

# Tema 5: Ensamblador para la CPU Teórica

## 5.1- Introducción al Lenguaje Ensamblador

## 5.2- Ensamblador para la CPU Teórica

### 5.2.1 – Formato del Lenguaje para la CPU Teórica

### 5.2.2 – Control del Flujo de Ejecución

### 5.2.3 – La Pila

### 5.2.4 – Subrutinas



## 5.1- Introducción al Lenguaje Ensamblador

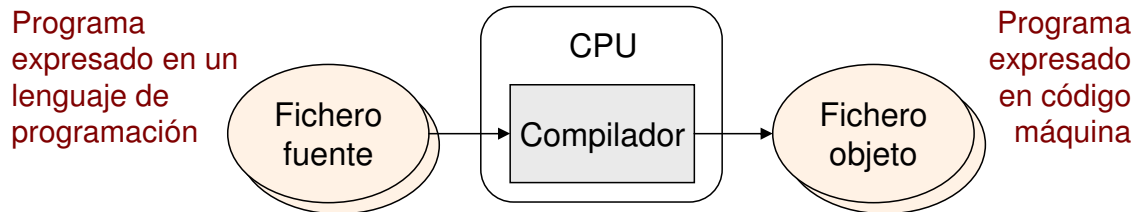
- El **código máquina** es una codificación en sistema binario, de una instrucción o de un conjunto de instrucciones, que puede ser ejecutado directamente por la CPU
- Para los humanos, programar en código máquina es complejo y propenso a errores
- El **lenguaje ensamblador** es un tipo de lenguaje de bajo nivel que surge como una herramienta para simplificar la programación en código máquina en **una CPU concreta**
- Esta simplificación se basa en dos aspectos fundamentales:
  - El empleo de **mnemónicos**, para representar las instrucciones
  - El empleo de **símbolos**, para representar datos y direcciones

### Objetivo del lenguaje ensamblador:

Liberar al programador de tener que recordar posiciones de memoria y códigos de instrucciones



- Un **compilador** es una herramienta software que permite traducir un programa escrito en un lenguaje de programación a código máquina



- Si el compilador traduce de **lenguaje ensamblador** a **código máquina**, se denomina **ensamblador**

## 5.2- Ensamblador para la CPU Teórica

### 5.2.1. FORMATO DEL LENGUAJE PARA LA CPU TEÓRICA

- ▶ Estructuración en sentencias
- ▶ Instrucciones
- ▶ Directivas
  - De definición de datos
  - Orientadas a dirigir el proceso de compilación
- ▶ Estructura de un programa en lenguaje ensamblador
- ▶ Constantes
- ▶ Símbolos
- ▶ Operadores
- ▶ Ejemplo

### • ESTRUCTURACIÓN EN SENTENCIAS

Un *programa* se estructura en una *secuencia de sentencias*.

En lenguaje ensamblador para la CPU Teórica, cada sentencia ocupa una línea.

Cada línea se compone de cuatro campos:

[**etiqueta**] [**operación**] [**operandos**] [**;comentario**]

Se suelen colocar en columnas en el fichero fuente, separados por tabuladores

Los corchetes indican que los campos son opcionales.

→ Una sentencia puede contener los cuatro campos, alguno o **ninguno**  
sentencia vacía

Dos *tipos de sentencias*, en función del campo operación:

- Instrucciones
- Directivas



## 5.2- Ensamblador para la CPU Teórica

### • INSTRUCCIONES

Una *instrucción en ensamblador* es una sentencia que da lugar a una *instrucción máquina* tras el proceso de ensamblado o compilación.

Las instrucciones del lenguaje ensamblador para la CPU Teórica son los mnemónicos de las instrucciones máquina de dicha CPU.

Las instrucciones representan *órdenes al procesador*.

Ejemplo → **MOV R3, R5**

operandos  
operación

### • DIRECTIVAS

Representan *órdenes al programa ensamblador* o compilador.

Se agrupan en dos clases:

- Directivas de *definición de datos*
- Directivas orientadas a *dirigir el proceso de compilación*



- DIRECTIVAS

**Directivas de definición de datos:** se utilizan para definir los datos de programa.

- Definición de un **dato de 16 bits**

Directiva → **VALOR**

Ejemplo →

```
dato1 VALOR 0
dato2 VALOR 5B3Ah
```

operandos  
operación  
etiqueta

- Definición de un **array de datos de 16 bits**

Directiva → **VECES**

Ejemplo →

```
lista VALOR 10 VECES 0
```

nº de elementos  
a definir

valor de  
inicialización



- DIRECTIVAS

**Directivas orientadas a dirigir el proceso de compilación**

- Definición de una **constante de 16 bits**

Directiva → **EQU**

Ejemplo →

```
dato EQU 58A4h
```

- Delimitación del **cuerpo de un procedimiento**

Directivas → **PROCEDIMIENTO, FINP**

Ejemplo →

```
PROCEDIMIENTO nombre
_____
_____
_____
FINP
```

conjunto de instrucciones  
del procedimiento



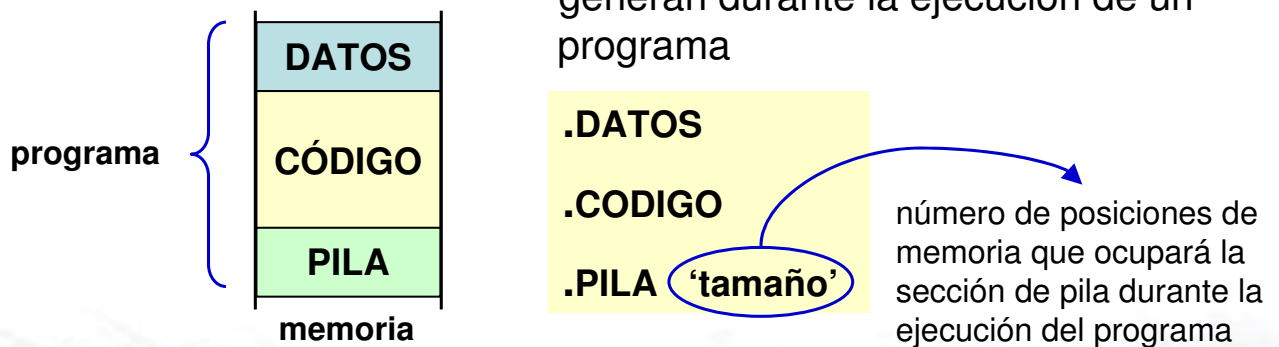
- DIRECTIVAS

### Directivas orientadas a dirigir el proceso de compilación

► **Estructuración de un programa** en secciones

Un programa, en memoria, se estructura en tres secciones:

- Sección de **datos**: Contiene los **datos** del programa
- Sección de **código**: Contiene las **instrucciones** del programa
- Sección de **pila**: Contiene los **datos temporales** que se generan durante la ejecución de un programa



- DIRECTIVAS

### Directivas orientadas a dirigir el proceso de compilación

► Control de **carga del programa**

Directiva → **ORIGEN** *dirección*

Dirección a partir de la cual se carga el programa en memoria.

Directiva → **INICIO** *etiqueta*

Primera instrucción del programa que la CPU ejecutará.

