

# **Bloque 2**

## **Ensamblador del computador elemental**



Prácticas de  
Introducción a los computadores  
**Curso 2007-2008**

## **SESIÓN 1**

# **Introducción al concepto y uso del ensamblador del computador elemental**

## **Objetivos**

En esta sesión se pretende que el alumno se familiarice con el concepto y el uso del ensamblador del computador elemental. En esta primera toma de contacto aprenderá a hacer archivos en ensamblador, ensamblarlos, corregir errores y cargar los programas resultantes en el simulador.

## **Conocimientos y materiales necesarios**

Para poder realizar esta sesión el alumno debe:

- Haber asimilado los conceptos de codificación y ejecución de instrucciones desarrollados en prácticas anteriores.
- Repasar la estructura de los archivo ejecutables (.exe).
- Repasar la teoría relativa al ensamblador para conocer el uso de las directivas básicas del ensamblador de la CPU elemental y la implementación de estructuras condicionales.

## Desarrollo de la práctica

### 1. El proceso de ensamblado

En prácticas anteriores has aprendido a codificar instrucciones para el computador teórico y a preparar archivos ejecutables (.exe) que pueden contener programas. Como habrás comprobado, el proceso de codificación es tedioso y se cometen errores con facilidad. Para simplificar este proceso se introduce una herramienta, denominada **ensamblador**, que permite realizar la codificación de instrucciones. Además esta herramienta puede interpretar ciertas directivas que le indican cómo debe codificar las instrucciones y facilitan el trabajo con datos.

El ensamblador acepta archivos escritos en un lenguaje denominado lenguaje ensamblador. Habitualmente, estos archivos tendrán extensión .ens y se crearán con un editor de texto como el Edit. El ensamblador transformará estos archivos en archivos .exe que podrán ser cargados en el simulador. Si durante el proceso de ensamblado se encuentran errores, el ensamblador generará un mensaje de error y habrá que solucionarlo volviendo a editar el archivo .ens.

Vamos a utilizar un archivo de ejemplo con errores para que veas cómo funciona el ensamblador. Sigue los siguientes pasos:

- ❑ Copia el archivo 2-1prog1.ens a tu disquete.
- ❑ Abre el interfaz de comandos. Sitúate en el disquete (poniendo a:) y edita el archivo (edit 2-1prog1.ens).
- ❑ Analizando el archivo, contesta a estas preguntas: ¿En qué dirección estará almacenado el dato etiquetado como operando1?<sup>[1]</sup>  
¿En qué dirección estará la primera instrucción?<sup>[2]</sup> ¿Qué valor habrá en la dirección 502h al final de la ejecución del programa?<sup>[3]</sup>
- ❑ Para comprobar que tus respuestas a las preguntas anteriores son correctas, tendrás que ensamblar el archivo y cargarlo en el simulador. Sal del editor y en el interfaz de comandos introduce la siguiente orden:  
ensambla 2-1prog1.ens
- ❑ Como el archivo contiene errores, te deberá aparecer un mensaje de error con una posible explicación. Fíjate en qué línea se encuentra el error. Vuelve a editar el archivo y arréglalo (el programa Edit indica en la esquina inferior derecha el número de línea en la que está el cursor). Tras guardar el archivo corregido y salir del editor, vuelve a intentar el ensamblado.
- ❑ Ahora no debería haber más errores. Si no es así, vuelve al paso anterior hasta que el ensamblador se ejecute sin mensajes de error. En este momento, comprueba que ha generado un archivo llamado 2-1prog1.exe introduciendo la orden dir en el interfaz de comandos.

1

2

3

- ❑ Edita el archivo 2-1prog1.eje para leerlo. ¿Cuál es la codificación de la instrucción ADD R5, R3, R4 (recuerda que era la penúltima instrucción)?<sup>[4]</sup> Sal del editor.
- ❑ Abre el simulador y carga el archivo 2-1prog1.eje. Comprueba con el desensamblador que las direcciones del operando1 y de la primera instrucción coinciden con las que tú habías respondido anteriormente. Fíjate que en esta primera instrucción, el ensamblador ha calculado la constante adecuada correspondiente al byte bajo de la dirección del operando. ¿Cuál es?<sup>[5]</sup>
- ❑ Ahora vas a ejecutar todas las instrucciones del programa. Fíjate en que PC ya está inicializado a la dirección de la primera instrucción. Vete pulsando **[F8]** hasta que se ejecute la última instrucción. Comprueba entonces que el valor que hay en la dirección 502h coincide con el que tú habías previsto anteriormente.

4

5

Como has podido comprobar, el ensamblador simplifica mucho el desarrollo de programas.

## 2. Directivas

Vamos a ver ahora el uso de las directivas principales.

En primer lugar, nos vamos a fijar en la directiva ORIGIN. Al principio de la práctica apuntaste en qué dirección estaban el operando1 y la primera instrucción. Además acabas de ver que el ensamblador sustituye BYTEBAJO DIRECCION dato1 por 0h. Todas estas operaciones se hacen basándose en la directiva ORIGIN. Vamos a modificar ahora la dirección de carga del programa para que sea ABCDh. Sigue estos pasos:

- ❑ Edita el archivo. Modifica la directiva ORIGIN para que la dirección de carga sea ABCDh. Fíjate que, para que el ensamblador sepa que ABCDh es un número y no una etiqueta, tienes que empezar el número con un cero.
- ❑ Guarda el archivo y ensámblalo.
- ❑ Carga el .eje en el simulador. Comprueba con el desensamblador que la dirección del operando1 es la esperada.
- ❑ Fíjate en la primera instrucción. ¿Por qué constante ha cambiado el ensamblador la directiva BYTEBAJO DIRECCION operando1?<sup>[6]</sup>
- ❑ ¿Cuál es el valor inicial de PC?<sup>[7]</sup>

6

7

Al igual que estas dos últimas direcciones, ahora han cambiado todas las direcciones de los datos del programa, lo que ha modificado la las instrucciones que tenían que ver con direcciones de memoria absolutas. Si hubieras tenido que realizar este proceso manualmente hubiera sido mucho más tedioso.

Fíjate que el cambio en la dirección de carga (la que indica la directiva `ORIGEN`) ha afectado, lógicamente, al valor inicial de `PC`. El ensamblador calcula este valor como la dirección a la que apunta la etiqueta que siga a la directiva `INICIO`. Vamos a comprobar cómo funciona esta directiva:

- ☐ Edita el archivo y cambia la etiqueta primera del tal forma que apunte a la segunda instrucción. ¿Con qué valor se debería cargar ahora `PC`?<sup>[8]</sup>
- ☐ Graba el archivo, ensámblalo y cárgalo en el simulador para comprobar si has respondido correctamente a la pregunta anterior.
- ☐ Vuelve a editar el archivo y vuelve a colocar la etiqueta primera apuntando a la primera instrucción de la sección de código.

8

¿Qué número tendrías que poner en `ORIGEN` para que al cargar en el simulador el archivo `.eje PC` valiese `5B37`?<sup>[9]</sup> Comprueba que tu respuesta es correcta poniendo ese valor en el archivo, ensamblándolo y cargándolo en el simulador.

