

Tema 4

Programación en lenguaje ensamblador

Curso 2008-2009

Objetivos

- Conocer el concepto de lenguaje ensamblador, y los procesos de ensamblado y carga.
- Conocer un lenguaje ensamblador de ejemplo.
- Ser capaz de traducir algoritmos sencillos, de *pseudocódigo* a lenguaje ensamblador.
- Conocer la existencia de la *pila* y comprender su función en la programación de procedimientos.
- Comprender cómo los conceptos de programación estructurada (procedimientos, paso de parámetros) son implementados en ensamblador.

Esquema

- 1 Conceptos
- 2 Ensamblador para la CPU elemental
- 3 Programación de algoritmos en ensamblador
- 4 La pila y las llamadas
- 5 Procedimientos

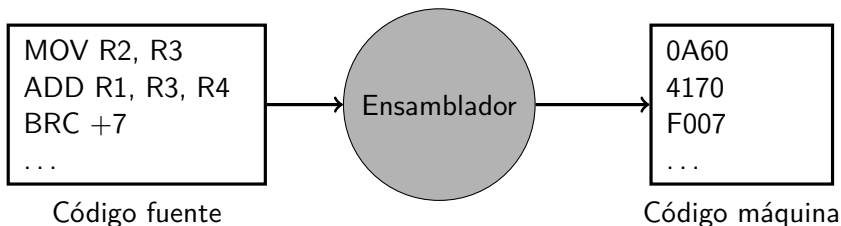
Esquema del apartado 1

- 1 Conceptos
 - Ensamblado y carga
 - Directivas y mnemónicos
- 2 Ensamblador para la CPU elemental
- 3 Programación de algoritmos en ensamblador
- 4 La pila y las llamadas
- 5 Procedimientos

Ensamblador

Cocepto

Es un programa que lee de un fichero de texto una secuencia de instrucciones, y genera en otro fichero su codificación.



También se llama *ensamblador* al lenguaje en que se escribe el código fuente para este caso.

Carga del programa

Para ejecutar un programa, no basta con tener su código máquina

- Es necesario cargarlo en la memoria
 - En una dirección concreta
- Es necesario inicializar algunos registros
 - En particular, PC debe apuntar a la primera instrucción que se debe ejecutar en el programa
 - También la pila debe inicializarse

0300	0A60
0301	4170
0302	F007
0303	...

PC 0300

Fichero ejecutable

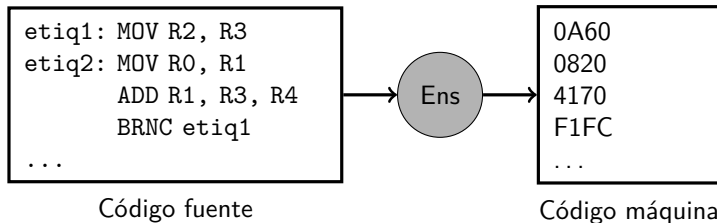
- El fichero “ejecutable” contiene no sólo código máquina, sino también información de carga.
- El ensamblador debe generar también esa información

Cálculo de saltos

Otro aspecto en el que puede ayudar el ensamblador es en el cálculo automático de la cantidad a sumar a PC en un salto.

Etiquetar las instrucciones

- Junto a cada instrucción se puede poner una *etiqueta*
- En los saltos escribimos la etiqueta de la instrucción destino
- El ensamblador calculará la cantidad correcta.



Otras utilidades

El ensamblador puede ayudarnos también...

- A codificar datos, escritos en base 2, 10 ó 16
- A colocar estos datos en la memoria
- A referirnos a esos datos por un “nombre” (etiqueta)
- A calcular en qué dirección de memoria está un dato

Por tanto...

