Bloque 3

El lenguaje de la máquina

Prácticas de
Fundamentos de Computadores y Redes
13 de febrero de 2017

SESIÓN 1

Sentencias de asignación

Objetivos

Este bloque tiene como objetivo comprender la traducción de programas escritos en lenguaje de alto nivel al lenguaje del Computador Teórico. En esta sesión se presentará el simulador del Computador Teórico que se usará a lo largo del bloque práctico y se utilizará para estudiar la traducción de sentencias de asignación y aritmético-lógicas desde un lenguaje de alto nivel que es `C++`.

Conocimientos y materiales necesarios

Para aprovechar adecuadamente esta sesión de prácticas, el alumno necesita:

- Conocer los sistemas de codificación binario natural y complemento a 2, así como tener soltura en la conversión a hexadecimal.
- Conocer el lenguaje de programación `C++`.
- Conocer el lenguaje ensamblador del Computador Teórico.
- Conocer conceptos básicos de traducción de sentencias de alto nivel, tales como sentencias de asignación y aritmético-lógicas, a lenguaje máquina.

Desarrollo de la práctica

A lo largo de esta sesión, se presentarán pequeños fragmentos de código en C++, junto con indicaciones de cómo se almacenarán las variables declaradas en esos fragmentos (si se almacenan en un registro, en cuál sería, o si por el contrario se almacenan en memoria, en qué dirección estarían). Se te pedirá que hagas el papel de un compilador y escribas un corto fragmento de código en lenguaje ensamblador del Computador Teórico que sea equivalente en su funcionalidad al fragmento de C++ propuesto.

Para cada ejercicio propuesto deberás crear un fichero llamado `Ejercicio-N.txt`, siendo `N` el número del ejercicio, y con un editor de texto escribir en ese fichero la solución al ejercicio. Cualquier editor es válido, como por ejemplo el Bloc de Notas de Windows, u otro que prefieras, como `Notepad++`.

Para comprobar si la solución que has ideado es correcta, accederás a la siguiente dirección web:

http://www.atc.uniovi.es/tools/CPU/simulador.php

En esta página encontrarás un simulador del Computador Teórico que te permitirá introducir el código que has escrito (copiando y pegando desde el editor) y ejecutarlo para comprobar que, tras la ejecución, todos los registros tienen el valor esperado. Antes de ejecutar el programa puede ser necesario cargar ciertos valores iniciales en los registros, o en ciertas posiciones de memoria. Si así fuera, el ejercicio lo indicará en su momento.

Importante: Todos los valores iniciales que se especifiquen para registros o memoria a través de la interfaz web, se interpretan en hexadecimal, siendo la `h` final opcional. Sin embargo, los datos inmediatos que aparezcan en el listado ensamblador de la interfaz web ***sí*** deberán llevar la h final, o de lo contrario se entenderá que están en base 10.

Por defecto, este simulador mostrará en pantalla los valores de todos los registros antes y después de que se ejecute el código introducido, coloreando en rojo aquellos que han cambiado de valor. Es posible asimismo indicar ciertas posiciones de memoria cuyo contenido se quiere observar también. Para ello basta escribir la dirección de memoria deseada bajo la columna "Dirección", y marcar la casilla "Mirar" de esa dirección. También aquí se puede especificar un valor inicial para el contenido de esa dirección.

Si el resultado no es el que se esperaba, habrá que localizar la causa y corregirla. Para este cometido puede ser muy útil activar la opción "Obtener trazado de cada instrucción". Cuando esta opción está activa, no sólo se muestran los estados inicial y final, sino también el estado de todos los registros tras la ejecución de cada una de las instrucciones. De este modo puedes comprobar en qué momento el resultado deja de ser el que se esperaba y comprender mejor las causas del fallo.

1. Asignación y expresiones simples

▶ **Ejercicio 1**. Traduce al ensamblador del Computador Teórico la siguiente asignación en `C++`. Cada variable se almacena en un registro, indicado en el comentario.

```
1 a=b;    // Con a en r3,  b en r5
```

Para probar tu código, pon en el simulador web un valor inicial de 7 al registro `r5`. Comprueba que, tras ejecutar el código, el registro `r3` adquiere también el valor 7.

