Prácticas de ensamblador ARM

Este documento recoge el material necesario para la realización de las prácticas de programación en ensamblador de la asignatura Tecnología de los Computadores. A continuación, se presenta un índice con el contenido del documento.

1. Introducción y Objetivos

Con esta práctica se pretende que el alumno se familiarice con el funcionamiento a nivel ISA (Instruction Set Architecture) de un procesador. En concreto vamos a utilizar un procesador basado en la arquitectura ARM. Para ello al alumno se le presentarán una serie de códigos en ensamblador para que los entienda estudiando su ejecución en un hardware real o en un simulador. Posteriormente el alumno deberá desarrollar una serie de programas en ensamblador para la resolución de los problemas propuestos.

Los objetivos a alcanzar son:

- Comprender el funcionamiento básico de un procesador ARM
- Familiarizarse con el nivel ISA del procesador ARM
- Introducción a la programación en ensamblador

2. Entornos de programación y ejecución

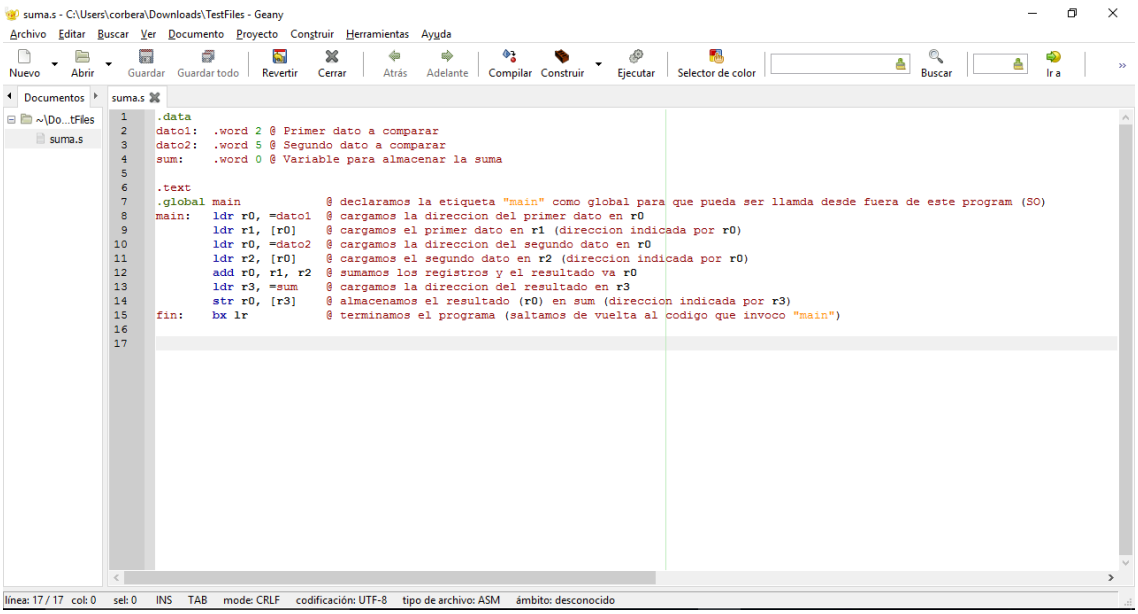
Para la programación y la ejecución de programas escritos en ensamblador ARM vamos a utilizar dos entornos:

1. Un entorno hardware real: Raspberry Pi
2. Un simulador ARM: ARMSim#

Programación

La programación en ambos entornos se realiza de la misma manera, consistirá en la edición del código fuente ensamblador con un editor de texto. Si estamos usando Windows como sistema operativo en el que desarrollar (PCs de los laboratorios), podemos usar el editor "scite". Si usamos Linux (Raspberry Pi) se recomienda el editor *geany,* aunque en cualquiera de los dos casos, cualquier otro editor de texto nos puede valer.