Necesitamos dos tipos de instrucciones en un programa:

- Las instrucciones del programa propiamente dichas (que deben ser convertidas a código máquina)
- Comandos para el programa ensamblador (que este debe “ejecutar” en lugar de codificar)

Hay dos tipos de texto en un listado ensamblador

- **Mnemónicos:** son nombres de instrucciones de la CPU
 - ⇒ El ensamblador obtiene su código máquina
 - Ej: `MOV R3, R2`
`ADD R2, R1, R4`
- **Directivas:** son instrucciones para el programa ensamblador
 - ⇒ Indican aspectos como: dónde cargar el programa, si lo que sigue son datos o instrucciones, cálculo de direcciones, estructura del fuente, etc.
 - Ej: `INICIO`
`.DATOS`
`DIRECCION`

Esquema del apartado 2

- 1 Conceptos
- 2 Ensamblador para la CPU elemental
 - Mnemónicos y datos
 - Directivas y etiquetas
 - Estructura del código fuente
 - Un ejemplo completo
- 3 Programación de algoritmos en ensamblador
- 4 La pila y las llamadas
- 5 Procedimientos

Mnemónicos

Concepto

Son los nombres de las instrucciones de la CPU.

Son convertidas en código máquina por el ensamblador.

Notas

- Delante de cada instrucción puede ir una *etiqueta*, finalizada por dos puntos
- Los saltos pueden especificar una *etiqueta* en lugar de la cantidad a sumar a PC
- Algunas instrucciones pueden llevar un *dato* (ej: MOVH y MOVL). Este dato puede especificarse en diferentes bases, o calcularse con directivas.

Especificación de datos

Un dato puede escribirse:

- En base 10
- En base 2:
 - Irá finalizado por “b”
 - Ejemplos: 1000 1010b, 1111 1111b
- En base 16:
 - Irá finalizado por “h”
 - La primera cifra ha de ser un número 0-9
 - Ejemplos: 8FA0h, 0FFFFh
- Como un carácter entre comillas simples:
 - El ensamblador codifica el código ASCII
 - Ejemplo: 'A' se codifica como 01000001

Ejemplo

Las cuatro expresiones siguientes se codifican igual:

- 65
- 01000001b
- 41h
- 'A'

Directivas

Concepto

Son “palabras especiales” que guían el proceso de compilación.

- Indican dónde cargar el programa
- Qué instrucción será la primera en ejecutarse
- El tamaño de la pila
- Permiten definir datos que se cargarán en memoria
- La estructura en *secciones* del programa
- Permiten calcular direcciones
- Dividir datos de 16 en mitades de 8 bits
- etc.

Estructura en memoria de un programa

¿Datos o instrucciones?

- La memoria no hace diferencia entre datos o instrucciones
- En teoría, pueden mezclarse libremente
- Pero se recomienda ser “ordenado” y seguir un esquema fijo
 - Juntar todos los datos
 - Juntar todas las instrucciones
 - Reservar un espacio para la *pila*

Estructura en memoria de un programa

Esquema generado por el ensamblador

El ensamblador proporciona directivas para estructurar el programa en tres secciones:

- **Datos:** comienzan en la dirección de carga
Tamaño depende de los datos contenidos
- **Código:** comienza tras el último dato
Tamaño depende de las instrucciones contenidas
- **Pila:** comienza tras la última instrucción
Tamaño prefijado por el programador



Directivas de estructura en memoria

División del fuente en secciones

.PILA N Declara el tamaño (N) a reservar para la pila

.DATOS Indica que comienza la sección de datos

.CODIGO Indica que comienza la sección de código

FIN Indica que finaliza la sección de código

Carga del programa

ORIGEN N Indica la dirección de memoria (N) en la que se cargará el programa

INICIO etiq Señala la etiqueta que lleva la primera instrucción a ejecutar

Etiquetas

Concepto

Es un texto libremente elegido por el programador para identificar una línea del programa.

