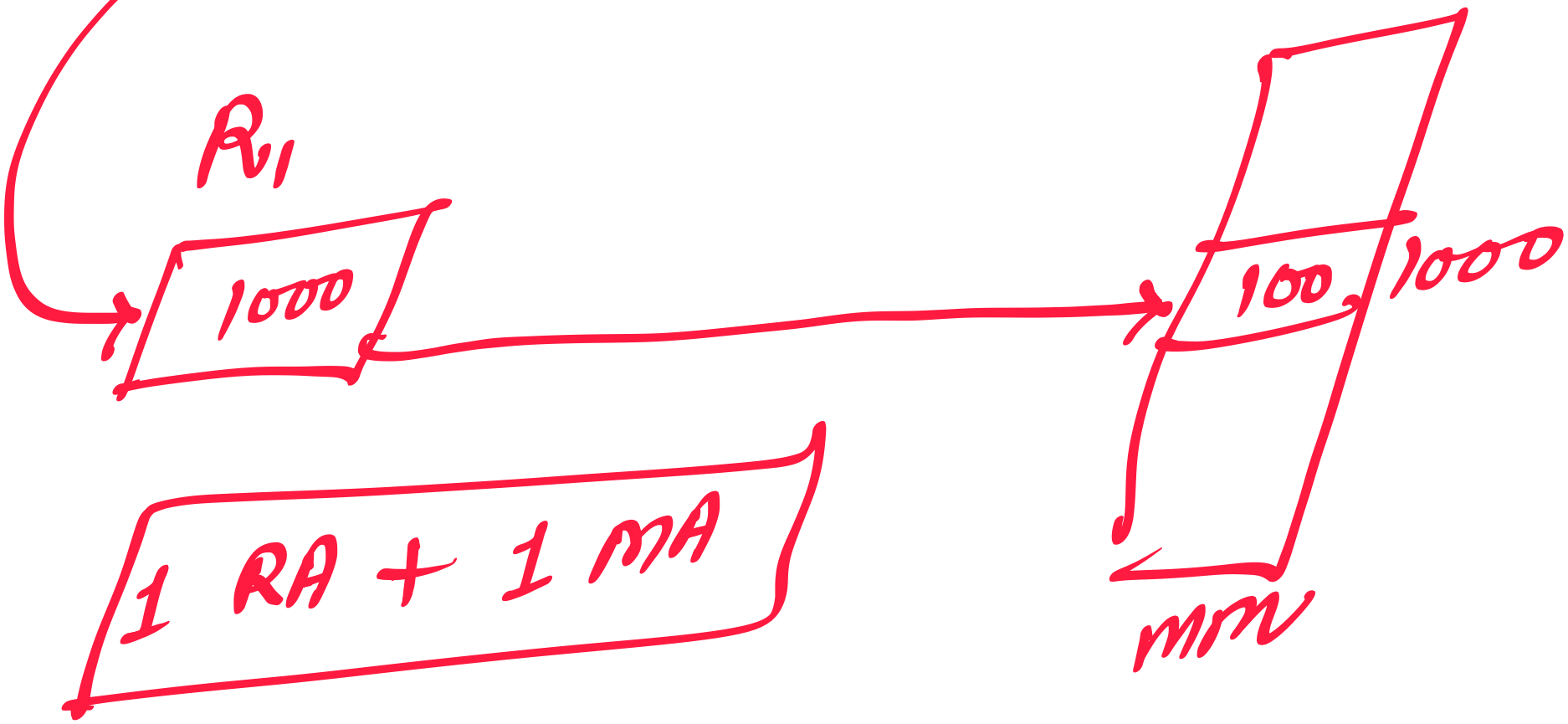
b. Memory Indirect AM

a. Register Indirect AM :-

* The Register contains the address of the operand & this Register is specified inside the instr.

Ex: LOAD @R1

LOAD R1 ⇒ Indirect Addr



Ex: ADD 1000, @2000

S_1 S_2, D

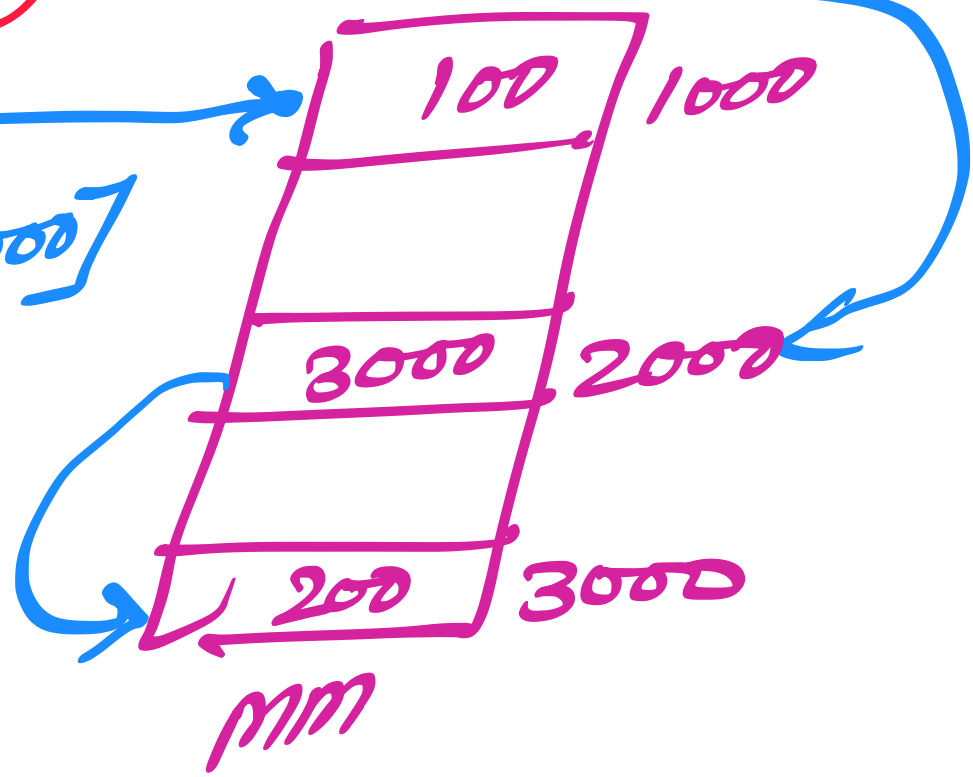
$M[M[2000]] \leftarrow M[1000] + M[M[2000]]$

① ① ②

$100 + M[3000]$

$M[3000] \leftarrow 100 + 200$

4 MA



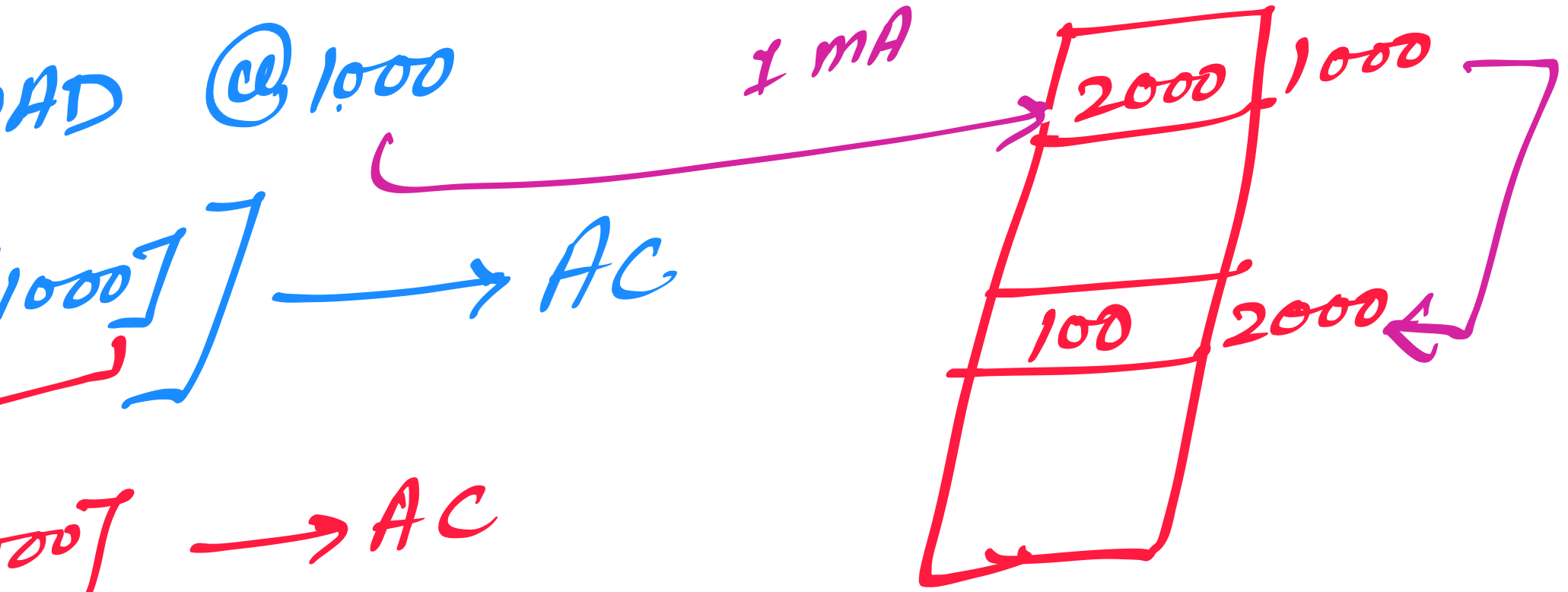
S.b. Memory Indirect AM:-

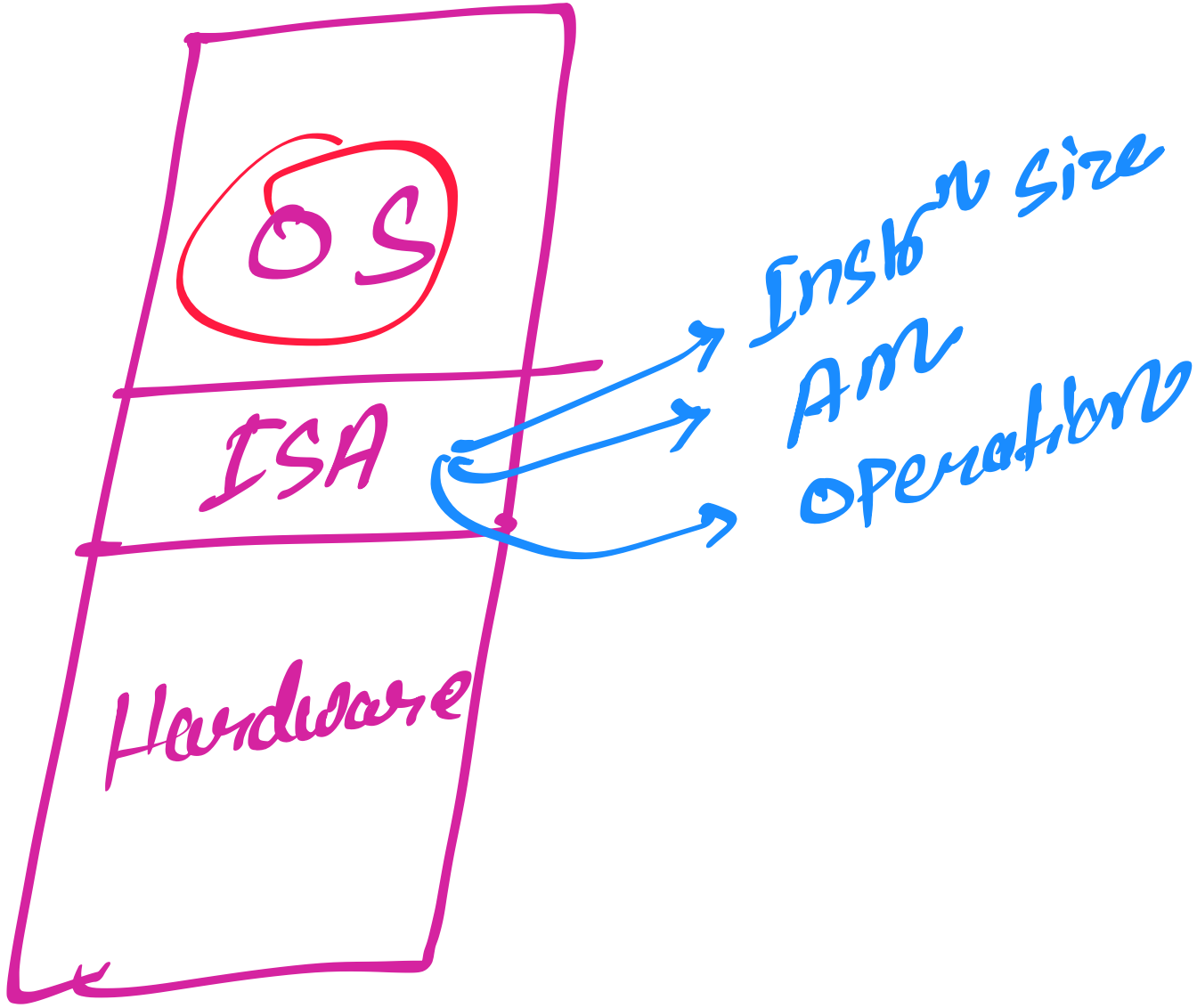
LOAD @1000

M[M[1000]] → AC

M[2000] → AC

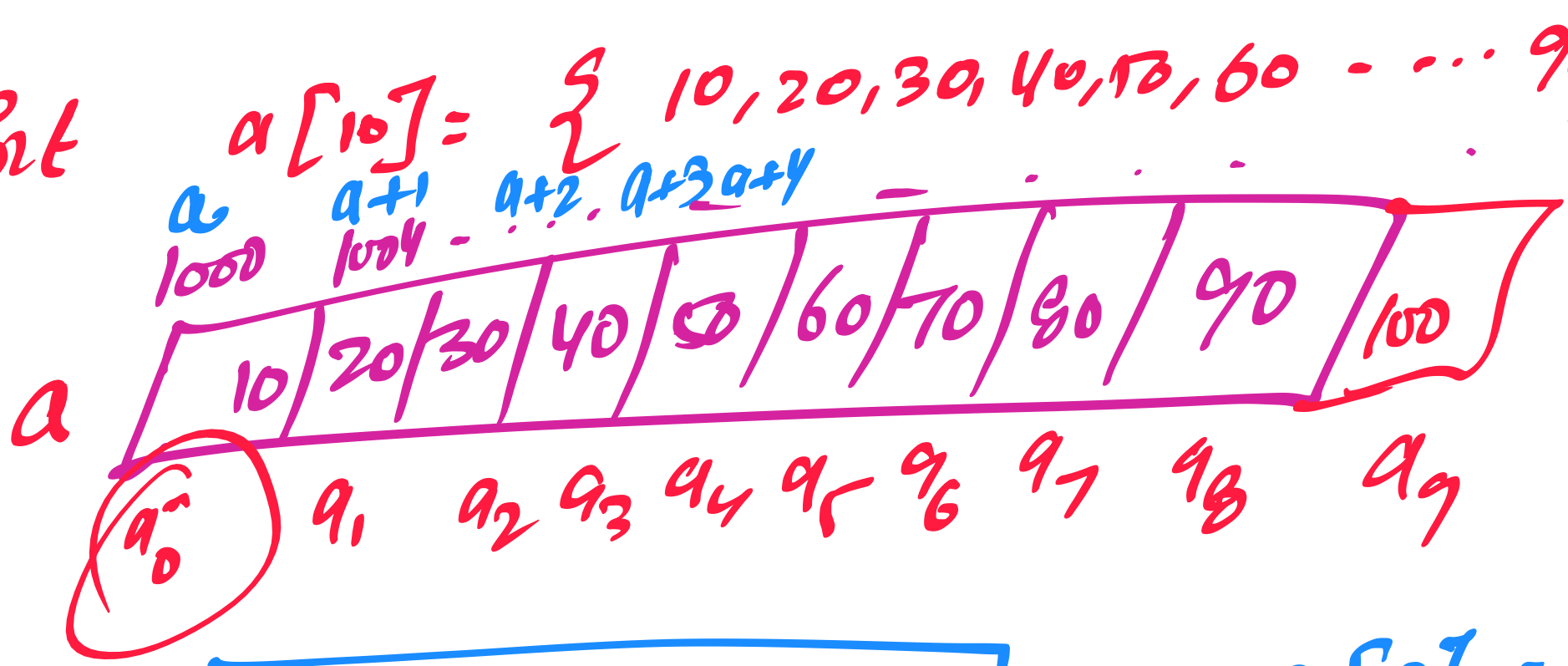
100 → AC




OS
ISA
Hardware
Instn size
Am
operation

(6) Auto Increment / Auto Decrement Am:-

int a[10] = {10, 20, 30, 40, 50, 60 ..... 90}

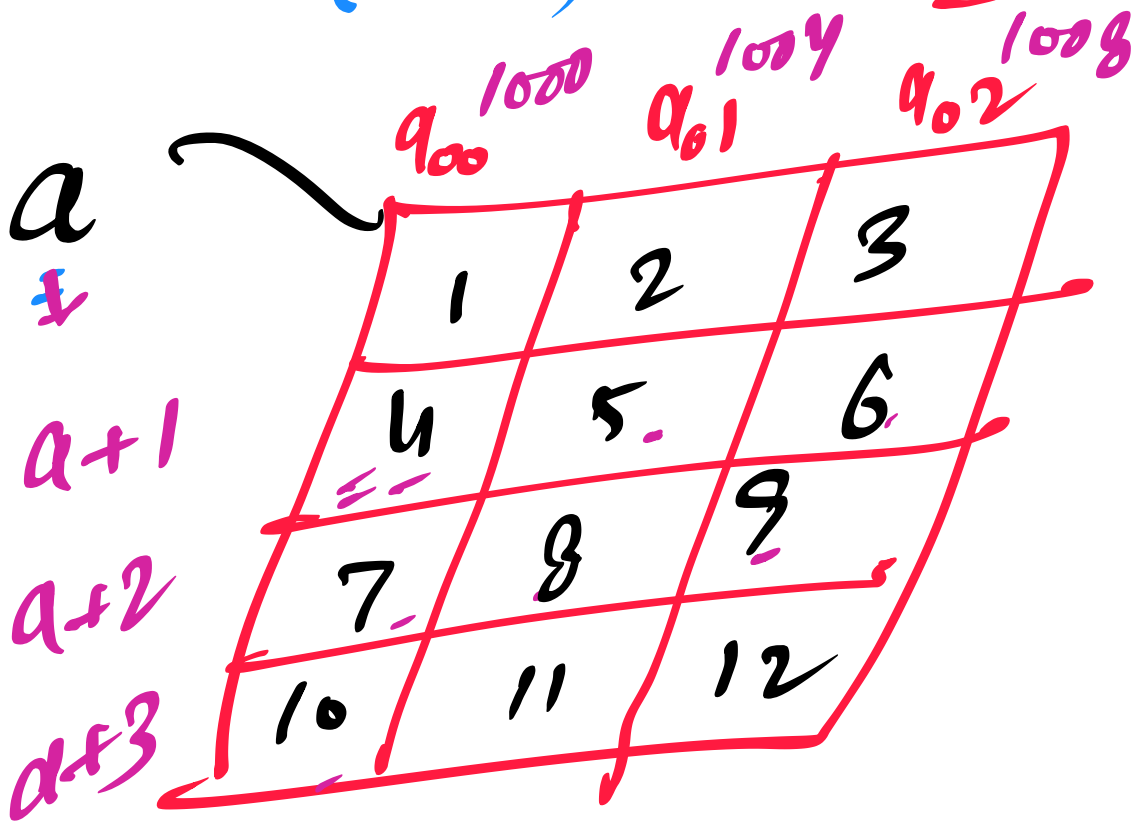


*(a) = 10

a[0] = 10

*a = a[0]

$*(a+1) = a[1]$, $*(a+2) = a[2]$



$*a = 1000$

$**a = 1$

a[4][3]

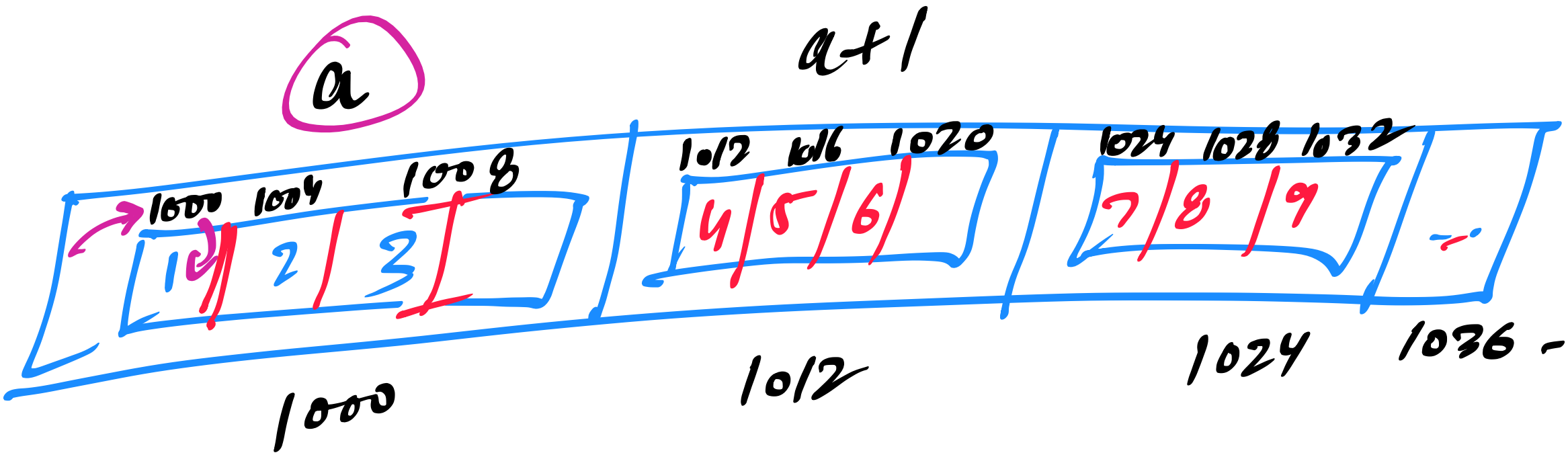
a → first zero address

a = 1000

a+1 = 1012

a+2 = 1024

a+3 = 1036



$a = 1000$

$*a = 1000$

$**a = 1$

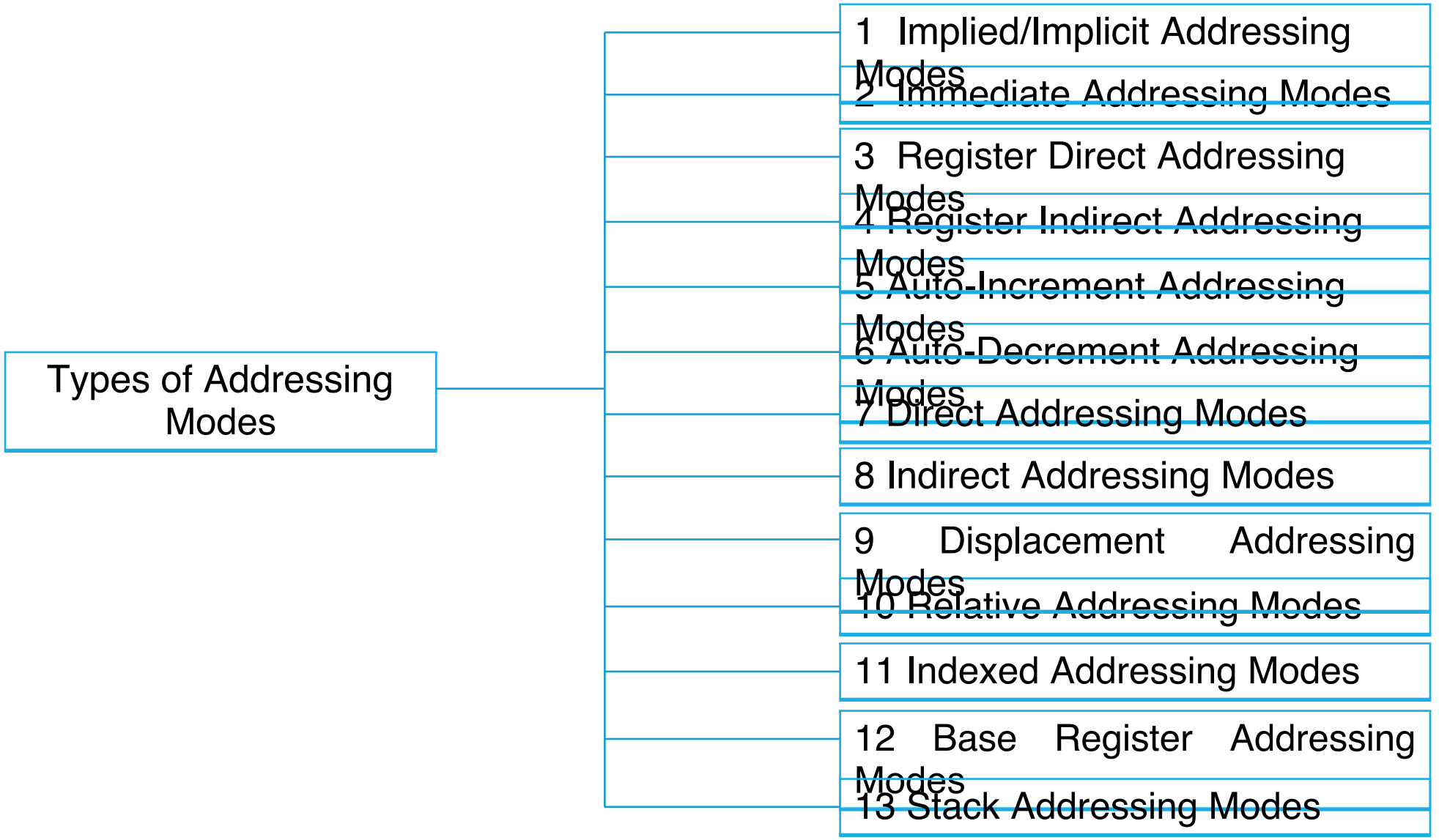
ADDRESSING MODES

The operation field of an instruction specifies the operation to be performed. This operation will be executed on some data which is stored in computer registers or the main memory.