Ejemplo →

```

ORIGEN  10F0h
INICIO  primera
.CODIGO
primera:  _____
          _____
          _____
FIN
```



- DIRECTIVAS

Directivas orientadas a dirigir el proceso de compilación

- **Finalización de la sección de código** de un programa

Directiva → **FIN**

Ejemplo →

```
.CODIGO
_____
_____
FIN
```



- ESTRUCTURA DE UN PROGRAMA EN ENSAMBLADOR

Etiqueta	Operación	Operandos	Comentario
	ORIGEN	constante	; Dirección carga programa
	INICIO	etiqueta_ini	; Etiqueta de la 1ª instrucción
	.PILA	constante	; Asignación de tamaño a la pila
	.DATOS		
etiqueta1	VALOR	constante	; Datos del programa
etiqueta2	VALOR	constante	
...	...	...	
	.CODIGO		
etiqueta_ini:	mnemónico	operandos	; Instrucciones del programa
	...	...	
	...	...	
	FIN		





- CONSTANTES

### Constantes Enteras

Se utilizan para representar **números enteros**

Pueden ser de tres tipos:

Tipo	Especificador
Binario	b
Decimal	(sin especificador)
Hexadecimal	h

Tienen que empezar por número:  
A3Bh → Mal  
0A3Bh → Bien

Ejemplo: Definición del mismo dato utilizando los tres tipos de constantes

<b>.DATOS</b>				<b>0031</b>
numero_1	VALOR	49	; constante decimal	<b>0031</b>
numero_2	VALOR	31h	; constante hexadecimal	<b>0031</b>
numero_3	VALOR	00110001b	; constante binaria	



- CONSTANTES

### Constantes de Tipo Carácter

Se utilizan para representar caracteres.

Se expresan mediante un carácter entre comillas simples.

<b>.DATOS</b>				
caracter	VALOR	'a'		<b>0061</b> 'a'

### Constantes de Tipo Cadena

Representan cadenas de caracteres.

Se expresan mediante un grupo de caracteres entre comillas dobles.

<b>.DATOS</b>				
palabra	VALOR	"texto"		<b>0074</b> 't'
				<b>0065</b> 'e'
				<b>0078</b> 'x'
				<b>0074</b> 't'
				<b>006F</b> 'o'



### • SÍMBOLOS

Las direcciones de  $\left\{ \begin{array}{l} \text{datos} \\ \text{instrucciones} \end{array} \right\}$  se pueden representar mediante *etiquetas*

Las *etiquetas de datos* son símbolos

Representan las variables de los lenguajes de alto nivel de forma simbólica sobre la memoria

etiqueta  $\approx$  nombre de variable

no van seguidas de ':'

Las *etiquetas de instrucciones* son símbolos

Representan las estructuras de control de los lenguajes de alto nivel de forma simbólica sobre la memoria

etiqueta  $\approx$  destino de salto

deben ir seguidas de ':' en su definición



## 5.2- Ensamblador para la CPU Teórica

### • SÍMBOLOS

**Palabras Reservadas**  $\left\{ \begin{array}{l} \text{Mnemónicos de las instrucciones} \\ \text{Nombres de los registros} \\ \text{Directivas y operadores} \end{array} \right\}$

Las palabras reservadas son *símbolos predefinidos*

No pueden utilizarse para definir nuevos símbolos

Símbolos predefinidos  $\rightarrow$  CASE INSENSITIVE

MOV = Mov = mov

Símbolos definidos por el usuario  $\rightarrow$  CASE SENSITIVE

Etiqueta1  $\neq$  etiqueta1

Normalmente  $\left\{ \begin{array}{l} \text{Símbolos reservados} \rightarrow \text{MAYÚSCULAS} \\ \text{Símbolos de usuario} \rightarrow \text{minúsculas} \end{array} \right\}$





### • OPERADORES

Realizar operaciones sobre constantes y datos durante el proceso de ensamblado (  $\equiv$  en tiempo de compilación)

#### ► DIRECCIÓN *etiqueta*

Calcula una constante de 16 bits que representa la dirección de la instrucción o del dato al que hace referencia *etiqueta*

#### ► BYTEALTO *cte16*

Calcula una constante de 8 bits que representa el byte más significativo de *cte16*

#### ► BYTEBAJO *cte16*

Calcula una constante de 8 bits que representa el byte menos significativo de *cte16*



## 5.2- Ensamblador para la CPU Teórica

```

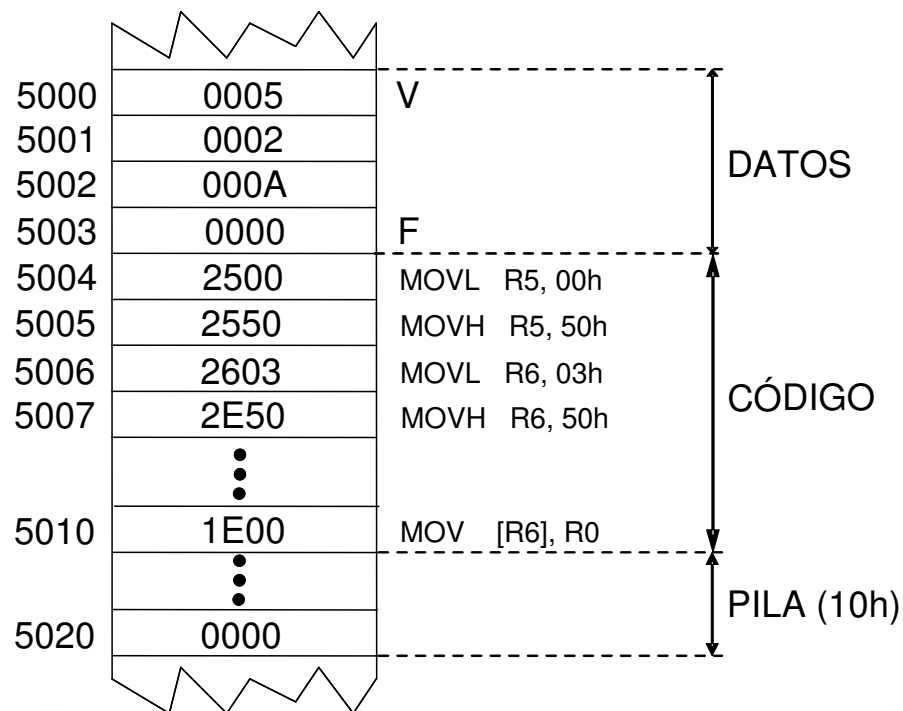
ORIGEN          5000h
INICIO          ini
.PILA           10h
.DATOS
V VALOR         5, 2, 10
F VALOR         0
.CODIGO
ini:  MOVL      R5, BYTEBAJO DIRECCION V
      MOVH      R5, BYTEALTO DIRECCION V
      MOVL      R6, BYTEBAJO DIRECCION F
      MOVH      R6, BYTEALTO DIRECCION F
      MOV       R0, [R5]
      ADD       R0, R0, R0
      INC       R5
      MOV       R1, [R5]
      ADD       R0, R0, R1
      INC       R5
      MOV       R1, [R5]
      SUB       R0, R0, R1
      MOV       [R6], R0
      FIN
    
```

#### EJEMPLO “Formula.ens”

Calcula:  $F = 2 \cdot V[0] + V[1] - V[2]$



Detalle del programa anterior, una vez ensamblado y cargado en memoria



### 5.2.2. CONTROL DEL FLUJO DE EJECUCIÓN

- Implementación de un condicional
- Ejemplos de condicionales
- Ejemplo de programa

### Problema

El código puramente *secuencial* tiene una *capacidad de resolución* de problemas muy *limitada*

### Solución

Utilizar *estructuras de control*

La CPU teórica proporciona *4 indicadores de estado* y su juego de instrucciones *2 instrucciones de salto condicional por indicador*