▶ **Ejercicio 2**. Convierte al ensamblador del Computador Teórico el siguiente fragmento de código, suponiendo que la variable `c` está almacenada en el registro `r1`.

```
1 c=65;   // Con c en r1
```

Observa que en el código `C++` el dato introducido se da en base 10. A la hora de escribirlo en ensamblador puedes darlo también en base 10, o en hexadecimal, como prefieras. En este segundo caso deberás añadir una `h` al final del dato en el listado ensamblador. Sin embargo, el simulador web te hará el volcado de los registros siempre en hexadecimal. Comprueba que tras la ejecución del programa, el registro `r1` contiene el valor `0041h` (hexadecimal).

▶ **Ejercicio 3**. Repite el primer ejercicio, pero ahora considera que la variable `b` está almacenada en la dirección de memoria `0500h`, es decir:

```
1 a=b;    // Con a en r3,  b en memoria (0500h)
```

Si necesitas registros auxiliares puedes utilizar cualquiera de los que no estén siendo usados para otra cosa. Para comprobar el correcto funcionamiento, usa el formulario web para indicar que la dirección de memoria `0500h` contiene el valor inicial `1234h` (puedes marcar también "Mirar" para comprobar el valor de esa dirección durante la ejecución del programa). Verifica que tras ejecutarse tu programa, el registro `r3` (que es el que contiene la variable `a`) termina con el valor `1234h`, que ha tomado de esa dirección de memoria.

► **Ejercicio 4**. Considerando, al igual que en el ejercicio anterior, que la variable `b` está almacenada en `0500h`, traduce el siguiente fragmento `C++` al ensamblador del Computador Teórico.

```
1 b=27;    // Con b en memoria, en la dirección 0500h
```

Si necesitas registros auxiliares puedes utilizar cualquiera de los que no estén siendo usados para otra cosa. Usa el formulario web para "mirar" la dirección de memoria `0500h`, que debe comenzar con el valor `0000h`, y comprueba que al terminar el programa tiene el valor `001Bh` (que es 27).

► **Ejercicio 5**. Traduce al ensamblador del Computador Teórico la siguiente asignación, que contiene una sencilla expresión. Supón que la variable `a` se almacena en la dirección de memoria `0501h`, mientras que las variables `b` y `c` lo hacen en los registros `r1` y `r2`, respectivamente.

```
1 a = (a + b) - (c + 2);    // Con a en memoria (0501h), b en r1, c en r2
```

Para comprobarlo, fijemos los siguientes valores iniciales (que damos aqui en base 10): `a=16`, `b=5`, `c=7`. Según estos valores ¿cuál habría de ser el valor final de `a`?[1]

1

Inicializa en el simulador web los registros `r1` y `r2` con los valores iniciales de `b` y `c`, respectivamente, y la dirección de memoria `0501h`, donde se guarda `a`, con el valor `0010h` (que es 16 en hexadecimal). Comprueba que tras ejecutarse tu código, la dirección `0501h` contiene el valor que habías predicho para `a`. Observa asimismo que la variable `a` es la única que resulta modificada en la expresión. Las variables `b` y `c` (es decir, los registros `r1` y `r2`) han de conservar sus valores iniciales. Si necesitas registros auxiliares para operaciones intermedias puedes usar `r3`, `r4` y `r5`.

Ya que esta expresión, a pesar de su aparente sencillez, da lugar a varias líneas de código en ensamblador, puede resultar útil activar la opción "Obtener trazado de cada instrucción" en el simulador web.

2. Ejercicios adicionales

Intenta realizar ahora los siguientes ejercicios que tienen algo más de dificultad:

✏ **Ejercicio 6**. Repite el ejercicio 5, pero en esta ocasión las tres variables estarán en memoria, en concreto, `a` estará en la dirección `0501h`, `b` en `0502h` y `c` en `0503h`:

```
1 a = (a + b) - (c + 2);    // Con a, b y c en memoria (0501h, 0502h y 0503h, resp)
```

Para comprobar el funcionamiento, inicializa las direcciones antes enumeradas de forma que los valores iniciales de las variables sean `a=9`, `b=8`, `c=3`. Con estos valores, ¿cuál sería el resultado de la expresión?[2]

[2]

Ejecuta tu programa "vigilando" la dirección `0501h` que corresponde a la variable `a`, para verificar que al finalizar el mismo esa dirección contiene el resultado que has predicho. Vigila también las direcciones de las otras variables para comprobar que no cambian de valor.