Addressing modes define the rules and mechanisms by which the processor calculates the effective memory address or operand location for data operations.

The way any operand is selected during the program execution is dependent on the addressing mode of the instruction. The purpose of using addressing modes is as follows:

1. To give the programming versatility to the user.
2. To reduce the number of bits in addressing the field of instruction.


Types of Addressing Modes
1 Implied/Implicit Addressing Modes
2 Immediate Addressing Modes
3 Register Direct Addressing Modes
4 Register Indirect Addressing Modes
5 Auto-Increment Addressing Modes
6 Auto-Decrement Addressing Modes
7 Direct Addressing Modes
8 Indirect Addressing Modes
9 Displacement Addressing Modes
10 Relative Addressing Modes
11 Indexed Addressing Modes
12 Base Register Addressing Modes
13 Stack Addressing Modes

1 Implied/Implicit Addressing Modes

2 Immediate Addressing Modes

3 Register Direct Addressing Modes

4 Register Indirect Addressing Modes

5 Auto-Increment Addressing Modes

6 Auto-Decrement Addressing Modes

7 Direct Addressing Modes

8 Indirect Addressing Modes

9 Displacement Addressing Modes

10 Relative Addressing Modes

11 Indexed Addressing Modes

12 Base Register Addressing Modes

13 Stack Addressing Modes

1. Implied/Implicit Addressing Modes:

Address of the operands is specified implicitly in the definition of the instruction.

All registers, reference the instructions that use an accumulator, and Zero-address instructions in a stack-organized computer are implied addressing mode instructions.

EA = AC, or EA = Stack[SP]

Example: CMA, CLA, PUSH, POP, etc.

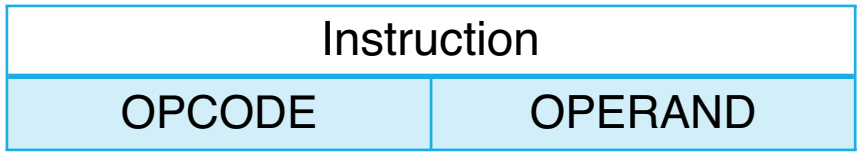
2. Immediate Addressing Modes

The operand is defined in the instruction itself which is used to perform a specified operation.

Instruction has an operand field instead of an address field.

Help initialize registers to a constant value.

Example: ADD 8 will increment the value stored in the accumulator by 8.



1 Implied/Implicit Addressing Modes

2 Immediate Addressing Modes

3 Register Direct Addressing Modes

4 Register Indirect Addressing Modes

5 Auto-Increment Addressing Modes

6 Auto-Decrement Addressing Modes

7 Direct Addressing Modes

8 Indirect Addressing Modes

9 Displacement Addressing Modes

10 Relative Addressing Modes

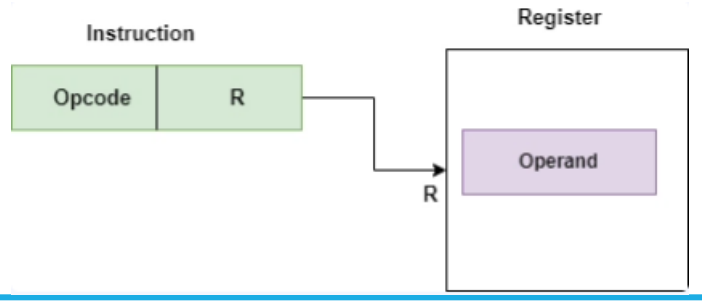
11 Indexed Addressing Modes

12 Base Register Addressing Modes

13 Stack Addressing Modes

3. Register Direct Addressing Modes:

Address specified in the instruction is the register address.
Designated operand need to be in a register.
Shorter address than the memory address. Faster to acquire an operand than the memory addressing.
Saving address field in the instruction.
EA = IR(R) (IR(R): Register field of IR)

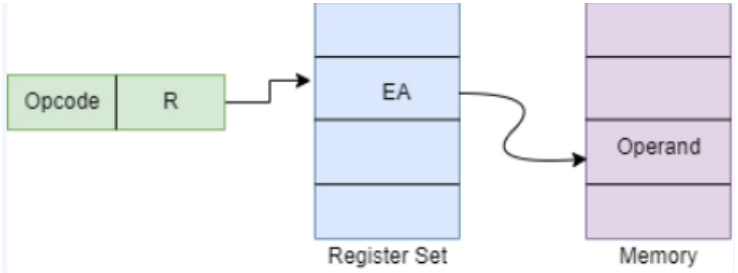


Example: MOV R1, 35H

4. Register Indirect Addressing Modes

The instruction defines a register in the CPU that stores the effective address of the operand in memory.
Only one reference to the memory is required to fetch the operand. The specified register contains the address of the operand instead of the operand.

Example: MOV R1, [R2]



1 Implied/Implicit Addressing Modes

2 Immediate Addressing Modes

3 Register Direct Addressing Modes

4 Register Indirect Addressing Modes

5 Auto-Increment Addressing Modes

6 Auto-Decrement Addressing Modes

7 Direct Addressing Modes

8 Indirect Addressing Modes

9 Displacement Addressing Modes

10 Relative Addressing Modes

11 Indexed Addressing Modes

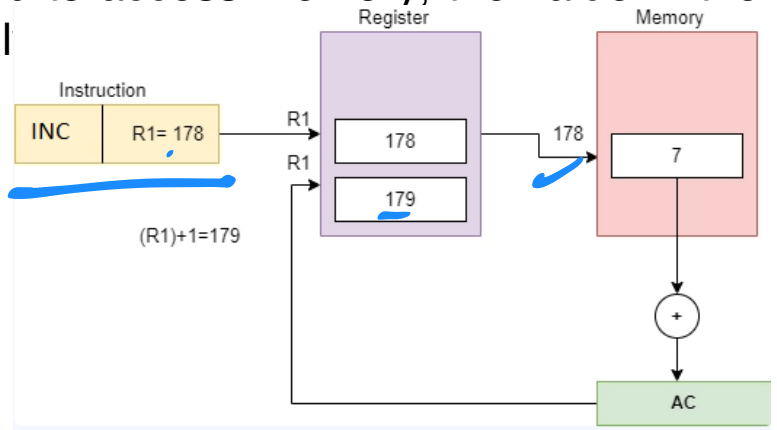
12 Base Register Addressing Modes

13 Stack Addressing Modes

5. Auto-Increment Addressing Modes:

When the address in the register is used to access memory, the value in the register is incremented by 1 automaticall
It follows a post-increment approach.
EA = content of the register

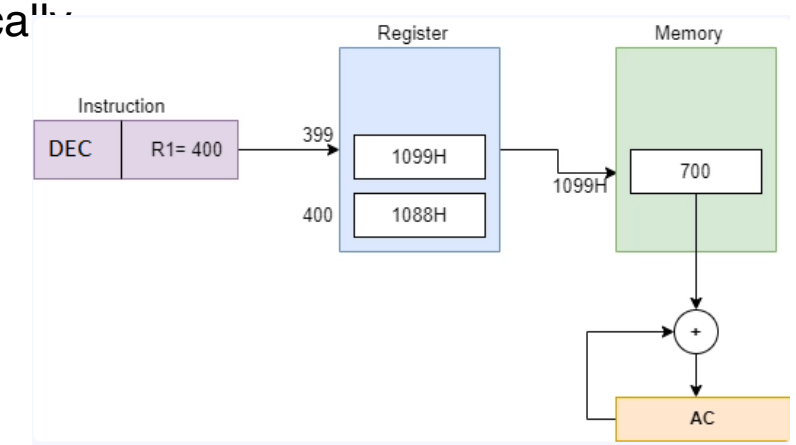
Example: EA = 178



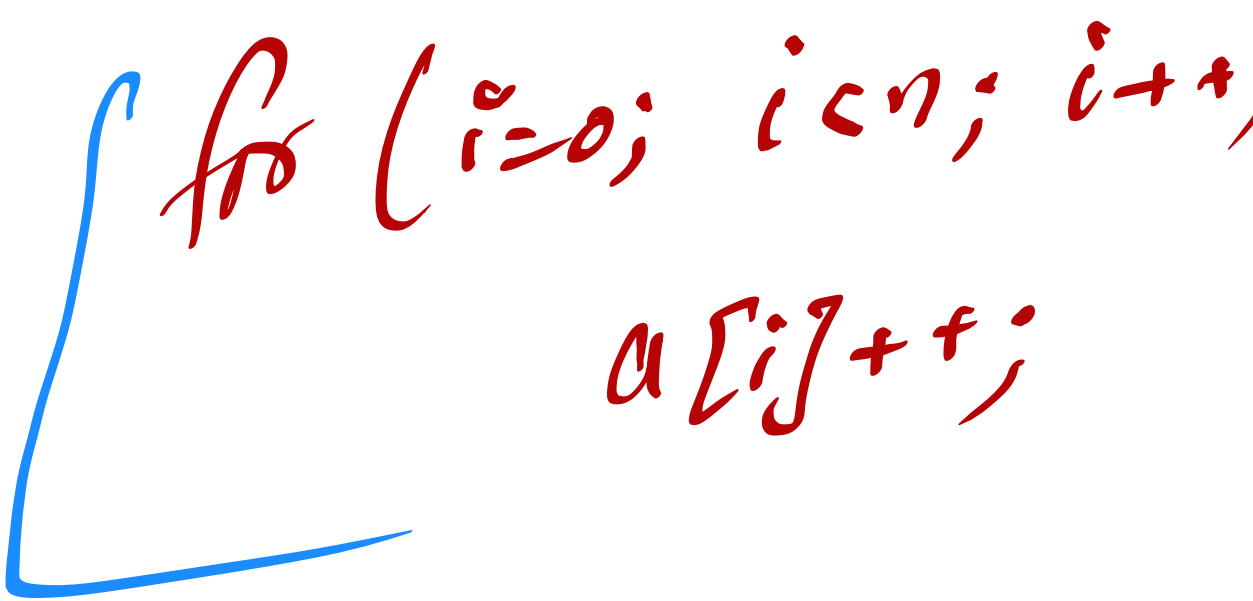
6. Auto-Decrement Addressing Modes

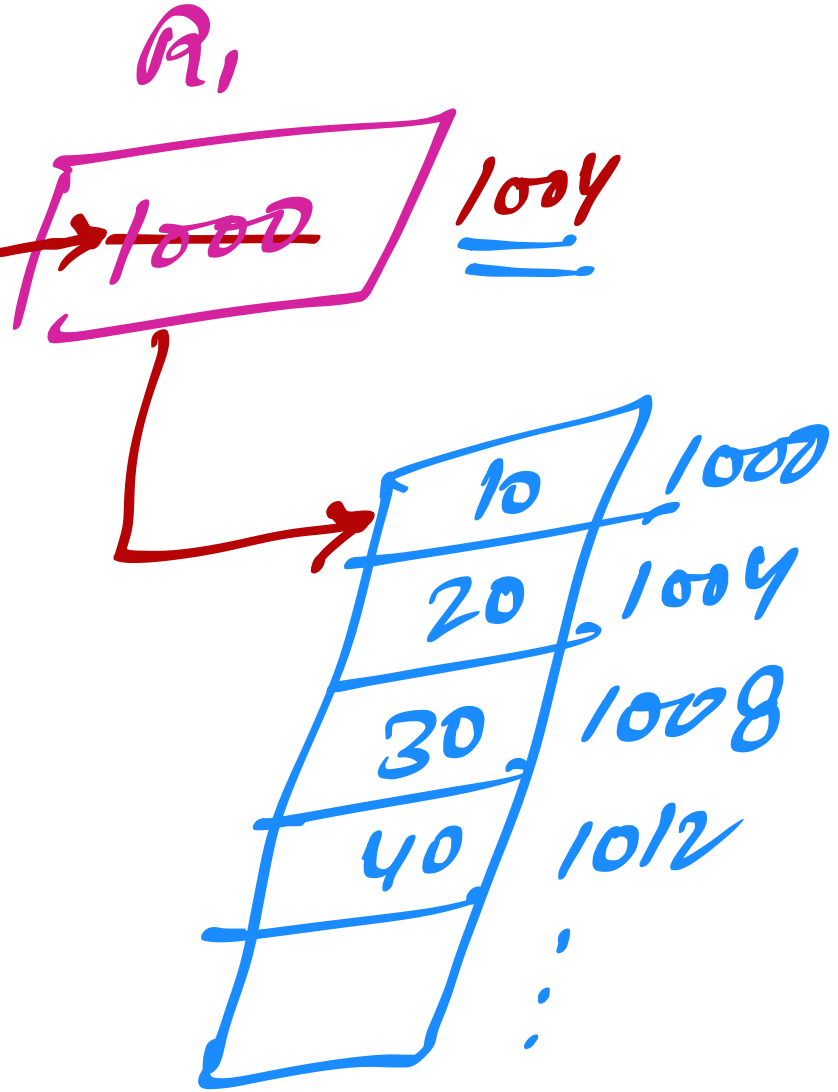
When the address in the register is used to access memory, the value in the register is decremented by 1 automatica
It follows a pre-decrement approach.
EA = content of the register

Example: EA = 399



INC R1





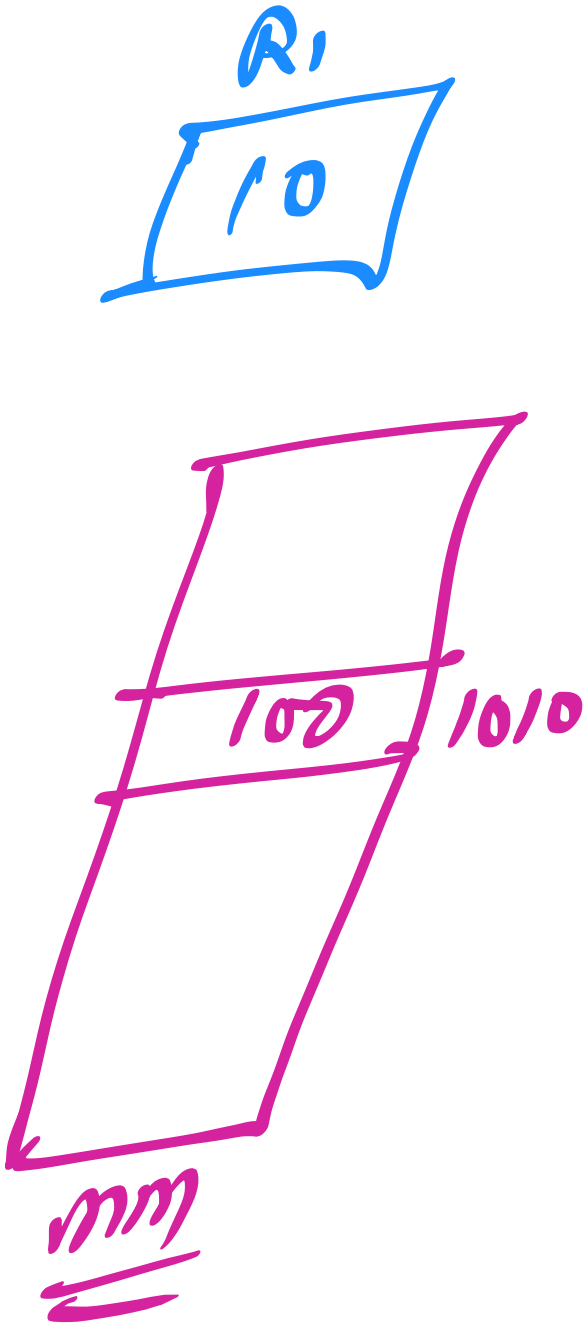
(8) Displacement Add:-

opcode	Address	Reg

LOAD 1000, R_1

EA = 1000 + [R_1]

EA = 1010



(8) Relative Am: / PC Relative Am: —

I_1: LOAD 17 1000

I_2: ADD R_1 R_2 1004 ← PC

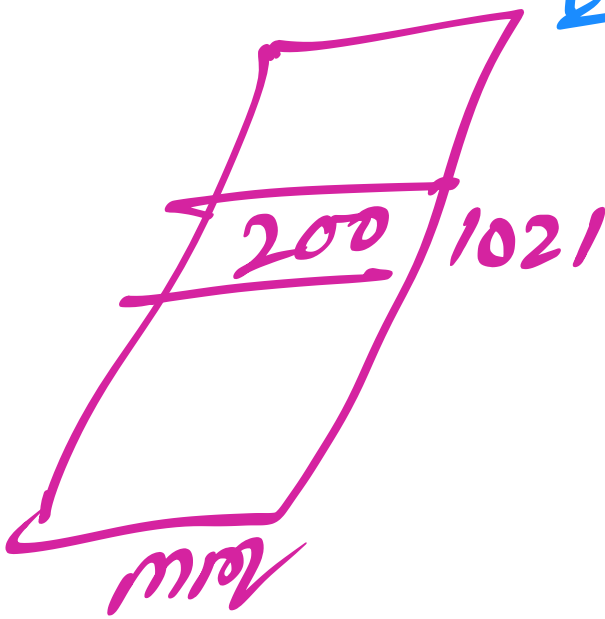
PC

1004

EA = PC + 17

EA = 1004 + 17

EA = 1021



1 Implied/Implicit Addressing Modes

2 Immediate Addressing Modes

3 Register Direct Addressing Modes

4 Register Indirect Addressing Modes

5 Auto-Increment Addressing Modes

6 Auto-Decrement Addressing Modes

7 Direct Addressing Modes

8 Indirect Addressing Modes

9 Displacement Addressing Modes

10 Relative Addressing Modes

11 Indexed Addressing Modes

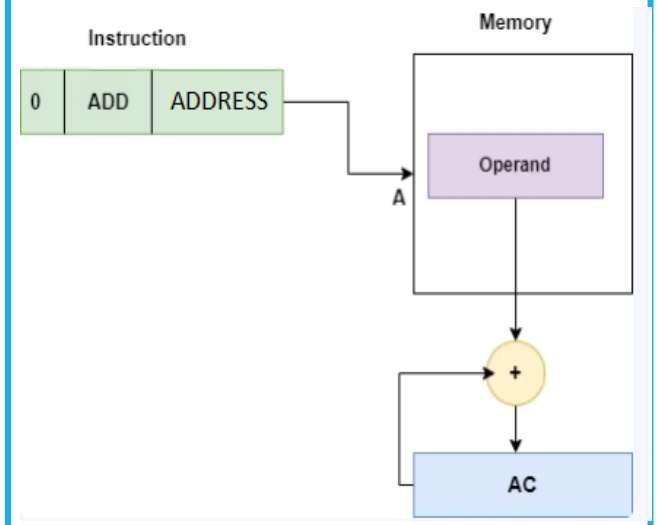
12 Base Register Addressing Modes

13 Stack Addressing Modes

7. Direct Addressing Modes:

The effective address of the operand resides in the address field of the instruction.
The operand resides in the memory, and the address field of the instruction gives its address.
One references to the memory is required to fetch the operand.
Also known as absolute addressing mode.

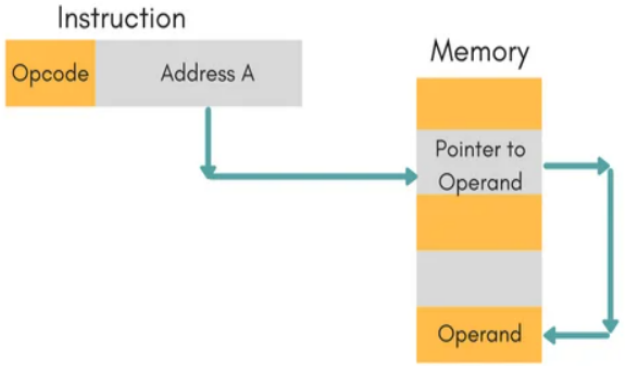
Example: ADD, [1000H]



8. Indirect Addressing Modes

The address field of the instruction gives the address of the memory location that contains the effective address of the operand.
Two references to the memory are required to fetch the operand
This addressing mode slows down the execution as it requires multiple memory lookups to find the operand.

Example: ADD [[1000H]]



1 Implied/Implicit Addressing Modes

2 Immediate Addressing Modes

3 Register Direct Addressing Modes

4 Register Indirect Addressing Modes

5 Auto-Increment Addressing Modes

6 Auto-Decrement Addressing Modes

7 Direct Addressing Modes

8 Indirect Addressing Modes

9 Displacement Addressing Modes

10 Relative Addressing Modes

11 Indexed Addressing Modes

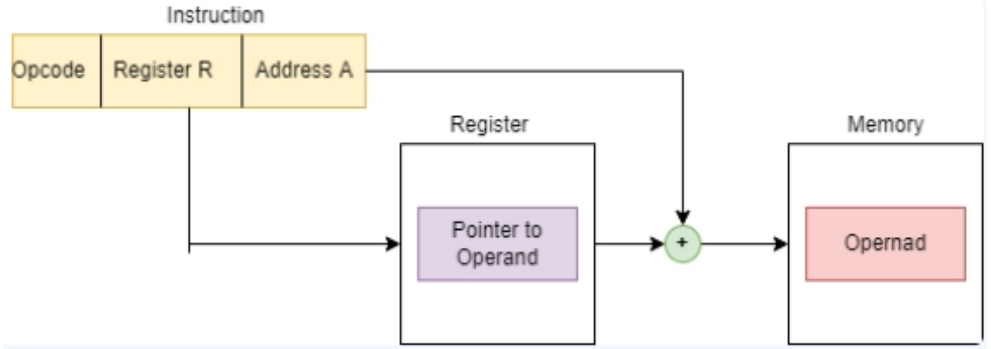
12 Base Register Addressing Modes

13 Stack Addressing Modes

9. Displacement Addressing Modes:

The displacement is added to the instruction's address part to obtain the effective address of the operand.