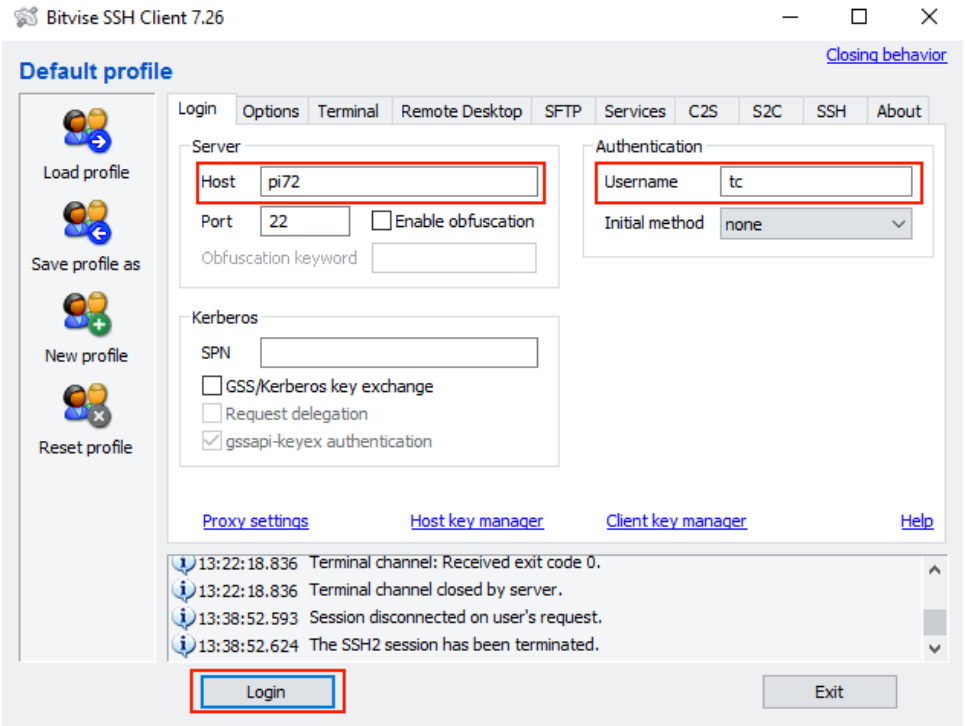
Una vez creado el código fuente con el editor se grabará en un fichero con extensión *".s"* que luego tendrá que ser ensamblado para poder ser ejecutado/simulado.



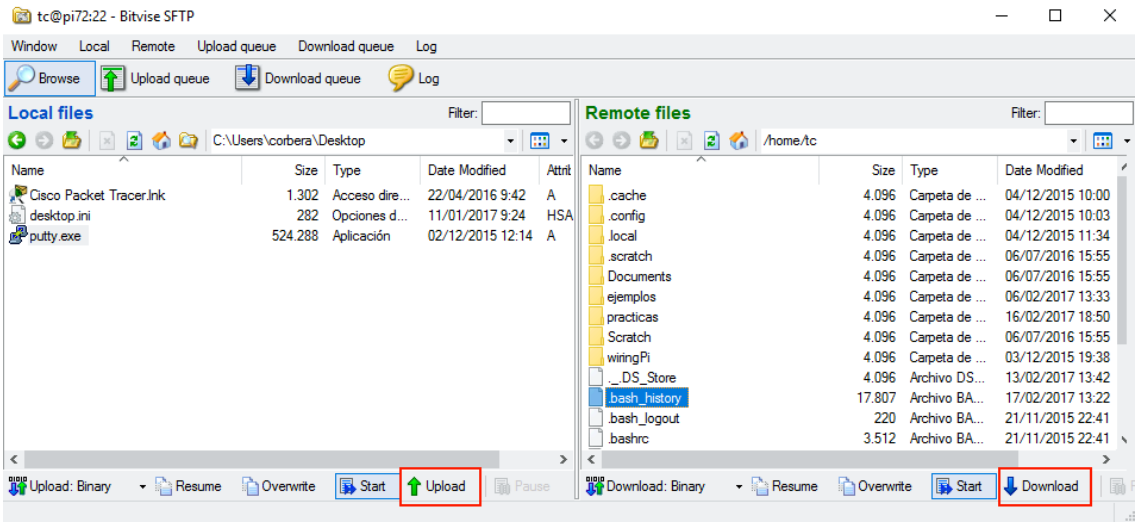
Ejecución en una Raspberry Pi

Para ejecutar el código ensamblador creado, primero habrá que ensamblarlo en la propia Raspberry Pi (rPi). Para ello, el fichero *".s"* deberá estar en la rPi. Para transferirlo desde un PC con Windows, utilizaremos el programa "Bitvise" que es un cliente SSH que nos permitirá conectarnos a la rPi, tener un terminal abierto con ella para ejecutar comandos y además una ventana para la transferencia de archivos desde el PC a la rPi.