✏ **Ejercicio 7**. Traduce a lenguaje máquina las siguientes instrucciones del lenguaje C. Como en el ejercicio anterior, las variables `a` y `b` están en las direcciones de memoria `0501h` y `0502h` respectivamente.

```
1 a++;
2 b-=a;
```

Si antes de ejecutar el fragmento de código anterior `a` vale 2 y `b` vale 4, ¿cuáles serán sus valores tras la ejecución?[3]. Inicializa las direcciones de memoria correspondientes a `a` y `b` con los valores 2 y 4 y verifica que al finalizar la ejecución de tu programa ambas contienen los valores predichos.

[3]

SESIÓN 2

Condicionales y bucles

Objetivos

Esta sesión prosigue el objetivo de comprender la traducción de programas escritos en lenguaje de alto nivel (C++) al lenguaje ensamblador del Computador Teórico. Se estudiará en esta sesión la traducción de sentencias y bucles.

Conocimientos y materiales necesarios

Para aprovechar adecuadamente esta sesión de prácticas, el alumno necesita:

- Conocer los sistemas de codificación binario natural y complemento a 2, así como tener soltura en la conversión a hexadecimal.
- Conocer el lenguaje de programación C++.
- Conocer el lenguaje ensamblador del Computador Teórico.
- Conocer conceptos básicos de traducción de sentencias de alto nivel a instrucciones de la máquina, tales como sentencias de asignación, aritmético-lógicas, condicionales, y bucles.

Desarrollo de la práctica

1. Condicionales y bucles

► **Ejercicio 8**. Traduce al ensamblador del Computador Teórico el siguiente fragmento de C++

```
1 if (a==b)    // a en r0, b en r1
2   c=1;       // c en r2
3 else
4   c=2;
5 d=c;         // d en r3
```

teniendo en cuenta las asignaciones de variables a registros que se indican en los comentarios. Si las variables a y b son iguales, ¿qué valor debería aparecer al final en d?[1]

1

Para comprobarlo inicializa en el simulador web los registros que almacenan dichas variables `a` y `b` con dos valores iguales. Ejecuta tu código y comprueba que al final el registro que almacena la variable `d` tiene el valor que has predicho. ¿Y si los valores iniciales de `a` y `b` fuesen diferentes? ¿Cuál sería en este caso el valor final de `d`?[2]. Compruébalo de nuevo en el simulador, ejecutando el mismo código pero con otros valores iniciales para `a` y `b`.

2

Nota: Si marcas la opción "Obtener trazado de cada instrucción", verás en la traza que junto a las instrucciones de salto condicional aparece una pequeña marca, que puede ser "✓" o "×". La primera (✓) indica que la condición del salto se cumple y por tanto se saltará a la etiqueta de destino[1]. La segunda (×) indica que la condición no se cumple y por tanto no se saltará a la etiqueta, sino que se continuará la ejecución por la instrucción siguiente al salto. Estas marcas son una mera ayuda, ya que tú mismo podrías averiguar si la condición se cumple o no mirando los valores que los bits de estado tienen en ese momento (los cuales también salen en la traza).

► **Ejercicio 9**. Considera el siguiente fragmento de código en C++, que asigna cero a la variable `a` si detecta que está fuera de un rango pre-establecido por las variables `b` y `c`. Las tres variables almacenan números naturales.

```
1 if ((a<b) || (a>c))  // a en r0, b en r1, c en r2
2   a=0;
```

Escribe las instrucciones de ensamblador que producen el mismo resultado, usando la asignación de variables a registros indicada en los comentarios.