EA = A + (R) Here, the address field holds two values, A: Base value R: displacement value.



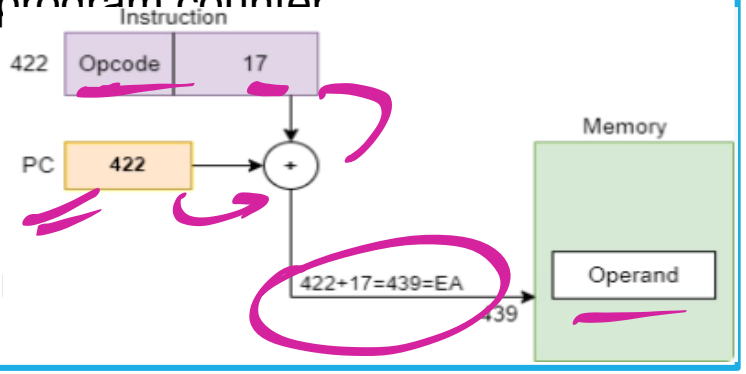
Example:

MOV R1, [Address Field+09H]

10. Relative Addressing Modes

This mode is another version of the displacement address mode. The program counter's content is added to the instruction's address part to obtain the effective address.

EA = A + (PC) Here, EA: Effective address, PC: program counter



Example: MOV R1, [PC + Address Field]

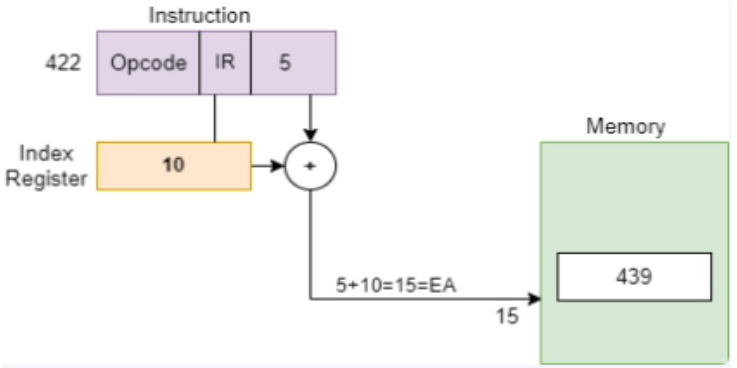
The instruction's address is usually a signed numbe

11. Indexed Addressing Modes:

The index register's content is added to the instruction's address to obtain the effective address.

EA = content of index register (XI) + Instruction address part

Example: MOV R1, [XI + Address Field].



12. Base Register Addressing Modes

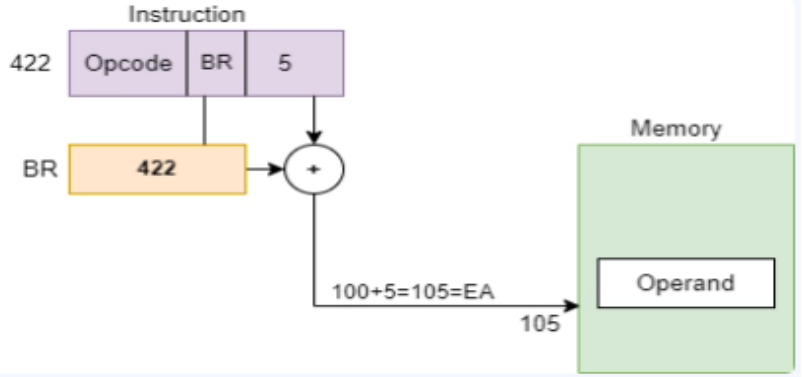
This mode is another version of the displacement address mode. To obtain the effective address, the base register's content is added to the instruction's address.

EA = A + (R)

A: Instruction address,

R: Pointer to the base address.

Example: MOV R1, [BX + Address Field]

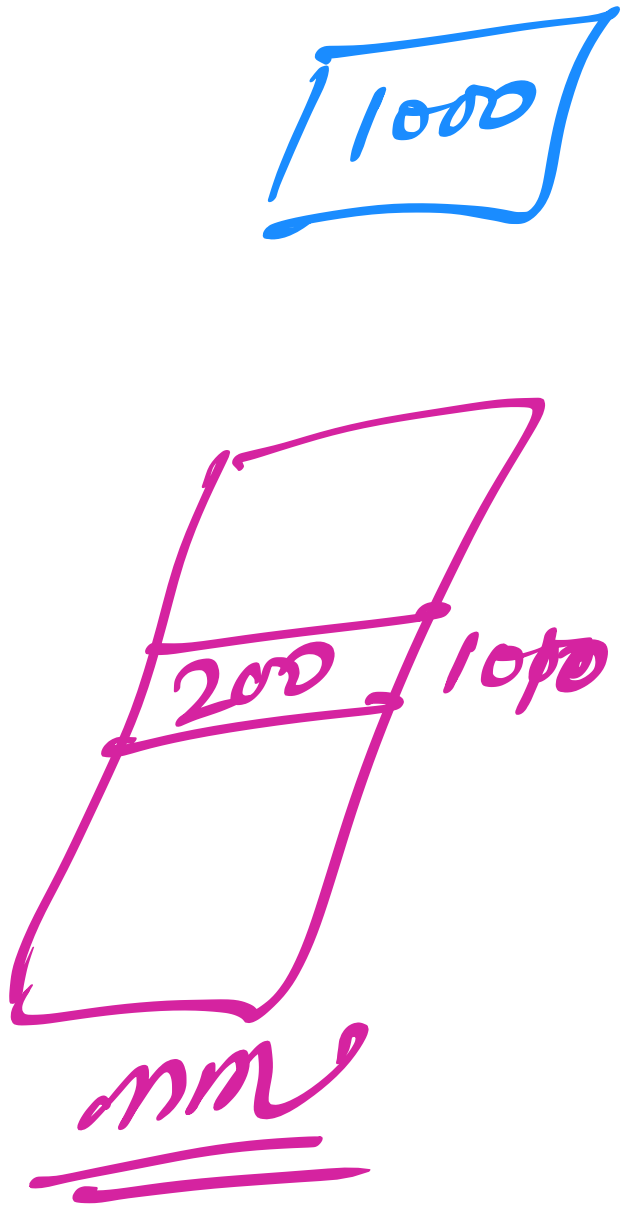


(10) Indexed Add:-

LOAD R_i, 10

EA = R_i + 10

EA = 1000 + 10 = 1010



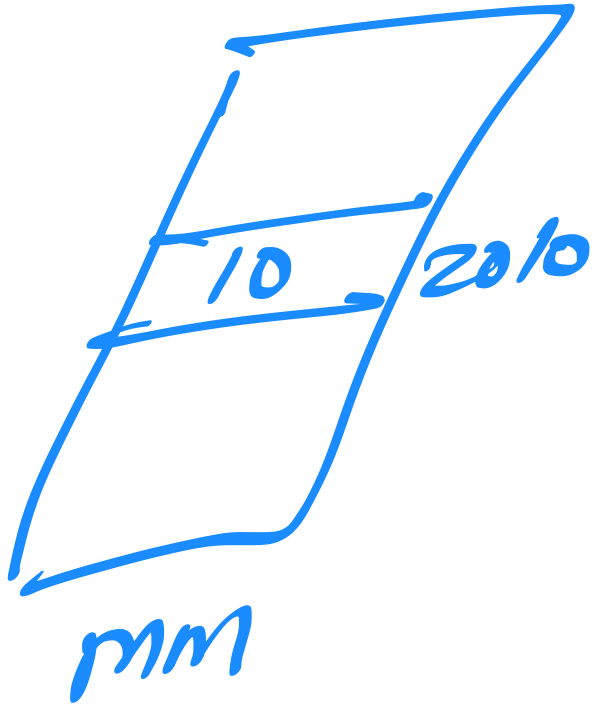
(11) Base Add:-

LOAD BR, 10

AC ← M[BR+10]

AC ← M[2000+10]

AC ← M[2010]



1 Implied/Implicit Addressing Modes

2 Immediate Addressing Modes

3 Register Direct Addressing Modes

4 Register Indirect Addressing Modes

5 Auto-Increment Addressing Modes

6 Auto-Decrement Addressing Modes

7 Direct Addressing Modes

8 Indirect Addressing Modes

9 Displacement Addressing Modes

10 Relative Addressing Modes

11 Indexed Addressing Modes

12 Base Register Addressing Modes

13 Stack Addressing Modes

13. Stack Addressing Modes:

In this mode, the operand is at the top of the stack.

For example: ADD, this instruction will POP the top two items from the stack, add them, and will then PUSH the result to the top of the stack.

It helps in reducing the number of bits in the instruction's addressing field.

It facilitates pointers, indexing of data, and counters for loop controls.

APPLICATIONS OF ADDRESSING MODES

Addressing Mode	Applications
Immediate Addressing Mode	Initialize the register to a constant value.
Direct Addressing Modes Register Direct Addressing Mode	Helps access static data and implement variables.
Indirect Addressing Modes Register Indirect Addressing Mode	Helps implement pointers and pass arrays as parameters.
Relative Addressing Mode	Helps in program relocation at runtime. And in changing the sequence of instructions during execution.
Index Addressing Mode	Helps in the array and record implementation.
Base Register Addressing Mode	Helps in writing relocatable code and handling recursive procedures.
Auto-Increment Addressing Mode Auto-Decrement Addressing Mode	Helps implements loops and stacks.

ADVANTAGES OF ADDRESSING MODES

They improve performance by efficiently utilizing the CPU cache and reducing the memory read latency.

Addressing Modes are used for implementing complex data structures in memory as they provide mechanisms such as indexing.

Program sizes can be reduced drastically as the code can be compacted which allows faster execution of instructions.

Addressing modes give you the flexibility to use different ways of specifying the address of operands in your instructions.

EXAMPLE: ADDRESSING MODES

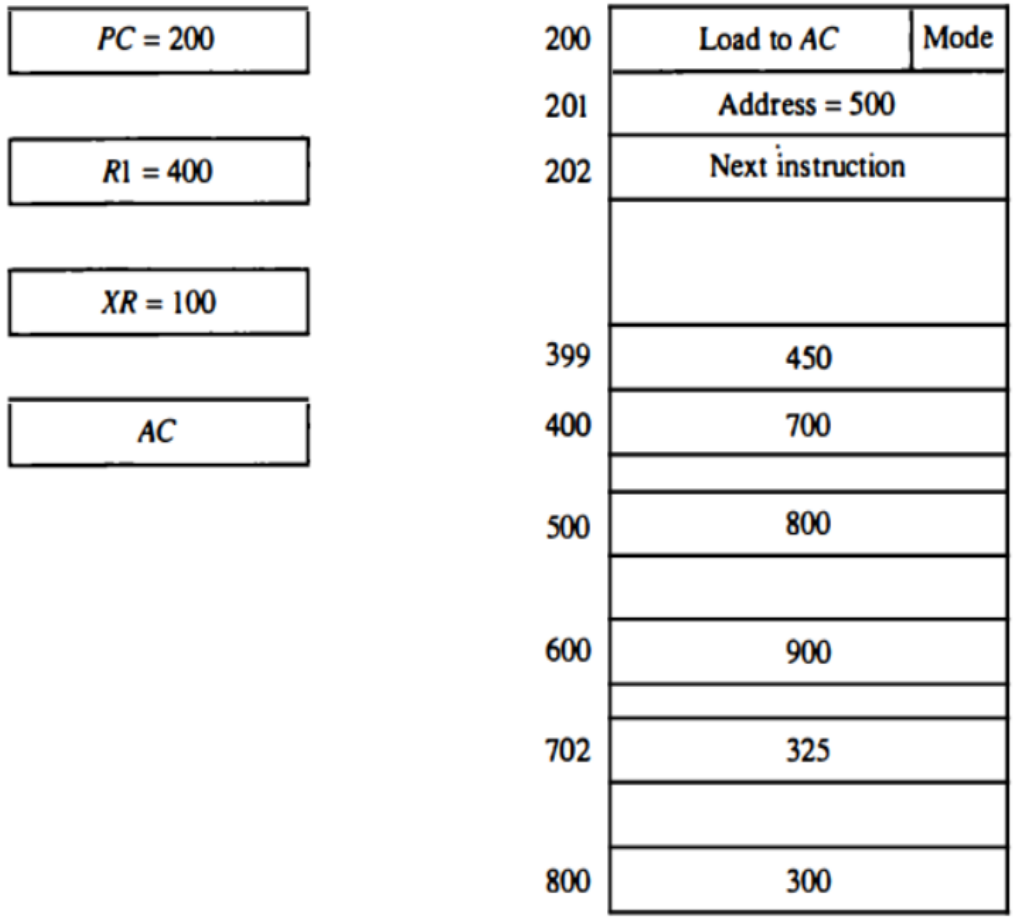
To show the differences between the various modes, we will show the effect of the addressing modes on the instruction defined in Fig.

The two-word instruction at addresses 200 and 201 is a "load to AC" instruction with an address field equal to 500.

The first word of the instruction specifies the operation code and mode, and the second word specifies the address part.

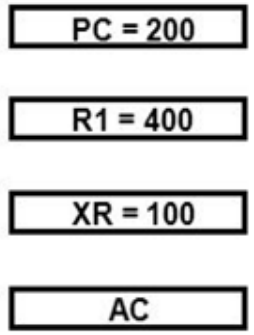
PC has the value 200 for fetching this instruction. The content of processor register R1 is 400, and the content of an index register or base register XR is 100.

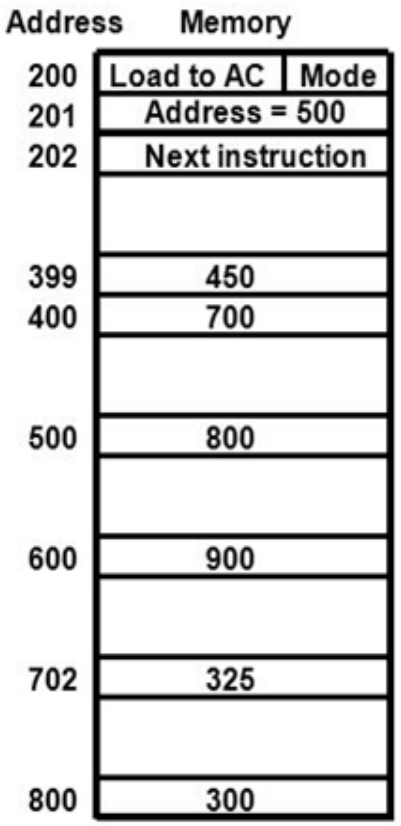
AC receives the operand after the instruction is executed. The figure lists a few pertinent addresses and shows the memory content at each of these addresses.



EXAMPLE: ADDRESSING MODES

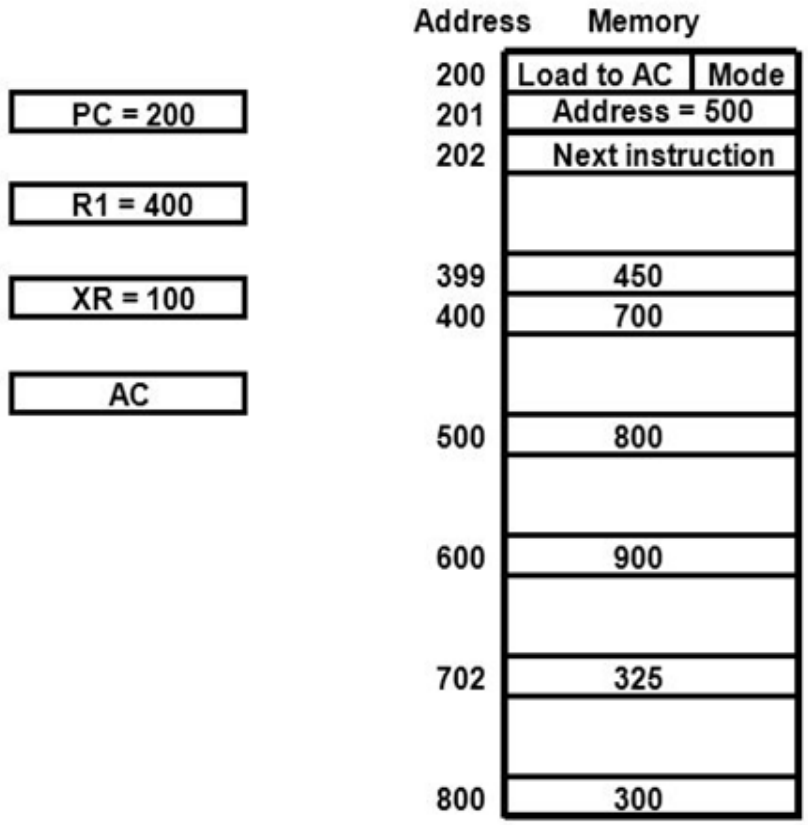
Addressing Mode	Instruction	Effective Address	Content of AC
Immediate AM	MOV A, 500		
Register AM	MOV A, R1		
Register Indirect AM	MOV A, [R1]		
Auto-Increment AM	INR A, [R1]		
Auto-Decrement AM	DCR A, [R1]		
Direct AM	MOV A, 500		
Indirect AM	MOV A, [[500]]		
Relative AM (EA=PC + Address Field)	MOV A, [PC + Address Field]		
Base Register AM (EA=XR + Address Field)	MOV A, [XR + Address Field]		
Indexed Register AM (EA=XR + Address Field/)	MOV A, [XR + Address Field]		





EXAMPLE: ADDRESSING MODES

Addressing Mode	Effective Address	Content of AC
Immediate AM	201	500
Register AM	--	400
Register Indirect AM	400	700
Auto-Increment AM	400	700
Auto-Decrement AM	399	450
Direct AM	500	800
Indirect AM	800	300
Relative AM (EA=PC + Address Field)	702	325
Base Register AM (EA=XR + Address Field/)	600	900
Indexed Register AM (EA=XR + Address Field/)	600	900



EXAMPLE: ADDRESSING MODES

Consider a 16-bit processor in which the following contents appear in main memory, starting at location 200:

The first part of the first instruction indicates that this instruction loads a value into an accumulator. The mode field specifies the addressing mode and, if appropriate a source register; assume that when used, the source register is R1, which has a value of 400. There is also a base register that contains the value 100. The value 500 in location 201 may be part of the address calculation. Assume that location 399 contains the value 999, location 400 contains a value 1000, and so on. Determine the effective address and operand to be loaded for the following address modes: Direct, Indirect, Immediate, Register Direct, and Register Indirect

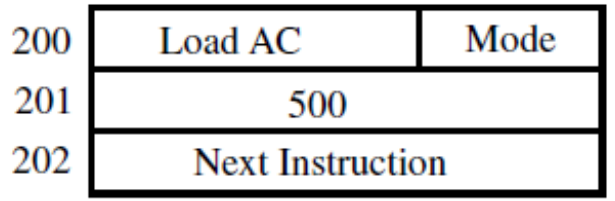


Figure 1: Three locations of main memory.

EXAMPLE: ADDRESSING MODES

[redacted] Base register, used as displacement has value = 100. R_1 has value 400. memory location 399 has value 999, 400 has value 1000, ..., x has value $x + 600$.

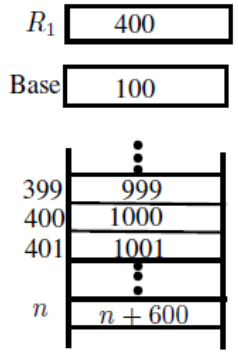


Figure 2: EA and Operand Computation.

i. Direct: EA=500, operand=1100
ii. Indirect: EA=1100, operand=1700
iii. Immediate: EA=201, operand=500
[redacted]
v. Register: EA=Register R_1, operand=400
vi. Register Indirect: EA=400, operand=1000

Review

Instⁿ: | AM | opcode | operand |

Instⁿ cycle : Fetch cycle, Execution cycle

Decode operand PO
Fetch

Instⁿ formats:

| opcode | op1 | op2 |
| 8 | 8 | 8 |

1. Zero Address

2. One "

3. Two "

4. Three "

CPU Org":-

1. Stack Org":-
2. Single Register (AC) Org"
3. General Register Org".

Addressing modes

Effective Address / Address of operand

1. Implicit Addr
2. Immediate Addr
3. Register Addr
4. Direct / Absolute Addr

Opcode | Index | Reg | Mem

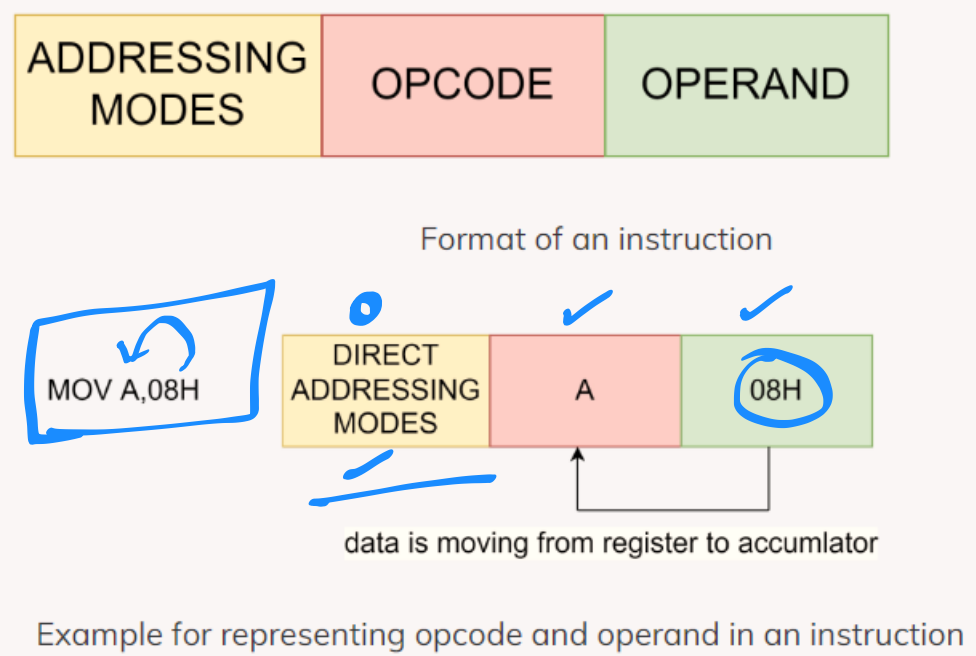
5. Indirect AM
6. Auto Inc/Dec AM
7. Displacement AM
8. Relative AM
9. Indexed AM
10. Base AM