Indicador	Instrucciones	
<b>Z</b>	<b>BRZ</b>	<b>BRNZ</b>
<b>C</b>	<b>BRC</b>	<b>BRNC</b>
<b>O</b>	<b>BRO</b>	<b>BRNO</b>
<b>S</b>	<b>BRS</b>	<b>BRNS</b>



## 5.2- Ensamblador para la CPU Teórica

### • IMPLEMENTACIÓN DE UN CONDICIONAL

El mecanismo para realizar un condicional elemental se basa en *dos fases sucesivas*:

- 1) Comparar dos magnitudes → **COMP** Rs1, Rs2  $\equiv$  Rs1 - Rs2

Resta el segundo operando del primero, igual que la instrucción SUB, modificando los bits del registro de estado pero sin almacenar el resultado de la resta

- 2) Cuando se cumple una determinada relación entre dos magnitudes, ciertos bits del registro de estado toman unos valores determinados y/o cumplen a su vez una relación entre ellos

$$Rs1 \begin{pmatrix} = & \neq \\ > & \geq \\ < & \leq \end{pmatrix} Rs2$$



- IMPLEMENTACIÓN DE UN CONDICIONAL

La relación a comprobar entre los bits de estado *depende* del tipo de magnitudes: *sin signo* o *con signo*.

Magnitudes sin signo	Relación	Magnitudes con signo
ZF = 1	=	ZF = 1
ZF = 0	≠	ZF = 0
CF = 0 AND ZF = 0	>	SF = OF AND ZF = 0
CF = 0	≥	SF = OF
CF = 1	<	SF ≠ OF
CF = 1 OR ZF = 1	≤	SF ≠ OF OR ZF = 1

Tras la instrucción COMP se ejecutan las instrucciones de salto condicional apropiadas según los bits de estado que sea necesario analizar → BRC, BRNC, BRO, BRNO, BRS, BRNS, BRZ, BRNZ



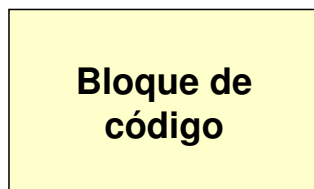
## 5.2- Ensamblador para la CPU Teórica

- EJEMPLOS DE CONDICIONALES

Interpretación de cantidades *sin signo*

Si ( R0 = R1 )

ejecutar



SI ( R0 ≠ R1 ) ENTONCES  
NO ejecutar Bloque de código  
SI NO  
ejecutar Bloque de código

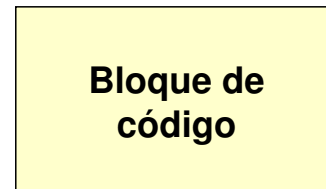


COMP

R0, R1

BRNZ

salto



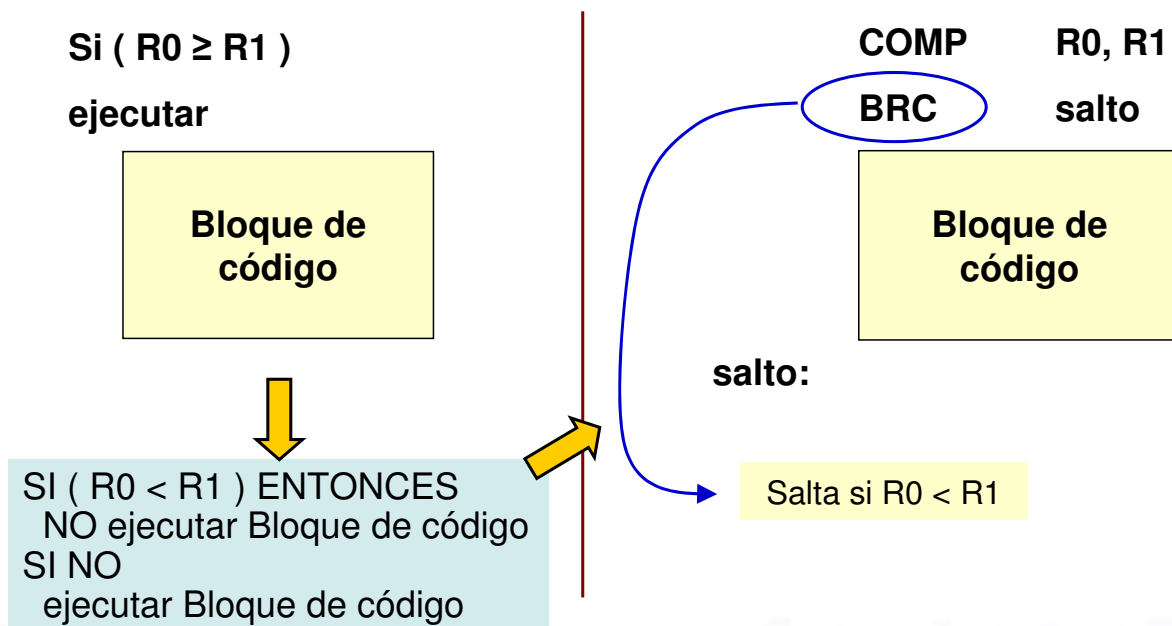
salto:

Salta si R0 ≠ R1



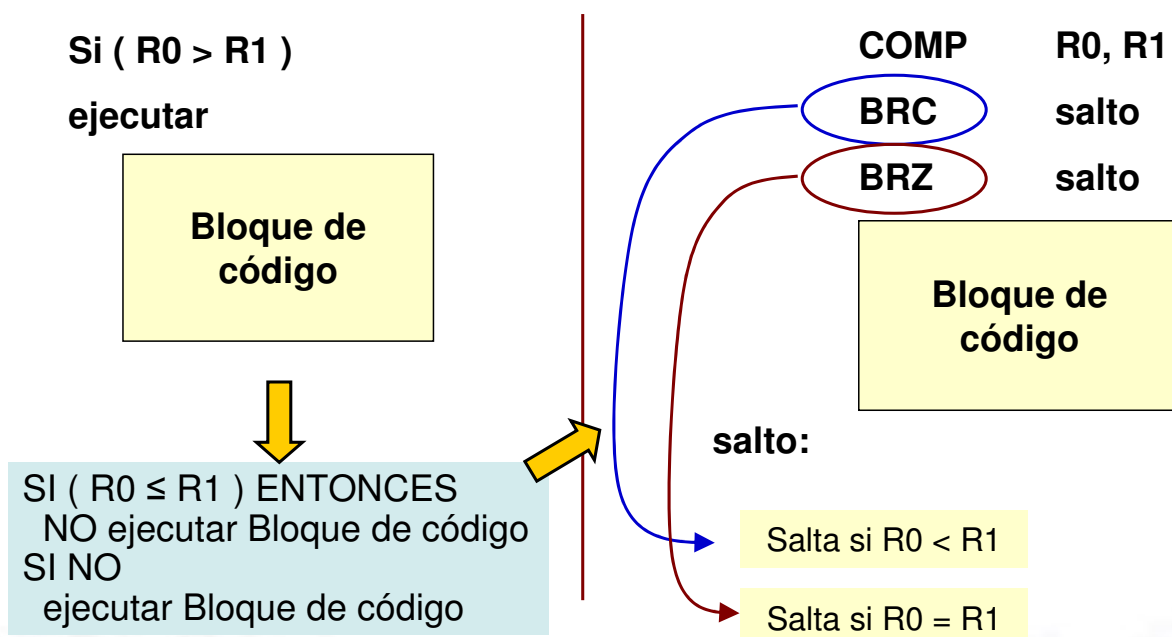
### • EJEMPLOS DE CONDICIONALES

Interpretación de cantidades *sin signo*



### • EJEMPLOS DE CONDICIONALES

Interpretación de cantidades *sin signo*



### • EJEMPLOS DE CONDICIONALES

Interpretación de cantidades *sin signo*

Si (  $R0 \leq R1$  )

ejecutar

Bloque de  
código

SI (  $R0 > R1$  ) ENTONCES  
NO ejecutar Bloque de código  
SI NO  
ejecutar Bloque de código