En la ventana principal de "Bitvise", solo tendremos que poner el nombre de la rPi en el campo "Host" y el nombre de usuario, en nuestro caso, "tc", en el campo "Username", como se puede apreciar en la siguiente imagen.



Una vez rellenados esos campos, tras pulsar el botón de "login", y meter el password de la cuenta, se nos abrirán dos ventanas. Un intérprete de comandos donde podremos dar las órdenes para ensamblar y ejecutar el programa, y una ventana para la transferencia de ficheros como la que se puede apreciar en la siguiente imagen:



A la izquierda aparece el contenido del directorio actual del PC y a la derecha el de la rPi. Una vez seleccionados los directorios y ficheros adecuados, podremos transferirlos del PC a la rPi (pulsando en "Upload") o viceversa (pulsando en "Download").

Una vez el fichero fuente (.s) se encuentre en la rPi, para ensamblar, enlazar y ejecutar el código, habrá que ejecutar los comandos adecuados en la ventana del terminal.

Los comandos para ensamblar, enlazar y ejecutar nuestro programa ensamblador son tres (que hay que teclear en la ventana del terminal):

1. **Ensamblado**: Con este paso generaremos un fichero objeto *".o"* a partir del fichero con el programa fuente *".s"*. El comando utilizado para ello es *"as"*:

   ```
   as –g programa.s –o programa.o
   ```

2. **Enlazado**: En este paso se generará el fichero ejecutable a partir de los ficheros objeto creados en el paso anterior, junto con las librerías que se utilicen dentro de los mismos. Para ello utilizaremos el comando *"gcc"*:

   ```
   gcc –g programa.o –o ejecutable
   ```

3. **Ejecución**: Tan solo nos queda invocar el programa para que sea ejecutado en la rPi:

   ```
   ./ejecutable
   ```

Si el programa que hemos escrito no utiliza la salida estándar (no imprime nada por pantalla), al ejecutar el programa parecerá que no ha ocurrido nada. Podremos ejecutar el programa paso a paso para depurarlo con el comando *"gdb"* cuyo uso aparece en los apéndices de este documento.

La imagen siguiente muestra la ventana del terminal en la cual se ha seguido todo el proceso de compilado y ejecución en una rPi a la que nos hemos conectado mediante SSH.