Sintaxis

- Están formadas por letras, números, y “_”
- En la sección de código deben terminar por “:”

Para qué sirven

- **En la sección de datos**, para identificar con un *nombre* las direcciones de memoria en las que están los datos
- **En la sección de código**, para identificar el destino de un salto.

Directivas para definir datos

Directiva VALOR

- Sólo puede aparecer en la sección de datos
- Permite definir un dato o una secuencia de ellos
- La línea con esta directiva puede llevar una etiqueta

Ejemplos

ORIGEN 0300h

.DATOS

VALOR 25

p VALOR 26h

VALOR 10

x VALOR -1, 0110b, 0FE45h

0300	0019	p ≡ 0301
0301	0026	
0302	000A	
0303	FFFF	x ≡ 0303
0304	0006	
0305	FE45	
...		

Directivas para definir datos

Directiva VECES

- Sólo puede aparecer tras la directiva VALOR
- Permite definir rápidamente un conjunto de datos iguales

Sintaxis

VALOR *M* VECES *N*

Crea una serie de *M* datos iguales, con el valor *N*.

Ejemplos

ORIGEN 0400h

.DATOS

lista VALOR 2 VECES 5

nums VALOR 4 VECES 0ACF2h

0400	0005	lista ≡ 0400
0401	0005	
0402	ACF2	nums ≡ 0402
0403	ACF2	
0404	ACF2	
0405	ACF2	
...		

Operadores

Concepto

Causan que el ensamblador realice ciertos cálculos y codifique el resultado.

BYTEALTO

Este operador recibe un dato de 16 bits, y calcula cuáles son los 8 bits superiores. Ejemplos:

BYTEALTO 12

BYTEALTO 7F45h

BYTEALTO 258

BYTEBAJO

Este operador recibe un dato de 16 bits, y calcula cuáles son los 8 bits inferiores.

Operadores

DIRECCION

Este operador recibe el nombre de una etiqueta, y resulta en 16 bits que son la dirección asociada a dicha etiqueta.

Ejemplo

Basándonos en los datos declarados en ejemplos previos:

DIRECCION p

DIRECCION x

BYTEALTO DIRECCION p

Uso

El uso típico es en conjunción con BYTEBAJO y BYTEALTO para inicializar registros con direcciones de datos.

Esquema de cada línea

El fichero fuente se compone de líneas

Una línea puede estar en blanco o contener (en este orden)

- Una etiqueta
- Una sentencia (mnemónico o directiva), con sus argumentos
- Un comentario, que comienza por el carácter “;”

Estos campos son opcionales. Pueden estar los tres o faltar cualesquiera de ellos.

Ensamblado

- Las líneas que contienen mnemónicos o datos, dan lugar a código en el fichero resultante
- Las que contienen directivas, sólo comentarios, o sólo etiquetas no dan lugar a código.

Esquema del fichero fuente

Estructura general

Un fichero fuente tiene extensión `.ens` y sigue la siguiente estructura:

- Una directiva `ORIGEN` para indicar dónde se cargará el programa
- Una directiva `INICIO` para indicar cuál es la primera instrucción a ejecutar
- Opcionalmente, una directiva `.PILA`
- Opcionalmente, una directiva `.DATOS` seguida de directivas de declaración de datos.
- Una directiva `.CODIGO` seguida de una serie de instrucciones (mnemónicos), uno por línea.
- La directiva `FIN`

Ejemplo completo

*; Este programa calcula la suma de dos datos llamados
; dato1 y dato2, y deja el resultado de la suma en
; otra variable llamada resultado.*

ORIGEN 200h

INICIO **primera**

.PILA 4

.DATOS

dato1 VALOR 30h

dato2 VALOR 20h

resultado VALOR 0 *; Inicialmente*

sigue ►►

Ejemplo completo

➤➤ *sigue*

.CODIGO

; El registro R0 contendrá las direcciones,

; R1 y R2 contendrán cada dato. R3 el resultado

primera:

MOVH R0, BYTEALTO DIRECCION dato1

MOVL R0, BYTEBAJO DIRECCION dato1

MOV R1, [R0]

INC R0

MOV R2, [R0]

ADD R3, R1, R2

INC R0

MOV [R0], R3

FIN

Esquema del apartado 3

- 1 Conceptos
- 2 Ensamblador para la CPU elemental
- 3 Programación de algoritmos en ensamblador**
 - Condicionales
 - Bucles
 - Estructura de datos array
 - Ejemplo de algoritmo
- 4 La pila y las llamadas
- 5 Procedimientos

Condición SI...ENTONCES

Pseudocódigo

```
SI (R1==R2)
{
    INSTR1
    INSTR2
    INSTR3
    . . .
}
INSTR4
INSTR5
```

Ensamblador

```
COMP R1,R2
BRNZ Son_diferentes
INSTR1
INSTR2
INSTR3
. . .
Son_diferentes:
INSTR4
INSTR5
```

Condición SI...ENTONCES...SI NO

Pseudocódigo

```
SI (R1==R2)
```

```
{
```

```
    INSTR1
```

```
    INSTR2
```

```
    INSTR3
```

```
    . . .
```

```
}
```

```
SI NO {
```

```
    INSTR4
```

```
    INSTR5
```

```
    . . .
```

```
}
```

```
INSTR6
```

```
INSTR7
```

Ensamblador

```
COMP R1,R2
```

```
BRNZ Son_diferentes
```

```
INSTR1
```

```
INSTR2
```

```
INSTR3
```

```
...
```

```
JMP Continuar
```

```
Son_diferentes:
```

```
INSTR4
```

```
INSTR5
```

```
...
```

```
Continuar:
```

```
INSTR6
```

```
INSTR7
```

Según Casos (*switch*)

Pseudocódigo

```

SEGUN (R1) {
    CASO 0: INSTR1
           INSTR2
           break;
    CASO 1: INSTR3
           INSTR4
           break;
    default: INSTR5
           INSTR6
}
INSTR7

```

Ensamblador

```

SUB  R2, R2, R2
COMP R1, R2
BRNZ No_es_cero
; Sí es cero
INSTR1
INSTR2
JMP Continuar
No_es_cero:
INC  R2
COMP R1, R2
BRNZ Tampoco_es_uno
; Sí es uno
INSTR3
INSTR4
JMP Continuar
Tampoco_es_uno:
INSTR5
INSTR6
Continuar:
INSTR7

```

Bucle HASTA QUE

Pseudocódigo

```
REPETIR {  
    INSTR1  
    INSTR2  
    INSTR3  
    . . .  
} HASTA QUE (R1==R2)  
INSTR4
```

Ensamblador

```
bucle:  
    INSTR1  
    INSTR2  
    INSTR3  
    . . . .  
    COMP R1, R2  
    BRNZ bucle  
; Fuera del bucle  
    INSTR4
```

Bucle MIENTRAS

Pseudocódigo

```
MIENTRAS (R1==R2)
{
    INSTR1
    INSTR2
    INSTR3
    . . .
}
INSTR4
```

Ensamblador

```
Bucle:
    COMP R1, R2
    BRNZ Fuera_bucle
    INSTR1
    INSTR2
    INSTR3
    . . .
    JMP Bucle

Fuera_bucle:
    INSTR4
```

Bucle REPETIR N VECES

Pseudocódigo

```
REPETIR 8 VECES  
(para R4 desde 8 hasta 0)  
{  
    INSTR1  
    INSTR2  
    INSTR3  
    . . .  
}  
INSTR4
```

Ensamblador

```
MOVH R4, 0  
MOVL R4, 8 ; Inicializar contador  
bucle:  
    INSTR1  
    INSTR2  
    INSTR3  
    . . .  
    DEC R4  
    BRNZ bucle  
; Salió del bucle  
INSTR4
```


El array

Concepto

Un *array* es una serie de datos, todos del mismo tamaño, almacenados en posiciones consecutivas de la memoria.