COMP

R0, R1

BRNC

compruebaZF

JMP

bloque

compruebaZF:

BRNZ

salto

bloque:

Bloque de  
código

salto:



## 5.2- Ensamblador para la CPU Teórica

### • EJEMPLOS DE CONDICIONALES

Interpretación de cantidades *sin signo*

Si (  $R0 = R1$  )

ejecutar

Bloque de  
código 1

si no, ejecutar

Bloque de  
código 2

COMP

R0, R1

BRNZ

si\_no

Bloque de  
código 1

JMP

salto

si\_no:

Bloque de  
código 2

salto:





### • EJEMPLO DE PROGRAMA

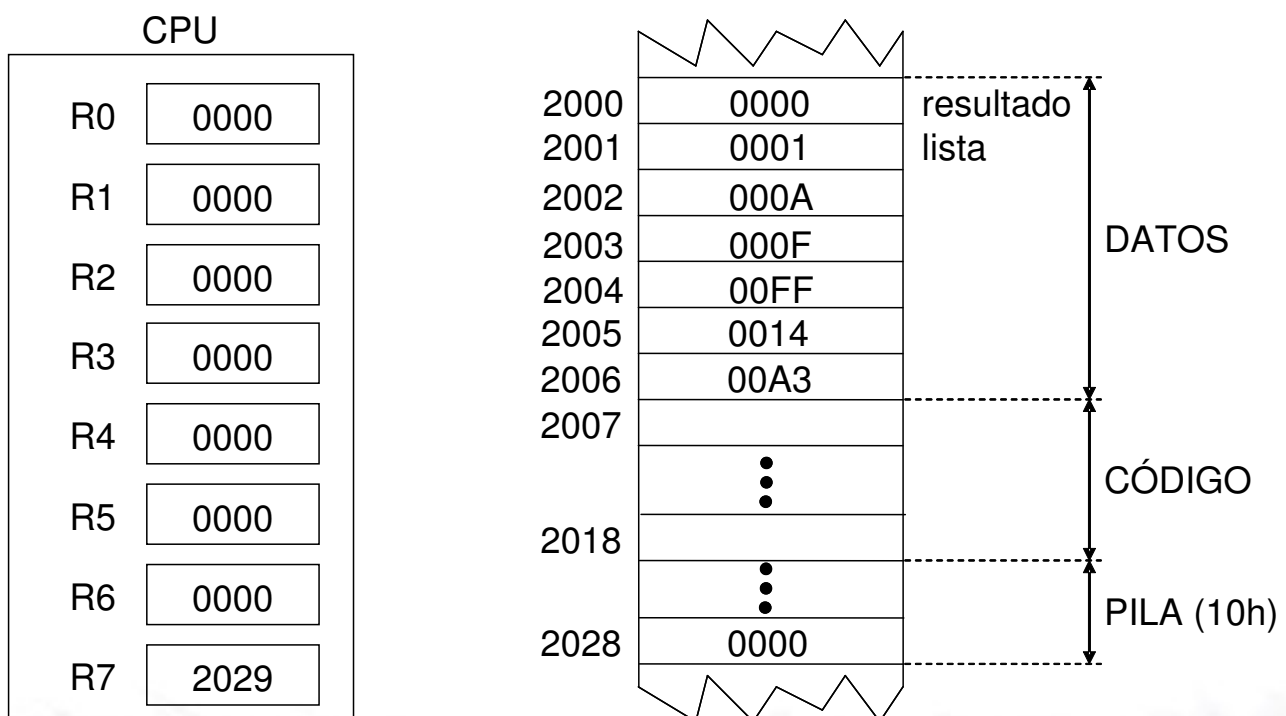
Escribir un programa en lenguaje ensamblador para la CPU teórica que procese una lista de 6 números (interpretados como magnitudes sin signo), sumando aquéllos que sean mayores o iguales a 10. El resultado de la suma se almacenará en una posición de la sección de datos etiquetada como resultado.

Inicio del programa en memoria → 2000h  
 Tamaño de la pila → 16  
 Números de la lista → 1, 10, 15, FFh, 20, A3h  
 R0 → Acumulador  
 R1 → Dato a procesar, temporalmente almacenado en el registro  
 R2 → Contador de elementos que aún quedan por procesar  
 R4 → Almacena el valor 10 (para realizar la comparación)  
 R5 → Dirección de memoria donde se almacenará el resultado  
 R6 → Dirección de memoria donde se encuentra el elemento de la lista que se procesa en cada momento



## 5.2- Ensamblador para la CPU Teórica

### • EJEMPLO DE PROGRAMA



### • EJEMPLO DE PROGRAMA

```

ORIGEN      2000h
INICIO      ini
.PILA       10h

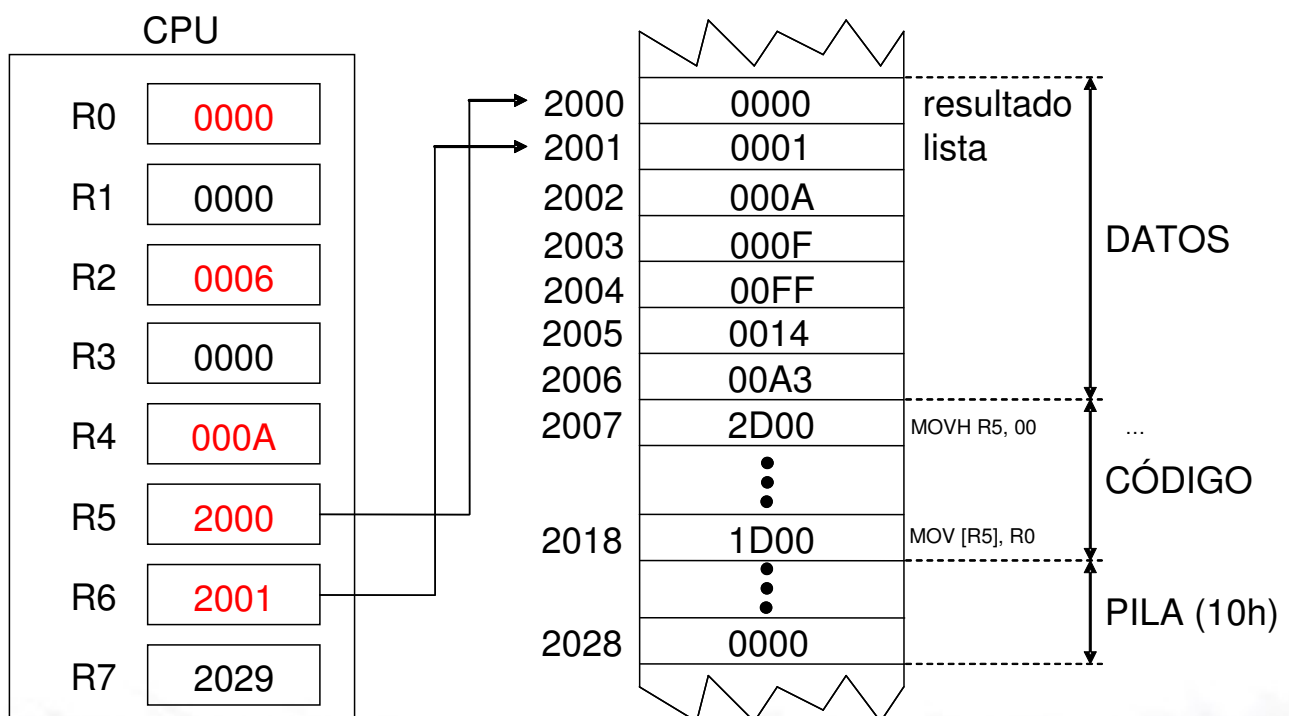
.DATOS
resultado VALOR      0
lista      VALOR      1, 10, 15, 0FFh, 20, 0A3h

.CODIGO
; inicializaciones
ini:        MOVL      R5, BYTEBAJO DIRECCION resultado
            MOVH      R5, BYTEALTO DIRECCION resultado
            MOVL      R6, BYTEBAJO DIRECCION lista
            MOVH      R6, BYTEALTO DIRECCION lista
            XOR       R0, R0, R0    ; acumulador = 0
            MOVH      R4, 00h
            MOVL      R4, 0Ah    ; el 10 para comparar
            MOVH      R2, 00h    ; número de elementos de
            MOVL      R2, 06h    ; la lista a procesar
    
```