9

Ahora vamos a fijarnos en la directiva `VALOR`. Esta directiva sirve para introducir datos. Tienes tres ejemplos en el archivo `2-1prog1.eje`. La directiva también permite introducir varios valores en la misma línea separados por comas. En ese caso la etiqueta apuntará al primer valor de la lista. Vamos a modificar el programa para que se declaren en una sola línea los dos operandos:

- ☐ Abre el archivo `2-1prog1.ens` en el editor y guárdalo con el nombre `2-1prog2.ens`. No salgas del editor.
- ☐ Sustituye las dos líneas donde se definen los operandos por la siguiente línea:  
`operandos VALOR 3, 0Ch`
- ☐ Modifica ahora las instrucciones de carga de los operandos para que funcionen con el cambio anterior. Fíjate en que ahora sólo tienes la etiqueta `operandos`, con lo que el primer operando estará en la dirección de operandos y el segundo operando estará una posición más allá.
- ☐ Guarda el archivo, sal del editor, ensámblalo, cárgalo en el simulador y ejecútalo para comprobar que funciona.

La directiva `VALOR` permite utilizar constantes en varios formatos. Vamos a ver algunos ejemplos:

- ☐ Edita el archivo `2-1prog2.ens`.
- ☐ Sustituye el valor del primer operando por la constante de carácter `'0'`.
- ☐ Guarda el programa y ensámblalo.

- ☐ Abre el archivo 2-1prog2.eje que se ha generado. ¿Por qué número ha cambiado el ensamblador la constante de carácter '0'?<sup>[10]</sup>
- ☐ Vuelve a editar el archivo fuente 2-1prog2.ens. Sustituye la constante '0' por la cadena .°PQR". Tras este cambio, ¿qué valor tendrá inicialmente PC cuando cargues el archivo ejecutable?<sup>[11]</sup> ¿Cuál será la dirección del resultado?<sup>[12]</sup> ¿Qué dos números se sumarán?<sup>[13]</sup>
- ☐ Guarda el programa y ensámblalo.
- ☐ Carga el archivo ejecutable generado en el simulador. Comprueba, ejecutando el programa, que tus respuestas a las preguntas anteriores son correctas.

10

11

12

13

Vamos a ver ahora el uso de la directiva VECES, que sirve para declarar varios datos con el mismo valor. Sigue los siguientes pasos:

- ☐ Abre en el editor el archivo 2-1prog2.ens y guárdalo con el nombre 2-1prog3.ens.
- ☐ Sustituye la primera línea de la sección datos por esta:  
operandos VALOR 8 VECES 3  
Esta línea hará que se declaren consecutivamente 8 datos con valor 3, al primero de los cuales apuntará la etiqueta operandos.
- ☐ Guarda el archivo, sal del editor y ensámblalo.
- ☐ Edita el archivo 2-1prog3.eje generado. ¿En qué dirección está ahora el resultado?<sup>[14]</sup> ¿Qué dos números se sumarán?<sup>[15]</sup>
- ☐ Carga el archivo 2-1prog3.eje en el simulador y comprueba si tu respuesta a las preguntas anteriores fue acertada.

14

15

### 3. Autoevaluación

#### 3.1. Archivos en el disco

En tu disco de prácticas deberás tener los archivos de programa 2-1prog1.ens, 2-1prog2.ens y 2-1prog3.ens, además de sus correspondientes .eje.

#### 3.2. Ejercicios

- ⇒ Puedes usar el ensamblador como una herramienta para pasar datos decimales a hexadecimal. ¿Se te ocurre cómo? Úsalo para convertir los siguientes datos a hexadecimal: 32, 212, 1000, 2007, 63000, 65535.

- ⇒ Escribe un pequeño programa que compara los registros R4 y R5 y si encuentra que son iguales incrementará R4, pero si son diferentes lo decrementará. Comprueba si el programa funciona correctamente, cargándolo en el simulador y poniendo “a mano” unos valores iniciales iguales en R4 y R5. Comprueba que también funciona correctamente cuando los valores que cargas en R4 y R5 son diferentes.

## SESIÓN 2

# Procesamiento de arrays en ensamblador

## Objetivos

En esta sesión se pretende que el alumno desarrolle un programa en ensamblador que requiera el uso de arrays y bucles.

## Conocimientos y materiales necesarios

Para poder realizar esta sesión el alumno debe:

- Conocer las directivas básicas del lenguaje ensamblador.
- Llevar al laboratorio los apuntes de teoría en los que se encuentran las tablas de condiciones necesarias para implementar estructuras condicionales.
- Conocer la forma de implementar bucles en el ensamblador de la CPU teórica.
- Conocer el concepto de array (unidimensional).

## Desarrollo de la práctica

---

### 1. Planteamiento del problema

En la práctica anterior vimos pequeños programas para practicar los conceptos básicos del ensamblador. En esta práctica vamos a intentar resolver un problema real que requerirá un programa mayor: hallar el máximo de un array de números enteros y almacenar este máximo en una posición de memoria. Un array, como sabrás, es un grupo consecutivo de posiciones de memoria con el mismo tipo y nombre base.

Para facilitar el trabajo, vamos a dividir la tarea en dos partes. En una primera haremos un programa que simplemente recorra el array trayendo los números



uno a uno desde memoria a un registro. En la segunda parte añadiremos a este programa las instrucciones necesarias para ir calculando el máximo y almacenar al final en memoria el resultado.

## 2. Implementación del bucle de lectura de datos

La forma más sencilla de que un programa recorra un array de números es utilizar un bucle. Para implementar este bucle, cuando el número de datos a recorrer (esto es, el tamaño del array) es conocido, se necesita:

- Disponer de un contador de datos que se irá decrementando a medida que se lean los datos del array. Cuando llegue a cero habremos acabado el recorrido de los datos. Para esta labor utilizaremos un registro, en concreto R0.
- Disponer de un puntero a los datos, es decir, de un elemento que señale en todo momento cuál es la dirección de memoria del dato que debe ser procesado. Para este cometido utilizaremos otro registro, R1. Inicialmente se cargará este registro con la dirección del primer número del array y en cada paso del bucle se irá incrementando en una unidad para que apunte al siguiente elemento.
- Un registro al que traer el número desde memoria. Emplearemos R2 para esta labor.

Este es el pseudocódigo que se corresponde al programa que vamos a hacer:

```
1 Inicializar contador
2 Inicializar puntero
3 Repetir
4   Leer el número
5   Incrementar el puntero
6   Decrementar el contador
7 Hasta fin del array (contador = 0)
```

Ahora debes hacer el programa que implemente este algoritmo siguiendo estas especificaciones:

- ☐ El archivo fuente se debe llamar `2-2bucle.ens`.
- ☐ El programa se debe cargar en la dirección `450h`.
- ☐ La lista de números a tratar debe ser esta: 7, 45, 78, 125, 678, 745, 345, 8, 764, 567, 324 y 23. Utiliza la etiqueta `array` para apuntar al principio del array.

Cuando hayas hecho el programa, ensámblalo y cárgalo en el simulador. Ejecútalo comprobando que todos los números pasan por el registro R2. ¿Cuánto vale R0 justo en el momento en el que el valor 8 aparece en R2?<sup>[1]</sup>

### 3. Implementación de la búsqueda del máximo

La implementación de la búsqueda del máximo del array necesitará:

- Un registro que almacene el máximo de forma temporal. Se utilizará R3.
- Una posición de memoria en la que almacenar el máximo obtenido tras recorrer todos los números. Se etiquetará como `máximo`.