Cadena de caracteres

Es un caso particular de array, en el que:

- Cada elemento es el código ASCII de una letra
- El último elemento vale cero (ASCII NULO)

Ejemplos

Array numérico

```
ORIGEN 0500h
```

```
.DATOS
```

```
nums VALOR 3,6,15,23
```

0500	0003	nums ≡ 0500
0501	0006	
0502	000F	
0503	0017	
...		

Cadena de caracteres

```
ORIGEN 0600h
```

```
.DATOS
```

```
mensaje VALOR 'H', 'o', 'l', 'a', 0
```

0600	0048	mensaje ≡ 0600
0601	006F	
0602	006C	
0603	0061	
0604	0000	
...		

Cadena (otra sintaxis)

```
ORIGEN 0600h
```

```
.DATOS
```

```
mensaje VALOR "Hola", 0
```

Programación con arrays

Método general

- Una etiqueta identifica la dirección en que comienza el array.
- Un registro se hace apuntar a esa dirección.
- Usando el registro entre corchetes, se obtiene el elemento
- Incrementando el registro se pasa al siguiente
- Una condición debe detectar si ya se ha llegado al final

Ejemplo de código

*; El siguiente ejemplo simplemente recorre el array. Cada elemento va
; siendo leído sobre R4*

ORIGEN 0400

INICIO primera

.DATOS

array VALOR 1,20,4,25

.CODIGO

primera:

MOVL R0, BYTEBAJO DIRECCION array

MOVH R0, BYTEBAJO DIRECCION array

MOVL R1, 4 *; Número de elementos*

MOVH R1, 0

bucle:

MOV R4, [R0]

; El dato está en R4

; ahora se procesaría

INC R0

DEC R1

BRNZ bucle

FIN

Cálculo de la longitud de una cadena

; Se declara un texto terminado en NULO y se cuenta la longitud

ORIGEN 0500

INICIO primera

.DATOS

texto VALOR "Esto es una prueba", 0

.CODIGO

primera:

MOVL R0, BYTEBAJO DIRECCION texto

MOVH R0, BYTEBAJO DIRECCION texto

SUB R1, R1, R1 ; Valor NULO para comparar con cada letra

SUB R2, R2, R2 ; Contador de cuántas letras llevamos

bucle:

MOV R4, [R0] ; Leer carácter

COMP R4, R2 ; ¿Es NULO?

BRZ Terminamos ; entonces salir del bucle

INC R2 ; No es nulo, incrementar contador

INC R0 ; apuntar a caracter siguiente

JMP bucle ; y repetir

Terminamos:

; En R2 tenemos cuántas letras tenía el texto

FIN

Problema propuesto

Variante

Modifica el programa anterior para que cuente cuántas letras hay en una cadena dada, pero sin contar los espacios en blanco.

Esquema del apartado 4

- 1 Conceptos
- 2 Ensamblador para la CPU elemental
- 3 Programación de algoritmos en ensamblador
- 4 La pila y las llamadas**
 - Concepto e implementación
 - Acceso de bajo nivel a la pila
 - Llamadas y retornos
- 5 Procedimientos

La pila: concepto

- La pila es una zona de memoria en la que se almacenan datos temporalmente.
- El acceso a una pila se basa en dos operaciones:
 - Almacenar dato. Se apila
 - Recuperar dato. Se obtiene el último introducido

Ejemplo

- ALMACENA 5
- ALMACENA 3
- ALMACENA 9
- RECUPERA
- RECUPERA
- ALMACENA 2
- RECUPERA

La pila en la CPU elemental

Instrucciones

Las instrucciones de acceso a la pila son:

- **PUSH** para almacenar
- **POP** para recuperar

Operandos

Tanto PUSH como POP reciben como operando un registro:

- PUSH Rx almacena en la pila una copia de Rx
- POP Rx recupera el dato que haya en la cima de la pila y lo deja en Rx

Observar que...