5. DATA TRANSFER AND MANIPULATION

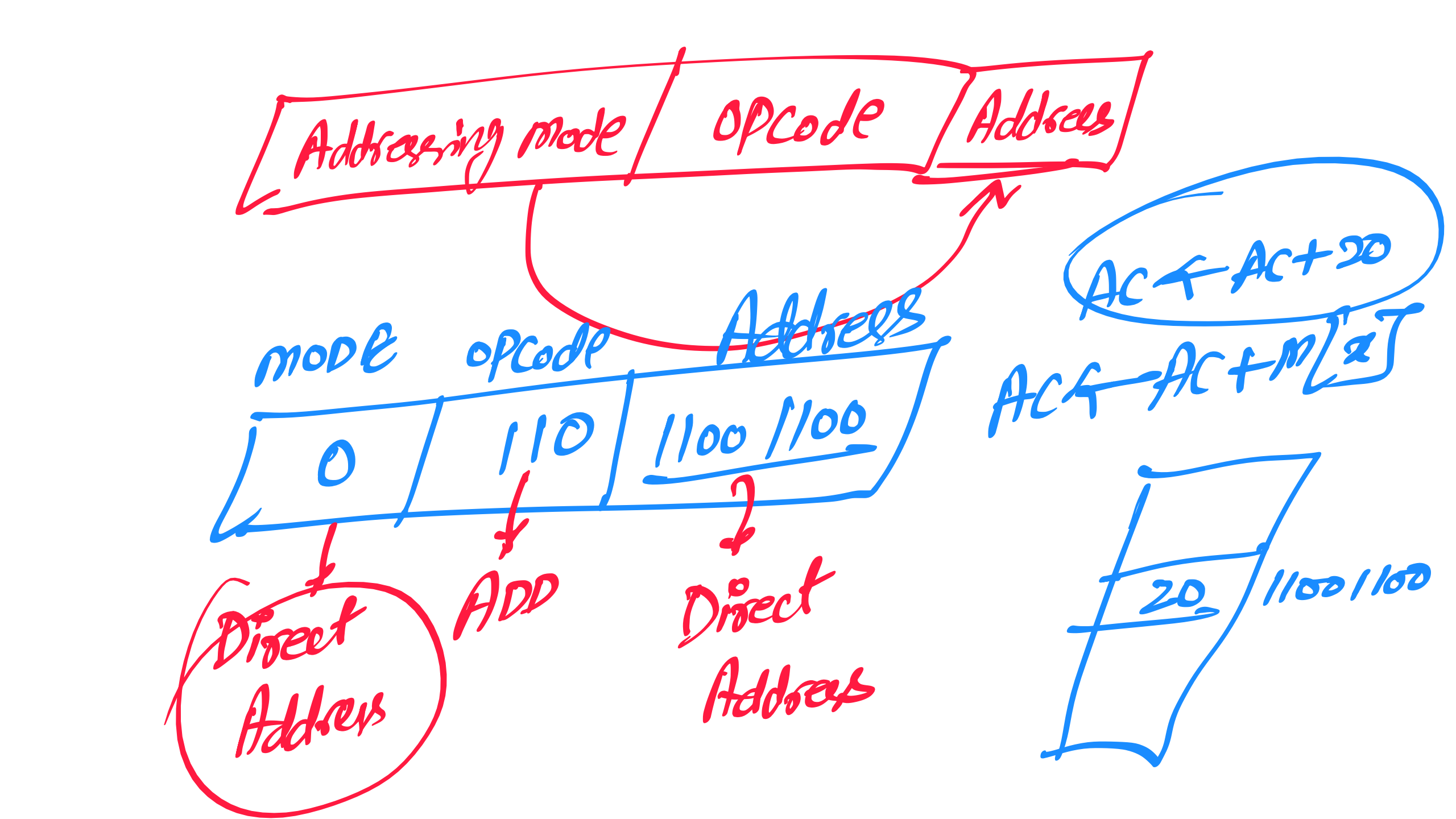
INTRODUCTION

Opcode: Operation codes are known as opcodes. The first component of an instruction, known as an opcode, instructs the computer on what task to carry out.

Operand: The second component of an instruction, known as an operand, instructs the computer where to locate or store the data or instructions.



Example for representing opcode and operand in an instruction


Addressing mode
OPcode
Address
mode
OPcode
Address
0
110
1100 1100
Direct Address
ADD
Direct Address
AC ← AC + 20
AC ← AC + M[x]
20
1100 1100

TYPES OF INSTRUCTIONS

There are certain basic operations included in every computer's instructions set. The computer instructions are classified into three categories:

1. Data Transfer Instructions
2. Data Manipulation Instructions
3. Program Control Instructions

DATA TRANSFER INSTRUCTIONS

Data transfer instructions move data from one location to another, without changing the binary information content. They are also called copy instructions.

Typically the transfers are between memory and processor registers, between processor registers and input and output registers, and among the processor registers themselves.

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

STORE:

Ex:

ST 1000 — AC → M[1000]

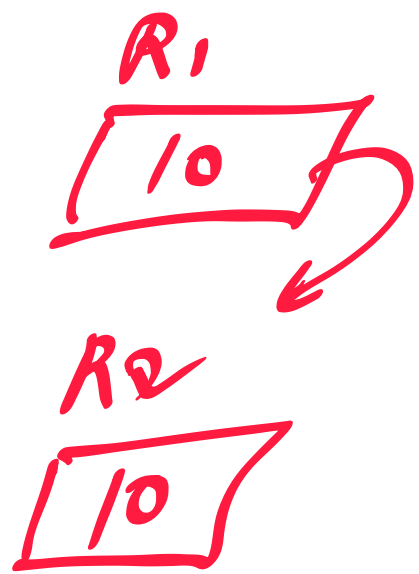
ST @1000 — Ac → M[M[1000]]

ST $1000 — Ac → M[PC+1000]

MOVE

Ex: MV

MV R_1 R_2



DATA TRANSFER INSTRUCTIONS

With different Addressing Modes: Eight addressing modes for Load instruction

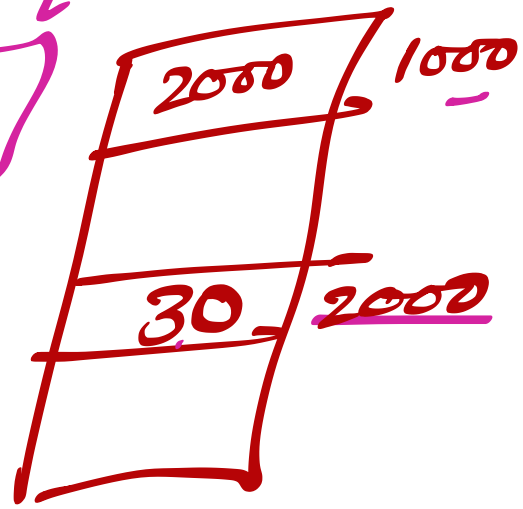
Mode	Assembly Convention	Register Transfer
Direct address	LD ADR	AC ← M[ADR]
Indirect address	LD @ADR	AC ← M[M[ADR]]
Relative address	LD $ADR	AC ← M[PC + ADR]
Immediate operand	LD #NBR	AC ← NBR
Index addressing	LD ADR(X)	AC ← M[ADR + XR]
Register	LD R1	AC ← R1
Register indirect	LD (R1)	AC ← M[R1]
Autoincrement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

1. LD 1000 ✓ 1MA AC ← M[1000] ✓ AC=2000

2. LD #1000 ✓ 0MA AC ← 1000 ✓ AC=1000
   ↳ immediate

3. LD @1000 → 2MA AC ← M[M[1000]]
   ↳ Indirect
   ⇒ AC ← M[2000]

AC=30



4. LD $1000
   ↳ Relative Addr

Opcode	Address

→ Displacement

Ex: 4. 3000 Fi: LD $1000

↳ Relative

$AC \leftarrow M[PC+1000]$

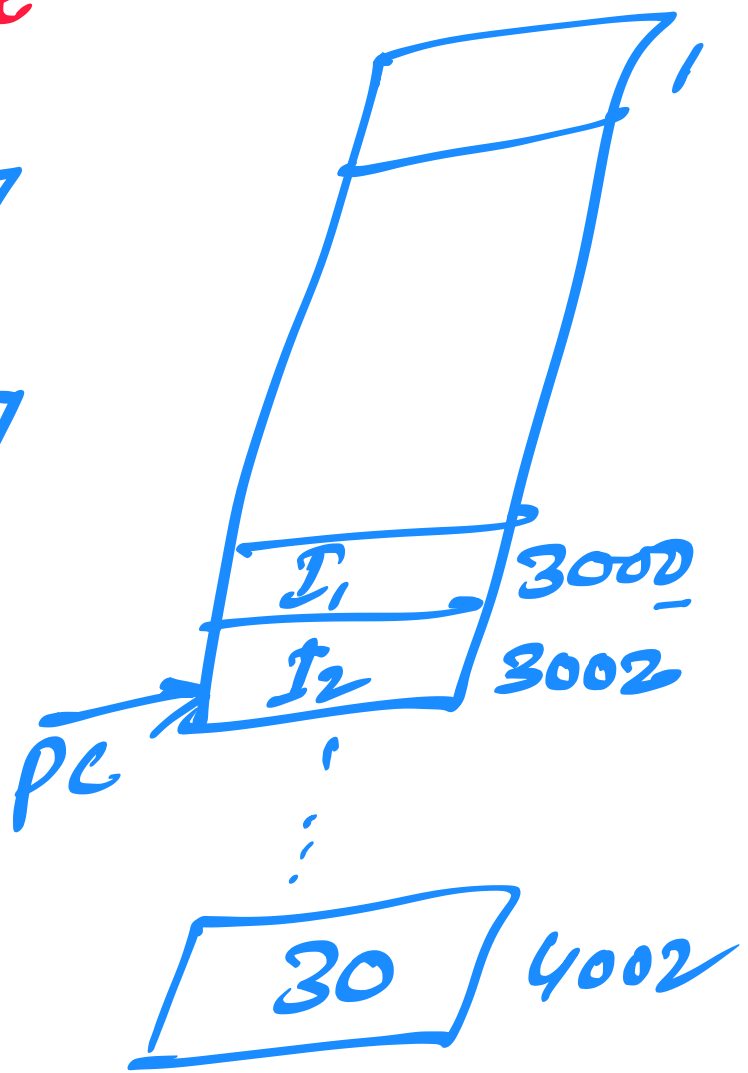
PC = 3002

$AC \leftarrow M[PC+1000]$

$\leftarrow M[3002+1000]$

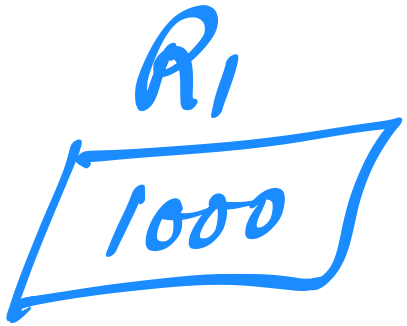
$AC \leftarrow M[4002]$

$AC \leftarrow 30$



Displacement Add.

LD R1 10



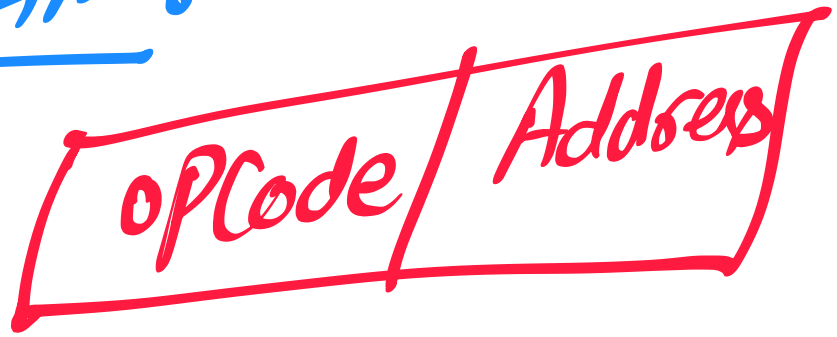
EA = [R1] + 10

EA = 1000 + 10 = 1010

AC ← M[R1 + 10] ⇒ M[1010]

AC ← M[1010]

* Relative Add:

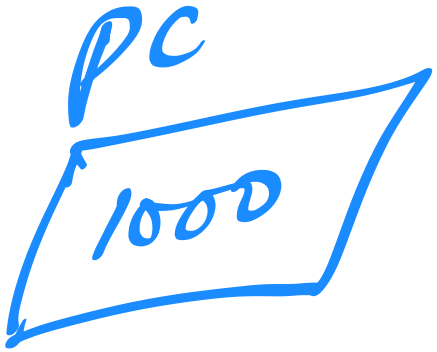


LD 10

EA = PC + Address

= 1000 + 10 = 1010

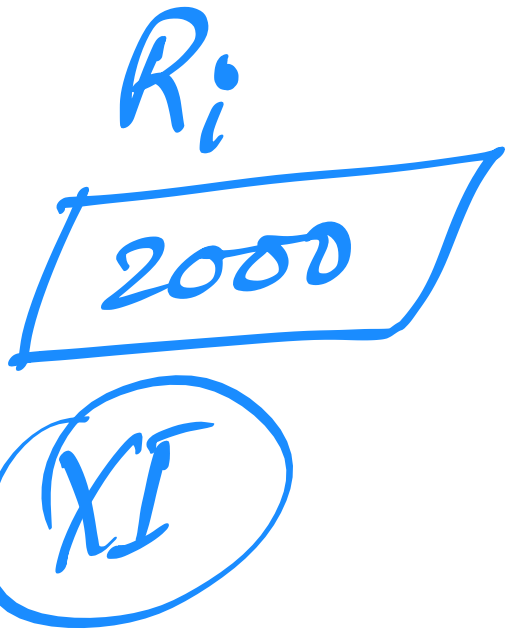
AC ← M[1010]



* Indexed Addr:—

LD R_i 10

EA = R_i + 10



* Base Addr:

LD BR 10

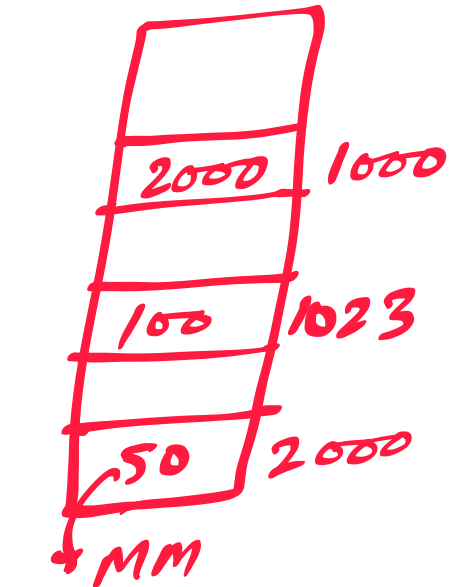
EA = [BR + 10]

Ex: LD @1000

$A \leftarrow M[M[1000]]$ — 1 mA

$A \leftarrow M[2000]$

$A \leftarrow 50$ — 1mA



Ex: LD 19 — 1002

Relative AM

EA = 1004 + 19 = 1023

1004 ← PC

PC: 1004

Ex: LDI 1000 ⇒ A ← 1000

LD #1000 ⇒ A ← 1000

Ex:

R1

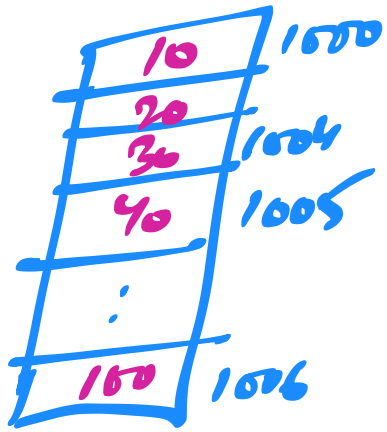
1006

LD (R1)+

AC ← M[R1]

AC ← M[1005]

R1 ← R1 + 1



AC

40

Ex: ADD @1000, R_1

↳ Register AM

↳ Indirect AM

(2)

$M[M[1000]] \leftarrow M[M[1000]] + R_1$

2 MA 1 RA

1 MA

$M[2000] \leftarrow$

DATA MANIPULATION INSTRUCTIONS

Data manipulation instructions are those instructions that manipulate or change the content of the data/registers/memory.

It performs operations on data and provides the computational capabilities of the Computer.

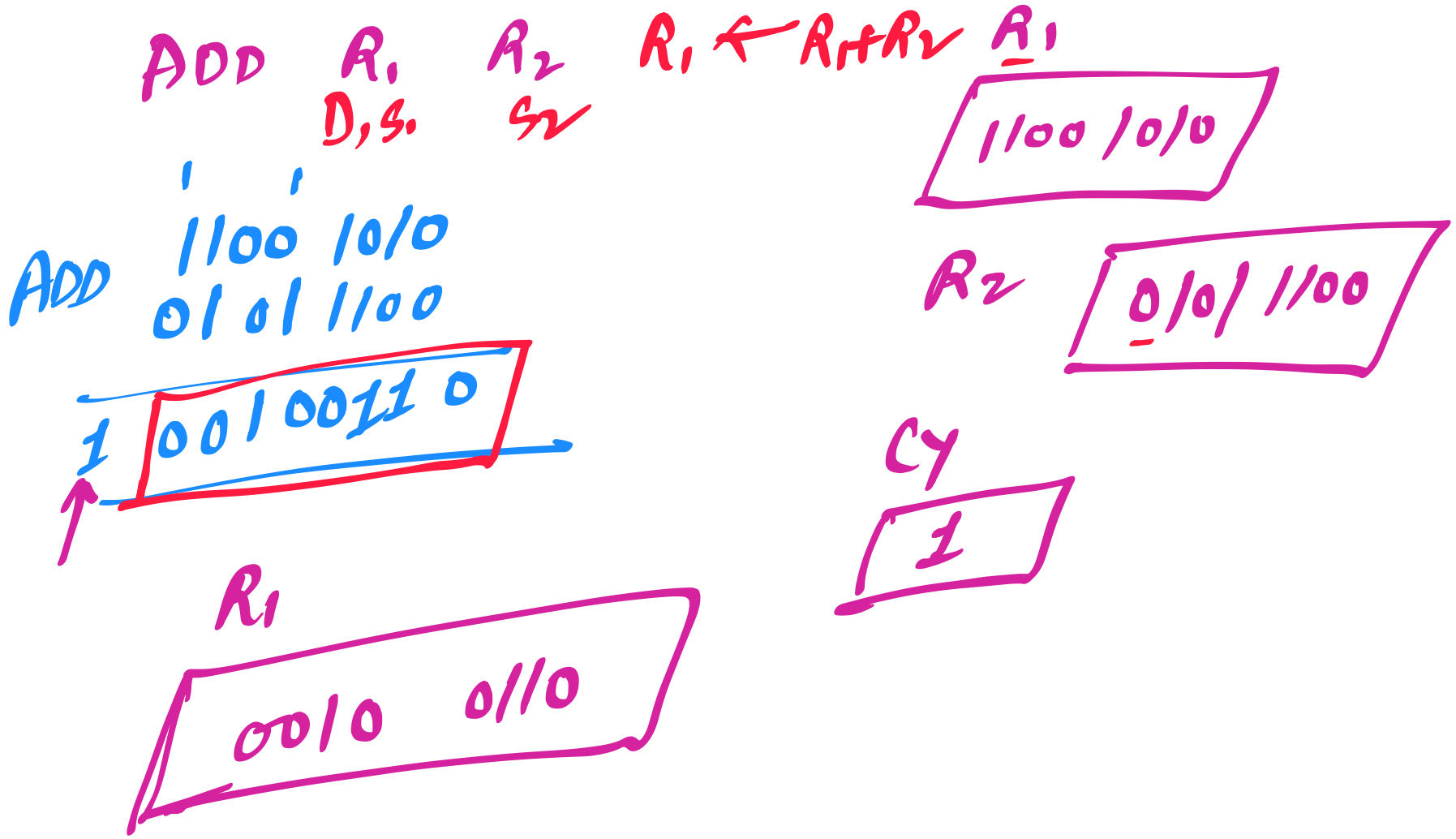
Data manipulation instructions can be categorized into three parts:

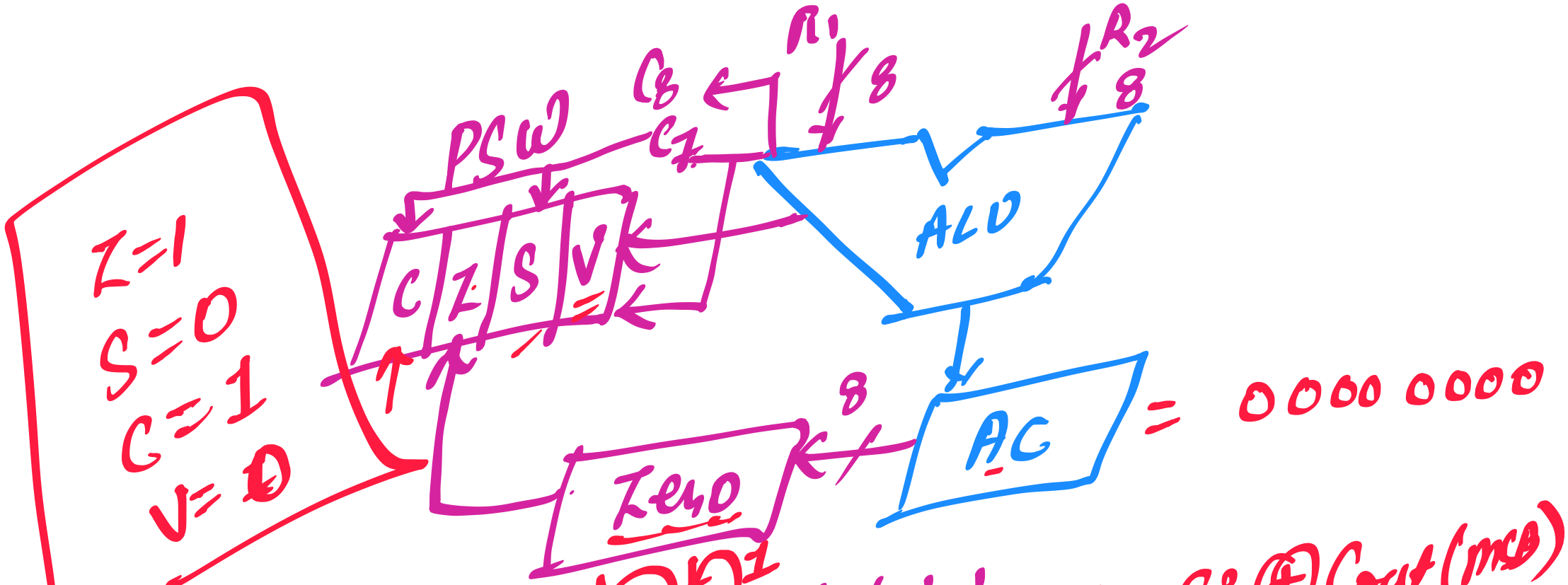
1. Arithmetic instruction
2. Logical and bit manipulation instructions
3. Shift instructions

1. ARITHMETIC INSTRUCTIONS

Arithmetic instructions include increment, decrement, add, subtract, multiply, divide, add with Carry, subtract with Borrow, and negate that is (2's) two's complement. If there's a negative number, it is considered as negate (so two's complement).

Name	Mnemonic
Increment	INC
Decrement	DEC
Add	ADD
Subtract	SUB
Multiply	MUL
Divide	DIV
Add with carry	ADDC
Subtract with borrow	SUBB


ADD R1 R2
D, S. Sr
R1 ← R1+R2
R1
1100 1010
R2
0101 1100
ADD 1100 1010
0101 1100
1 0010 0110
Cy
1
R1
0010 0110



Ex:

$$\begin{array}{r} 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1 \\ 0000\ 0001 \\ +\ 1111\ 1111 \\ \hline 1\ \ 0000\ 0000 \end{array}$$

C_0

C_B = 1

C_Z = 1

$V = C_{in} \oplus C_{out}$ (MSB)

$= 1$

$1 \oplus 1 = 0$

$V = 0$

$A = 1001\ 0011$

$B = 0101\ 1011$

$1110\ 1110$

$C = 0$

$A = 1011\ 0110$

$B = 1001\ 0101$

$0100\ 1011$

$+1$

$0100\ 1100$

$CY = 1$

2. LOGICAL AND BIT MANIPULATION INSTRUCTIONS

These logical instructions consider each operand bit individually and treat it as a Boolean variable.