Para completar el programa, sigue estos pasos:

- ❑ Edita el fichero `2-2bucle.ens`. Guárdalo con el nombre `2-2max.ens`.
- ❑ En la sección de datos, añade la variable `maximo` tras el array. No importa qué valor inicial le des.
- ❑ Al principio del código debes inicializar el máximo temporal (que, como se dijo más arriba, se guardará en R3). Esta inicialización dependerá de si los valores son números naturales o enteros. Considera que los números son naturales. ¿Cuál sería el valor de inicialización si fuesen enteros?<sup>[2]</sup>
- ❑ Dentro del bucle debe compararse el valor de R3 con el del dato del array que se está procesando (que estará en R2). Introduce el código necesario justo después de traer el dato a R2. Si de esta comparación se deduce que R3 es mayor o igual que R2, debe continuar la ejecución del bucle (usa una instrucción de salto condicional). Si resulta que el valor de R2 es mayor que el de R3, éste último será el nuevo máximo y, por lo tanto, debe actualizarse el valor de R3 antes de continuar con el bucle.
- ❑ Terminado el bucle, añade las instrucciones necesarias para almacenar el máximo temporal (R3) en la variable `maximo`. Utiliza R4 como registro para el direccionamiento indirecto que tendrás que hacer.
- ❑ Guarda el fichero, ensámblalo y carga el ejecutable en el simulador.
- ❑ Ejecuta el programa y comprueba que al final aparece en la posición de memoria correspondiente el valor del máximo. ¿En qué dirección está la variable `maximo`?<sup>[3]</sup> ¿Cuál es el valor que tiene al final?<sup>[4]</sup>

2

3

4

## 4. Autoevaluación

### 4.1. Archivos en el disco

En tu disquete deberías tener los archivos `2-2bucle.ens` y `2-2max.ens`.

### 4.2. Ejercicios

- ⇒ Si la lista estuviera formada por números enteros en lugar de sólo números naturales, tendrías que hacer varios cambios. Intenta hacerlos, copiando el archivo `2-2max.ens` a `2-2max2.ens`, y comprueba que el nuevo programa funciona introduciendo algún número negativo en la lista.
- ⇒ Codifica la instrucción de salto condicional del programa `2-2max.ens` y comprueba con el depurador que lo has hecho correctamente.
- ⇒ Crea un nuevo programa a partir del `2-2max.ens` en el que se calcule el mínimo a la vez que el máximo.

## SESIÓN 3

# La pila: Uso y funcionamiento

## Objetivos

En esta sesión el alumno conocerá la utilidad de la estructura de datos “pila” como almacén temporal de información, y las instrucciones específicas que existen para acceder a ella (PUSH y POP).

También se estudiará cómo la CPU utiliza un registro especial para mantener esta pila, las consecuencias que ello tiene, qué tipos de problemas pueden aparecer usando la pila, y algunos “trucos” para acceder a la pila mediante instrucciones diferentes de PUSH y POP.

## Conocimientos y materiales necesarios

Para poder realizar esta sesión el alumno debe:

- Saber escribir programas en el lenguaje ensamblador de la CPU elemental y utilizar el programa ensamblador para obtener archivos .exe.
- Repasar en los apuntes de teoría el concepto de “pila” como almacén temporal en el cual el último dato en entrar es el primero en salir.

---

## Desarrollo de la práctica

---

### 1. Directiva `.PILA` y ejemplos básicos de uso de la pila

El lenguaje ensamblador tiene una directiva llamada `.PILA` que no habíamos usado aún, y que sirve para definir el tamaño de la pila. Esta directiva debes usarla al principio del programa, como se muestra en el ejemplo `2-3pila1.ens`, que vas a ver ahora.

- ❑ Copia el archivo `2-3pila1.ens` en tu disquete.
- ❑ Ejecuta el “Interfaz de comandos” y escribe `A:` (para situarte en el disquete) seguido de `EDIT 2-1pila1.ens` para abrir el archivo.

Puedes ver ahora que la segunda línea de este archivo pone:

```
2 .PILA 4
```

lo cual indica que durante la ejecución de este programa, la pila tendrá un tamaño máximo de 4 posiciones. Es nuestra responsabilidad como programadores el determinar un tamaño suficiente y asegurarnos de que nuestro programa nunca intenta guardar en la pila más datos de los que hemos reservado con esta directiva (al final de la práctica veremos qué ocurre si no tenemos cuidado con esto).

El resto del programa demuestra uno de los usos más habituales de la pila: almacén temporal de información, junto con un ejemplo de utilización menos corriente: intercambio de registros. Veamos ambos con detenimiento:

#### 1.1. Guardando temporalmente datos

Observa el archivo `2-3pila1.ens` que tienes abierto en el editor. Las primeras instrucciones del programa sirven simplemente para inicializar los registros `R0` y `R1` con los datos `1234h` y `ABCDh`, respectivamente.

Justo a continuación, usando la instrucción `PUSH` guardamos en la pila estos registros, para borrarlos seguidamente con la instrucción `XOR` (la instrucción `XOR`, al ser aplicada sobre dos datos idénticos siempre produce como resultado cero, por lo que suele usarse para borrar registros).

Ahora `R0` y `R1` tendrán el valor cero, pero en la pila tenemos guardada una copia de sus valores originales. Las dos siguientes instrucciones (que han sido sustituidas por interrogantes) tendrían como finalidad devolver a estos registros sus valores originales utilizando la instrucción `POP`. Ahora tienes que hacer lo siguiente:

- ☐ Cambia los interrogantes por las instrucciones apropiadas.
- ☐ Sal del editor y ensambla el listado, mediante el comando `ensambla` que ya conoces de las sesiones previas. Asegúrate de que no hay errores y obtienes un fichero llamado `2-3pila1.eje`.
- ☐ Carga el `.eje` en el simulador. Pulsa cuatro veces la tecla `[F8]` para ejecutar las cuatro primeras instrucciones y comprueba cómo los registros `R0` y `R1` adquieren los valores `1234h` y `ABCDh` respectivamente.
- ☐ Pulsa `[F8]` dos veces más para ejecutar las instrucciones `PUSH`. Los valores de `R0` y `R1` no han cambiado, pero ahora hay una copia de ellos en la pila.
- ☐ Pulsa `[F8]` dos veces más. Esto ejecutará las instrucciones `XOR`, borrando los registros `R0` y `R1`. Verifica que ocurre así.
- ☐ Ahora, pulsa `[F8]` dos veces más para ejecutar las instrucciones que tú has añadido. ¿Qué valor aparece en `R0`?<sup>[1]</sup> ¿Y en `R1`?<sup>[2]</sup> ¿Son los mismos valores que tenían al principio del programa?<sup>[3]</sup> Si no fuera así, es que no has puesto las instrucciones `POP` correctas. Piensa otra vez sobre ello y repite desde el primer punto.

1

2

3

Si has superado correctamente esta fase, te habrás dado cuenta de que los registros deben sacarse de la pila en orden inverso a como fueron introducidos, si el objetivo es recuperar sus valores iniciales. Pero podemos desear otros objetivos, como es el caso del ejemplo siguiente.

## 1.2. Un uso poco frecuente: intercambiar datos

Las cuatro siguientes instrucciones que va a ejecutar el programa son:

```

23  PUSH R0
24  PUSH R1
25  POP  R0
26  POP  R1

```

Antes de ejecutarlas trata de predecir ¿Qué valor tendrá al final `R0`?<sup>[4]</sup> ¿Y `R1`?<sup>[5]</sup> Comprueba que es así pulsando cuatro veces `[F8]`

4

5

Ahora una pregunta con trampa. Fíjate en cuántas veces hemos ejecutado la instrucción `PUSH` en el programa. ¿Cuál sería el valor más pequeño que podríamos usar en la directiva `.PILA`?<sup>[6]</sup> La respuesta no es 4, piensa sobre ello. Si no ves por qué, pregunta al profesor.