## 5.2- Ensamblador para la CPU Teórica

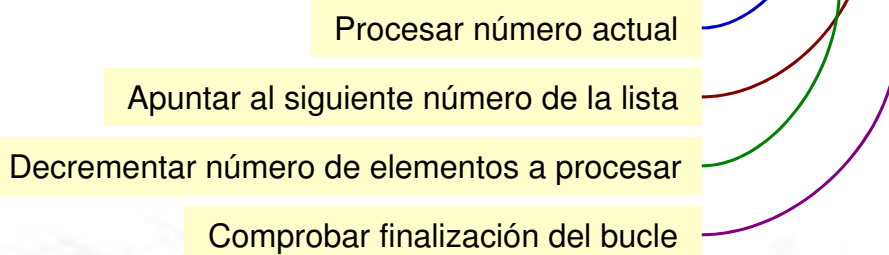
### • EJEMPLO DE PROGRAMA



### • EJEMPLO DE PROGRAMA

```

; comienzo del bucle
bucle:  MOV     R1, [R6]
        COMP   R1, R4
        BRC    sigue
sigue:  ADD     R0, R0, R1
        INC    R6
        DEC    R2
        BRNZ   bucle
        MOV    [R5], R0
        FIN
    
```



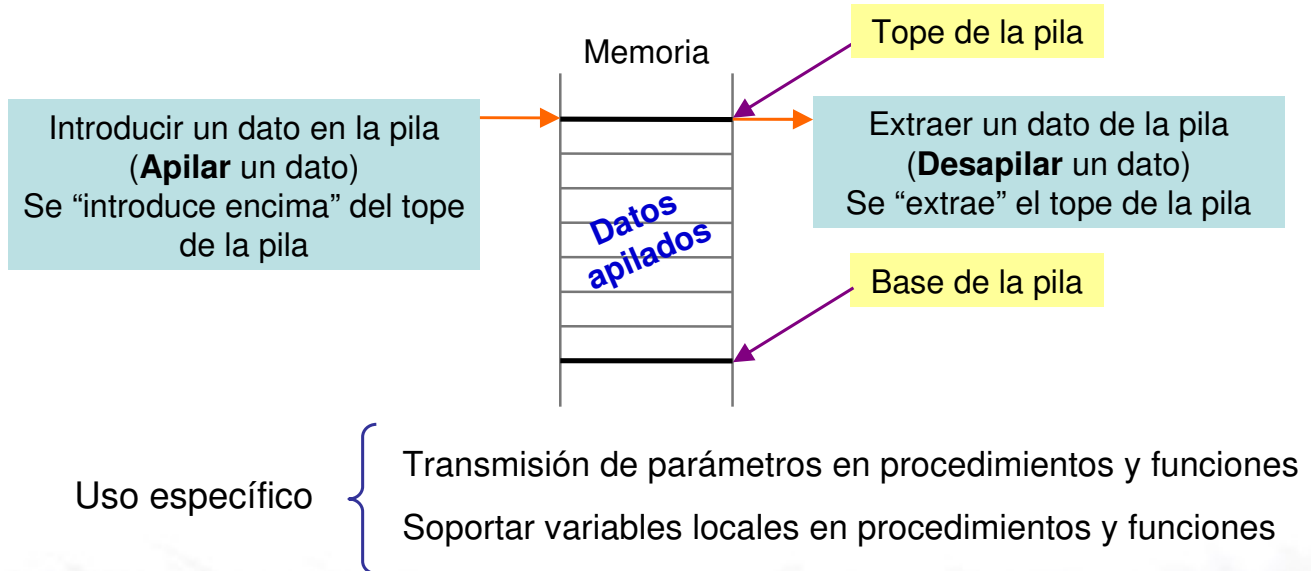
### 5.2.3. LA PILA

- ▶ Concepto de pila
- ▶ Implementación de la pila en la CPU Teórica
- ▶ Instrucciones de manejo de la pila
- ▶ Ejemplos de utilización de la pila



### • CONCEPTO DE PILA

Zona de memoria dedicada a almacenar datos temporales que se generan durante la ejecución de un programa



## 5.2- Ensamblador para la CPU Teórica

### • IMPLEMENTACIÓN DE LA PILA EN LA CPU TEÓRICA

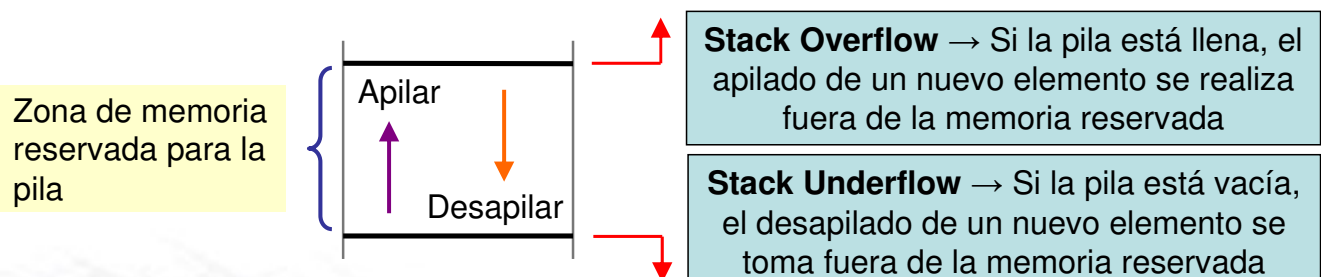
Se reserva una zona de memoria (un número concreto de palabras) para la pila utilizando la directiva **.PILA**

**.PILA 100h** ; Reserva 256 posiciones de memoria para la pila

Se utiliza un registro denominado **puntero de pila** (stack pointer o SP) que apunta siempre al último elemento apilado → al **tope de la pila**