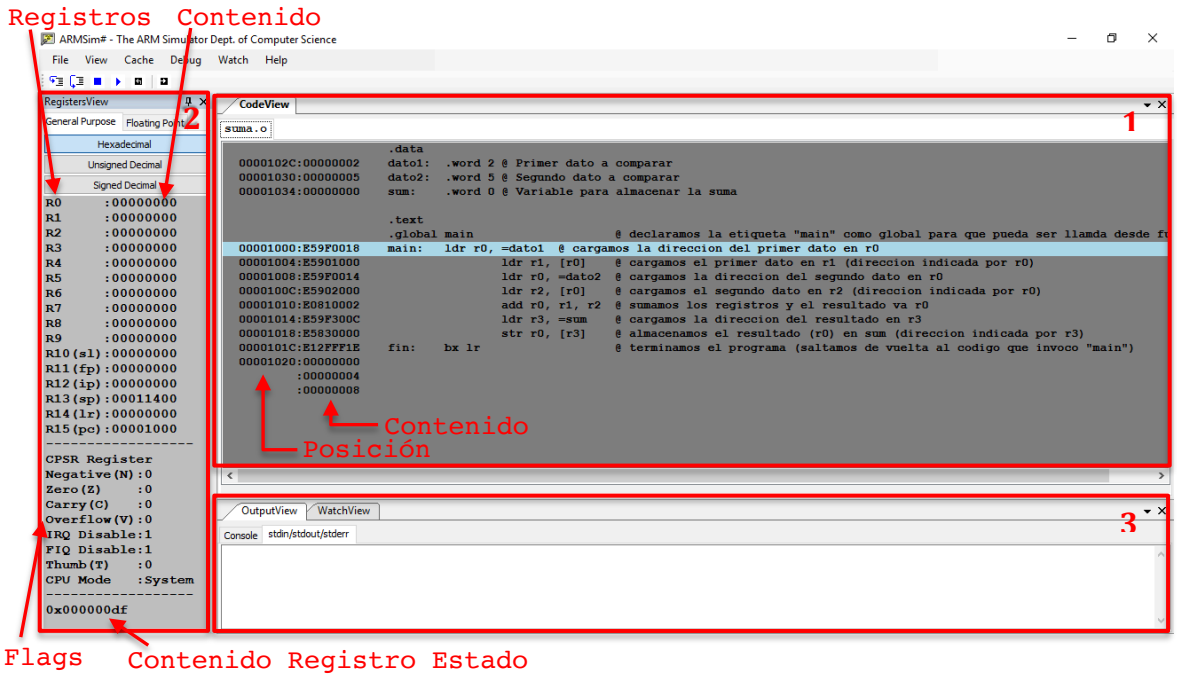
```
tc@raspberrypi:~/practicas $ as -g suma.s -o suma.o
tc@raspberrypi:~/practicas $ gcc -g suma.o -o suma
tc@raspberrypi:~/practicas $ ./suma
tc@raspberrypi:~/practicas $ echo $?
7
tc@raspberrypi:~/practicas $ gdb suma
GNU gdb (Raspbian 7.7.1+dfsg-5) 7.7.1
Copyright (C) 2014 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "arm-linux-gnueabihf".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from suma...done.
(gdb) list
1       .data
2       dato1:  .word 2 @ Primer dato a comparar
3       dato2:  .word 5 @ Segundo dato a comparar
4       sum:    .word 0 @ Variable para almacenar la suma
5
6       .text
7       .global main                    @ declaramos la etiqueta "main" como global para que pueda ser l
lamda desde fuera de este program (S0)
8       main:   ldr r0, =dato1  @ cargamos la direccion del primer dato en r0
9                       ldr r1, [r0]    @ cargamos el primer dato en r1 (direccion indicada por r0)
10                      ldr r0, =dato2  @ cargamos la direccion del segundo dato en r0
(gdb)
11                      ldr r2, [r0]    @ cargamos el segundo dato en r2 (direccion indicada por r0)
12                      add r0, r1, r2  @ sumamos los registros y el resultado va r0
13                      ldr r3, =sum    @ cargamos la direccion del resultado en r3
14                      str r0, [r3]    @ almacenamos el resultado (r0) en sum (direccion indicada por r
3)
15      fin:    bx lr                   @ terminamos el programa (saltamos de vuelta al codigo que invoc
o "main")
16
17
(gdb)
Line number 18 out of range; suma.s has 17 lines.
(gdb)
```

Ejecución en el simulador ARMSim#

También podemos utilizar un simulador de una plataforma ARM si no disponemos de una rPi. Como ya hemos comentado la edición del programa fuente no cambia con respecto a la ejecución en una rPi. Por tanto los ficheros de entrada del simulador serán los ficheros fuente *".s"* con el programa.

Las indicaciones para poder descargar e instalar el simulador ARMSim# os serán proporcionadas por vuestro profesor.

En la siguiente figura se muestra el aspecto del simulador una vez cargado un programa ensamblador:



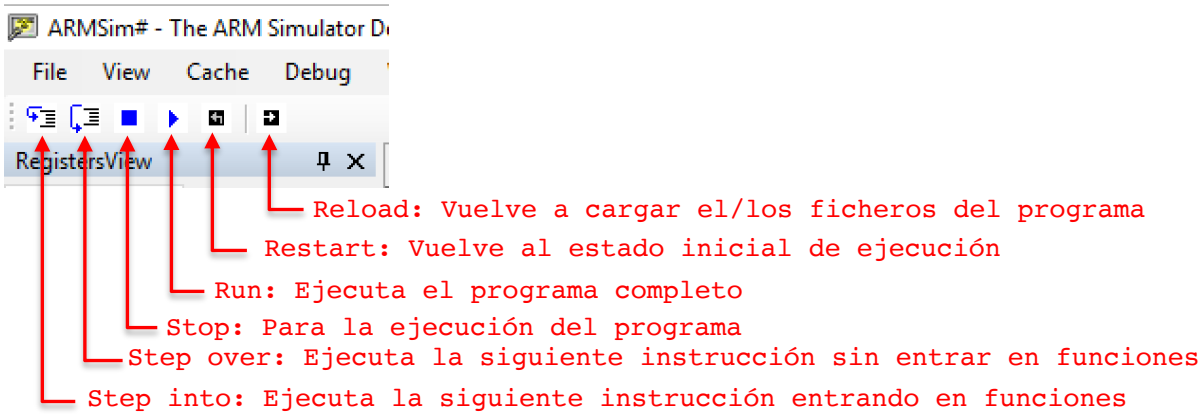
Se puede observar que aunque se le indique al simulador que cargue el fichero fuente "suma.s", realmente lo que ha hecho ha sido compilar dicho fichero y generar el fichero "suma.o" que es el que aparece en el panel *"CodeView"*.

En esa imagen aparecen tres paneles:

1. **CodeView**: En este aparece el código del programa (programa + datos). Además de aparecer el código fuente tal y como se tecleó en el fichero *".s"* (a la derecha del panel), también aparecen las posiciones de memoria y los valores introducidos en dichas posiciones de memoria para cada dato/instrucción (todo en hexadecimal, a la izquierda del panel).
2. **RegistersView**: Este panel que aparece a la izquierda de la ventana del simulador, muestra el contenido (a seleccionar entre hexadecimal, decimal con o sin signo) de todos los registros del procesador (r0-r15 y el registro de estado, mostrando por separado el valor de los flags N, Z, C y V almacenados en dicho registro).
3. **OutputView:** En este panel se podrá ver la salida del programa, si es que este imprime algo por ella (salida estándar). En este panel, también se puede seleccionar la pestaña **WatchView** que nos servirá para poder ver el contenido de variables de memoria mientras se ejecuta el programa.

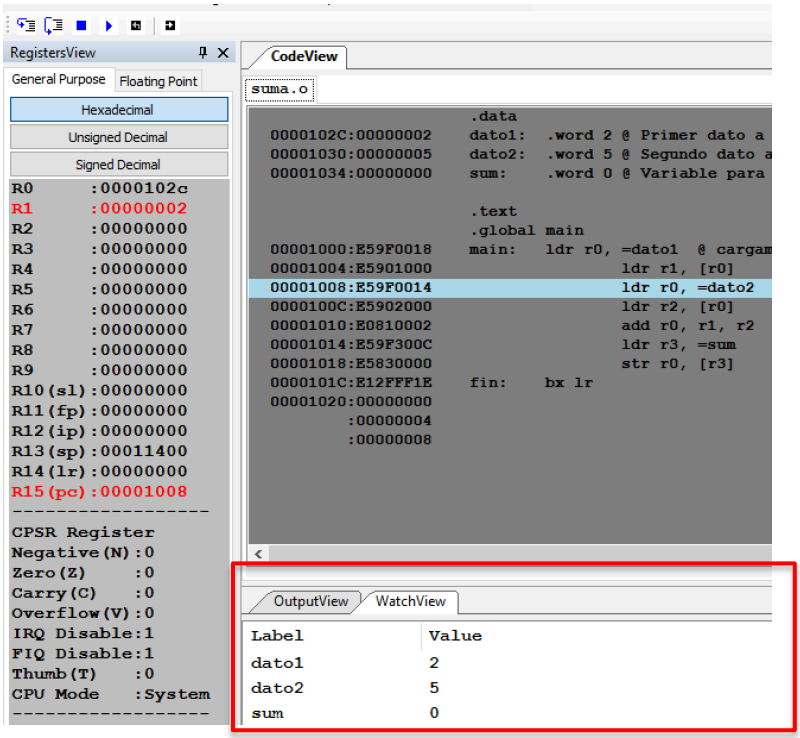
Si no ha habido ningún problema a la hora de compilar y cargar el programa, el simulador nos permitirá ejecutar el código cargado. Podremos simular una ejecución completa del código o simular una ejecución "paso a paso" del mismo, para poder ver la evolución de los registros o de las variables de memoria en cada instrucción del programa.