6

## 2. Cómo funciona la pila

### 2.1. El papel de R7

Como ya te han explicado en teoría, la pila no es más que una parte de la memoria, como pudiera ser la sección de datos o la sección de código del programa.

En particular, el registro R7 es el que señala a la dirección de memoria donde está la pila. En la dirección señalada por R7 es donde se encuentra almacenado el último dato que se ha introducido en la pila. Cada vez que se introduce un dato nuevo (PUSH), R7 se decrementa y el dato se almacena en la nueva posición a la que apunta. Cada vez que se saca un dato de la pila (POP) lo que se obtiene es lo que había en la posición apuntada por R7, tras lo cual R7 se incrementa.

Por tanto, el valor inicial de R7 nos está determinando dónde se irán guardando los datos de la pila. Observa que la pila va creciendo hacia las direcciones bajas a medida que se introducen más datos en ella. El valor inicial de R7 determina dónde estará la “base” de la pila, mientras que el valor de R7 en cualquier otro instante determina dónde está la “cima” de la pila (cuando la cima coincide con la base, es que la pila está vacía, lo que ocurre al cargar el programa).

En los archivos .eje, la tercera línea indica cuál debe ser el valor inicial de R7. Veámoslo:

- ☐ En el interfaz de comandos, escribe `EDIT 2-3pila1.eje` para ver los contenidos del archivo .eje
- ☐ Observa la tercera línea ¿qué valor aparece?<sup>[7]</sup> Este debería ser el valor inicial de R7. Comprobémoslo:
- ☐ Sal del editor y abre (de nuevo) el archivo .eje con el simulador. Observa qué valor toma R7 inicialmente. ¿Coincide con el que has visto en el .eje?<sup>[8]</sup>

7

8

### 2.2. La ejecución de PUSH

Ahora veremos qué ocurre cuando se ejecuta una instrucción PUSH. Para ello:

- ☐ Pulsa **[F8]** cuatro veces para inicializar R0 y R1. La próxima pulsación de **[F8]** ejecutaría la instrucción PUSH R0. Antes de pulsarla responde a lo siguiente: ¿En qué dirección de memoria se guardará el dato?<sup>[9]</sup> ¿Qué valor aparecerá en esa dirección de memoria?<sup>[10]</sup> ¿Qué valor aparecerá en R7?<sup>[11]</sup> ¿Resultará modificado R0?<sup>[12]</sup>
- ☐ Pulsa **[F8]** y comprueba tus respuestas anteriores. Para verificar que se ha escrito la dirección de memoria que habías previsto deberás usar el editor hexadecimal de la memoria e ir a la dirección que has respondido antes, comprobando si hay allí el dato que has dicho.

9

10

11

12

- ☐ Si pulsaras de nuevo **F8** ¿qué dirección de memoria resultaría modificada?<sup>[13]</sup> ¿Qué valor se escribirá en esa dirección?<sup>[14]</sup> ¿Qué valor tomará R7?<sup>[15]</sup>
- ☐ Pulsa **F8** y verifica tus respuestas.

13
15
14

### 2.3. La ejecución de POP

Pulsa **F8** dos veces más (para ejecutar las instrucciones XOR que borran los registros). Ahora estamos preparados para analizar lo que ocurre al ejecutar instrucciones POP. Para ello ve respondiendo a lo siguiente:

- ☐ ¿Qué valor tiene R7?<sup>[16]</sup> Abre el editor hexadecimal de la memoria y ve a la dirección que acabas de responder. ¿Qué valor hay en ella?<sup>[17]</sup> Ese será el valor que se traerá cuando se ejecute POP R1, y se copiará en el registro R1. Pulsa **F8** una vez.
- ☐ ¿Qué valor ha aparecido en R1?<sup>[18]</sup> Tiene que ser el mismo valor que habías visto en la memoria. ¿Qué valor tiene ahora R7?<sup>[19]</sup> Tiene que ser uno más que el valor que tenía en la pregunta anterior.
- ☐ Mediante el editor hexadecimal ve a la dirección de memoria 310h. Puedes ver que aún sigue allí el valor ABCDh. La instrucción POP no elimina de la memoria el dato, pero no es necesario hacerlo porque la próxima vez que haga PUSH, ese dato será sustituido por uno nuevo.
- ☐ Pulsa **F8** otra vez y comprueba cómo R7 se incrementa de nuevo. En este momento, R7 debe tener el mismo valor con el que comenzó la ejecución, lo cual significa que la pila está de nuevo vacía.
- ☐ Observa ahora cómo varía R7 al ejecutar el resto del programa (cuatro pulsaciones más de **F8**). ¿Cuál es el valor más pequeño que llega a alcanzar R7?<sup>[20]</sup> ¿Cuál es la diferencia entre este valor y el valor inicial?<sup>[21]</sup> Este será el tamaño mínimo que deberíamos usar en la directiva .PILA

16
----

17
----

18
----

19
----

20
----

21
----

### 2.4. Trucos “sucios” con la pila

La forma “educada” de acceder a la pila es usar las instrucciones PUSH y POP. Sin embargo, sabemos que estas instrucciones usan de forma implícita el registro R7 para mantener un indicador de la cima de la pila. Haciendo uso de este conocimiento ¿qué pasaría si modificamos R7 o leemos su valor? Podríamos lograr interesantes resultados que serán útiles en la sesión siguiente.



### 2.4.1. Modificar R7

La instrucción POP, además de recuperar un dato de la pila, incrementa el registro R7. ¿Qué pasaría si ejecutáramos la instrucción INC R7? El resultado sería el mismo que el de POP, pero sin recuperar el dato de la pila. De hecho, es como si hubiéramos eliminado un dato de la pila. Veámoslo con un ejemplo:

- ❑ Abre de nuevo el fichero 2-3pila1.ens con el editor y ve a la línea 18 (se trata de la línea POP R1, que has escrito como parte de esta sesión).
- ❑ Cambia esa línea por INC R7.
- ❑ Ensambla el programa y cárgalo en el simulador .
- ❑ Pulsa **[F8]** para ejecutar las instrucciones que cargan R0 y R1, las que meten una copia en la pila (PUSH) y las que borran los registros (XOR). Cuando IR contenga la instrucción XOR R1,R1,R1, no pulses más **[F8]**.
- ❑ Ahora irá la ejecución de INC R7 que, como hemos dicho antes, tendrá resultados similares al POP. ¿Cuánto valdrá R7 cuando pulses **[F8]**?<sup>[22]</sup> Pulsa **[F8]** y compruébalo. Observa que es lo mismo que si hubieras hecho POP R1, con la diferencia de que R1 no ha recuperado su valor inicial.
- ❑ Ahora vendrá la instrucción POP R0 ¿Qué valor aparecerá en R0?<sup>[23]</sup> ¿De qué dirección de memoria se obtiene ese valor?<sup>[24]</sup>

22

23

24

Con el experimento anterior se demuestra que INC R7 equivale a sacar datos de la pila, pero sin guardar estos datos en otros registros, sino simplemente descartándolos.

### 2.4.2. Acceder a la pila a través de R7

Podemos usar R7 entre corchetes para acceder al dato que hay en la cima de la pila, por ejemplo poniendo MOV R0, [R7] ¿Equivale esto a POP R0?<sup>[25]</sup> ¿Por qué?