- PUSH es una operación de escritura Registro→Memoria
- POP es una operación de lectura Memoria→Registro
- Pero en ningún caso se especifica la dirección
- ¿?

Implementación

El secreto

- PUSH y POP no especifican direcciones porque la CPU mantiene en un registro especial la dirección en la que ha almacenado el último dato de la pila.
- Este registro se denomina “Puntero de pila”
- En la CPU elemental es R7

Cómo funcionan PUSH y POP

- **PUSH Rx**
 - Decrementa R7
 - Escribe en la dirección apuntada por R7 el dato Rx
- **POP Rx**
 - Lee de la dirección apuntada por R7 y deja el resultado en Rx
 - Incrementa R7

Ejemplo

Suponer que R7 comienza con el valor 0522h ¿qué valor tendrán los registros R5, R6 y R7 al final del siguiente código? Dibujar la evolución de R7 y la pila.

Listado

```
XOR  R5, R5, R5
PUSH R5
INC  R5
INC  R5
POP  R6
PUSH R5
ADD  R5, R5, R5
PUSH R5
PUSH R6
ADD  R6, R5, R5
POP  R6
DEC  R6
PUSH R6
POP  R5
```

Observaciones

- Cuando se saca un dato de la pila, no se “borra”. Queda ahí, y será reescrito por el siguiente que entre.
- La pila necesita una zona “libre” en la que evolucionar.
- Si se meten más datos de los que cabrían en la zona reservada, se produce un *stack overflow* (el dato introducido sobrescribirá parte del código del programa, pero no se detecta el error)
- Tras su uso como almacén temporal, la pila debería volver a quedar vacía
- **Debe haber** el mismo número de POP que de PUSH
- Si se sacan más datos de los que se han metido, se produce un *stack underflow*, (el dato obtenido con POP es incorrecto, pero no se detecta el error)

Usando el puntero de pila

Ya que sabemos que R7 es el puntero de pila, *pero a la vez es un registro que puedo manipular*, es posible hacer “cosas raras” con la pila:

Por ejemplo:

- ¿Qué obtenemos si hago MOV R1, [R7]?
- ¿Y si hago MOV [R7], R2?
- ¿Y si hago INC R7 varias veces?
- ¿Y si hago DEC R7 varias veces?
- Además, puedo copiar R7 en otro registro: MOV R5, R7
- ¿Qué obtengo si hago:
MOV R5, R7
INC R5
PUSH R2
INC R5
MOV R2, [R5]

Manipulaciones habituales

- ① **Copiar R7** a otro registro:
Permite que el otro registro pueda ser usado para acceder a los datos de la pila (sin ser afectado por otros PUSH y POP)
- ② **Incrementar R7:**
Permite “eliminar” datos de la pila, sin necesidad de hacer POP
- ③ **Decrementar R7:**
Permite “hacer un hueco” en la pila

Llamadas

Una llamada es un tipo especial de salto que permite volver en el futuro a la instrucción siguiente a la que realizó este salto.

Sintaxis

- CALL etiqueta

Realiza la llamada (salto especial) a “etiqueta”

- RET

Cuando se encuentra esta instrucción se salta a la instrucción que había después del CALL (se retorna)

Observar que:

- Si se hacen varios CALL, la instrucción RET debe volver al último CALL que se hubiera hecho
- Un CALL etiqueta puede aparecer en varios puntos de un programa, y el RET debe volver al CALL correcto en cada caso.

¿Cómo es posible?

RET no lleva ningún parámetro. Entonces, ¿cómo sabe RET a qué dirección debe saltar para “volver”?

