

## Laboratorio 6. Procesadores hardware. Simulación con ARMSim#

### Objetivos:

- Experimentar la ejecución simulada de programas escritos en ensamblador de ARM.
- Comprobar los resultados de la ejecución de instrucciones de movimiento y procesamiento (mov, add, and, cmp), de instrucciones de bifurcación (b, bge), de instrucciones de acceso a la memoria (ldr, str) y de llamadas al S.O. (swi) con interrupciones software.

### Recursos:

- El ordenador del laboratorio o cualquier equipo que tenga instalado el simulador ARMSim#.
- El Moodle de la asignatura

### Actividades previas:

Antes de la sesión de laboratorio, debe leer previamente este documento e intentar realizar las actividades indicadas para asegurar que puede completarlo antes de finalizar la sesión. No se olvide de transferir a su cuenta en el laboratorio los ficheros con el trabajo realizado previamente, llevándolos en una memoria USB o transfiriéndolos por email.

### Resultados:

Como resultado de esta práctica debe subirse a Moodle un fichero ZIP que contenga el fichero con el código fuente del programa ensamblador resultado de esta tarea (Grupo-Apellido1-Apellido2-L6.s), junto con una captura gráfica de la ventana del simulador ARMSim# con el programa cargado y ejecutado, donde se vea la salida por consola .

### Programa objeto de la práctica

Empiece desde el entorno gráfico, abra un terminal de texto, cree un nuevo directorio **lab6**, entre en este directorio (**lab6**) y realice paso a paso las siguientes actividades:

1. Descargue desde Moodle en **lab6** el fichero **loteria.js**. Este fichero contiene un programa JavaScript que simula una lotería, que da premios un 25% de las veces que se ejecuta. El código del programa es el siguiente:

```

1 // número aleatorio entre 0 y 255
2 var num = Math.random(); // Entre 0 y 1
3 num = Math.floor(num * 256); // Entre 0 y 255
4
5 var str = "\nLo sentimos, no hay premio";
6
7 if (num < 64) { // probabilidad=25% (64/256)
8     str = "\nHa ganado 10 Euros";
9 }
10
11 console.log(str);
12
13 console.log("num: " + num);
14

```

2. Ejecute el programa **loteria.js** en el terminal de texto con el interprete node varias veces y verá como el programa puede cambiar el mensaje a consola de una ejecución a otra. Unas veces indica que se ha ganado un premio de 10 euros y otras que no se ha ganado premio.

3. Descargue desde Moodle en [lab6](#) el fichero `loteria.s` que contiene una implementación a medio terminar en lenguaje ensamblador de ARM del programa JavaScript `loteria.js`, cuyo contenido se muestra a continuación:

```
1 .equ      stdout,      1
2 .equ      timer,       0x6d
3 .equ      writeNum,    0x6b
4 .equ      writeString, 0x69
5 .equ      halt,        0x11
6
7  /* Generar número aleatorio entre 0 y 255:
8
9      var num = Math.random();      // Entre 0 y 1
10     num = Math.floor(num * 256); // Entre 0 y 255 */
11
12 start:
13     swi     timer          ; Número de milisegundos a R0
14     and     r0,r0,#0xFF    ; Aplicar máscara que deja 8 últimos bits
15     ldr     r4,=num        ; Cargar puntero a variable num en R4
16     str     r0,[r4]        ; Guardar número aleatorio (en R0) en num
17
18     /* var str = "\nLo sentimos, no hay premio"; */
19
20     ldr     r5,=str        ; Cargar puntero a variable str en R5
21     ldr     r0,=SinPremio  ; Cargar puntero a msj: SinPremio en R0
22     str     r0,[r5]        ; Guardar puntero (en R0) en variable str
23
24     /* if (num < 64) { // probabilidad=25% (64/256)*/
25
26     ldr     r0,[r4]        ; restaurar num en R0
27     cmp     r0,#64         ; R0 < 64 (probabilidad=25%)
28     bge     seguir         ; Saltar si mayor o igual que 64
29
30     /* str = "\nHa ganado 10 Euros"; */
31
32 /*>>>> Incluir aquí las instrucciones ensamblador necesarias
33 para implementar la sentencia JavaScript anterior */
34
35     /* } // Etiqueta seguir: permite saltar fuera del bloque */
36
37 seguir:
38
39     /* console.log(str); */
40
41 /*>>>> Incluir aquí las instrucciones ensamblador necesarias
42 para implementar la sentencia JavaScript anterior */
43
44     /* console.log("num: " + num); */
45
46 /*>>>> Incluir aquí las instrucciones ensamblador necesarias
47 para implementar la sentencia JavaScript anterior */
48
49 final:
50     swi     halt           ; Finalizar ejecución
51
52
53 .data      /* definición de variables y mensajes */
54 num:       .word 0         ; variable num
55 str:       .word 0         ; variable str
56
57 /*>>>> incluir la directiva para definir el string "num: " */
58
59 Premio:    .asciz "\nHa ganado 10 Euros\n"
60 SinPremio: .asciz "\nLo sentimos, no hay premio\n"
61 .end
62
```