25

Pero más interesante aún que el uso de R7 entre corchetes, es la posibilidad de copiar lo que hay en R7 a otro registro, y después usar ese otro registro entre corchetes. Esto nos permite leer cualquiera de los datos que hay en la pila, sin necesidad de sacarlos de ella ni alterar el valor de R7. Veámoslo con un ejemplo.

- ❑ Haz una copia del fichero 2-3pila1.ens y renombra la copia para que se llame 2-3pila2.ens.
- ❑ Abre 2-3pila2.ens con el editor y borra todas las líneas que hay entre XOR R1,R1,R1 y FIN.
- ❑ Completa el siguiente dibujo que representa el estado que tendría la pila al ejecutar el programa hasta este punto (escribe direcciones y contenidos):

Dirección	Contenido	
		← Cima de la pila
		← Base de la pila

- ❑ Debajo de `XOR R1,R1,R1` escribe la instrucción `MOV R6,R7` la cual sirve para hacer una copia del registro R7 en el registro R6. Gracias a esto, R6 apuntará a la cima de la pila. Marca con una flecha en el dibujo la posición a que apunta R6 dentro de la pila.
- ❑ Escribe la instrucción que serviría para recuperar el dato que hay en la cima de la pila y guardarlo en el registro que tenía originalmente ese valor **sin usar** POP. Debes usar R6 para acceder a la dirección donde está el dato ¿Qué instrucción es la que debes poner?<sup>[26]</sup>
- ❑ Para recuperar el otro registro, debes acceder al interior de la pila. Esto puedes hacerlo gracias a la copia que tienes en R6, ya que el dato que buscas está justo una posición después. Añade las instrucciones necesarias para hacer que R6 apunte a ese dato, y para transferir ese dato desde la memoria al registro que originalmente tenía ese valor.
- ❑ Ensambla el programa que has escrito y comprueba su funcionamiento mediante el simulador. Al finalizar el programa los registros R0 y R1 deben tener los valores 1234h y ABCDh respectivamente.

26

### 3. Detalles avanzados

#### 3.1. ¿Quién decide dónde poner la pila?

Si hicieras los programas “a mano”, tú mismo podrías decidir dónde colocar la pila, y deberías comenzar tu programa inicializando R7 con la dirección de memoria donde hayas decidido poner la base de la pila. Deberías elegir una dirección base de tal forma que las direcciones ocupadas por la pila no se solaparan con las direcciones ocupadas por otras partes del programa (código o datos).

Cuando programas en ensamblador, es el propio programa ensamblador el que decide dónde poner la pila, y basa su decisión en la directiva `.PILA` y en el tamaño de tu código. En concreto, el ensamblador sitúa la pila justo después de tu código, dándole un tamaño igual al que tú hayas especificado en `.PILA`. Veamos qué significa esto.

- ❑ Examina de nuevo el listado `2-3pila1.ens` abriéndolo con EDIT, y carga en el simulador de la CPU el ejecutable `2-3pila1.eje`.

- ❑ ¿En qué dirección de memoria comienza tu programa?<sup>[27]</sup> Recuerda que se trata del valor que hemos puesto en la directiva ORIGIN.
- ❑ Mediante el desensamblador examina lo que hay a partir de esa dirección de memoria. ¿Cuál es la última dirección de memoria ocupada por el código?<sup>[28]</sup>
- ❑ La directiva .PILA especificaba el valor 4. Si sumas 4 a la dirección que has respondido en el apartado anterior ¿qué sale?<sup>[29]</sup> ¿Este es el valor inicial de R7!

27

28

29

Con lo anterior se pretende que veas que la pila del programa anterior es una pequeña zona de 4 posiciones de memoria, que comienza justamente en la dirección donde termina el código.

### 3.2. El problema del desbordamiento de pila

Imagina que el programa realiza cinco PUSH seguidos. En principio esto sería un error de programación, ya que la pila no tiene espacio más que para cuatro datos. No obstante, ¿qué crees que ocurriría? Recuerda dónde está ubicada la pila en la memoria.

Vamos a hacer un experimento que te mostrará lo que ocurre. Escribe el siguiente programa en el editor y guárdalo con el nombre 2-3pila3.ens

```

1  ORIGIN 0300h
2  .PILA 2
3  .CODIGO
4  MOVH R0, 0C0h
5  MOVL R0, 0FFh
6  PUSH R0
7  PUSH R0
8  PUSH R0
9  PUSH R0
10 MOV R1, R0
11 FIN

```

Es evidente que el programa está mal hecho, puesto que sólo reserva espacio para dos datos en la pila (.PILA 2) y sin embargo introduce cuatro datos (cuatro PUSH). Ensámblalo para obtener 2-3pila3.eje y carga este en el simulador de la CPU. Veamos qué ocurre al ejecutarlo:

- ❑ Pulsa **[F8]** dos veces, para inicializar R0 con el valor C0FFh .
- ❑ Pulsa **[F8]** dos veces más, para ejecutar los dos primeros PUSH. Hasta aquí todo bien, pero ahora la pila ya está llena. ¿Qué ocurrirá si pulsamos **[F8]** otra vez? Veámoslo.
- ❑ Pulsa **[F8]**, el siguiente PUSH se ejecuta, aparentemente sin problemas. Sin embargo ha habido un desbordamiento de pila. El dato que acabamos de introducir en la pila, realmente se ha guardado fuera de ella. ¿En qué dirección se ha guardado?<sup>[30]</sup>

30

- ❑ Pulsa **F8** otra vez. De nuevo tenemos un PUSH que se ejecuta aparentemente sin problemas. Los problemas aparecerán ahora:
- ❑ Si pulsáramos de nuevo **F8** esperaríamos la instrucción `MOV R1, R0`, de acuerdo con el listado anterior. Sin embargo, pulsa **F8** ¿qué instrucción aparece en IR?<sup>[31]</sup> ¿De dónde ha salido esa instrucción? Observa su código máquina ¿te suena ese número? Coincide con el valor de R0. Lo que ha ocurrido es que los últimos PUSH han invadido la zona de código. El número `C0FFh` que intentábamos meter en la pila, se ha metido en realidad en la zona de código sustituyendo las últimas instrucciones del programa.
- ❑ Abre el desensamblador y ve a la dirección `300h`. Verás allí el programa, pero en la zona final aparece la instrucción `JMP -1`, que es la decodificación del dato `C0FFh`

31

Conclusión: el desbordamiento de la pila no se detecta cuando ocurre, pero puede tener consecuencias imprevisibles (en nuestro caso, el programa se quedaría “colgado”, ya que `JMP -1` causará un salto a la misma dirección de memoria una y otra vez).

Nuestra CPU sencilla no tiene mecanismos para detectar cuándo va a producirse un desbordamiento de pila, y es responsabilidad del programador asegurarse de que esto no ocurra nunca. Otras CPUs tienen la capacidad de detectar este problema y abandonar la ejecución del programa que lo haya causado.

## 4. Autoevaluación

### 4.1. Archivos en el disco

En tu disco de prácticas deberás tener los archivos de programa `2-3pila1.ens`, `2-3pila2.ens` y `2-3pila3.ens`, si bien estos ficheros no serán necesarios para las sesiones futuras.

### 4.2. Ejercicios

- ⇒ Escribe un programa que comience haciendo PUSH de los registros R0 a R6 (ambos inclusive). Asegúrate de que en la directiva `.PILA` reservas tamaño suficiente para esto.