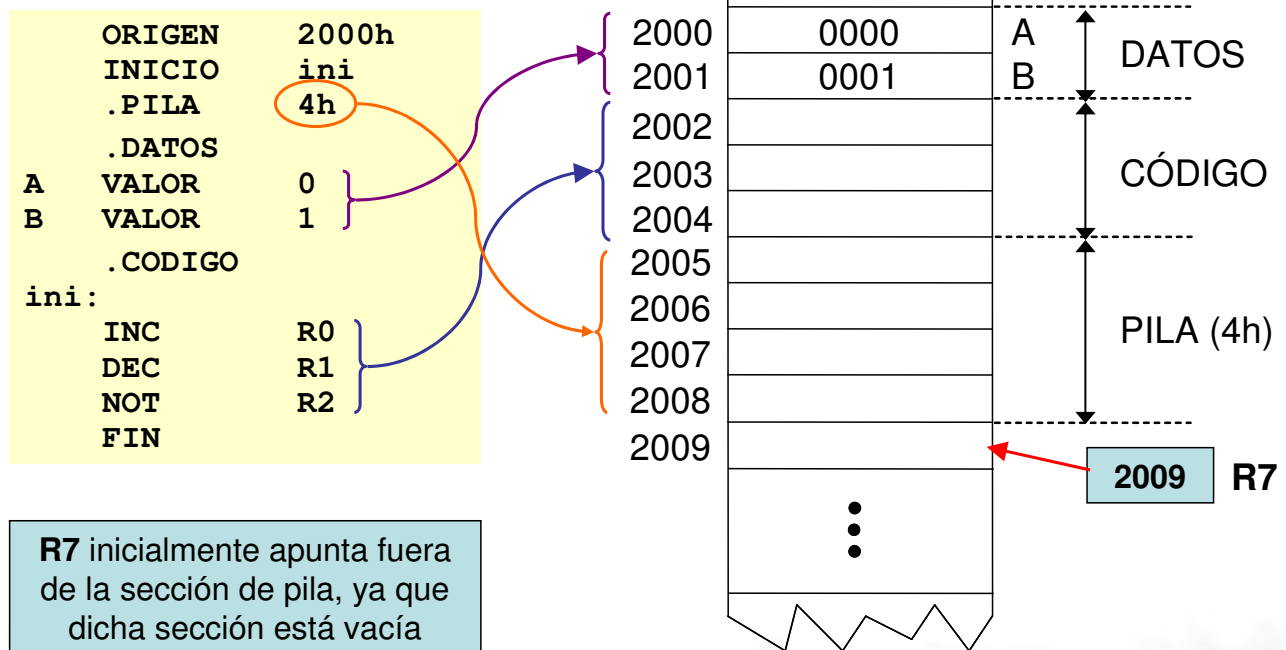
En la CPU Teórica no hay registro SP específico

→ se utiliza como **puntero de pila** el registro **R7**



### • IMPLEMENTACIÓN DE LA PILA EN LA CPU TEÓRICA

Ejemplo



## 5.2- Ensamblador para la CPU Teórica

### • INSTRUCCIONES DE MANEJO DE LA PILA

En el juego de instrucciones de la CPU Teórica hay varias instrucciones que durante su ejecución utilizan la pila como zona de almacenamiento temporal

Las instrucciones cuyo objetivo es introducir (*apilar*) y extraer (*desapilar*) datos de la pila son PUSH y POP, respectivamente

**PUSH Rs → Apila el contenido del registro Rs (R0...R7) en la pila**

**Funcionamiento:**  $R7 \leftarrow R7 - 1$   
 $[R7] \leftarrow Rs$

**POP Rd → Desapila la palabra del tope de la pila en el registro Rd (R0...R7)**

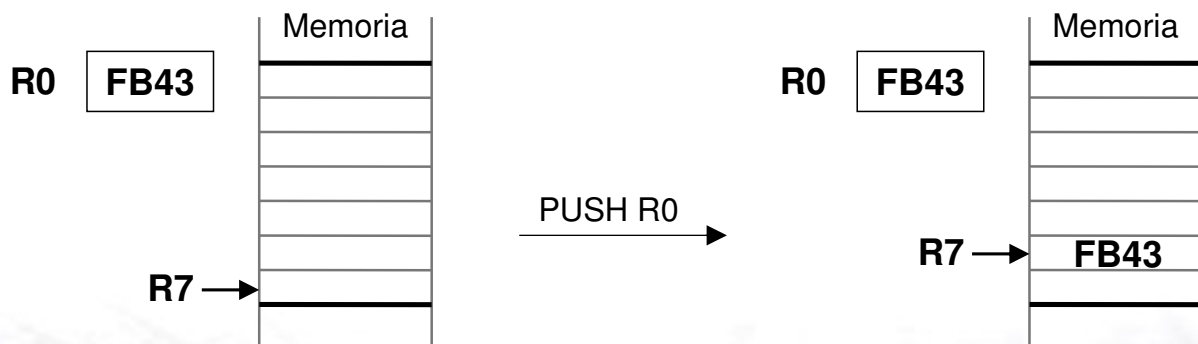
**Funcionamiento:**  $Rd \leftarrow [R7]$   
 $R7 \leftarrow R7 + 1$