El esqueleto del programa contenido en el fichero `loteria.s` implementa en ensamblador algunas partes del programa JavaScript con un bloque de instrucciones equivalentes.

El programa empieza definiendo las constantes (`stdout`, `timer`, `writeNum`, `writeString` y `halt`) que se van a utilizar en el resto del programa y que lo hacen más legible.

Las 4 primeras instrucciones en ensamblador a partir de la etiqueta `start` implementan las 2 primeras intrucciones de JavaScript. Estas inicializan la variable `num` con un número aleatorio entre 0 y 255. Para generar el número aleatorio, la primera instrucción lee el número de milisegundos del reloj del sistema con la primitiva `swi 0x6d (timer)`. La segunda instrucción aplica una mascara que extrae los últimos 8 bits con la operación lógica `and`, con lo cual se obtiene un valor aleatorio en coma fija entre 0 y 255. A continuación, la instrucción `ldr r4,=num` carga un puntero a la variable `num` en el registro 4. La siguiente instrucción `ldr r0,[r4]` utiliza este puntero para guardar el valor aleatorio en la variable `num`, reservada con la directiva `num: .word 0;`, utilizando direccionamiento indirecto a través del registro R4. Estas 4 primeras instrucciones ensamblador son equivalentes a los 2 primeras sentencias de JavaScript.


La siguiente sentencia JavaScript (`var str = "\nLo sentimos, no hay premio";`) se implementa con el bloque de 3 instrucciones en ensamblador que viene a continuación, junto con las definiciones de la variable `str` (etiqueta `str:`) y del string (etiqueta `SinPremio:`) que se realizan en el bloque de datos al final. La primera instrucción (`ldr r5,=str`) carga un puntero a la variable `str` en el registro R5. La segunda (`ldr r0,=SinPremio`) carga un puntero al string `SinPremio` en el registro R0. Y la tercera guarda este ultimo puntero en la variable `str` utilizando direccionamiento indirecto a través del registro R5.

De la tercera instrucción JavaScript solo se implementa la comprobación de la condición de la sentencia `if (if (num < 64))` y la no ejecución del bloque, en el caso de no cumplirse dicha condición. La primera sentencia carga la variable `num` en R0 utilizando direccionamiento indirecto a través del registro R4, que contiene un puntero a la variable `num`. La siguiente instrucción (`cmp r0,#64`) compara el número aleatorio, cargado en R0, con el valor 64 para colocar los flags del registro de estado CPSR y poder condicionar la siguiente instrucción al resultado de la comparación. Si el número es mayor o igual que 64 se ejecutará la instrucción de salto condicional, `bge seguir` (saltar si mayor o igual a etiqueta `salir:`) y el bloque de instrucciones de la sentencia `if` de JavaScript no se ejecutará. Como el número aleatorio es equiprobable entre 0 y 255 al comparar con 64 obtenemos una probailidad del 25% de ejecutar el bloque de accione, es decir de dar un premio de 10 Euros.



La implementación en ensamblador de las sentencias JavaScript del programa `loteria.js` que no se han incluido aquí, deben añadirse en esta práctica siguiendo las instrucciones que se dan a continuación, de forma que el programa completo se comporte igual que `loteria.js`. Es decir, cuando se complete la implementación en ensamblador del resto de las instrucciones, la ejecución del programa en el simulador ARMSim# debe modificar de forma similar el contenido de sus variables (estado) y enviar los mismos mensajes a la consola del simulador.