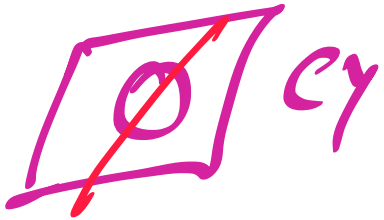
Basically, logical instructions help perform binary operations on strings of bits stored in registers.

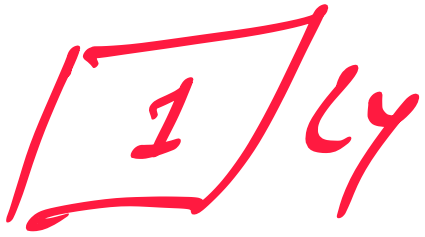
Name	Mnemonic
Clear	CLR
Complement	COM
AND	AND
OR	OR
Exclusive-OR	XOR
Clear carry	CLRC
Set Carry	SETC
Complement Carry	COMC
Enable Interrupt	EI
Disable Interrupt	DI


SETC
CLRC
COMC


1
cy


1
cy


0
cy


1
cy

$R_1 =$ 1100 1010

① CLR R_1 → 0000 0000

② COM R_1 → 1111 1111 R_1

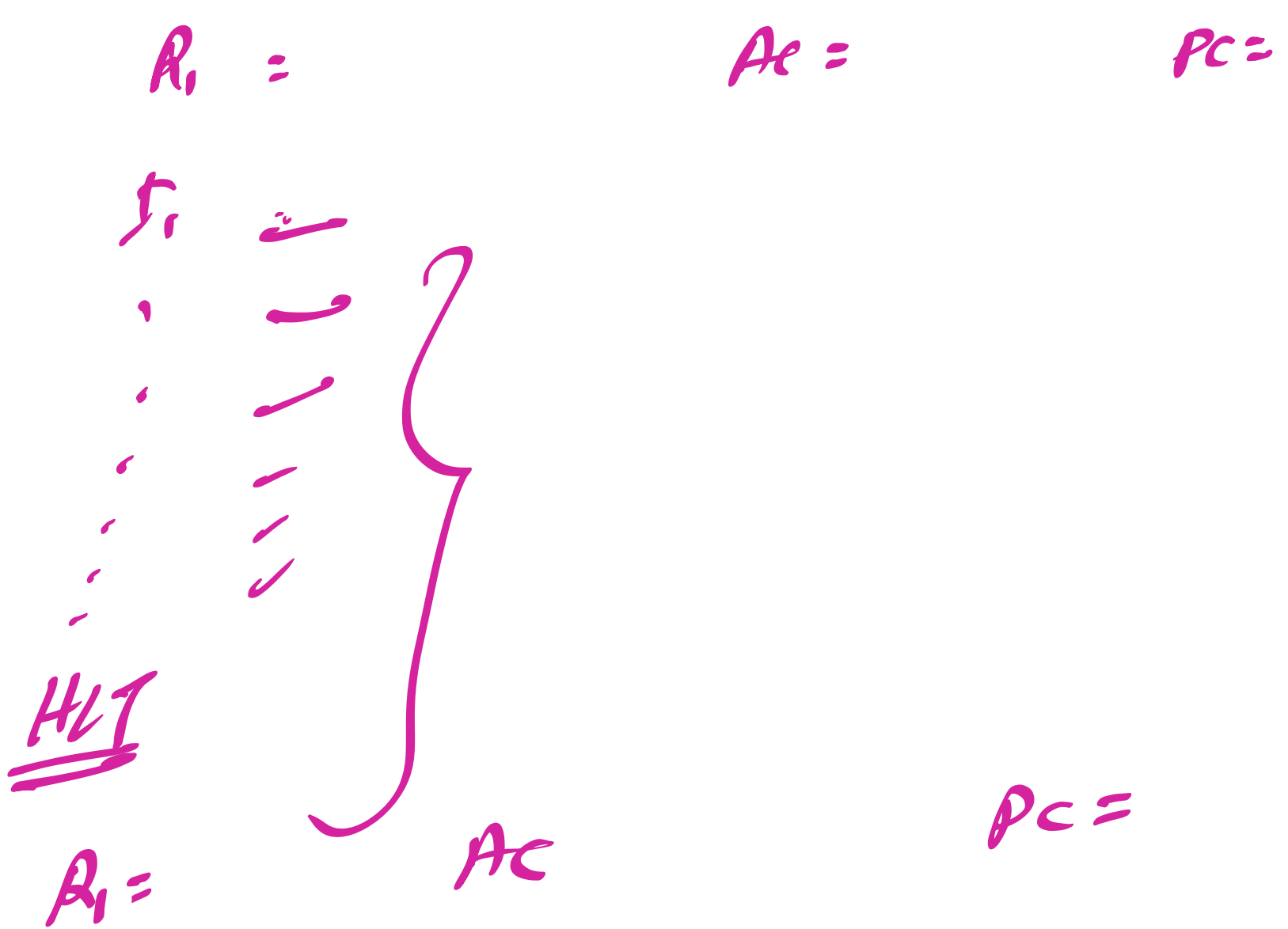
③ AND R_1 → AC ← AC AND R_1

④ OR R_1 → AC = 0101 1010

⑤

1111 1111

0101 1010


R1 =
Ac =
PC =
HLT
R1 =
Ac
PC =

R_1 1011 0110

CLR R_1

0000 0000

3. SHIFT INSTRUCTIONS

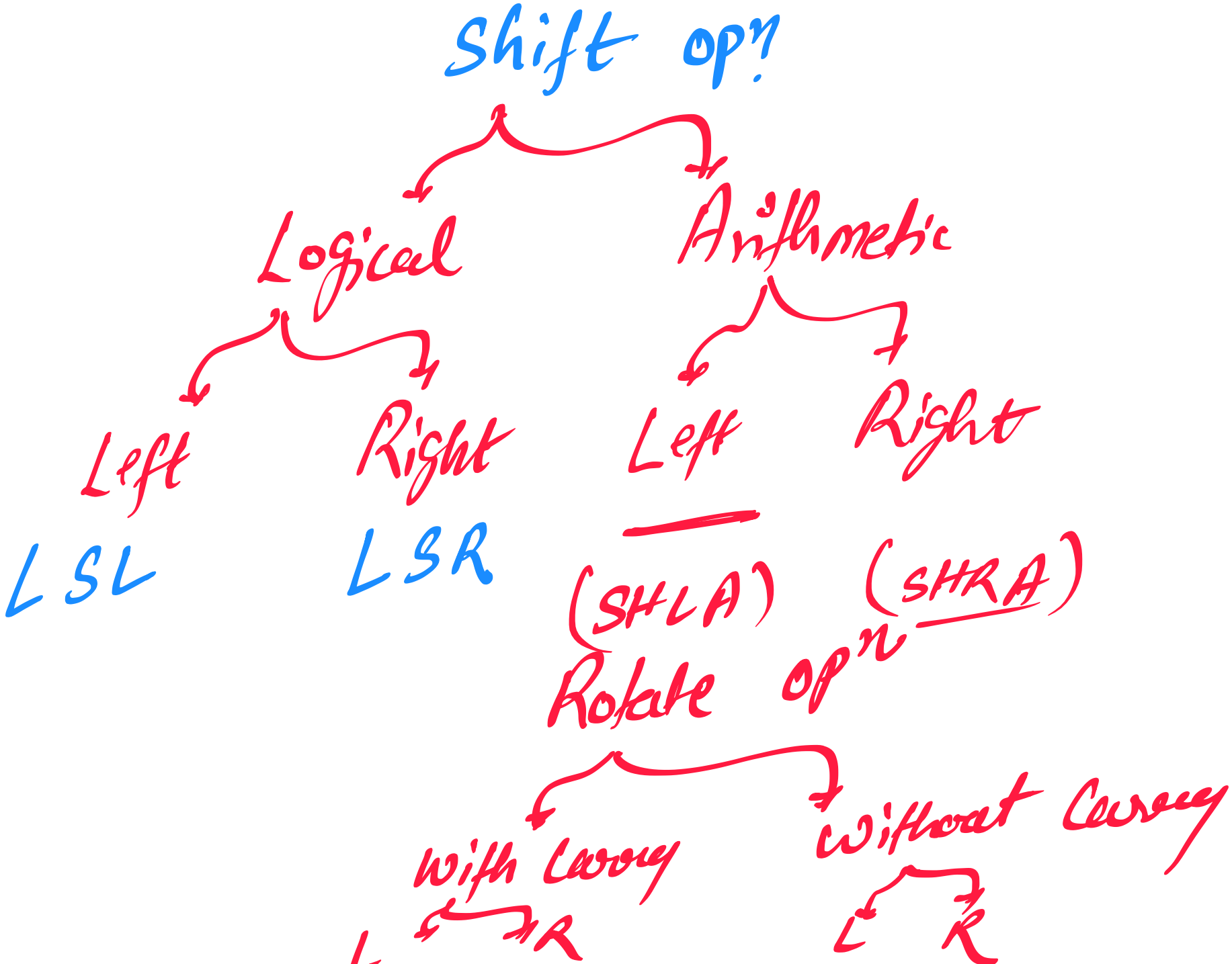
Shift instructions allow the bits of a memory byte or register to be shifted one-bit place to the right or the left.

There are basically two types of shift instructions – arithmetic and logical.

Arithmetic shifts consider the contents of the memory byte or register to be a signed number. So, when the shift is made, the number is arithmetically divided by two (right shift) or multiplied by two (left shift).

Logical shifts consider the contents of the register or memory byte to be just a bit pattern when the shift is made.

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


Shift opn
Logical
Arithmetic
Left
Right
Left
Right
LSL
LSR
(SHLA)
(SHRA)
Rotate opn
With Carry
Without Carry
L
R
L
R

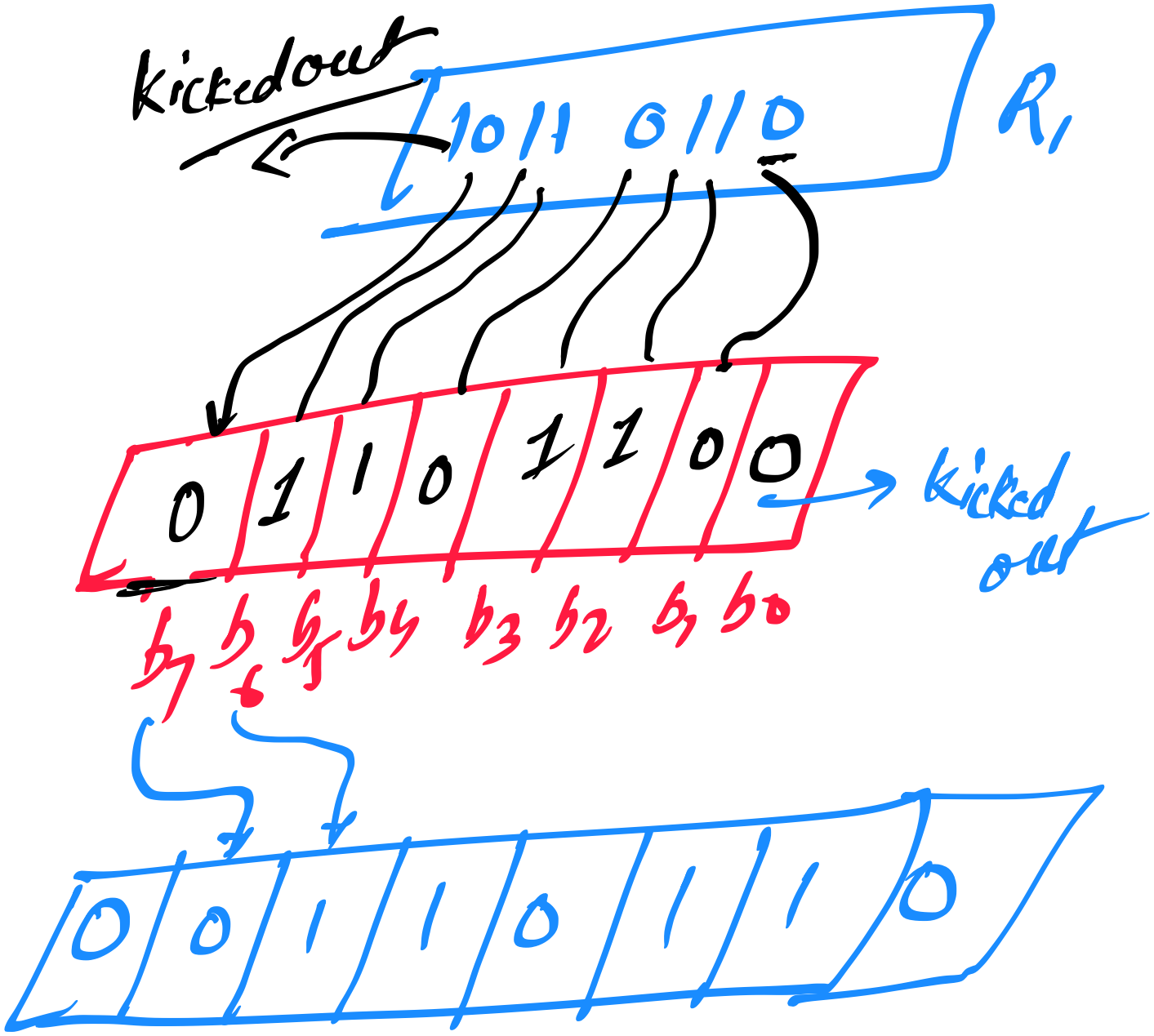
* Logical Shift :-

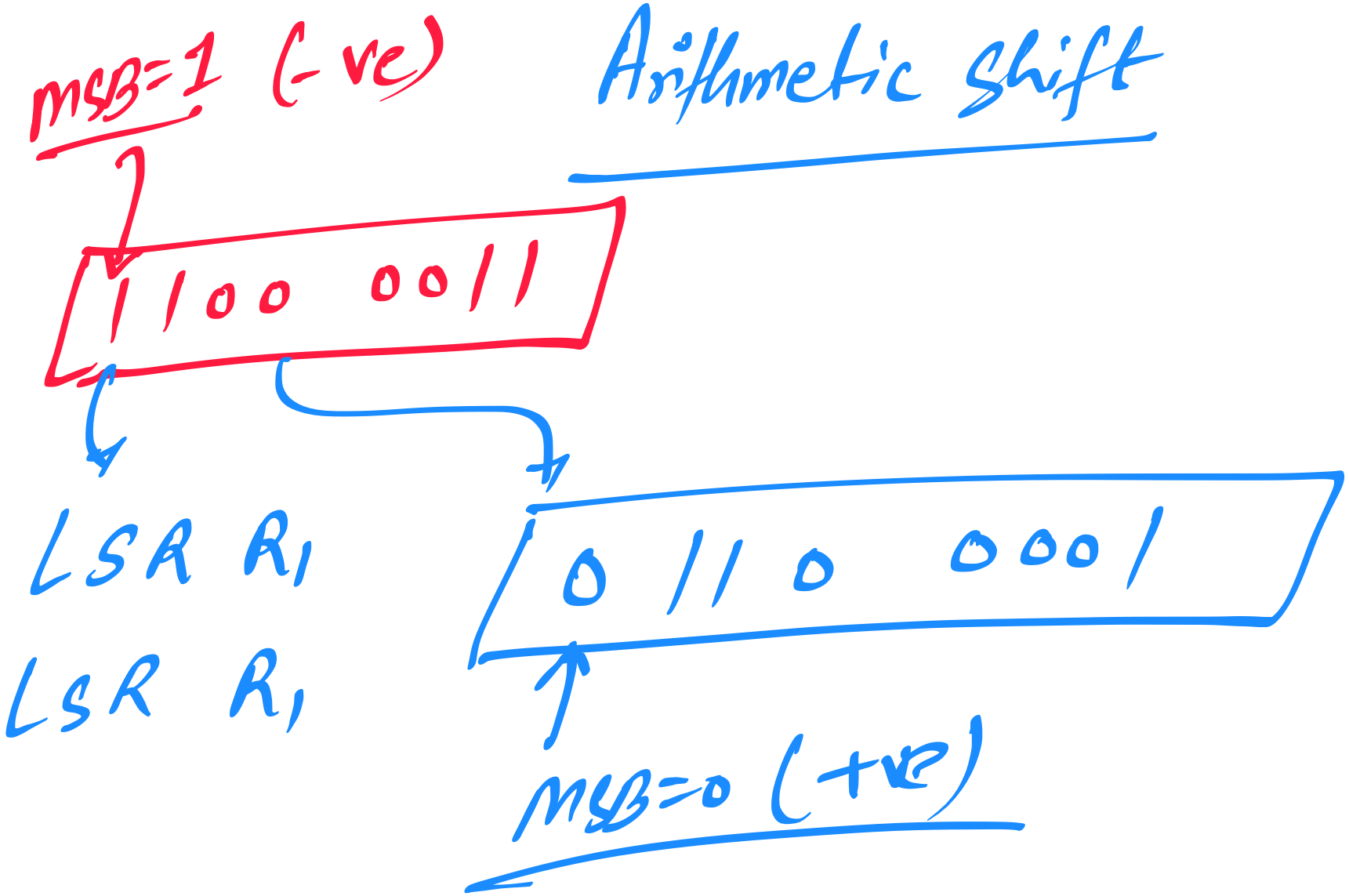
(a) Left (LSL):

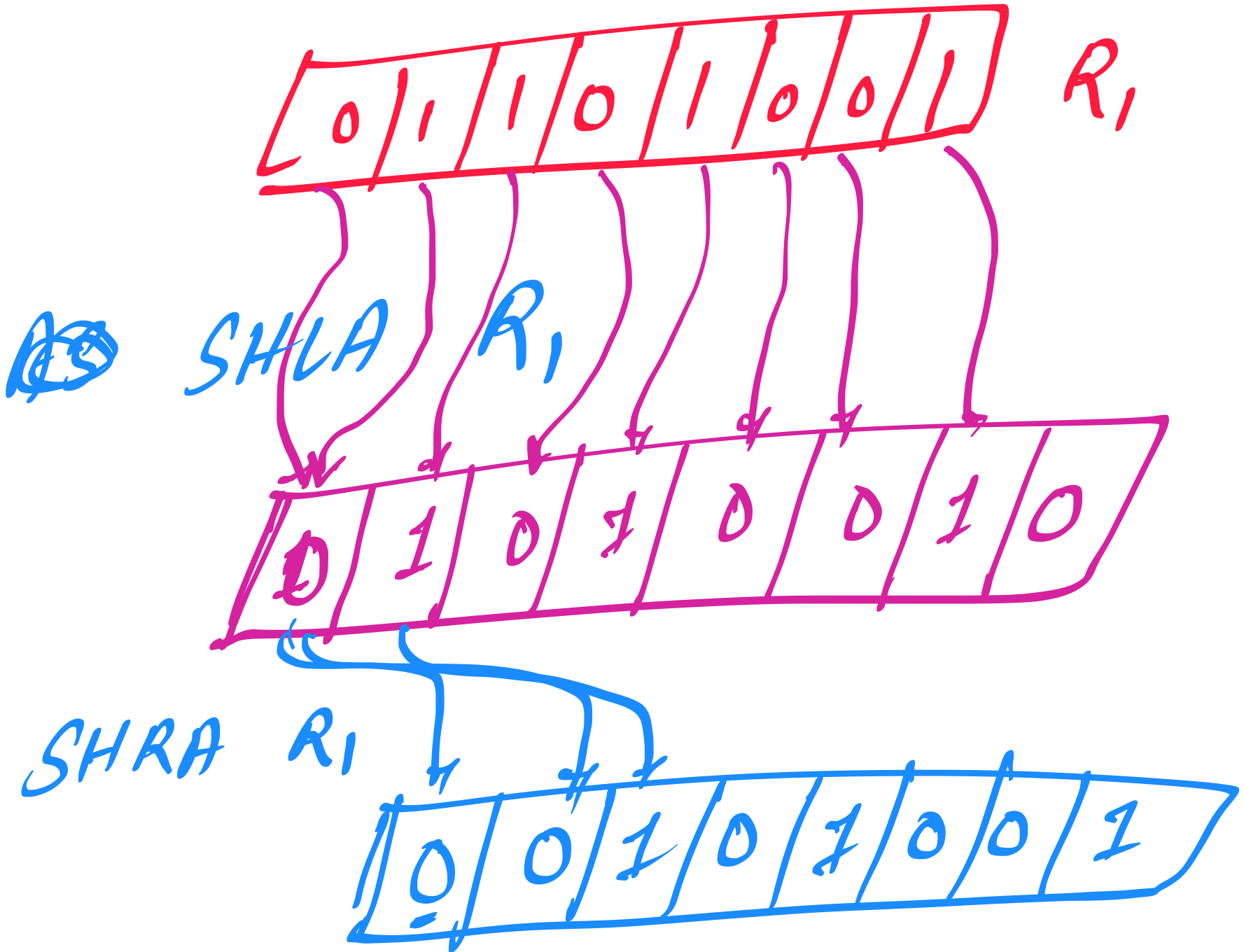
LSL R_1

(b) Right (LSR)

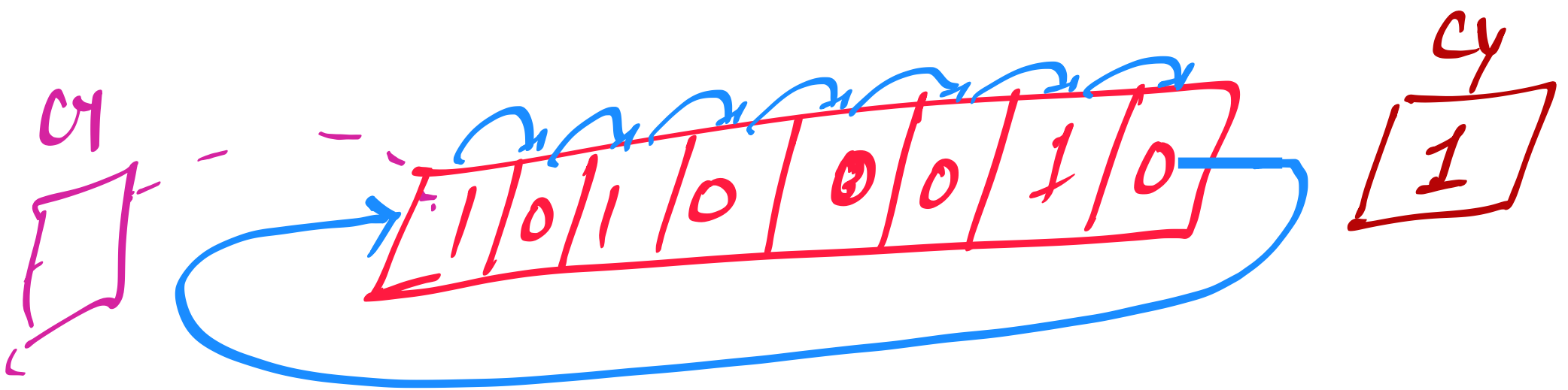
LSR R_1



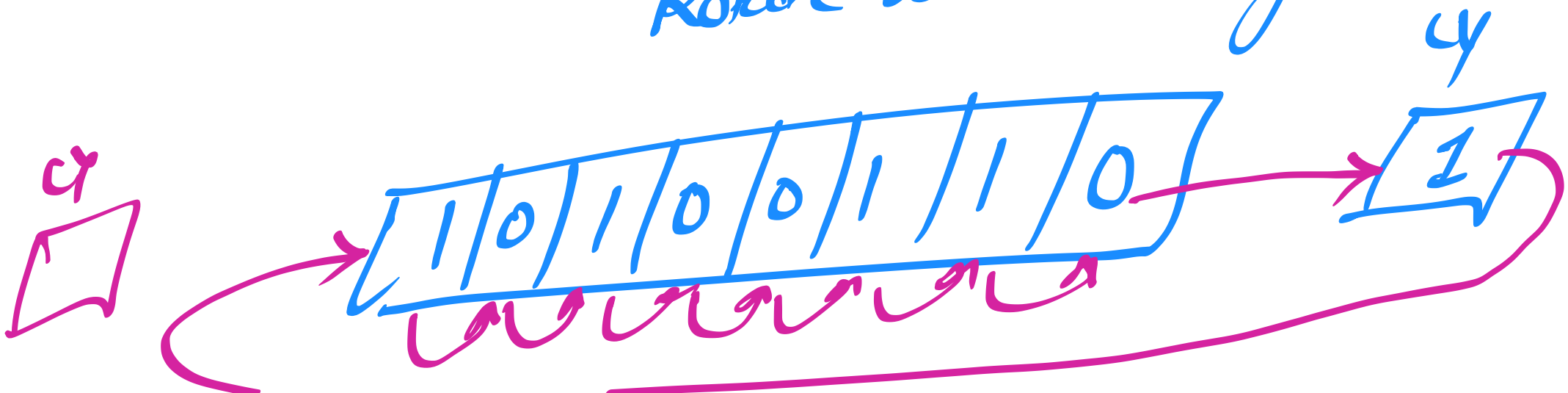

MSB=1 (-ve)
Arithmetic shift
1100 0011
LSR R1
LSR R1
0110 0001
MSB=0 (+ve)