Si ahora quisieras eliminar todo lo que has metido en la pila (descartándolo), ¿cuántas instrucciones necesitarías?<sup>[32]</sup> Puede hacerse con tres instrucciones, usando un registro de forma auxiliar. Pien-  
sa cómo lo harías, escribe el programa y comprueba su funcionamiento.

32

- ⇒ Además del desbordamiento de pila, existe el llamado “*stack underflow*” (desbordamiento por debajo), que ocurre cuando realizas más instrucciones POP que el número de datos almacenados en la pila.

Escribe un programa que cause *stack underflow* (por ejemplo, metiendo dos datos en la pila, y tratando de sacar tres). ¿Qué crees que pasará cuando se ejecute el POP extra? Comprueba tu intuición con el simulador.

Aparentemente el *underflow* es menos peligroso que el *overflow*, ya que (aparentemente) no causa que se escriba en zonas de la memoria que no pertenecen a nuestro programa. ¿Estás seguro de esto?<sup>1</sup>

- ⇒ Escribe un programa que comience haciendo PUSH de todos los registros del procesador (puede servirte el del primer ejercicio). A continuación añade las instrucciones necesarias para traer al registro R2 el dato que se almacenó en el quinto PUSH (PUSH R4). Deberás usar el truco de registro intermedio (R6) y te ayudará hacer un dibujo de la pila para averiguar a qué distancia de la cima se halla el dato que quieres leer.

Ensambla el programa y cárgalo en el simulador. Modifica a mano el registro R4 y dale el valor AAAAh. Ejecuta tu programa y verifica que efectivamente R2 recibe el valor AAAAh (leído de la pila a través de R6)

---

<sup>1</sup>Pregunta difícil

## SESIÓN 4

# Procedimientos

## Objetivos

En esta sesión el alumno practicará con el concepto de procedimiento, estudiando en detalle el mecanismo de llamada y retorno y el papel que desempeña la pila en este mecanismo.

Posteriormente, se verá cómo la pila puede usarse también como lugar de intercambio de información (paso de parámetros) y se practicará este concepto con un sencillo ejemplo.

## Conocimientos y materiales necesarios

Para poder realizar esta sesión el alumno debe:

- Saber escribir programas en el lenguaje ensamblador de la CPU sencilla y utilizar el programa ensamblador para obtener archivos .exe.
- Comprender claramente el funcionamiento de la pila y el papel que juega el registro R7 en este funcionamiento, así como ser capaz de acceder a los datos que hay en la pila sin necesidad de usar la instrucción POP (esto ha sido practicado en la sesión anterior).
- Repasar en los apuntes de teoría el concepto de procedimiento y de parámetro.

## Desarrollo de la práctica

### 1. Directiva PROCEDIMIENTO y ejemplo básico

Un procedimiento no es más que un trozo de código que sólo es ejecutado cuando se le “llama” desde otra parte del programa. El procedimiento debe contener una instrucción que cause el retorno al punto desde el cual fue llamado, una vez que ha cumplido con su cometido.

Por tanto, las dos instrucciones básicas relacionadas con los procedimientos son CALL (para llamarlo) y RET (para retornar). Desde el punto de vista del código máquina, no hay más: un procedimiento no se diferencia de otro código, salvo porque en algún momento ha de ejecutar una instrucción RET. Pero desde el punto de vista del ensamblador resulta conveniente tener alguna forma de distinguir fácilmente lo que es un procedimiento de lo que no lo es, pues esto mejora la legibilidad del programa y ayuda a evitar errores.

Para esto existen las directivas PROCEDIMIENTO y FINP que se colocan al principio y al final de un procedimiento. No obstante, es importante que comprendas que estas directivas sólo están para ayudarte a leer los programas, y que no se convierten en código máquina como veremos enseguida con un ejemplo. En particular, debes tener siempre presente que la directiva FINP **no causa retorno**. Debes usar siempre una instrucción RET para esto.

### 1.1. Ejemplo minimo

El procedimiento más pequeño posible, por tanto, será el que contenga tan solo la instrucción RET. Vamos a escribir uno así y llamarlo desde otros sitios, para estudiar cómo funciona el mecanismo de llamada y retorno.

En ensamblador, los procedimientos pueden ponerse en cualquier lugar de la sección de código. Al principio o al final. Lo único que tienes que tener en cuenta es que, a menos que utilices la directiva INICIO, el programa comenzará a ejecutarse por la primera instrucción que escribas *incluso si esta está dentro de un procedimiento*. Por eso no debes olvidar usar INICIO si decides poner los procedimientos delante del “programa principal”.

- ❑ Copia el ejemplo siguiente en el editor y guárdalo con el nombre 2-4proc1.ens:

```

1  ORIGEN 500h
2  .PILA 4
3  .CODIGO
4      INC R0
5      CALL minimo
6      INC R1
7      CALL minimo
8      NOP ; Este NOP marca el fin del ‘‘programa principal’’
9
10  PROCEDIMIENTO minimo
11      RET
12  FINP ; Fin del procedimiento
13  FIN ; Fin del codigo

```

- ❑ Sobre el sencillo listado anterior cabe hacer algunos comentarios:
- El programa necesita una pila desde el momento que usa una instrucción CALL (pues la dirección de retorno se almacena en la pila).
  - Al no utilizar directiva INICIO, la primera instrucción en ejecutarse será la primera instrucción del código, es decir, INC R0.

- El procedimiento lo hemos puesto después del “programa principal”. Es un procedimiento absurdo, puesto que no hace nada salvo retornar, pero servirá para ver cómo funciona RET.
  - ❑ Teniendo en cuenta que el programa se cargará en 500h y que, una vez convertido en código máquina, las directivas habrán desaparecido ¿en qué dirección se cargará la instrucción RET?<sup>[1]</sup>
  - ❑ Ensambla el listado anterior para obtener 2-4proc1.eje y cárgalo en el simulador.
  - ❑ Usando el desensamblador, observa lo que hay a partir de la dirección 500h. Verifica que RET está en la posición que habías respondido antes. Observa que CALL mínimo ha sido sustituido por CALL seguido de un número.
  - ❑ Anota el valor con que se inicializa R7.<sup>[2]</sup> (Nota, ¿podrías haber deducido tú mismo este valor a partir de la respuesta anterior y del valor de .PILA?)
  - ❑ Pulsa [F8]. La instrucción INC R0 se ejecuta, y PC se incrementa. Todo está listo para ejecutar el primer CALL ¿Qué valor tomará PC cuando CALL se ejecute?<sup>[3]</sup>
  - ❑ Pulsa [F8] y comprueba que efectivamente PC toma el valor que habías predicho. Por tanto CALL se comporta como un salto que te lleva a la dirección de memoria donde está la primera instrucción del procedimiento mínimo. Pero eso no es todo, observa que R7 ha cambiado de valor, ¿qué valor tiene ahora?<sup>[4]</sup>. Esto indica que algo ha sido almacenado en la pila. Vamos a ver qué es.
  - ❑ Abre el editor hexadecimal de la memoria y ve a la dirección indicada en R7, ¿qué valor encuentras allí?<sup>[5]</sup> Este es el valor que tenía PC cuando la instrucción CALL se ejecutó, o sea, la *dirección de retorno*.
  - ❑ Todo está listo para ejecutar la primera instrucción del procedimiento, que en este caso es un simple RET. La instrucción RET se limita a sacar un valor de la cima de la pila y guardarlo en PC. ¿Qué valor será en este caso?<sup>[6]</sup> Pulsa [F8] y comprueba que PC toma el valor predicho. ¿Qué otro registro ha cambiado de valor?<sup>[7]</sup> ¿Por qué?
  - ❑ Teniendo en cuenta el valor que ahora tiene PC, ¿cuál será la próxima instrucción que se ejecutará?<sup>[8]</sup> Por tanto se ha *retornado* al punto en que se llamó, y la ejecución prosigue.
  - ❑ Pulsa [F8] para ejecutar esta instrucción.
  - ❑ La próxima instrucción es de nuevo un CALL. Antes de pulsar [F8] trata de responder:
    - ¿Qué valor tomará R7?<sup>[9]</sup>
    - ¿Qué valor tomará PC?<sup>[10]</sup>
    - ¿Qué valor quedará almacenado en la cima de la pila?<sup>[11]</sup>
- Pulsa [F8] y verifica tus respuestas

1

2

3

4

5

6

7

8

9

10

11



- ❑ Finalmente, pulsando `F8` una vez más observa cómo regresas a la instrucción siguiente a la llamada.