4. Utilizando el terminal de texto, cambie el nombre del fichero `loteria.s` por `Grupo-Apellido1-Apellido2-L6.s`. Una vez cambiado el nombre, añada en la primera línea de este fichero un nuevo comentario indicando su nombre, apellidos y grupo. Una vez realizados estos cambios debe familiarizarse con el simulador ARMSim#:

Arranque el simulador ARMSim# y compruebe que tiene habilitado SWIInstructions en `File>Preferences>Plugins`. Cargue (`File>Load`) el

fichero Grupo-Apellido1-Apellido2-L6.s en el simulador. Observe que, al cargar el fichero al simulador, éste automáticamente ensambla el programa antes de cargarlo. Además genera las instrucciones de maquina (binarias), que son las que en realidad se ejecutan. Si le diera algún error al cargarlo, es porque, al modificar el programa ha cometido algún error. En ese caso, abra el editor de texto, corrija lo necesario, sávelo y vuelva a cargarlo en el simulador (File>Reload, o icono  en la barra de herramientas).

A continuación seleccione Hexadecimal en la tabla de registros (a la izquierda). Observará que todos los registros están a cero salvo el R13 (sp) que contiene el valor 0x5400 y el R15 (pc), que tiene el valor 0x1000 (dirección de comienzo de la ejecución del programa cargado) y verá que en el registro CPSR los indicadores N, Z, C y V están a 0.

Para comprobar que el programa funciona correctamente debe ejecutarlo. El programa se ejecuta instrucción a instrucción con el icono «Step Into»  de la barra de herramientas. De esta forma podrá ver que el cambio del estado de los registros o de los indicadores es el previsto. Si quiere volver al estado inicial del programa, en cualquier punto de la ejecución puede pulsar el icono «Restart»  y reiniciar la ejecución desde el principio otra vez.

También puede ejecutar el programa entero de una vez pulsando el icono «Run» . En este caso el programa no para hasta llegar a la primitiva halt del S.O. (instrucción swi 0x11 (halt)).


Para inspeccionar la parte de la memoria principal que esta utilizando seleccione el desplegable Memory desde el menú View. Una vez seleccionado aparecerá MemoryView en la parte inferior. (Si tiene activado Stack, puede desactivarlo porque no se usa). En MemoryView ponga 00001000 como valor de dirección para visualizar la zona de MP donde esta cargado el programa. Seleccionando WordSize>8bits o WordSize>32bits visualizara el contenido de MP como bytes o palabras.

Seleccionando Stdin/Stdout/Stderr en OutputView de visualizará la consola de ARMSIM#. De esta forma podrá ver los mensajes que se envían a consola desde el programa. El bloque que acaba de implementar no envía ningún mensaje a consola, por lo que todavía no debe visualizar nada.

5. Con un editor de texto plano debe añadirse al fichero Grupo-Apellido1-Apellido2-L6.s lo que falta para implementar un programa ensamblador equivalente al programa JavaScript `loteria.js`. Deberá añadirse lo siguiente:

- a) Añada a Grupo-Apellido1-Apellido2-L6.s el primer bloque no incluido (de instrucciones ensamblador) utilizando un editor de texto plano. Este bloque corresponde a la implementación de la instrucción `str = "\nHa ganado 10 Euros"`; que está dentro del bloque de la sentencia `if` delimitado entre llaves. Dicho bloque debe sustituir el primer comentario que empieza por `>>>>>` y que está resaltado en el listado anterior con un rectángulo en azul transparente.

Lase instrucciones de este bloque son similares a las del segundo bloque de instrucciones en el que se inicializó la variable para el caso de no haber premio, pero ahora deben guardar en la palabra de memoria asignada a la variable `str` el puntero al string que indica que se ha ganado el premio.


Una vez añadido este bloque de instrucciones, arranque el simulador ARMSim# y cargue (File>Load) el fichero Grupo-Apellido1-Apellido2-L6.s en el simulador. Si le diera algún error al cargarlo es porque, al modificar el programa, ha cometido algún error. Abra el editor de texto, corrija lo necesario, sávelo y vuelva a cargarlo en el simulador (File>Reload, o icono  en la barra de herramientas). Una vez cargado sin errores, compruebe que funciona correctamente ejecutándolo y viendo que realiza el salto solo cuando: `num < 64`.

- b) Añada el segundo bloque no incluido (de instrucciones ensamblador) utilizando un editor de texto plano. Este bloque corresponde a la implementación de la instrucción `console.log(str)`; que esta a continuación. Dicho bloque debe sustituir el segundo comentario que empieza por `>>>>>` y que está resaltado en el listado anterior con un rectángulo en azul transparente.