0 1 1 0 1 0 0 1
R1
SHLA R1
1 1 0 1 0 0 1 0
SHRA R1
0 0 1 0 1 0 0 1

Rotate without carry



Rotate with carry



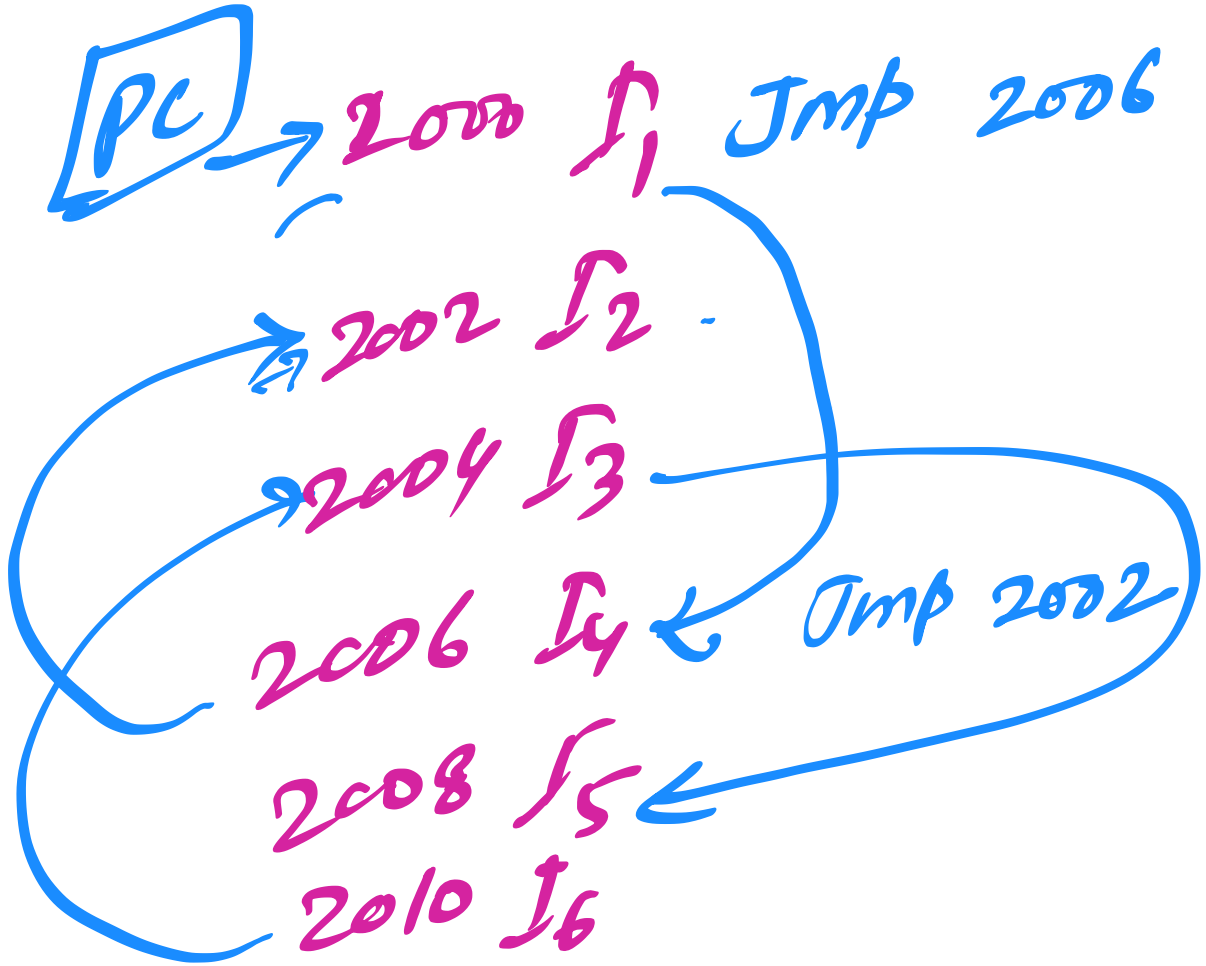
6. PROGRAM CONTROL

Program Execution Sequence

Sequential

PC → 1000 I_1
↓
PC → 1002 I_2
↓
PC → 1004 I_3
↓
PC → 1006 I_4
↓
PC → 1008 I_5
↓
PC → 1010 I_6
⋮

Transfer of control




```
a = b+c;
if (a>10)
{
a--;
}
else if (a==10)
{
a++;
}
else
{
--
}
```

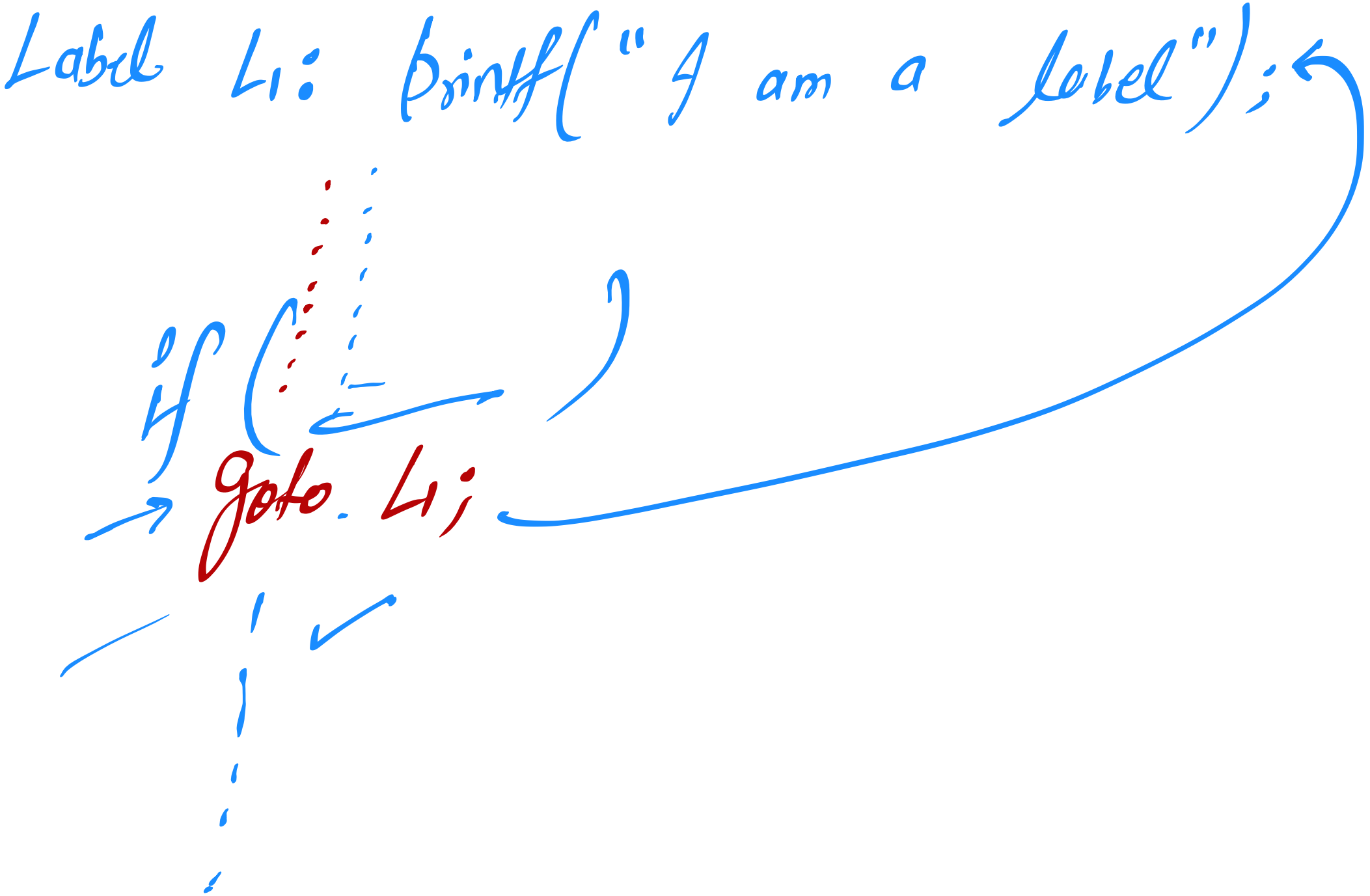
do while

while

```
a = b + c;
while( a > 0)
{

   a--;
   c= a++;
}
printf("  ");
```

a = 10


Label L1: printf("I am a label");
if (
goto L1;

PROGRAM CONTROL INSTRUCTIONS

Program control instructions modify or change the flow of a program.

It is the instruction that alters the sequence of the program's execution, which means it changes the value of the program counter, due to which the execution of the program changes.

Features:

- These instructions cause a change in the sequence of the execution of the instruction.
- This change can be through a condition or sometimes unconditional.
- Flags represent the conditions.
- Flag-Control Instructions.
- Control Flow and the Jump Instructions include jumps, calls, returns, interrupts, and machine control instructions.
- Subroutine and Subroutine-Handling Instructions.
- Loop and Loop-Handling Instructions.

Program Execution Sequence

Sequential

1000 : I_1 ← PC = 1000

1002 : I_2 ← PC = 1002

1004 : I_3 ← PC = 1004

1006 : I_4 ← PC = 1006

1008 : I_5 ← PC = 1008

1010 : HLT ← PC = 1012

Transfer of Control

2000 I_1 PC = 2000

2002 I_2 Jmp 2008 PC = 2008

2004 I_3

2006 I_4

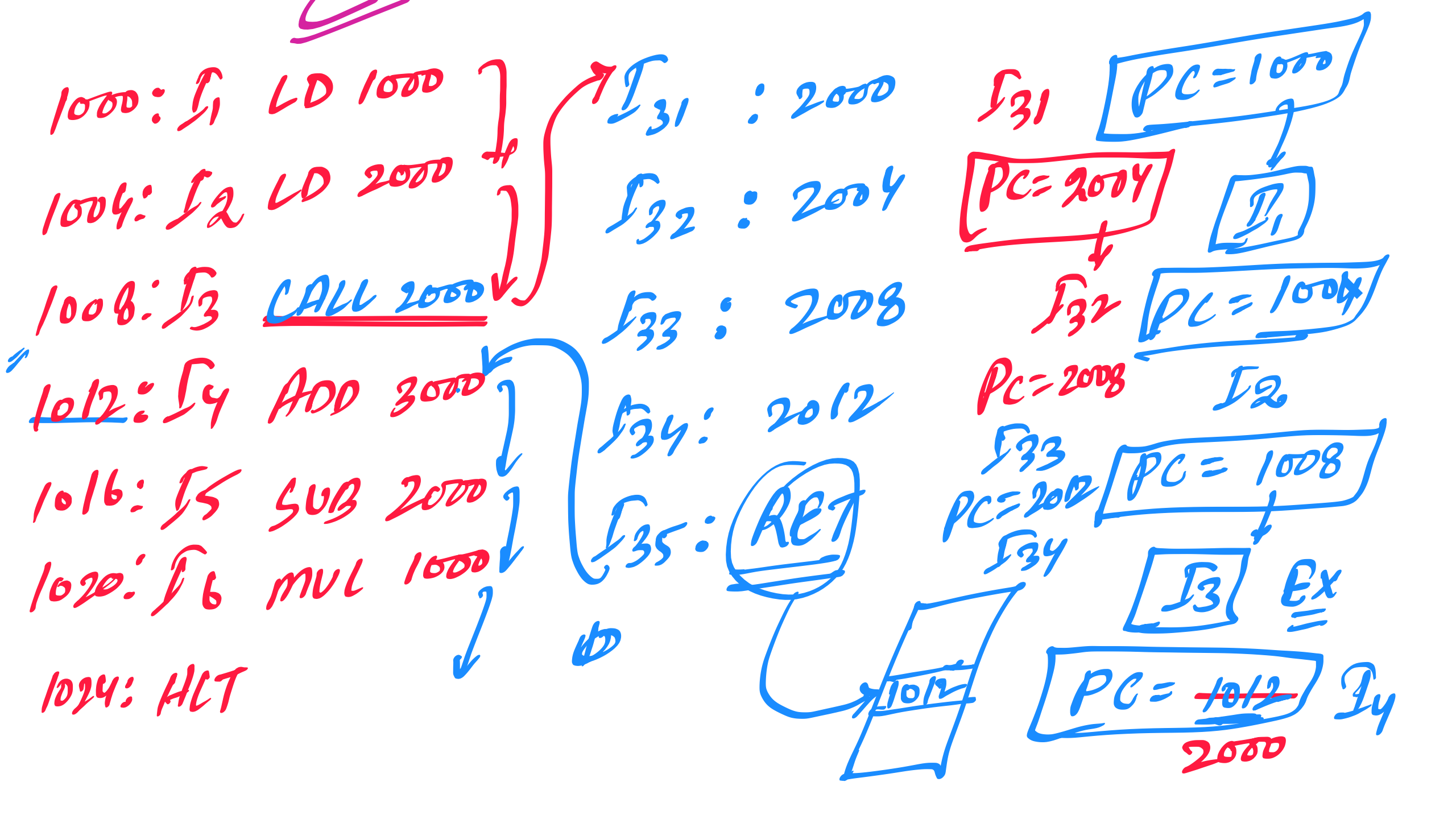
2008 I_5 Jmp 2002 PC = 2010

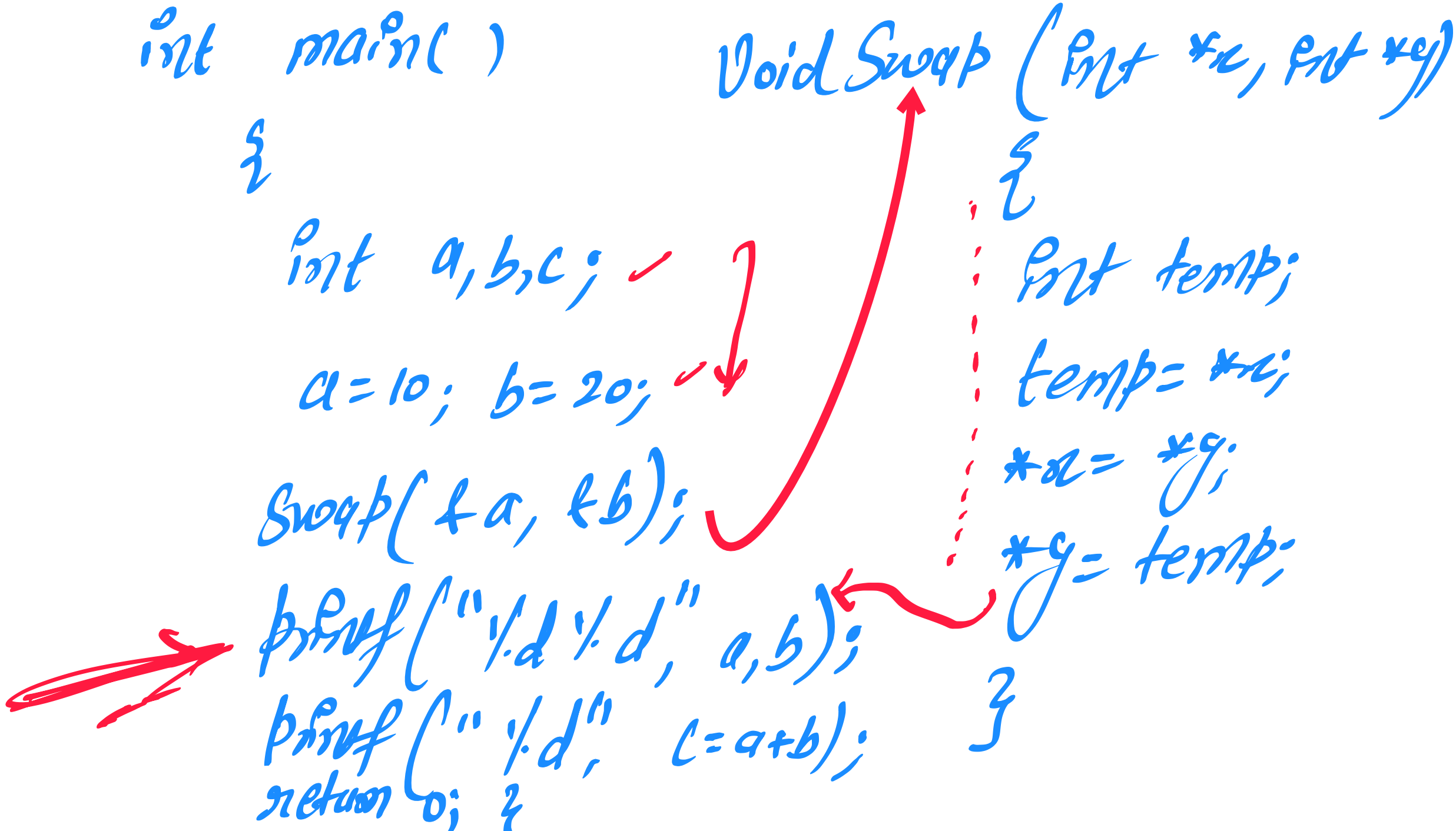
2010 I_6

* if-else
* loops
* goto

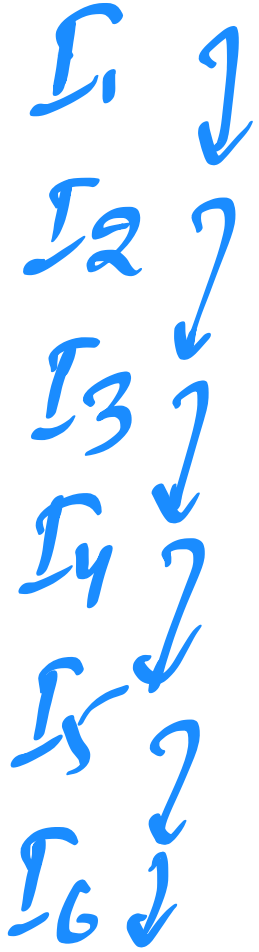
→ if (a>10)
{
a++;
}
else
a--;

while(a>0)
{
≡
≡
}

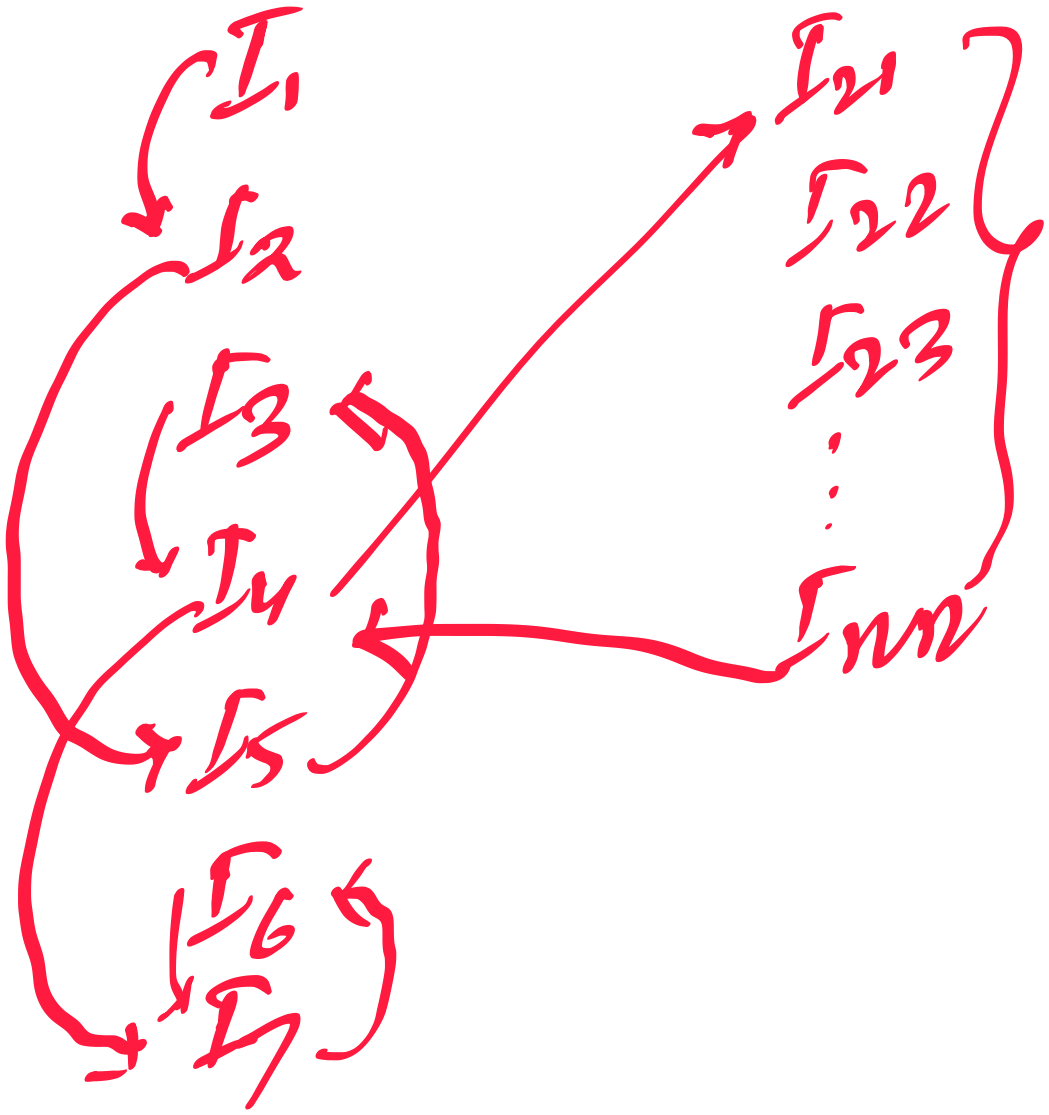

1000: I1 LD 1000
1004: I2 LD 2000
1008: I3 CALL 2000
1012: I4 ADD 3000
1016: I5 SUB 2000
1020: I6 MUL 1000
1024: HLT
I31 : 2000
I32 : 2004
I33 : 2008
I34: 2012
I35: RET
I31
PC=1000
I1
I32
PC=2004
I2
I33
PC=2008
I3
Ex
I34
PC=2012
1012
PC= 1012
2000
I4


int main( )
{
int a,b,c;
a=10; b=20;
Swap(&a, &b);
printf("%d %d", a,b);
printf("%d", c=a+b);
return 0; }
Void Swap (int *x, int *y)
{
int temp;
temp= *x;
*x= *y;
*y= temp;
}