Todo lo anterior pretende clarificar varios hechos importantes: el destino de CALL se calcula como en un salto; además de saltar CALL guarda en la pila el valor de PC, es decir, la dirección de retorno; RET se limita a sacar de la pila un número y guardarlo en PC.

Como consecuencia, se entiende que si se hacen modificaciones en la pila (o en el registro R7) dentro del procedimiento, RET no funcionará correctamente. Verificaremos esto en uno de los ejercicios de autoevaluación, al final de la práctica.

## 1.2. Una variación

- ❑ Copia el programa `2-4proc1.ens` y llama a la copia `2-4proc2.ens`
- ❑ Edita `2-4proc2.ens` con EDIT y mueve el procedimiento al principio del código (justo después de la directiva `.CODIGO`)
- ❑ Piensa si necesitas hacer más modificaciones en el programa para que funcione.
- ❑ Ensámblalo y comprueba que se ejecuta correctamente. Observa ahora cómo las instrucciones CALL deben realizar un salto *hacia atrás*, lo que se traduce en que el desplazamiento es negativo. No obstante, el mecanismo de llamada y retorno es el mismo que en el caso anterior.

## 2. Paso de parámetros

### 2.1. Introducción (repaso de teoría)

Vamos a construir ahora un procedimiento más útil. La verdadera potencia de los procedimientos se demuestra mediante el paso de parámetros, gracias al cual el resultado que devuelve el procedimiento puede ser diferente cada vez que se le llama, si en la llamada se le pasa un parámetro diferente.

Por ejemplo, imagina un procedimiento que sirva para calcular el triple de una cantidad. Si le “pasamos” la cantidad 3, debe responder 9, si le “pasamos” la cantidad 5, responderá 15, etc.

El algoritmo en sí es muy simple, basta que el procedimiento calcule `cantidad + cantidad + cantidad`. La complicación está en realidad en ¿cómo recibe el procedimiento esta cantidad? ¿Cómo le responde al programa que llamó el resultado que ha obtenido? El ensamblador no proporciona respuestas a estas preguntas. A diferencia de otros lenguajes como el C que permiten especificar los parámetros cuando se hace una llamada, y asignar los resultados que la función devuelve, el

ensamblador carece de estos mecanismos. Por tanto todo se deja a la creatividad del programador.

Los programadores han encontrado una forma de pasar parámetros y recibir resultados cuando programan en ensamblador. Se trata de usar la pila para pasar los parámetros, y un registro de la CPU para devolver el resultado. Así, antes de realizar CALL, meten en la pila las cantidades con las que debe operar el procedimiento, y cuando el procedimiento haya finalizado, obtienen en un registro del procesador la respuesta deseada.

Usaremos este mismo sistema, suponiendo que el registro que tendrá la respuesta será R0. Así, si tenemos un procedimiento llamado `triple` y queremos usarlo para calcular el triple de R2, guardando el resultado en R2, la llamada será:

1	PUSH	R2	;	<i>‘pasar’ el parámetro</i>
2	CALL	triple	;	<i>llamar al procedimiento</i>
3	MOV	R2, R0	;	<i>obtenemos la respuesta en R0</i>

- ❑ Si R2 vale 0003, R7 vale 0419h y la instrucción CALL `triple` está almacenada en la dirección 040Fh, completa el siguiente dibujo que representa el estado de la pila cuando la primera instrucción de `triple` va a ejecutarse (recuerda que CALL también mete cosas en la pila).

Dirección	Contenido

Toda la dificultad del paso de parámetros se halla en realidad dentro del procedimiento `triple`, ya que es ahí donde hay que usar algunos trucos para poder acceder al parámetro sin tener que usar POP (no podemos usar POP porque sacaríamos la dirección de retorno y entonces RET no funcionaría).

## 2.2. Programación del procedimiento

Un procedimiento debe tener cuidado de no alterar ningún registro del procesador, excepto R0 que es el que usa para devolver el resultado. Por tanto es conveniente guardar en la pila todos los registros que el procedimiento utilice para sus cálculos y restaurarlos antes de retornar. Sin embargo, al meter registros en la pila se modifica R7, complicándose los cálculos necesarios para encontrar el lugar con respecto a R7 donde están los parámetros. Por esto, es habitual seguir siempre una estructura en la programación de procedimientos que simplifique al máximo posible estos problemas.

Esta estructura es la que se muestra en el programa 2-4proc3.ens que está en la carpeta de prácticas de la asignatura (debes copiarlo a tu disquete) y cuyo listado tienes en la figura 4.1 en la página siguiente para mayor comodidad.

```

1  ORIGEN 400h
2  INICIO principal
3  .PILA 8
4  .CODIGO
5  PROCEDIMIENTO triple
6      ; Paso 1 --- Guardar R6
7      PUSH R6
8      ; Paso 2 --- Copiar R7 en R6
9      MOV R6, R7
10     ; Paso 3 --- Salvaguardar registros
11     ; Se guardan todos los registros que el procedimiento
12     ; vaya a modificar, salvo R0
13     ; (y salvo R6 que ya se ha guardado y R7 que es
14     ; el puntero de pila)
15     PUSH R3
16
17     ; Paso 4
18     ; Incrementar R6 lo necesario hasta que apunte al parámetro
19     ???
20     ; Paso 5
21     ; Copiar el parámetro a un registro (R3)
22     ???
23     ; Paso 6 --- Realizar el algoritmo
24     ; (en este caso, hallar el triple de R3 dejando el
25     ; resultado en R0)
26     MOV R0, R3 ; R0=R3
27     ADD R0, R0, R3 ; R0=R3+R3
28     ADD R0, R0, R3 ; R0=R3+R3+R3
29
30     ; Paso 7
31     ; Recuperar los registros guardados en la pila en el paso 3
32     POP R3
33     ; Paso 8
34     ; Recuperar el valor original de R6
35     ???
36     ; Paso 9 y final. Retornar
37     RET
38 FINP
39
40 principal:
41     MOVH R2, 0
42     MOVL R2, 3
43
44     ; Llamar a triple pasando R2 como parámetro
45     PUSH R2
46     CALL triple
47     MOV R2, R0
48     ; Eliminar el parámetro que aun está en la pila
49     ; ???
50 FIN

```

Figura 4.1: Listado del fichero 2-4proc3.ens

- ☐ Lee con atención los comentarios del programa. Los pasos que aquí se describen son los recomendados para todo procedimiento.
- ☐ Vamos a completar el dibujo siguiente (la pila):