En la parte superior del panel *"RegisterView"* encontramos una barra de herramientas con acceso a las funciones principales para llevar a cabo estas tareas. En la siguiente figura aparecen cada uno de ellos:



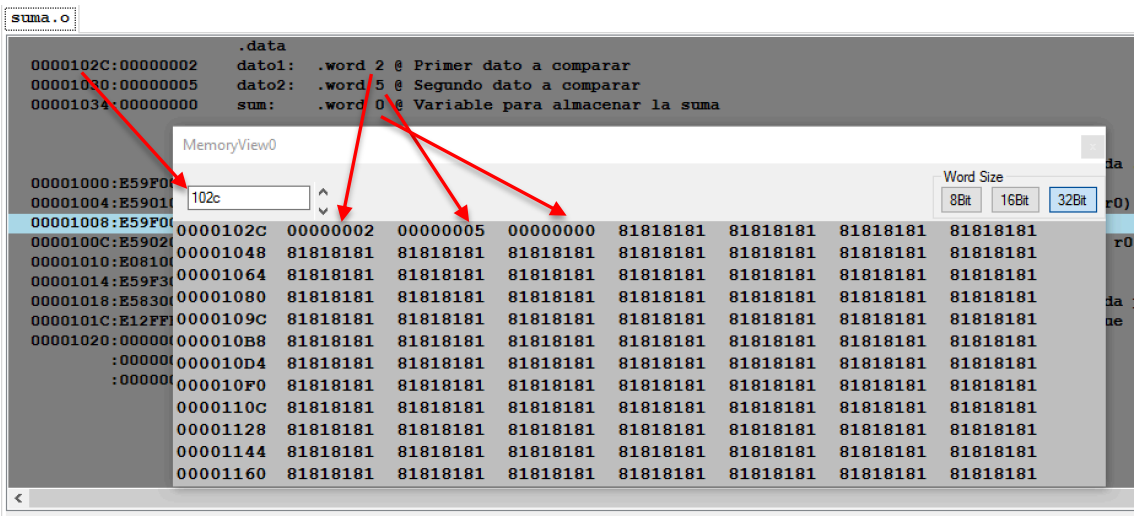
Con los botones "*Step into*" y "*Step over*" ejecutamos instrucción a instrucción el código (entrando y sin entrar en las llamadas a función, respectivamente).

En la siguiente figura se puede ver el estado del simulador tras ejecutar paso a paso ("*Step into*") las dos primeras instrucciones. La instrucción resaltada en celeste es la próxima a ejecutar, y en la vista de los registros, en color rojo aparecen aquellos registros modificados por la ejecución de la última instrucción (en este caso, la última instrucción ha cargado el valor de "*dato1*" en el registro "*r1*").



También podemos observar como en el panel inferior hemos seleccionado la pestaña "*WatchView*" y en ella hemos introducido 3 variables (*labels*) a observar. Para añadir una variable a dicho panel, pinchando con el botón derecho del ratón seleccionamos la opción "*add watch*" y en la nueva ventana que se abre seleccionamos el fichero al que pertenece dicho "*label*", luego seleccionamos el "*label*" y por último el formato de visualización que queremos (byte, word, string, ...). En ese panel podremos ir viendo la evolución del contenido de esas variables conforme se van ejecutando las instrucciones.