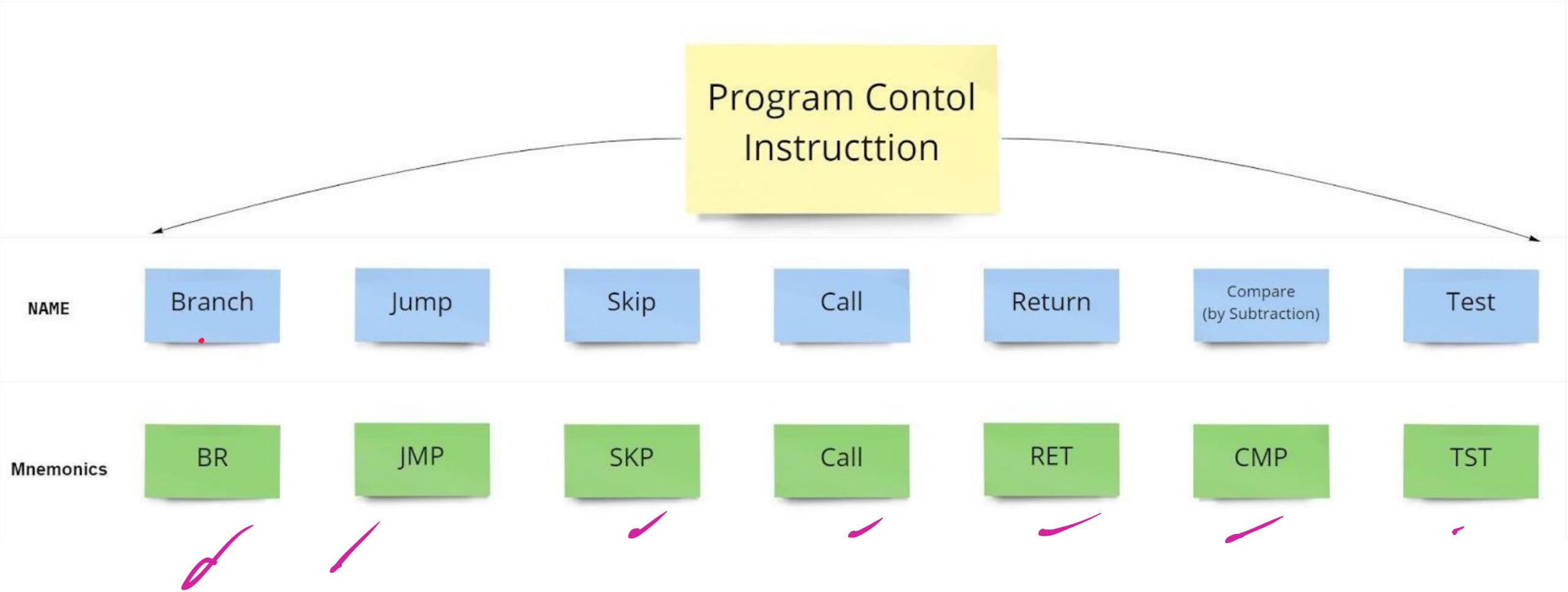
Program



Program

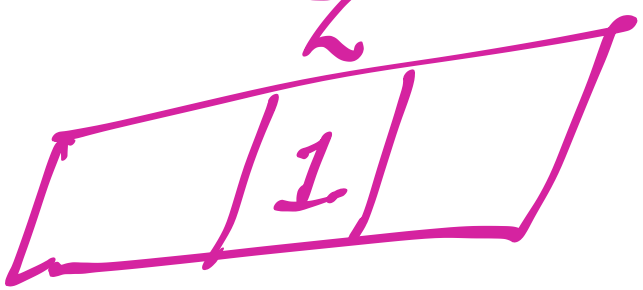


PROGRAM CONTROL INSTRUCTIONS



Program Control Instrn

- Conditional
  - Ex: JNZ, JZ, JC, JS
  - Jmp if Non-zero
- Unconditional
  - Ex: CALL




```
L1:
.....
goto L1
```

```
if (a>10)
{
=
=
}
else
{
=
=
```

PROGRAM CONTROL INSTRUCTIONS

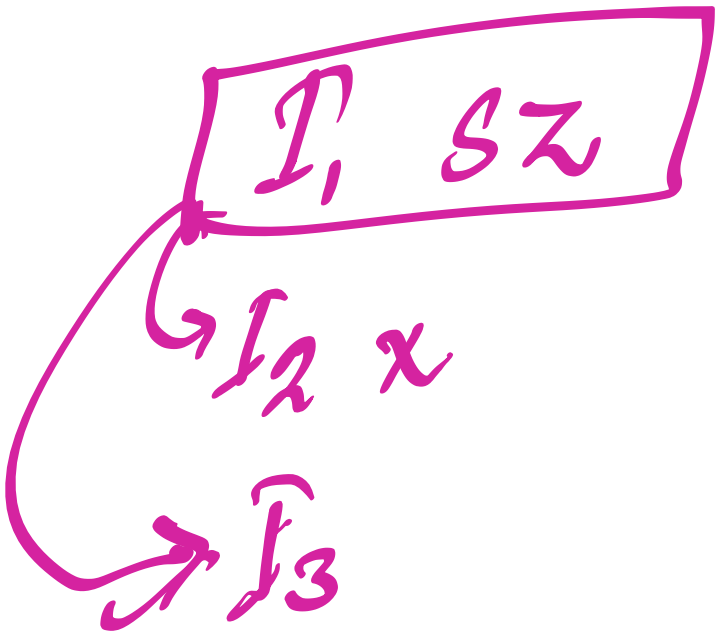
Program Control Instructions	Description
Branch (BR)	Branches are conditional and are used to provide logic to the program and make it do different things. Branches require several bits in the machine code instruction for the condition so they have less bits to use for the location of the branch. That's why branches use a specified offset from the current program counter and can't go as far as a jump.
Skip (SKP)	Skip instructions is used to skip one(next) instruction. It can be conditional or unconditional. It does not need an address field. In the case of conditional skip instruction, the combination of conditional skip and an unconditional branch can be used as a replacement for the conditional branch.
Jump (JMP)	The jump instruction transfers the program sequence to the memory address given in the operand based on the specified flag. For a larger distance.
Compare (CMP)	The Compare instruction performs a comparison via a subtraction, with difference not retained. CMP compares register sized values, with one exception.
CALL and RETURN	The CALL and RETURN instructions interrupt the flow of a program by passing control to an internal or external subroutine. An external subroutine is another program. The RETURN instruction returns control from a subroutine back to the calling program and optionally returns

SKP:

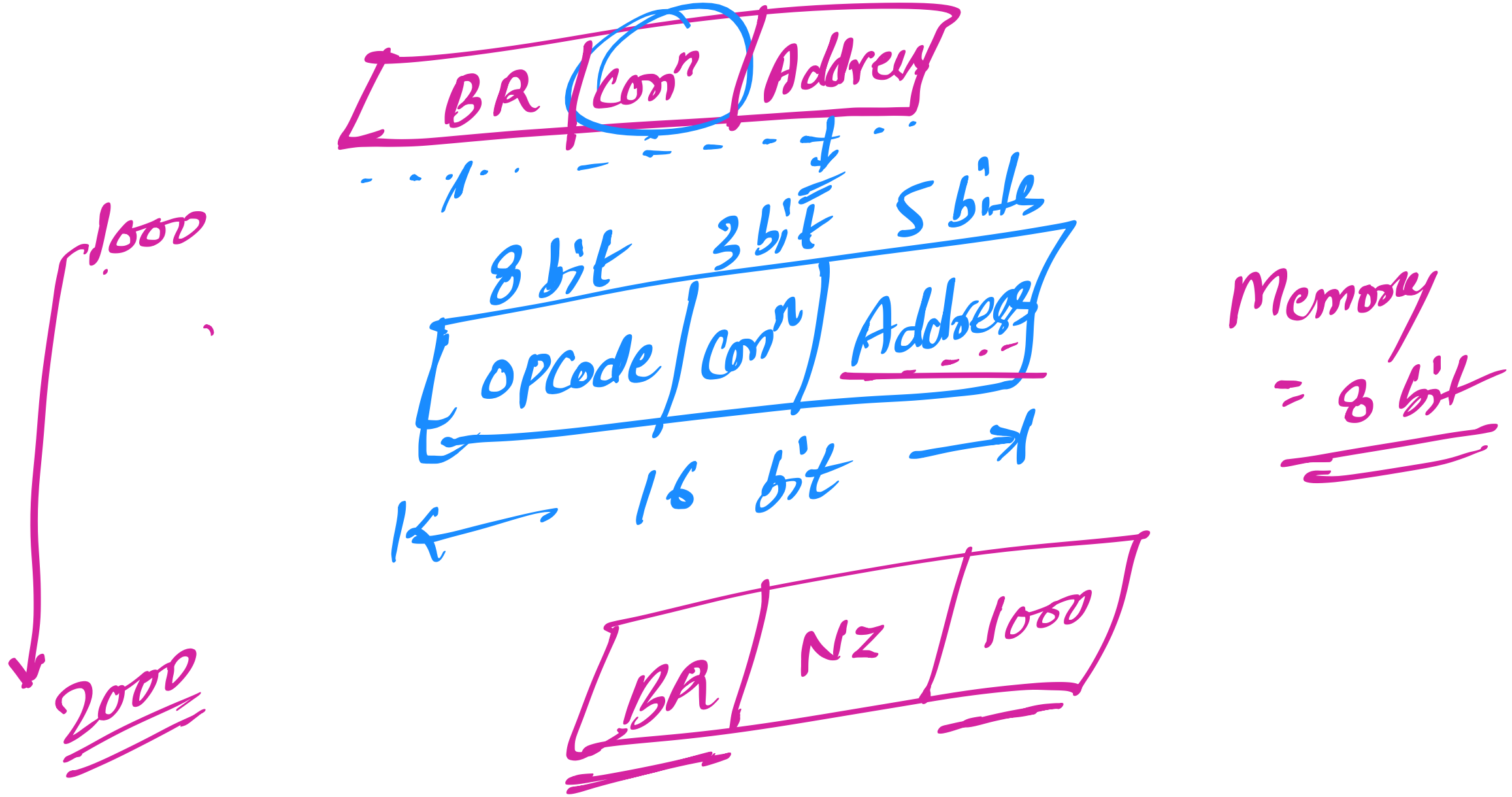
SKP

PC → I_2 x

PC → I_3 ✓



0
Z


BR
Comⁿ
Address
8 bit
3 bit
5 bits
OPCode
Comⁿ
Address
16 bit
1000
2000
BR
NZ
1000
Memory
= 8 bit

STATUS BIT CONDITIONS/ FLAG, PROCESSOR STATUS WORD

To check different conditions for branching instructions like CMP (compare) or TEST can be used. Certain status bit conditions are set as a result of these operations.

V	Z	S	C

Status bits mean that the value will be either 0 or 1 as it is a bit. We have four status bits:

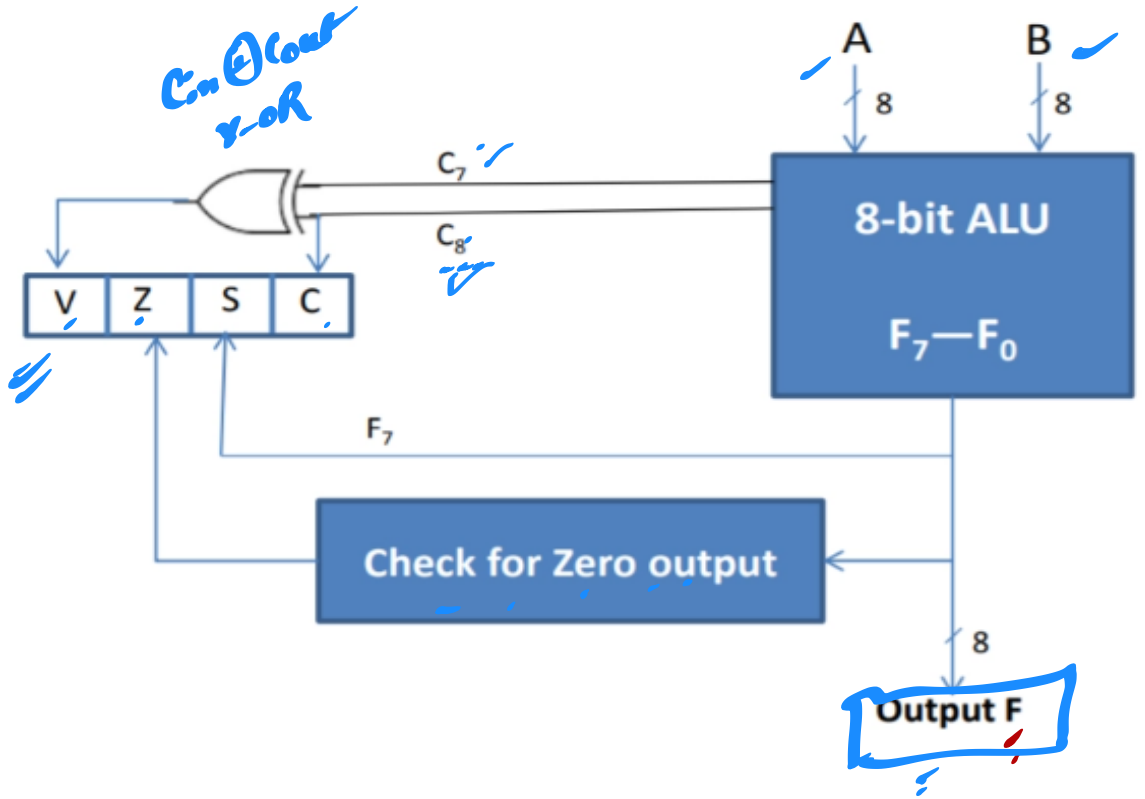
"V" stands for Overflow: based on certain bits, i.e., if extra bits are generated into our operation.

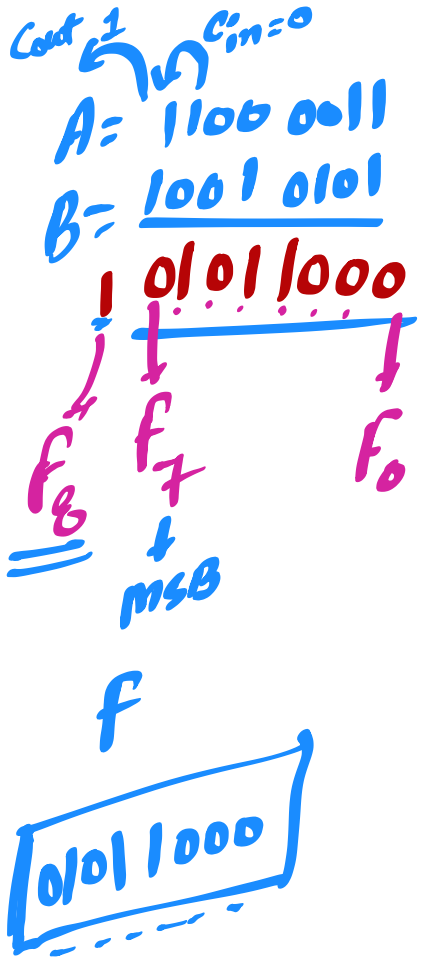
"Z" stands for Zero: If the output of the ALU(Arithmetic Logic Unit) is 0, then the Z flag is set to 1, otherwise, it is set to 0.

"S" stands for the Sign bit: If the number is positive, the Sign(S) flag is 0, and if the number is negative, the Sign flag is 1.

"C" stands for Carry: if the output of the ALU operation generates Carry, then C is set to 1, else C is set to 0.

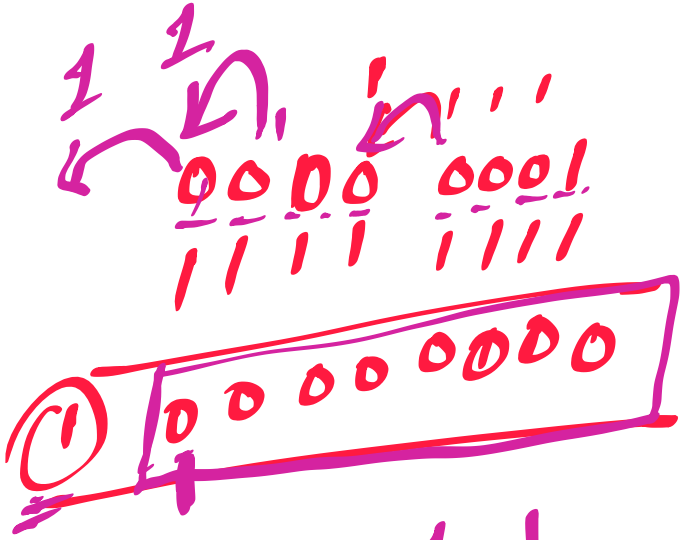
STATUS BIT CONDITIONS/ FLAG, PROCESSOR STATUS WORD





4 bit → Nibble

Ex:



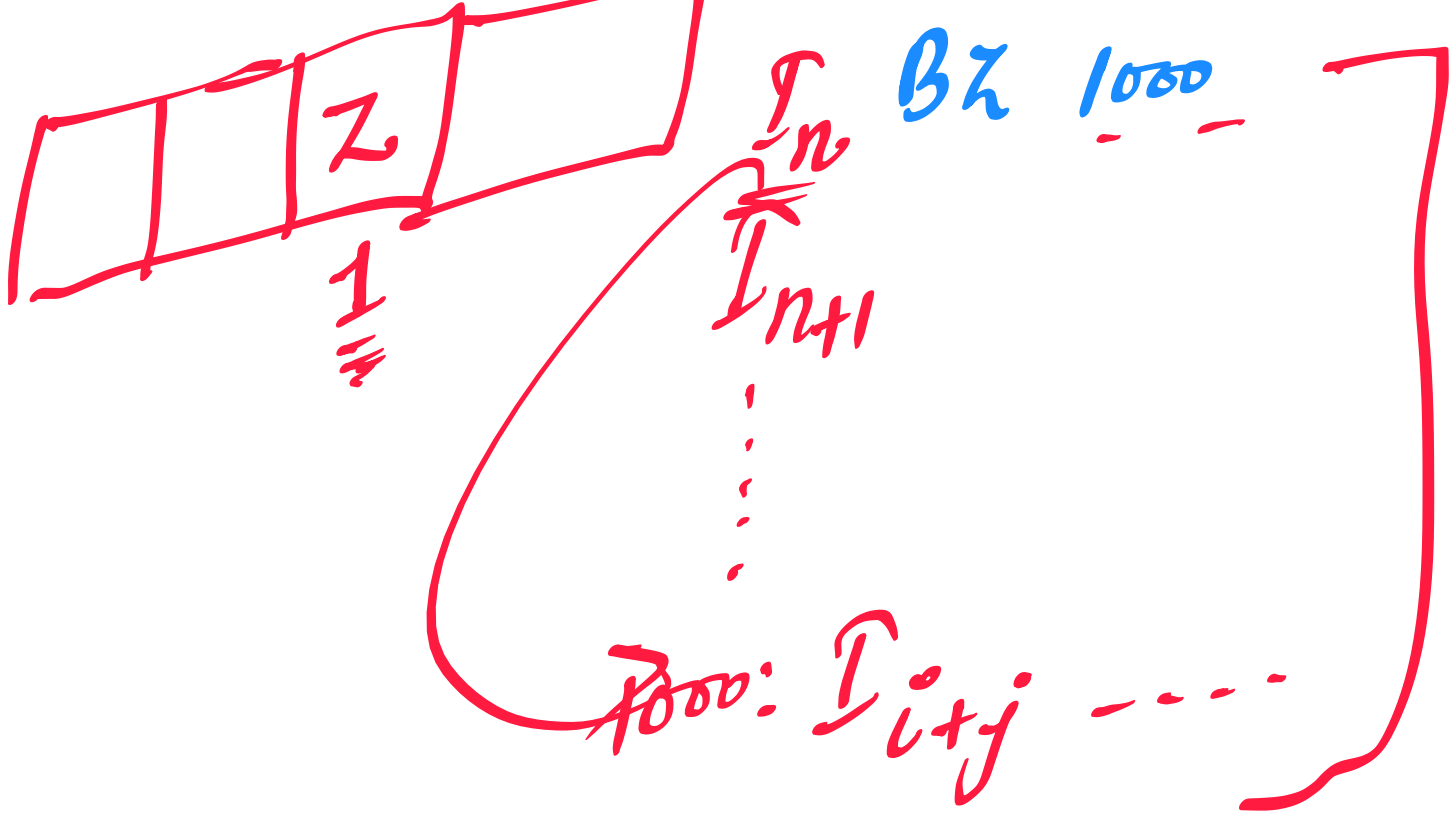
- Zero Flag (Z) = 1
- Sign flag (S) = 0
- Overflow Flag (V) = 0
- Carry flag (C) = 1
- Auxiliary Carry (AC) = 1
- Parity flag (P) = if #1's are odd = 1

1. CONDITIONAL BRANCH INSTRUCTIONS

A conditional branch instruction is basically used to examine the values that are stored in the condition code register to examine whether the specific condition exists and to branch if it does.

Each conditional branch instruction tests for a different combination of Status bits for a condition.