Comprueba si funciona correctamente, poniendo en las variables `b` y `c` (es decir, en los registros correspondientes a estas variables) los valores `5` y `10`, respectivamente, y haz tres pruebas. En la primera, la variable `a` está dentro del rango (por ejemplo, vale `7`), en la segunda se sale del rango por debajo (por ejemplo, vale `2`) y en la tercera se sale del rango por encima (por ejemplo vale `12`). Comprueba que al finalizar tu programa la variable `a` (es decir, el registro que la contiene) conserva su valor inicial en la primera prueba, pero es cambiado por `0000h` en las otras dos.

► **Ejercicio 10** . El siguiente código en `C++` calcula la suma de los 100 primeros números naturales, dejando el resultado en la variable `a`:

```
1   a=0;                    // a en r0
2   for (i=0; i<100; i++)   // i en r1
3     a=a+i;
```

¿Cuál será el resultado?[3] Puedes obtener la respuesta escribiendo un pequeño programa en `C++` como el anterior, que imprima el valor de `a` al finalizar, y ejecutándolo. O evaluar en un intérprete python la expresión `sum(range(100))`. Pásalo a hexadecimal para comparar con la salida del simulador (en python tienes la función `hex()`).

3

Escribe el equivalente en ensamblador, usando la asignación de variables a registros indicada en los comentarios. Comprueba que el valor final de `a` (es decir, del registro en que se almacena) es el esperado, pero ten en cuenta que el simulador web te dará el resultado en hexadecimal. Si no sale lo que esperas, puedes activar la traza, pero en este caso mejor reduces el número de iteraciones del bucle hasta que localices el problema. Una vez corregido, lo vuelves a 100, pero desactiva la traza. ¡De lo contrario la traza tendría cientos de líneas!

[1] En la traza las etiquetas ya no son visibles y han sido sustituídas por números, que indican cuántas instrucciones se saltarán para llegar al destino. En los saltos hacia atrás este número es negativo, y cuenta a la propia instrucción de salto.

► **Ejercicio 11**. El siguiente fragmento de código C++ va contando cuántos elementos distintos de cero hay en un array dado. Por no complicar el código, se supone que el array es de tamaño suficiente y siempre se encuentra un cero que finaliza el bucle antes de "salirse" de este tamaño.

```
1  contador=0;
2  i=0;
3  while (datos[i]!=0) {
4    contador++;
5    i++;
6  }
```

Para traducir el código anterior a ensamblador, usa `r0` para el `contador`, `r1` para el índice `i` del array, y `r2` para contener la dirección de memoria en la que comienza el array, que será `0740h`. Necesitarás otro registro auxiliar que puede ser `r4` para contener la dirección del dato concreto al que se accede en cada iteración del bucle, el cual se calcula como la suma de la dirección en la que comienza el array más el índice `i`. También necesitarás un registro que contenga cero para comparar con el dato que estás procesando.

Para probar tu código, pon en las direcciones `0740h`, `0741h` y `0742h` tres valores cualesquiera distintos de cero. Al terminar tu programa, el `contador` debe valer 3, ya que habrá encontrado tres valores consecutivos distintos de cero en el array. Prueba a poner un cero en `0741h` y ejecutar de nuevo, y el contador deberá ahora valer `1`, ya que tras contar el primer dato ya encuentra un cero. Finalmente prueba a poner un cero en `0740h` y comprueba que en este caso el contador vale cero.

2. Ejercicios adicionales

Ejercicios adicionales, por si te ha sobrado tiempo.

✏ **Ejercicio 12**. Cuando necesitamos saber si una variable contiene un valor par o impar, habitualmente usamos en C++ o en python el operador "módulo" que da el resto de la división, dividiendo por 2. Si ese resto es 0, es que el número era par y si es 1 es que era impar. Así por ejemplo:

```
1  if (a % 2 == 0)  // Si a es par, incrementar b
2    b++;
```

Sin embargo, traducir directamente eso a código máquina no es trivial, ya que no tenemos una instrucción máquina que nos de el resto de una división, y si tuvieramos que calcular ese resto mediante restas sucesivas sería muy ineficiente.