### • PUSH

PASO	SEÑALES DE CONTROL
4	R7-IB, TMPE-SET, ADD, ALU-TMPS
5	Rs-IB, IB-MDR
6	TMPS-IB, IB-R7, IB-MAR, ESCRIBIR
7	FIN

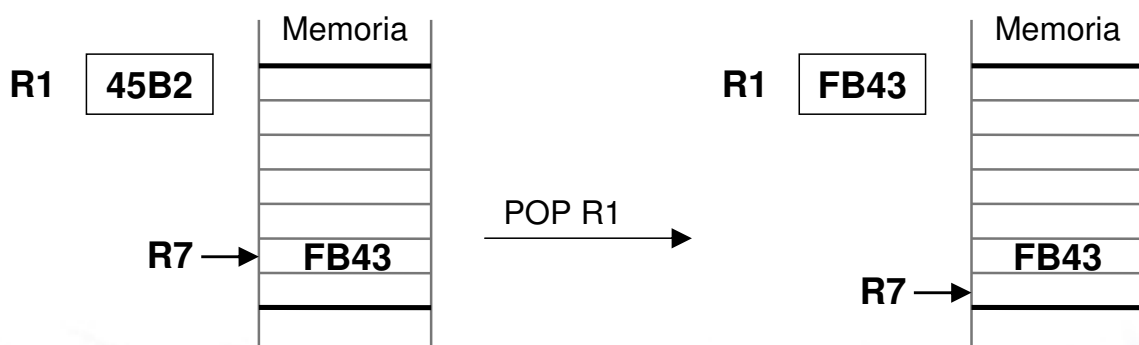
Ejemplo: PUSH R0



### • POP

PASO	SEÑALES DE CONTROL
4	R7-IB, IB-MAR, LEER, TMPE-CLR, CarryIn, ADD, ALU-TMPS
5	TMPS-IB, IB-R7
6	MDR-IB, IB-Rd, FIN

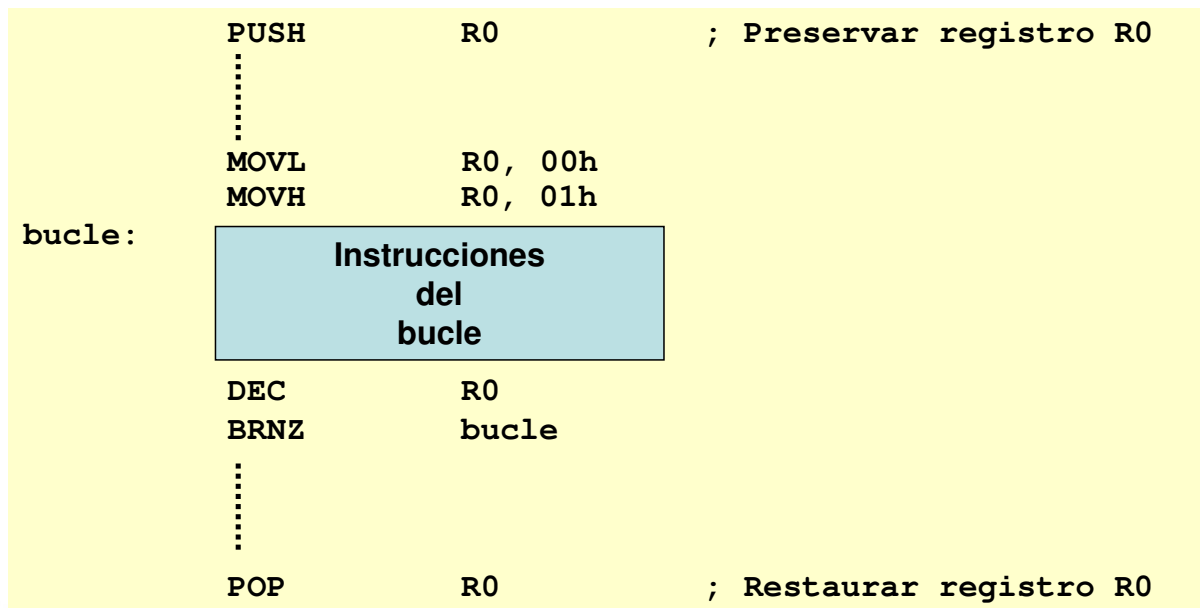
Ejemplo: POP R1





- EJEMPLOS DE UTILIZACIÓN DE LA PILA

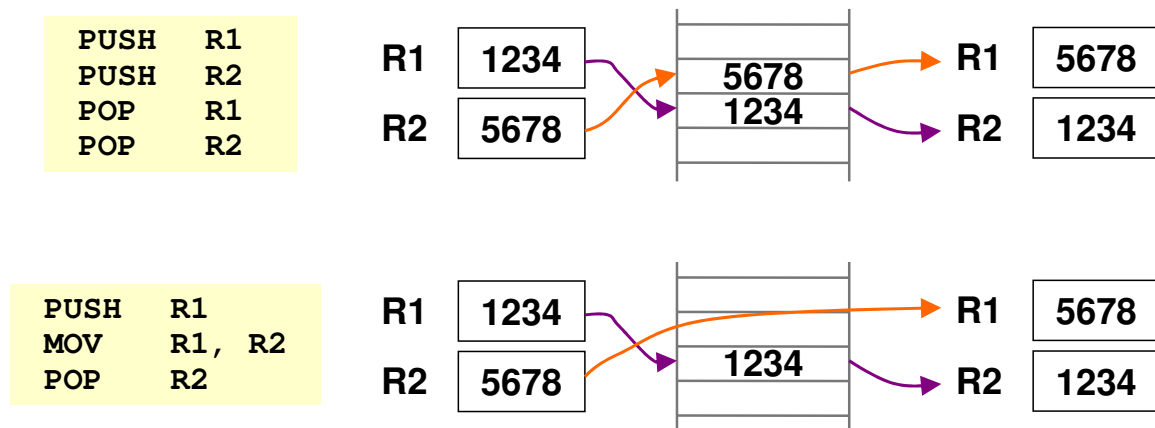
- 1) Almacén temporal



- EJEMPLOS DE UTILIZACIÓN DE LA PILA

- 2) Intercambio de datos entre registros

Intercambiar el contenido de los registros R1 y R2:  $R1 \leftrightarrow R2$



### 5.2.4. SUBROUTINAS

- ▶ Introducción
- ▶ Definición de subrutinas
- ▶ Instrucción de llamada
- ▶ Instrucción de retorno
- ▶ Ejemplo
- ▶ Transmisión de parámetros a subrutinas a través de la pila



## 5.2- Ensamblador para la CPU Teórica

### • INTRODUCCIÓN

Subrutina  $\approx$  Método  $\approx$  Función  $\approx$  Procedimiento  $\approx$  Subprograma

Fragmento de código dentro de un programa que...

- realiza una **tarea específica**
- es **relativamente independiente** del resto del código
- puede ser llamado desde cualquier punto del programa
- está preparado para retornar, una vez ejecutado, a la instrucción siguiente a la que provocó su ejecución

Componentes de una subrutina

- ▶ **Bloque de código**, que se ejecuta cuando se invoca a la subrutina
- ▶ **Parámetros de llamada**, que recibe la subrutina desde el punto de invocación
- ▶ **Valor de retorno**, de la subrutina al punto de invocación



### • INTRODUCCIÓN

Ventajas de dividir un programa en subrutinas:

- ▶ Reducción de redundancia en el programa
- ▶ Reutilización de código entre múltiples programas
- ▶ Descomponer problemas complejos en subproblemas (más fáciles de resolver, implementar y mantener)
- ▶ Incremento de la legibilidad del código

Esta forma de programación, utilizando subrutinas, se denomina **programación estructurada**

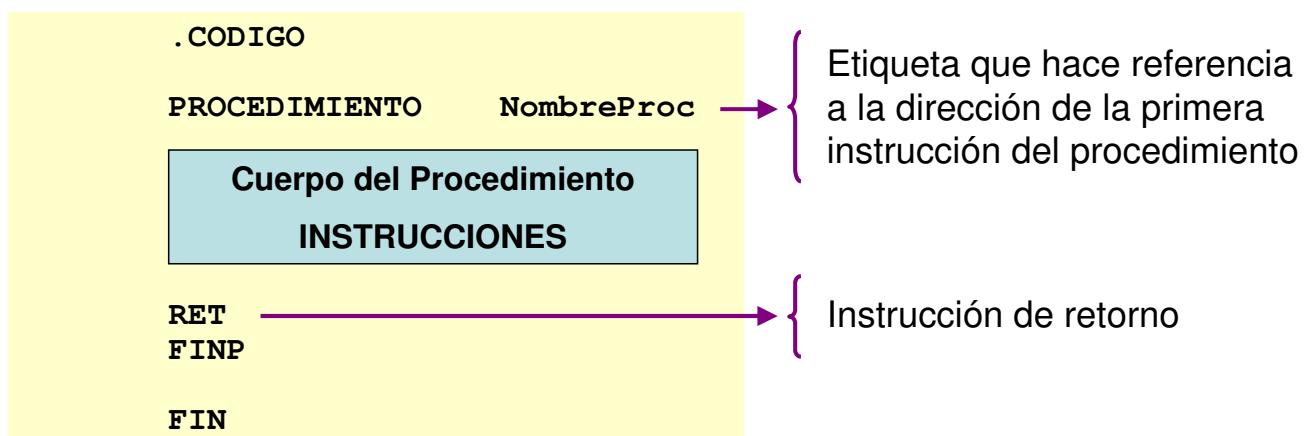


## 5.2- Ensamblador para la CPU Teórica

### • DEFINICIÓN DE SUBROUTINAS

Para definir una subrutina dentro de un fragmento de código se utilizan las directivas **PROCEDIMIENTO** y **FINP**

Ejemplo



## • INSTRUCCIÓN DE LLAMADA

Para llamar a una subrutina se dispone de **dos formas**

CALL **Etiqueta**

Dato inmediato de 8 bits  
Inm8

Se basa en un **salto relativo**

$$\begin{aligned} R7 &\leftarrow R7 - 1 \\ [R7] &\leftarrow PC \\ PC &\leftarrow PC + \text{Inm8} \end{aligned}$$

CALL **RX**

RX contiene la dirección  
de la primera instrucción  
del procedimiento

Se basa en un **salto absoluto**

$$\begin{aligned} R7 &\leftarrow R7 - 1 \\ [R7] &\leftarrow PC \\ PC &\leftarrow RX \end{aligned}$$

La instrucción CALL funciona de forma similar a la instrucción  
JMP equivalente (relativa o absoluta) con una diferencia