Implementación

Llamada

Antes de saltar a la etiqueta en cuestión, CALL almacena el valor de PC *en la pila*

Retorno

La instrucción RET simplemente extrae de la (cima de la) pila un dato y lo carga en PC

Por tanto puede decirse que:

- CALL etiqueta equivale a PUSH PC más JMP etiqueta
- RET es simplemente POP PC

Verificar que este simple mecanismo cumple las propiedades buscadas.

Esquema del apartado 5

- 1 Conceptos
- 2 Ensamblador para la CPU elemental
- 3 Programación de algoritmos en ensamblador
- 4 La pila y las llamadas
- 5 Procedimientos**
 - Procedimientos en ensamblador
 - Paso de parámetros en registro
 - Paso de parámetros en la pila

Definición de procedimiento

Definición

Es un trozo de código al que se “llama” desde otra parte del programa, que puede recibir parámetros y devolver un resultado.

Sintaxis

El procedimiento comienza con una etiqueta, y termina con una instrucción RET

Ejemplo

El siguiente procedimiento intercambia los registros R1 y R2:

`intercambiar_R1_R2:`

`PUSH R1`

`PUSH R2`

`POP R1`

`POP R2`

`RET`

Directivas para procedimientos

Para hacer más visible dónde empieza y acaba un procedimiento existen las directivas:

- `PROCEDIMIENTO etiqueta`
- `FINP`

La instrucción para llamar al procedimiento es la misma, tanto si se usa la directiva `PROCEDIMIENTO` como si no.

¡CUIDADO!

La directiva `FINP` no genera código. En particular *no equivale a `RET`*

Ejemplo de programa completo con procedimiento

```
ORIGEN 200h
INICIO ini
.CODIGO
PROCEDIMIENTO intercambiar_R1_R2
    PUSH R1
    PUSH R2
    POP R1
    POP R2
    RET
FINP
ini:
    XOR R1, R1, R1
    XOR R2, R2, R2
    INC R2
    CALL intercambiar_R1_R2
    MOV R3, R1 ; ¿Qué tenemos en R3?
    CALL intercambiar_R1_R2
    MOV R3, R1 ; ¿Y ahora?
FIN
```

Concepto de parámetro

Parámetro

Es un dato que se suministra al procedimiento. Sirve para que el procedimiento pueda producir diferentes resultados cada vez que se le llame.

Ejemplo

Un procedimiento llamado “Multiplica” debe recibir los datos a multiplicar.

De este modo `Multiplica(3,5)` producirá diferente resultado que `Multiplica(4,2)`

Sin embargo...

La instrucción `CALL` no admite parámetros, sólo etiqueta.

Una solución

Los parámetros con los que el procedimiento debe operar se pasan en *ciertos* registros. El resultado de la operación se devuelve en otro registro.

Por ejemplo

El procedimiento `Multiplica` puede funcionar de modo que:

- Espera los datos a multiplicar en R1 y R2
- Devuelve el resultado en R0

Un programa que quiera usar ese procedimiento debería:

- Preparar en R1 y R2 los datos a multiplicar
- Ejecutar un `CALL Multiplica`
- A la vuelta, tendrá en R0 el resultado.

Ejemplo

```
ORIGEN 200h
INICIO ini
.CODIGO
PROCEDIMIENTO Multiplica    ; Parámetros: R1, R2, resultado:      R0
    XOR    R0, R0, R0
bucle:
    ADD    R0, R0, R1
    DEC    R2
    BRNZ   bucle
    RET
FINP

ini: ; Ejemplo, quiero obtener:      R5 = 12 x 14
    MOVH   R1, 0
    MOVL   R1, 12
    MOVH   R2, 0
    MOVL   R2, 14
    CALL   Multiplica
    MOV    R5, R0
FIN
```

Inconvenientes del paso de parámetros en registros

El paso de parámetros a través de registros es sencillo y fácil de entender, pero tiene inconvenientes:

- Si el procedimiento necesita muchos parámetros, podemos no tener suficientes registros.
- La programación de llamadas “anidadas” se hace compleja:
 - Si el procedimiento llama a su vez a otro procedimiento, ¿en qué registros le pasa los parámetros a éste otro?
 - Si usa los mismos registros, perderíamos su valor previo
 - Si usa otros, se nos pueden acabar

La solución

La solución a estos problemas es usar la pila en lugar de los registros para pasar los parámetros.