En lugar de eso, suele utilizarse otra estrategia. Si te das cuenta, al expresar un número en binario, se ve claramente que si el número es par su último bit será cero, mientas que si el último bit es uno, eso implica que el número es impar. De este modo evitamos el tener que hacer divisiones. ¿Cómo comprobar entonces si el último bit es 1 ó 0? En código máquina resulta muy sencillo. Basta hacer la operación `AND` entre el dato en cuestión y el valor `0001h`. Si el resultado de esa operación es `0000h`, es que el último bit del dato era cero y por tanto el número par. Y este caso se detecta fácilmente a través del flag Z.

Así pues, podríamos reescribir en C el programa anterior haciendo uso de esta propiedad, en lugar de usar divisiones o módulos. El lenguaje C también tiene el operador `AND` bit a bit, que se denota por `&`.

```
1  if (a & 0x0001 == 0)  // Si a es par, incrementar b
2    b++;
```

Traduce a ensamblador el listado anterior, haciendo `a=R0`, `b=R1` y comprueba su funcionamiento con el simulador web. Prueba con diferentes valores de `a` (pares e impares) para ver si en cada caso `b` se incrementa adecuadamente o no. Ten cuidado de que al finalizar el programa no haya resultado modificada la variable `a`, que debe terminar con el mismo valor con el que comenzó.

✏ **Ejercicio 13**. Modifica el ejercicio 10 haciendo uso del "truco" que has aprendido en el ejercicio anterior, para que sume los números impares comprendidos entre 0 y 99. Comprueba si obtienes el resultado correcto (que es 2500 o `09C4h`)

SESIÓN 3

Procedimientos y funciones

Objetivos

Esta sesión prosigue el objetivo de objetivo comprender la traducción de programas escritos en lenguaje de alto nivel (C++) al lenguaje del Computador Teórico. Se estudiará en esta sesión la traducción de los procedimientos y sus llamadas. Además, se trabajará con variables locales y con la pila del programa.

Conocimientos y materiales necesarios

Para aprovechar adecuadamente esta sesión de prácticas, el alumno necesita:

- Conocer los sistemas de codificación binario natural y complemento a 2, así como tener soltura en la conversión a hexadecimal.
- Conocer el lenguaje de programación C++.
- Conocer el lenguaje ensamblador del Computador Teórico.
- Conocer conceptos básicos de traducción de sentencias de alto nivel, tales como sentencias de asignación, aritmético-lógicas, condicionales, bucles y procedimientos, a instrucciones de la máquina.

Desarrollo de la práctica

1. Procedimientos

▶ **Introducción**. Para los ejercicios relacionados con procedimientos, a fin de poder probar si funcionan correctamente mediante el simulador web, será necesario no sólo codificar el procedimiento, sino también una llamada al mismo y verificar si a su retorno los registros y posiciones de memoria tienen los valores esperados. Comenzaremos por tanto por un ejemplo guiado que puede servirte de modelo para los siguientes.

Se tiene el siguiente código C++ que implementa una función trivial que no recibe parámetros y siempre retorna cero. A la vez, el programa principal `main()` llama a esta función y deja el resultado en una variable llamada `a`.

```
1 int Cero(void)
2 {
3   return 0;
4 }
5 void main(void)
6 {
7   int a;
8   a=Cero();
9 }
```

A la hora de convertirlo en ensamblador, la función `Cero` será una etiqueta a la que se "saltará" mediante la instrucción `call` desde el programa principal, y que terminará con una instrucción `ret` para volver al punto desde el que se saltó. El `0` que la función debe retornar lo guardará en `r0` antes del `ret`, siguiendo el convenio de que siempre retornaremos el resultado de la función en el registro `r0`. Según esto, la traducción de esta función es muy sencilla:

```
1 Cero:         ; Etiqueta ''nombre'' de la función
2   movh r0, 0  ; preparamos un 0 en r0 para retornarlo
3   movl r0, 0  ;
4   ret         ; y saltamos al punto desde donde la función fue llamada
```

En cuanto al programa principal, en el lenguaje `C++` se trata de otra función llamada `main()` y en realidad, como función que es, contiene un retorno (al sistema operativo). Sin embargo, cuando hagamos la conversión a código ensamblador no la trataremos como una función, sino que escribiremos "directamente" su código sin que sea llamado desde ningún sitio ni tenga que retornar a otro.