Este bloque debe enviar el string identificado por el puntero guardado en la variable `str` a consola utilizando la primitiva del S.O. `swi 0x69 (writeString)`. Esta primitiva del S.O. recibe 2 parámetros:

1. Debe recibir en el registro R0 el identificador del fichero donde escribir, que en este caso debe ser `stdout` para que el mensaje salga por consola.
2. Debe recibir en el registro R1 el puntero al string que debe escribir en el fichero identificado.

Las instrucciones de este bloque deberán cargar primero dichos valores en los registros R0 y R1, para invocar a continuación la primitiva con el código 0x69.

Una vez añadido este bloque de instrucciones, arranque el simulador ARMSim# y cargue (File>Load) el fichero Grupo-Apellido1-Apellido2-L6.s en el simulador. Si le diera algún error al cargarlo es porque, al modificar el programa, ha cometido algún error. Abra el editor de texto, corrija lo necesario, sávelo y vuelva a cargarlo en el simulador (File>Reload, o icono  en la barra de herramientas). Una vez cargado sin errores, compruebe que funciona correctamente ejecutándolo y viendo que envía el mensaje a consola que corresponde al valor de `num`, habrá premio solo si (`num < 64`).

Para comprobar si el mensaje ha salido por consola, se debe seleccionar `Stdin/Stdout/Stderr` en `OutputView`, de forma que se visualice la consola en la parte inferior de ARMSIM#. De esta forma podrá ver los mensajes que se envían a consola desde el programa. El bloque que acaba de implementar enviará uno de estos dos mensajes: `"\nHa ganado 10 Euros\n"` o `"\nLo sentimos, no hay premio\n"`.

- c) Añada el tercer bloque no incluido (de instrucciones ensamblador) utilizando un editor de texto plano. Este bloque corresponde a la implementación de la instrucción `console.log("num: " + num)`; que esta a continuación. Dicho bloque debe sustituir el tercer comentario que empieza por `>>>>>` y que está resaltado en el listado anterior con un rectángulo en azul transparente. Este bloque debe realizar 2 escrituras por consola.

La primera escritura debe enviar el string `"num: "` utilizando la primitiva del S.O. `swi 0x69 (writeString)`. Notese que el string no lleva el character `\n` (nueva línea al final para que cuando se escriba el número a continuación, este salga por consola en la misma línea. El string `"num: "` debe definirse sustituyendo el cuarto comentario que empieza por `>>>>>` y que está resaltado en el listado anterior con un

rectángulo en azul transparente.

La segunda escritura debe enviar a consola el valor del número aleatorio guardado en la variable `num` utilizando la primitiva del S.O. `swi 0x6b (writeNum)`, de forma que el mensaje muestre el número aleatorio generado en esta ejecución por consola.

Una vez añadido este bloque de instrucciones, depure el programa ya complete, igual que en los bloques anteriores.

Cuando se haya completado la implementación en ensamblador de las partes no incluidas, la ejecución del programa en el simulador ARMSim# deberá enviar a la consola del simulador los mismos mensajes que el programa JavaScript `loteria.js`, además de producir cambios de estado similares.

6. Una vez completado el programa en `Grupo-Apellido1-Apellido2-L6.s` debe cargarlo en el simulador ARMSim# y ejecutar el programa hasta el final. Además debe seleccionar la consola (`Stdin/Stdout/Stderr` en `OutputView`) para visualizar los mensajes que el programa ha enviado a `stdout`. Realice además la captura de la ventana de ARMSim# para incluirla en la entrega de esta práctica. La captura debe mostrar el contenido de los registros y la salida por consola, además del programa cargado.

La captura de la pantalla se puede realizar de la siguiente forma. Cuando muestre en el navegador la página creada con la calculadora, maximice la ventana del navegador para que ocupe toda la pantalla del PC y no haya otras ventanas sobre el navegador. Haga una captura de pantalla (pulse la tecla "Impr Pant" del teclado). La imagen está en la carpeta "Imágenes" de su cuenta. Puede acceder a ella en el menú: Lugares>Imágenes. Tendrá un nombre como "Screenshot from ...".png, en el que los puntos suspensivos representan la fecha y hora de la captura.