Dirección	Contenido
0419	—

Para ello, mira el listado comenzando en la etiqueta principal y ve ejecutando mentalmente el programa, escribiendo en la pila cada vez que ejecutes una instrucción que la modifique (recuerda que CALL lo hace), hasta llegar al Paso 4. [Nota: el valor inicial de R7 es 0419h y la instrucción CALL triple está en la dirección 040Fh]

- ☐ ¿Qué valor tiene R7 en ese instante?<sup>[12]</sup> ¿Y R6?<sup>[13]</sup> ¿Por qué no son iguales?
- ☐ En la figura de la pila, pinta una flecha señalando al lugar al que apunta R6. ¿Cuántas unidades por debajo de ese lugar está el parámetro?<sup>[14]</sup>
- ☐ Abre con EDIT el archivo 2-4proc3.ens
- ☐ Completa lo que falta en el paso 4, sustituyendo los interrogantes por una secuencia de instrucciones INC, tantas como creas necesarias
- ☐ Completa lo que falta en el paso 5. Tras esto, tendremos una copia del parámetro en el registro R3
- ☐ Completa lo que falta en el paso 8 (sacar el valor que habíamos guardado en la pila al principio del procedimiento)
- ☐ Guarda el archivo y sal de EDIT.

12

13

14

Aún faltan unos interrogantes por completar, al final del programa, pero de momento vamos a dejarlo así para obtener una primera versión de prueba.

- ☐ Ensambla el programa para obtener 2-4proc3.eje (comprueba que no obtienes errores de ensamblado) y cárgalo en el simulador.
- ☐ Pulsa **[F8]**. Observa cómo la primera instrucción que se ha ejecutado es la del programa “principal”. Pulsa **[F8]** de nuevo para terminar de inicializar R2 con el valor 0003.
- ☐ Pulsando **[F8]** otra vez, metemos el parámetro en la pila. Hazlo.
- ☐ Al pulsar de nuevo **[F8]**, ejecutaremos CALL. Hazlo. ¿Qué valor toma PC?<sup>[15]</sup> ¿Sabrías decir por qué este valor, en este caso, coincide con el de la directiva ORIGEN?

15

- ❑ Sigue pulsando **[F8]**, hasta que la instrucción que veas en IR sea `MOV R0, R3`. En este punto ya se han ejecutado las instrucciones que habías añadido para incrementar R6 y para cargar el parámetro en R3. Si lo habías hecho bien, R3 debería valer en este instante 0003, puesto que ese era el valor del parámetro, ¿cuánto vale R3 en tu caso?<sup>[16]</sup> Si no es 0003 has hecho mal el paso de parámetros. Vuelve al EDIT y repasa tu programa. También puede ayudarte el mirar en este momento en el simulador los valores de R7 y de R6, y examinar los contenidos de la pila con el editor hexadecimal.
- ❑ Pulsa **[F8]** hasta que la instrucción que aparece en IR sea `RET`. En ese instante `RET` ya se habrá ejecutado y si todo estaba bien, debes tener en R0 el resultado de la operación. ¿Cuánto vale R0?<sup>[17]</sup> Observa cómo R2 aún tiene su valor original, ¿cuál?<sup>[18]</sup> Todos los restantes registros deben tener valor cero (salvo R7). Si no es así, es que no has restaurado correctamente R6 en el Paso 8.
- ❑ Pulsa de nuevo **[F8]**. La instrucción que se ejecuta es `MOV R2, R0` para almacenar el resultado que ha devuelto el procedimiento. Si no te aparece esa instrucción en IR es que no has retornado al punto correcto. Esto se deberá a un error en el manejo de la pila (has olvidado sacar de ella un dato que habías metido).
- ❑ El programa ha terminado (la instrucción siguiente sería ya `NOP`). No obstante, hay un problema ¿cuál es el valor ahora de R7?<sup>[19]</sup> ¿Y cuál era su valor inicial?<sup>[20]</sup> ¿Por qué no coinciden?
- ❑ La respuesta es que antes del `CALL` hicimos un `PUSH` para meter un parámetro. Este dato aún sigue en la pila (compruébalo con el editor hexadecimal). Es necesario *eliminarlo* antes de que el programa pueda darse por correctamente finalizado. Edita de nuevo `2-4proc1.ens` y añade al final la instrucción o instrucciones necesarias para eliminar los parámetros (sin usar `POP`).
- ❑ Ensambla el programa y cárgalo en el simulador. ¿Qué valor tiene inicialmente R7?<sup>[21]</sup> (no es el mismo que antes porque ahora tu programa tiene más instrucciones, y por tanto su código ocupa más, recuerda que el ensamblador pone la pila donde termina el código).
- ❑ Pulsa repetidas veces **[F8]** hasta que aparezca la instrucción `NOP` en IR. Cuando esto ocurra, tu programa habrá terminado. Entonces todos los registros deben valer 0000, excepto R0 que valdrá 0009 (el resultado), y R2 que tendrá una copia de R0. Además, R7 debe tener el mismo valor que tenía al principio (el que has respondido en el punto anterior). Si es así ¡enhorabuena! Tu programa ha funcionado perfectamente.

16

17

18

19

20

21

## 3. Autoevaluación

### 3.1. Archivos en el disco

En tu disco de prácticas deberás tener los archivos de programa `2-4proc1.ens`, `2-4proc2.ens` y `2-4proc3.ens`, si bien estos ficheros no serán necesarios para las sesiones futuras.

### 3.2. Ejercicios

- ⇒ Modifica el programa `2-4proc1.ens`, alterando el valor de `R7` dentro del procedimiento `minimo` (por ejemplo, puedes hacer un `INC R7` o cualquier otra cosa). Trata de anticipar qué ocurrirá cuando se ejecute `RET`. El retorno no se producirá al lugar correcto, pero ¿eres capaz de decir a dónde se retornará y por qué?
- ⇒ En otra práctica (sesión 3 del bloque 2) habíamos programado un algoritmo que calculaba la suma de los  $N$  primeros números naturales. En aquella práctica el valor  $N$  estaba prefijado en 5. Programa aquel algoritmo de nuevo en forma de procedimiento, haciendo que  $N$  sea un parámetro. El procedimiento devolverá el resultado en `R0`
- ⇒ Convierte en un procedimiento el programa que has desarrollado en la sesión 2 de este mismo bloque de prácticas, que encontraba el máximo entre una serie de números. En este caso, el parámetro debería ser la serie de números, pero puesto que estos números se encuentran almacenados en la memoria (en un array), lo que le pasaremos como parámetro al procedimiento será la dirección de memoria donde está el primero de ellos. El número de datos a procesar lo consideraremos fijo de momento, igual a 12.  
El procedimiento irá procesando el array a partir de esa dirección hasta terminar con los 12 elementos, y retornará en `R0` el valor del mayor de ellos.  
Para llamar al procedimiento, necesitarás meter en la pila la dirección donde están los datos. Para averiguar esa dirección tienes la directiva `DIRECCION`. Debes usarla como ya sabes en combinación con las directivas `BYTEALT0` y `BYTEBAJO` para guardar esa dirección en un registro, y después copiar ese registro a la pila mediante `PUSH`, tras lo cual ya podrás llamar (`CALL`) al procedimiento.
- ⇒ Modifica el procedimiento anterior para que reciba también como parámetro el número de elementos a procesar.