En el ejemplo anterior, ya que no consideraremos a `main()` como una función, la variable local `a` tampoco la consideraremos variable local, y no se guardará en la pila. Supondremos que está en una posición de memoria prefijada, igual a `0400h`. Por tanto, el código que llama a la función `Cero()` y guarda en `a` el resultado de esta función sería:

```
1 principal:      ; Esta etiqueta no es necesaria, es para mayor claridad
2   movh r1, 04h
3   movl r1, 00h  ; r1 apunta a 0400 que es donde está la variable a
4   call Cero     ; Llamamos a la función. El ''resultado'' viene en r0
5   mov  [r1], r0 ; y lo guardamos en a
```

A fin de probar este código, es necesario juntarlo en un solo listado. Dado que el simulador web siempre comienza ejecutando las instrucciones por la primera que escribamos, la función `Cero()` no puede ir al principio del listado. Si así lo hiciéramos, la ejecución empezaría por esa función, en lugar de hacerlo por el "programa principal" y, cuando alcanzara la instrucción `ret`, ¿a dónde retornaría?

Por tanto al principio del código ha de ir el programa "principal", como muestra el siguiente listado:

```
1 principal:      ; Esta etiqueta no es necesaria, es para mayor claridad
2   movh r1, 04h
3   movl r1, 00h  ; r1 apunta a 0400 que es donde está la variable a
4   call Cero     ; Llamamos a la función. El ''resultado'' viene en r0
5   mov  [r1], r0 ; y lo guardamos en a
6
```

```
 7 Cero:         ; Etiqueta ‘‘nombre’’ de la función
 8   movh r0, 0  ; preparamos un 0 en r0 para retornarlo
 9   movl r0, 0  ;
10   ret         ; y saltamos al punto desde donde la función fue llamada
```

Sin embargo téngase en cuenta otro detalle. Cuando el programa ejecute `call Cero`, se producirá un salto hasta la etiqueta `Cero:`, se ejecutará esa función y cuando se encuentre la instrucción `ret` se volverá a la instrucción que hay tras el `call`, es decir, tras el `ret` se ejecutará `mov [r1], r0`. Pero ¿qué ocurrirá después?

El procesador "no sabe" que el programa ha terminado ahí. Continuará ejecutando por la instrucción siguiente a esa, y resulta que la siguiente instrucción será la primera de la función `Cero`. Es decir, ¡se ejecutará otra vez esta función! Y lo que es peor, cuando se alcance por segunda vez el `ret` no hay a dónde retornar, ya que esta segunda vez no ha sido llamada realmente mediante `call`.

Es necesario indicar de alguna forma al procesador que el programa principal ha finalizado. En un procesador real, con un sistema operativo, se colocaría ahí una instrucción que retornara al sistema operativo (por eso en el `C++`, `main()` es otra función, que retorna al operativo cuando termina). En nuestro simulador del Computador Teórico no hay operativo, por lo que no hay a dónde retornar.

Por suerte el simulador tiene implementado que, cuando la instrucción a ejecutar sea `nop` (de *No operation*, una operación que no hace nada), el simulador debe detenerse y dar el programa por finalizado. Esta es la razón por la que todos los ejercicios que hemos hecho hasta ahora se detenían al terminar, y es que todas las posiciones de la memoria en las que no hayamos puesto algo, están por defecto rellenas con `0000h`, que es el código máquina de la instrucción `nop`. Por eso, al final de cada uno de los ejercicios anteriores había instrucciones `nop` aunque no las hubiéramos escrito.