Mnemonic	Branch condition	Tested condition
BZ	Branch if zero	Z = 1
BNZ	Branch if not zero	Z = 0
BC	Branch if carry	C = 1
BNC	Branch if no carry	C = 0
BP	Branch if plus	S = 0
BM	Branch if minus	S = 1
BV	Branch if overflow	V = 1
BNV	Branch if no overflow	V = 0
	Unsigned **compare conditions (A - B)**	
BHI	Branch if higher	A > B
BHE	Branch if higher or equal	A ≥ B
BLO	Branch if lower	A < B
BLOE	Branch if lower or equal	A ≤ B
BE	Branch if equal	A = B
BNE	Branch if not equal	A ≠ B
	Signed **compare conditions (A - B)**	
BGT	Branch if greater than	A > B
BGE	Branch if greater or equal	A ≥ B
BLT	Branch if less than	A < B
BLE	Branch if less or equal	A ≤ B
BE	Branch if equal	A = B
BNE	Branch if not equal	A ≠ B


Z
1
I_n
I_{n+1}
BZ 1000
1000: I_{i+j} ---

2. SUBROUTINE CALL AND RETURN

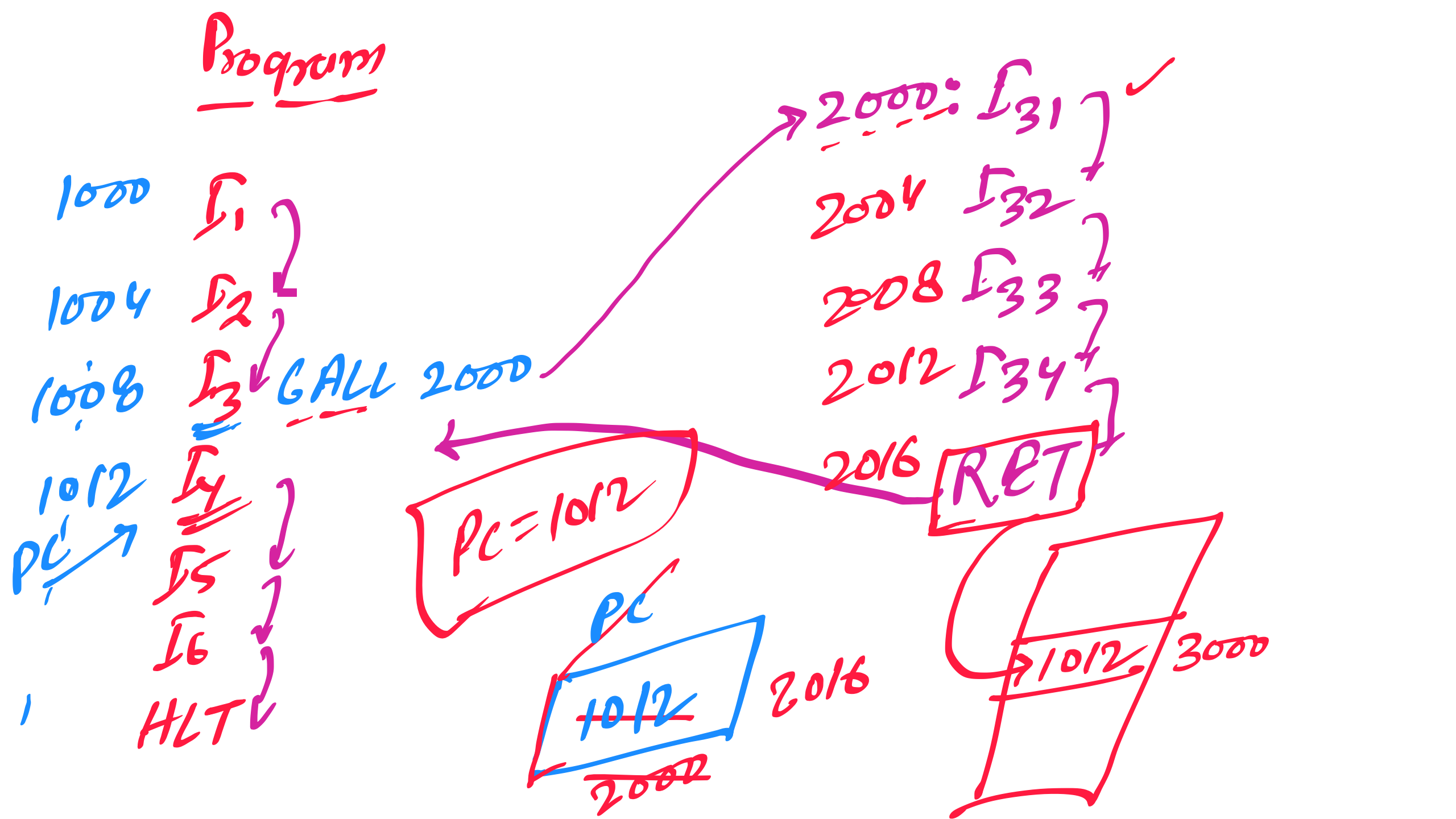
Subroutine is a self-contained sequence of instructions that performs a given computational task.

It may be called many times at various points in the main program.

When called, branches to 1st line of the subroutine and at the end, returned to the main program.

Different names for the instruction that transfers program control to a subroutine

- Call subroutine
- Jump to subroutine
- Branch to a subroutine
- Branch and save the return address


Program
1000 I1
1004 I2
1008 I3 CALL 2000
1012 I4
PC
I5
I6
HLT
2000: I31
2004 I32
2008 I33
2012 I34
2016 RET
PC=1012
PC
1012
2016
2000
1012
3000

2. SUBROUTINE CALL AND RETURN

Two Most Important Operations are Implied:

Branch to the beginning of the Subroutine - Same as the Branch or Conditional Branch.

Save the Return Address to get the address of the location in the Calling Program upon exit from the Subroutine

Locations for storing Return Address

Fixed Location in the subroutine (Memory)

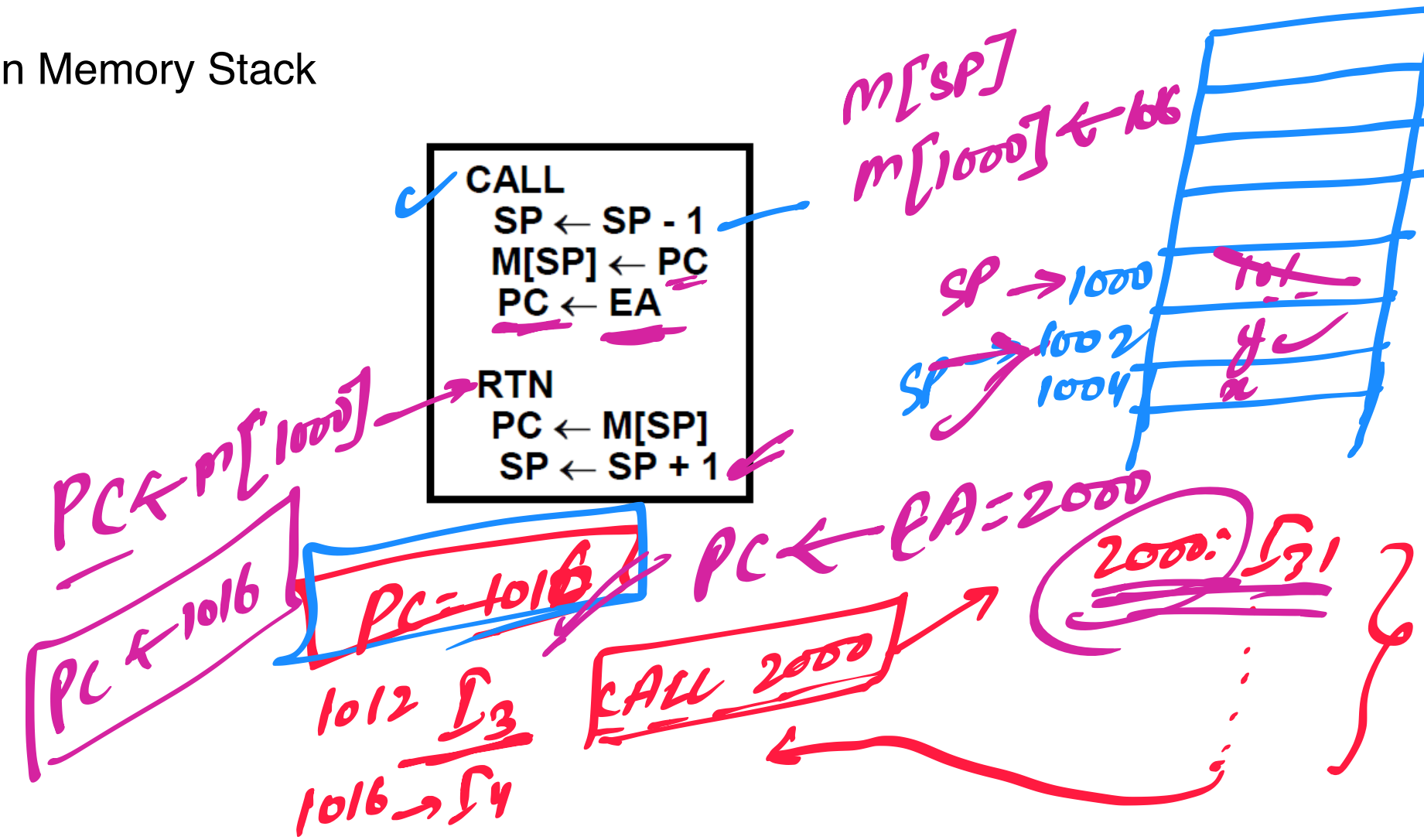
Fixed Location in memory

In a processor Register

In memory stack - most efficient way

2. SUBROUTINE CALL AND RETURN

In Memory Stack



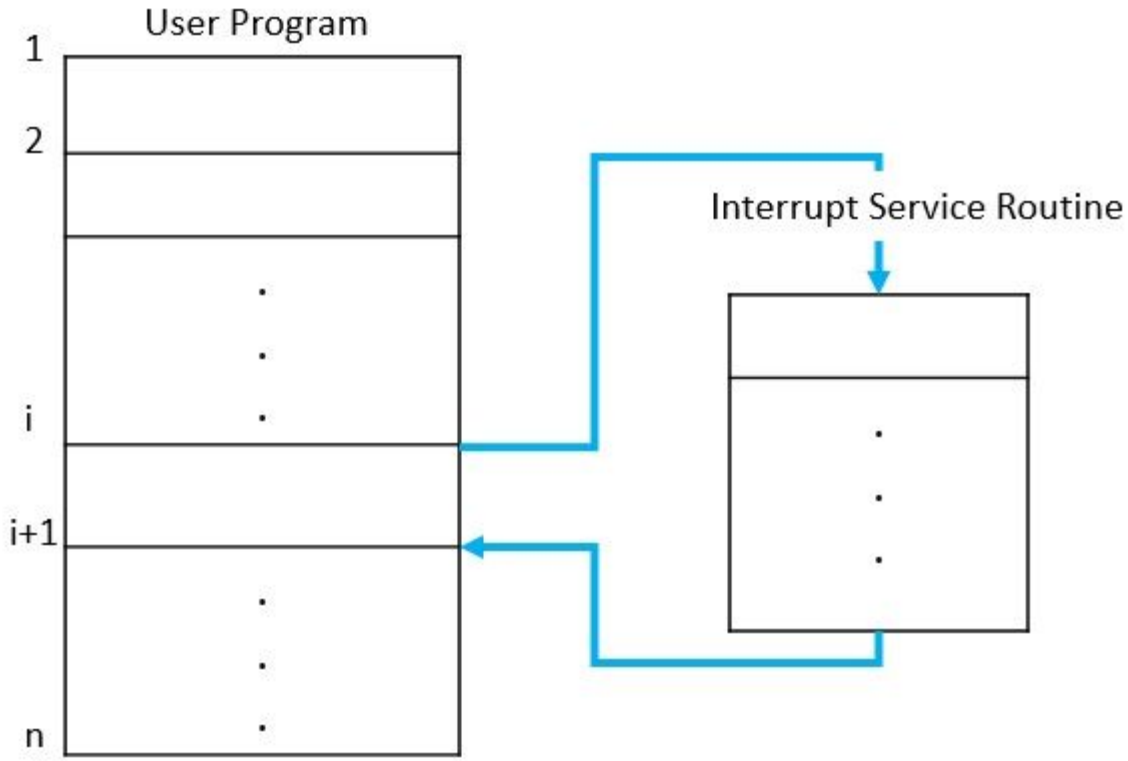
3. PROGRAM INTERRUPT

An interrupt is a signal emitted by a device attached to a computer or from a program within the computer.

An interrupt temporarily stops or terminates a service or a current process.

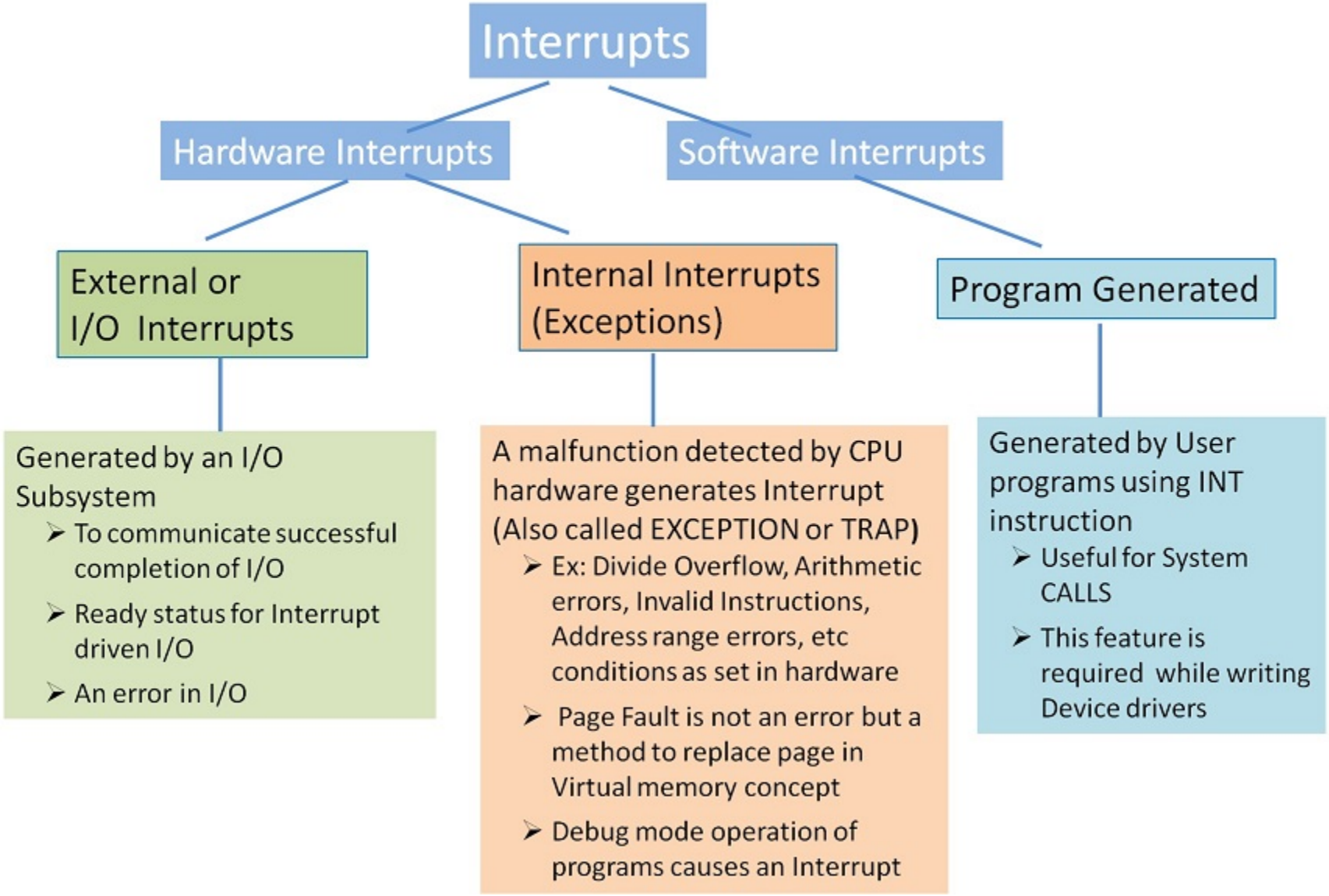
Types of Interrupts:

- External interrupts
- Internal interrupts
- Software interrupts



Transfer Control Via Interrupts

3. PROGRAM INTERRUPT



3. INTERRUPT PROCEDURE

Step 1 – First device issues interrupt to CPU.

Step 2 – Then, the CPU finishes the execution of the current instruction.

Step 3 – CPU tests for pending interrupt requests. If there is one, it sends acknowledgment to the device which removes its interrupt signal.

Step 4 – CPU saves program status word onto the control stack.

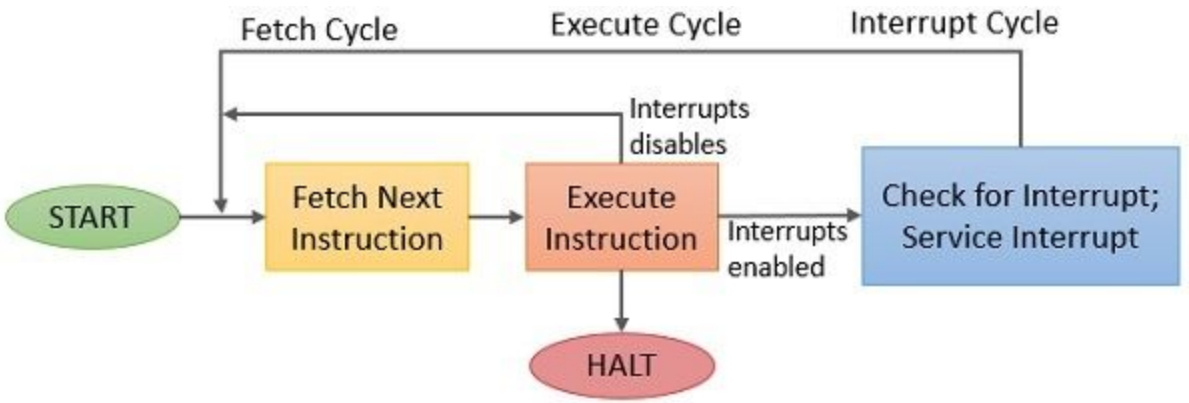
Step 5 – CPU loads the location of the interrupt handler into the PC registe

Step 6 – Save the contents of all registers from the control stack into memc

Step 7 – Find out the cause of interrupt, or interrupt type, or invokes t appropriate routine.

Step 8 – Restore saved registers from the stack.

Step 9 – Restore the PC to dispatch the original process



Instruction Cycle with Interrupts

7. RISC, AND CISC

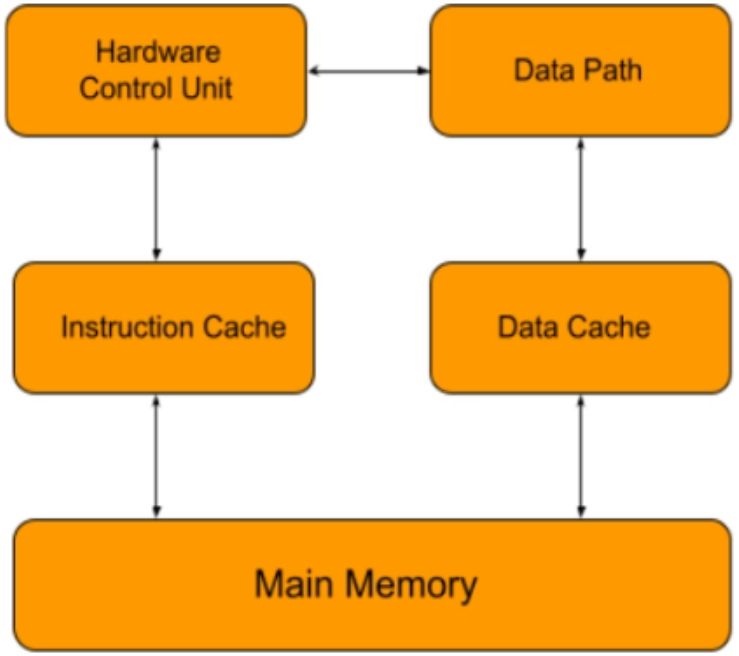
REDUCED INSTRUCTION SET COMPUTER PROCESSOR (RISC)

RISC is a microprocessor architecture that uses a simple set of instructions that can be substantially modified.

It is designed to reduce the time it takes for instructions to execute by optimizing and reducing the number of instructions. It means that each instruction cycle has only one clock per cycle, and each cycle consists of three parameters: fetch, decode, and execute.

The RISC processor can also combine multiple complex instructions into a simple one. RISC chips require several transistors, making them less expensive to develop and reducing instruction execution time.

Examples of RISC processors are PowerPC, Microchip PIC, SUN's SPARC, RISC-V.



RISC Architecture

FEATURES OF RISC PROCESSOR

RISC processors use one clock per cycle (CPI) to execute each instruction in a computer. Each CPI also comprises the methods for fetching, decoding, and executing computer instructions.

Multiple registers in RISC processors allow them to hold instructions, reply fast to the computer, and interact with computer memory as little as possible.

The RISC processors use the pipelining technique to execute multiple parts or stages of instructions to perform more efficiently.

RISC has a simple addressing mode and fixed instruction length for the pipeline execution.

It uses LOAD and STORE instructions to access the memory location.

COMPLEX INSTRUCTION SET COMPUTER (CISC)

Intel developed the CISC processor.

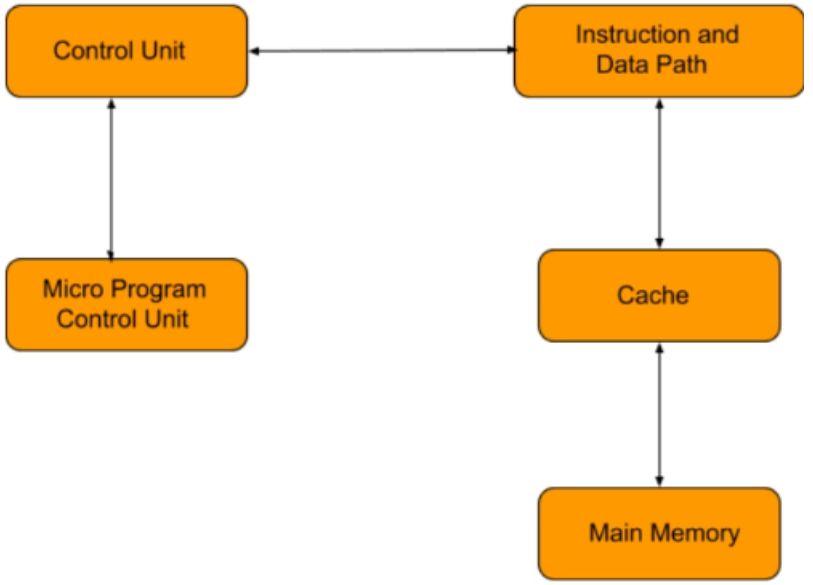
It has an extensive collection of complex instructions that range from simple to very complex and specializes in the assembly language level, which takes a long time to execute the instructions.

CISC approaches reducing the number of instructions on each program and ignoring the number of cycles per instruction.

It emphasizes building complex instructions directly in the hardware because the hardware is always faster than the software.

CISC chips are relatively slower than RISC chips but use little instructions.

Examples of CISC processors are AMD, Intel x86, and the System/360.



CISC Architecture

FEATURES OF CISC PROCESSOR

CISC may take longer than a single clock cycle to execute the code.

The length of the code is short, so it requires minimal RAM.

It provides more accessible programming in assembly language.

It focuses on creating instructions on hardware rather than software because they are faster to develop.

It comprises fewer registers and more addressing nodes, typically 5 to 20.

DIFFERENCE BETWEEN RISC AND CISC

	RISC	CISC
1	**Instructions of a fixed size**	Instructions of **variable size**
2	Most instructions take **same time to fetch**.	Instructions have **different fetching times**.
3	Instruction set **simple and small**.	Instruction set **large and complex**.
4	**Less addressing modes** as most operations are register based.	**Complex addressing modes** as most operations are memory based.
5	**Compiler design is simple**	**Compiler design is complex**
6	**Total size of program is large** as many instructions are required to perform a task as instructions are simple.	**Total size of program is small** as few instructions are required to perform a task as instructions are complex & more powerful.
7	Instructions use a **fixed number of operands.**	Instructions have **variable** number of operands.
8	Ideal for processors performing a **dedicated operation**.	Ideal for processors performing a **verity of operations**.
9	Since instructions are simple, they can be decoded by a **hardwired control unit**.	Since instructions are complex, they require a **Micro-programmed Control Unit**.
10	Execution speed is **faster** as most operations are register based.	Execution speed is **slower** as most operations are memory based.
11	As No. of cycles per instruction is fixed, it gives a **better degree of pipelining**	Since number of cycles per instruction varies, **pipelining has more bubbles** or stalls.
12	E.g.:: **ARM7**, **PIC 18** Microcontrollers.	E.g.: Intel **8085, 8086** Microprocessors.