7. Al llegar a la sesión Lab6 deberá realizar las modificaciones específicas de su grupo y subir la entrega a moodle.



## Tablas con instrucciones y directivas que pueden ser de utilidad

(Aparecen en orden alfabético.)

| Instrucciones                 | Significado   |
|-------------------------------|---|
| <code>add rd, rf1, rf2</code> | Suma los contenidos de los registros <i>rf1</i> y <i>rf2</i> y guarda el resultado en <i>rd</i> .   |
| <code>b etiq</code>           | Bifurca a la instrucción que tiene la etiqueta <i>etiq</i> <sup>4</sup> .   |
| <code>bge etiq</code>         | Si como consecuencia de una operación anterior se cumple la condición mayor-o-igual (ge), bifurca a la instrucción de etiqueta <i>etiq</i> .  |
| <code>cmp rf, #val</code>     | Compara el contenido del registro <i>rf</i> con el valor <i>val</i> . Si son iguales, pone el indicador Z = 1 y, si no, pone Z = 0. Si el contenido del registro <i>rf</i> es menor que <i>val</i> , pone el indicador N = 1 y, si no, pone N=0.  |
| <code>ldr rd, =etiq</code>    | Carga en el registro <i>rd</i> la dirección de la memoria etiquetada con <i>etiq</i> .  |
| <code>mov rd, rf</code>       | Copia el contenido del registro <i>rf</i> en el registro <i>rd</i> .  |
| <code>mov rd, #val</code>     | Copia el valor <i>val</i> al registro <i>rd</i> .   |
| <code>ldr rf1, [rf2]</code>   | Lleva al registro <i>rf1</i> la palabra de memoria cuya dirección es el contenido de <i>rf2</i>   |
| <code>str rf1, [rf2]</code>   | Almacena el contenido del registro <i>rf1</i> en la palabra de memoria cuya dirección es el contenido de <i>rf2</i>   |
| <code>sub rd, rf1, rf2</code> | Guarda en el registro <i>rd</i> el resultado de restar los contenidos de los registros <i>rf1</i> y <i>rf2</i> .  |
| <code>swi 0x11</code>         | Llamada al sistema operativo que para la ejecución del programa   |
| <code>swi 0x69</code>         | Llamada al sistema operativo para escribir en un fichero una cadena de caracteres almacenada en memoria con la directiva <i>.asciz</i> . Espera recibir en el registro R0 el identificador del fichero y en R1 el puntero a la cadena. Las cadenas acaban siempre con un carácter nulo (0). |
| <code>swi 0x6b</code>         | Llamada al sistema operativo para escribir en un fichero un número. Espera recibir en el registro R0 el identificador del fichero y en R1 el número. El número se interpreta en complement a 2.   |
| <code>swi 0x6b</code>         | Llamada al sistema operativo para leer el reloj del sistema. Devuelve en R0 el tiempo en milisegundos. No espera recibir parámetros.  |

| Directivas                 | Significado   |
|----------------------------|---|
| <code>.ascii "... "</code> | Indica al ensamblador que cargue en memoria, en la zona de datos, la cadena de caracteres "... " <sup>5</sup> , justo detrás de los datos que se hayan definido previamente con directivas similares (o al principio de la zona de datos si no hay directivas previas). |
| <code>.asciz "... "</code> | Hace lo mismo que <i>.ascii</i> y además añade al final el carácter nulo (0)  |
| <code>.data</code>         | Indica que empieza la parte de definición de datos  |
| <code>.byte valores</code> | Indica al ensamblador que cargue en memoria, en la zona de datos, la lista de <i>valores</i> especificados (tipo byte), a continuación de los definidos previamente   |
| <code>.equ cte, val</code> | Establece la equivalencia entre el nombre de una constante <i>cte</i> <sup>6</sup> y su valor <i>val</i>  |
| <code>.end</code>          | Indica al ensamblador el fin del programa (incluido código y datos)   |
| <code>.skip val</code>     | Indica que debe reservarse memoria para almacenar un número <i>val</i> de bytes   |
| <code>.text</code>         | Indica al ensamblador que empieza la zona de código ejecutable  |

<sup>2</sup> *val* representa un valor en decimal (por defecto), o en hexadecimal si empieza por 0x.

<sup>3</sup> *rf* (registro fuente 1 o 2) y *rd* (registro destino) representan a alguno de los 16 registros (r0-r15).

<sup>4</sup> *etiq* debe interpretarse como una etiqueta del programa

<sup>5</sup> "... " representa una cadena de caracteres ASCII de cualquier longitud

<sup>6</sup> *cte* representa el nombre de una constante