En este caso, para forzar a que el simulador se detenga una vez haya finalizado el programa principal, escribiremos explícitamente la instrucción `nop` al final del mismo. El código completo queda por tanto así:

```
 1 principal:      ; Esta etiqueta no es necesaria, es para mayor claridad
 2   movh r1, 04h
 3   movl r1, 00h  ; r1 apunta a 0400 que es donde está la variable a
 4   call Cero     ; Llamamos a la función. El ‘‘resultado’’ viene en r0
 5   mov  [r1], r0 ; y lo guardamos en a
 6   nop           ; FIN DEL PROGRAMA
 7
 8 Cero:         ; Etiqueta ‘‘nombre’’ de la función
 9   movh r0, 0  ; preparamos un 0 en r0 para retornarlo
10   movl r0, 0  ;
11   ret         ; y saltamos al punto desde donde la función fue llamada
```

Puedes probar a ejecutar este código en el simulador web. Activa el modo traza para ver cómo se produce la llamada y el retorno. Inicializa la dirección `0400h` (variable `a`) con un valor distinto de cero y activa la casilla "Mirar", para comprobar cómo al final de la ejecución esa posición ha cambiado de valor y es cero.

En los ejercicios que siguen deberás ceñirte al esquema anterior, es decir, programa principal al principio del listado, que se ocupa de apilar los parámetros si los hubiere (en el ejemplo anterior no los hay), llamar a la función, eliminar los parámetros incrementando `r7`, guardar el resultado en la variable apropiada y terminar con `nop`, y tras éste, vendría el código de la función a implementar.

Importante. Como sabes, la pila utiliza implícitamente el registro r7, por lo que éste irá cambiando de valor a lo largo del programa (con cada `push`, `pop`, `call` y `ret`). Sin embargo, al finalizar la ejecución, el valor final de `r7` ha de ser igual al que tenía al principio, pues de no ser así ello significaría que la pila no ha sido usada de forma equilibrada. Asegúrate de que así es en todos los casos (lo cual es sencillo porque si `r7` cambia de valor aparecerá en rojo).

► **Ejercicio 14**. Implementa la función `Max()` que recibe dos números naturales y retorna el valor del mayor de ellos. Implementa también el programa principal que llama a esa función, como se muestra en el siguiente código `C++`. Las variables deben asignarse a los registros que se indican en los comentarios.

```
1  unsigned int Max(unsigned int a, unsigned int b)  // a y b son parámetros en la pila,
2                                                    // apilados de dcha. a izda.
3  {
4    if (a>b)
5      return a;          // retornado en r0
6    else
7      return b;
8  }
9  void main(void)
10 {
11     unsigned int x=5, y=9;       // x en r1, y en r2
12     unsigned int resultado;      // resultado en memoria, dirección 0500h
13
14     resultado=Max(x, y);
15 }
```

Compruébala en el simulador, trazando el contenido de la dirección `0500h` y comprobando que al final contiene `9`, ya que éste es el mayor entre `x` e `y`.

► **Ejercicio 15**. Implementa la función `MaxMin()` que recibe dos números naturales y retorna por separado el valor del mayor y del menor de ellos. Para poder retornar dos resultados, recibirá dos parámetros por referencia. Ya que los resultados los "devuelve" en esos parámetros, no necesita retornar nada a través del `return`, por lo que la función se declara de tipo `void` (lo cual en ensamblador implica que no va a devolver nada en `r0`).

Traduce al ensamblador del Computador Teórico el siguiente programa en `C++`, siguiendo las indicaciones de los comentarios para mapear las variables en registros o memoria.

```
1  void MaxMin(unsigned int a, unsigned int b, unsigned int& max, unsigned int& min)
2  {   // a, b, max y min son parámetros en la pila, apilados de dcha. a izda.
3     if (a>b) {
4       max=a;       // Observar que max y min se han pasado por referencia
5       min=b;       // En ensamblador se usa la dirección a que apuntan estos parámetros
6     }
7     else {
8       max=b;
9       min=a;
10    }
11 }
12 void main(void)
13 {
14    unsigned int x=5, y=9;        // x en r1, y en r2
15    unsigned int mayor, menor;    // ambas en memoria, mayor en 0500h, menor en 0501h
16    MaxMin(x, y, mayor, menor);
17 }
```

Una vez traducido a ensamblador, ejecutarlo en el simulador web, "mirando" las direcciones `0500h` y `0501h` y comprobando que una vez finalizado el programa cada una contiene, respectivamente, el mayor y el menor de los datos.