Esquema general

El código que llama debe:

- 1 Poner en la pila los datos con que operará el procedimiento
- 2 Ejecutar `CALL procedimiento`
- 3 Al retorno, eliminar de la pila lo introducido en el paso 1

El código llamado debe:

- 1 Hacer una copia del puntero de pila en otro registro (ej, R6)
- 2 Guardar en la pila todos los registros que vaya a modificar
- 3 Recuperar de la pila los parámetros (ej, usando R6 como puntero)
- 4 Efectuar la operación pertinente
- 5 Recuperar de la pila los registros almacenados en el paso 2
- 6 RETornar.

Ejemplo: código que llama a Multiplica

Ejercicio

Escribir un código que llame dos veces a Multiplica:

- ① La primera vez para realizar la operación 12×14 , guardando el resultado en R5
- ② La segunda para 3×7 , guardando el resultado en R6

Importante

La pila se usa sólo para pasar los datos (parámetros) al procedimiento.

Los resultados se retornarán por registro (en R0)

Solución

Acceso a los parámetros desde el procedimiento

Aunque hay muchas posibles formas de hacerlo, seguiremos siempre los mismos pasos para no equivocarnos:

Esqueleto del procedimiento

PROCEDIMIENTO nombre

```
PUSH    R6           ; Guardar R6
MOV     R6, R7       ; Copiar en él R7
PUSH    R1           ; Guardar los restantes registros
PUSH    R2
. . .
```

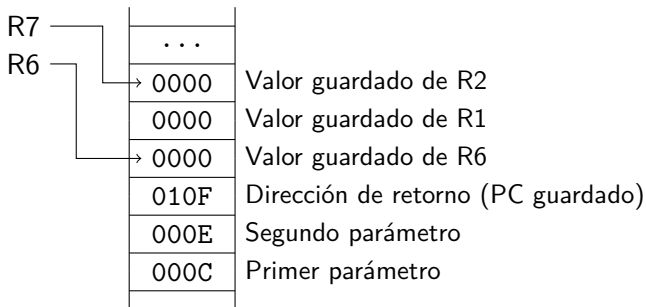
; Código del procedimiento

```
. . .               ; Recuperar registros guardados
POP     R2
POP     R1
POP     R6
RET           ; Retornar
```

FINP

Estado de la pila siguiendo ese esquema

Cuando comience a ejecutarse la línea “Código del procedimiento”, la pila tendrá el siguiente estado (los números son un ejemplo, correspondiente a una llamada del procedimiento para multiplicar 12×14)



Uso de R6 para acceder a los parámetros

Si programamos siempre los procedimientos siguiendo el esquema anterior tenemos que:

- Incrementando R6, podemos acceder a la dirección de retorno
- Incrementando R6 de nuevo, podemos acceder al último parámetro introducido antes de la llamada
- Incrementando R6 de nuevo, podemos acceder al parámetro anterior
- etc.

Ejercicio

Escribir el código del procedimiento Multiplica que accede a los parámetros de la pila y los multiplica, retornando el resultado en R0

Solución

Observaciones

- Es importante asegurarse de que al final de la rutina hay tantos POP como PUSH hubo al principio. ¿Qué pasaría si se olvida uno?
- Es importante que el código que llama elimine de la pila los parámetros que puso (INC R7), para devolver la pila a su estado inicial

Paso por referencia

Los parámetros que se ponen en la pila pueden ser datos , o direcciones de memoria donde hay otros datos.

Este segundo caso se denomina paso por referencia.