→ antes de saltar, almacena el valor del **PC renovado** en la **pila**



## • INSTRUCCIÓN DE RETORNO

Para retornar de un procedimiento se dispone de la instrucción

RET

$$\begin{aligned} PC &\leftarrow [R7] \\ R7 &\leftarrow R7 + 1 \end{aligned}$$

El retorno a la instrucción siguiente a la que provocó el salto, se  
consigue actualizando el registro PC con el valor del tope de la pila

### MUY IMPORTANTE:

En el momento de ejecutarse la instrucción RET, el  
registro que actúa como puntero de pila (R7) debe  
estar apuntando a la dirección de retorno almacenada  
por la instrucción CALL

En otro caso, el retorno es errático



## • EJEMPLO

Escribir un programa principal capaz de **multiplicar dos números naturales**

La operación de multiplicación la debe realizar la subrutina **multiplica**

El programa principal y la subrutina se **comunican** utilizando el siguiente convenio:

- El programa principal indica los dos números a multiplicar en los registros **R4** y **R5**
- La subrutina devuelve el resultado en el registro **R0**



## • EJEMPLO

	ORIGEN	1000h
	INICIO	ini
	.PILA	10h
	.DATOS	
Multiplicador	VALOR	2
Multiplicando	VALOR	3
Resultado	VALOR	0
	.CODIGO	
ini:	MOVL	R2, BYTEBAJO DIRECCION Multiplicador
	MOVH	R2, BYTEALTO DIRECCION Multiplicador
	MOV	R4, [R2]
	MOVL	R2, BYTEBAJO DIRECCION Multiplicando
	MOVH	R2, BYTEALTO DIRECCION Multiplicando
	MOV	R5, [R2]
	CALL	Multiplica
	MOVL	R2, BYTEBAJO DIRECCION Resultado
	MOVH	R2, BYTEALTO DIRECCION Resultado
	MOV	[R2], R0



## • EJEMPLO

```

PROCEDIMIENTO Multiplica
PUSH R4
XOR          R0, R0, R0 ; Inicializar R0 a 0
COMP        R4, R0      ; Comprobar si R4 es ≠ 0
BRZ         finbucle
bucle:      ADD          R0, R0, R5 ; Sumar Multiplicando
            DEC          R4          ; Hasta que
            BRNZ         bucle      ; Multiplicador sea 0
finbucle:   POP R4
            RET
            FINP
            FIN

```



## • TRANSMISIÓN DE PARÁMETROS A SUBROUTINAS A TRAVÉS DE LA PILA

El paso de parámetros a subrutinas utilizando la pila es el mecanismo estándar utilizado en los lenguajes de programación de alto nivel

### EJEMPLO

Implementar el procedimiento → **suma(lista, numele, total)**

Suma los 'numele' primeros elementos de 'lista' y almacena el resultado en 'total'

El procedimiento recibe tres parámetros:

- **lista**, por ***dirección***  
Dirección del primer elemento de la lista
- **numele**, por ***valor***  
Número de elementos a sumar
- **total**, por ***dirección***  
Dirección donde se almacena el resultado de la suma





### • EJEMPLO

	ORIGEN	1000h
	INICIO	ini
	.PILA	10h
	.DATOS	
numele	VALOR	4
lista	VALOR	1, 3, 5, 7, 9, 2, 4, 8
total	VALOR	0
	.CODIGO	
ini:	MOVL	R0, BYTEBAJO DIRECCION lista
	MOVH	R0, BYTEALTO DIRECCION lista
	PUSH	R0 ; Apilar dirección de lista
	MOVL	R0, BYTEBAJO DIRECCION numele
	MOVH	R0, BYTEALTO DIRECCION numele
	MOV	R1, [R0]
	PUSH	R1 ; Apilar número de elementos
	MOVL	R0, BYTEBAJO DIRECCION total
	MOVH	R0, BYTEALTO DIRECCION total
	PUSH	R0 ; Apilar dirección de total

Fase 1  
Apilado de  
parámetros



### • EJEMPLO

Fase 2 Llamada	→	CALL	suma
Fase 9 Limpieza de parámetros	{	INC	R7
		INC	R7
		INC	R7
Fase 3 Secuencia de entrada	{	PROCEDIMIENTO	suma
		PUSH	R6 ; Utilizar R6 como base de
		MOV	R6, R7 ; la pila
Fase 4 Preservar registros	{	PUSH	R0
		PUSH	R1
		PUSH	R2
		PUSH	R3
		PUSH	R4

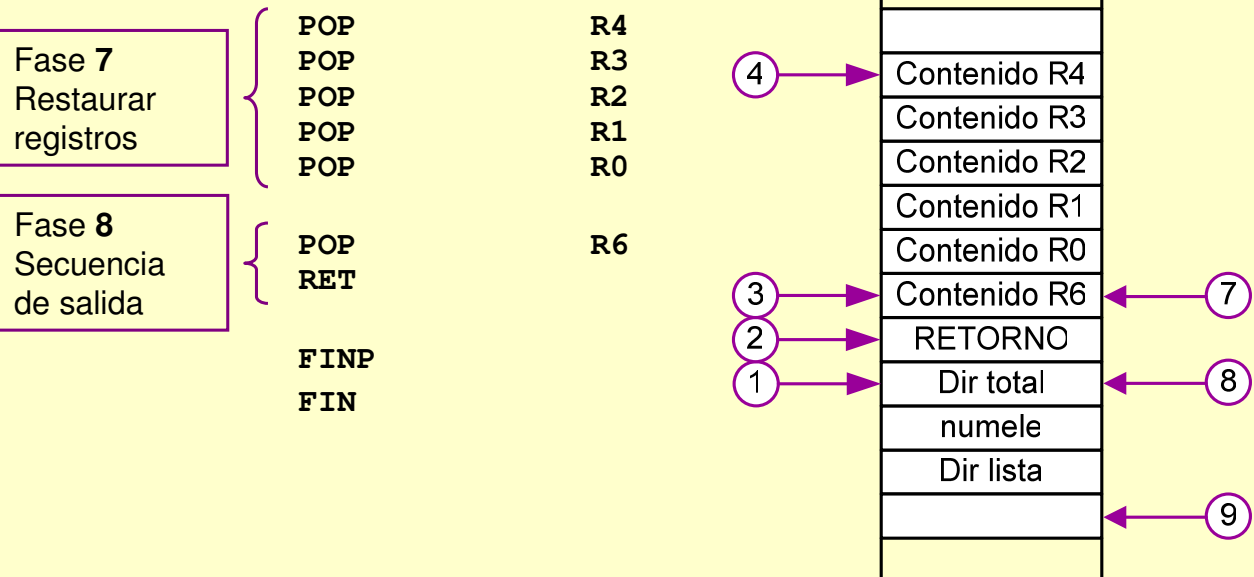


### • EJEMPLO

Fase 5 Acceso a parámetros	{	INC	R6	
		INC	R6	
		MOV	R1, [R6]	; R1 ← Dir Total
		INC	R6	
		MOV	R2, [R6]	; R2 ← Num Elementos
		INC	R6	
		MOV	R3, [R6]	; R3 ← Dir Lista
bucle:	{	XOR	R0, R0, R0	Fase 6 Cuerpo de la rutina
		COMP	R2, R0	
		BRZ	finproc	
		MOV	R4, [R3]	
		ADD	R0, R0, R4	
		INC	R3	
		DEC	R2	
		BRNZ	bucle	
finproc:		MOV	[R1], R0	



## 5.2- Ensamblador para la CPU Teórica



Estado de R7 al finalizar cada fase