2. Ejercicios adicionales

Intenta realizar ahora los siguientes ejercicios que tienen algo más de dificultad:

✏ **Ejercicio 16**. El siguiente código en `C++` implementa una función llamada `Maximo()` que recibe dos parámetros. El primero es un array pasado por referencia. El segundo es un entero que indica el tamaño del array (pasado por copia). La función recorre el array para encontrar el elemento de mayor valor, y retorna el valor encontrado.

Para probar la función, se escribe un programa principal que tiene un array con 5 elementos, inicializados como se indica en la declaración del array.

```
 1 unsigned int Maximo(unsigned int datos[], unsigned int cuantos)
 2 {
 3    unsigned int i;      // i en registro r5
 4    unsigned int mayor;  // mayor en registro a elegir por el alumno
 5
 6    mayor=0;
 7    for (i=0; i<cuantos; i++)
 8      if (datos[i]>mayor)     // datos[i] en r4
 9        mayor=datos[i];
10    return mayor;
11 }
12
13 void main()
14 {
15    unsigned int d[5]={3, 2, 9, 1, 4};
16    unsigned int n=5;                  // n en r1
17    unsigned int resultado;            // resultado en r2
18
19    resultado=Maximo(d, n);
20 }
```

En tu traducción a ensamblador, el array estará almacenado a partir de la dirección `0600h`. Por tanto, usando el simulador web, debes inicializar las posiciones de memoria `0600h`, `0601h`, `0602h`, `0603h` y `0604h` con los valores `3`, `2`, `9`, `1`, y `4`, respectivamente. No es necesario "mirar" estas posiciones de memoria, ya que en teoría no cambian de valor durante la ejecución del programa. No obstante, si algo no funciona bien puedes activar la casilla "mirar" para comprobar que efectivamente no cambien.

Tras ejecutar tu programa, `r2` (que contiene el resultado) ha de tomar el valor `9`, ya que este es el máximo entre los datos. Puedes probar a ejecutar varias veces tu código, cambiando los valores iniciales que hay en memoria, o incluso el valor inicial de `r1` (que es el que almacena la variable `n` que dice cuántos elementos tiene el array), reduciéndolo a valores inferiores a `5`, para comprobar que la función busca el máximo sólo entre los primeros `n` elementos. Por ejemplo, si con los datos anteriores `n` fuera 2, el resultado sería `3`, ya que el máximo entre los dos primeros elementos es `3`.

✏ **Ejercicio 17**. Se tiene un array conteniendo varios números naturales, y se quiere buscar dentro del array la primera aparición de un dato dado. Para ello se implementa una función

`Busca()` que recibe el array (por referencia), un número natural indicando cuántos elementos tiene el array, y otro número natural que es el dato que se busca. La función retorna, en un número entero, el índice del primer elemento cuyo valor coincida con el dato buscado. Si el dato buscado no aparece, la función retorna -1 (`FFFFh`).

El siguiente ejemplo muestra cómo se llamaría a la función, que en este caso retornaría 3, ya que el dato buscado aparece en el elemento 3 del array (recuerda que los índices de los arrays comienzan en `0`, por lo que el elemento de índice `3` es en realidad el cuarto elemento del array).

```
1 void main()
2 {
3    unsigned int datos[]={1, 7, 2, 4, 5};
4    unsigned int n = 5;           // numero de elementos del array, r1
5    unsigned int busco = 4;       // dato buscado, r2
6    int encontrado;               // resultado, r3
7
8    encontrado=Buscar(datos, n, busco);
9 }
```

La implementación de la función én `C++` sería como sigue:

```
1 int Buscar(unsigned int dat[], unsigned int cuantos, unsigned int cual)
2 {
3     unsigned int i;          // Indice de bucle, r1
4     for (i=0; i<cuantos; i++)
5       if (dat[i]==cual)      // Si se encuentra
6         return i;            // retornamos la posición
7     // Si hemos llegado aqui sin encontrarlo, retornamos -1
8     return -1;
9 }
```

Traduce al ensamblador del Computador Teórico la función anterior y el programa principal de prueba. Ejecútalo en el simulador web con los datos suministrados (que guardarás en memoria a partir de la posición `0500h`) y pruébalo también con otros datos, con datos repetidos, buscando otros números, etc. para asegurarte de que funciona en todos los casos.