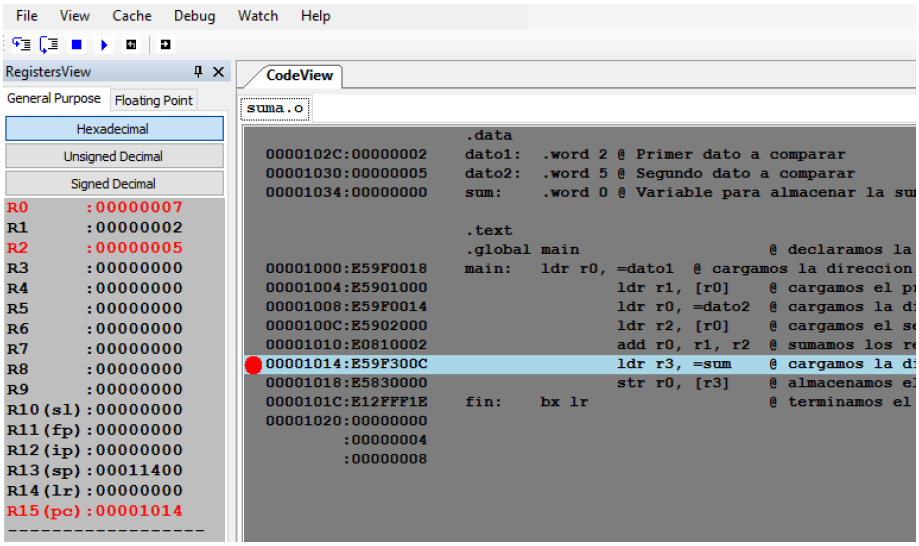
También se puede observar el contenido de la memoria directamente. Para ello del menú "*View*" seleccionamos la opción "*Memory*", lo que nos abrirá una nueva ventana como la mostrada en la siguiente imagen. En ella bastará con indicar la primera posición de memoria que queremos observar (en el ejemplo se selecciona la posición *102c* ya que, si nos fijamos en el panel principal, es la dirección de comienzo del segmento de datos) y si queremos ver las posiciones de memoria agrupadas por byte (8Bit), hword (16Bit) o Word (32Bit).



Breakpoints

Si el código es muy largo o tiene bucles con muchas iteraciones, quizás la ejecución paso a paso sea muy tediosa hasta llegar a la instrucción que nos interesa. Para poder llegar a dicha instrucción de una forma más rápida, saltándonos todas las anteriores, se puede establecer un "*breakpoint*" en dicha instrucción.

Basta con hacer doble click con el ratón sobre la instrucción en cuestión y aparecerá un punto rojo a la izquierda de la dirección de memoria de dicha instrucción. Si ahora le damos a ejecutar el programa (ejecución completa, no paso a paso), el programa empezará a ejecutarse y no parará hasta que termine o llegue a la instrucción marcada con el "*breakpoint*". A partir de ese momento podemos continuar con la ejecución paso a paso a paso de las instrucciones que nos interesan.



3. Programación en ensamblador ARM

Para programar en ensamblador ARM además de conocer las instrucciones disponibles (tanto su semántica como su gramática), se tienen que conocer una serie de reglas sintácticas y léxicas que debe respetar todo fichero con un programa en ensamblador para que pueda ser entendido por el ensamblador.

Para programar en ensamblador de una manera cómoda y sencilla se tienen que utilizar etiquetas y directivas para el compilador.

Las **etiquetas** identifican una instrucción o dato, su valor es la posición que ocupa en memoria dicha instrucción o dato.

Las **directivas** sirven para dar estructura al programa (`.data, .text`) y para definir tipos de datos (`.byte, .word, .ascii, …).`

La estructura de un programa en ensamblador consta, al menos, de dos bloques: i) el segmento de datos, identificado con la directiva `.data`, donde se definen las variables de memoria del programa, y ii) el segmento de código, identificado con la directiva `.text`, donde aparecen una serie de instrucciones consecutivas que forman el código del programa.

En el apartado `.data,` en la declaración de variables cada línea está formada por:

- una etiqueta (seguida de ":") (opcional)
- una directiva de definición de datos
- los datos (opcional)
- comentarios (comenzando por "@") (opcional)

Por ejemplo:

```
.data
micadena: .ascii "cadena ejemplo" @ caracteres
palabra:  .word 50            @ esto ocupa 4 bytes
vectorW:  .word 2,6,9,1       @ 4 posiciones de 4 bytes consecutivas
bloque:   .space 30           @ 30 bytes no inicializados
```

En la siguiente tabla se presentan algunas de las directivas más comunes para la reserva de espacio e inicialización de variables de memoria en el segmento de datos:

Directiva Ensamblador	Descripción
`.ascii "<string>"`	Inserta la cadena "string" como datos en memoria
`.asciz "<string>"`	Igual que .ascii pero añade un byte 0 al final de la cadena "string" en memoria
`.byte <byte1> {,<byte2>} …`	Inserta una lista de valores de tamaño byte
`.word <word1> {,<word2>} …`	Inserta una lista de valores de tamaño word (32 bits)
`.space <num_bytes> {,<fill_byte>}`	Reserva el número de bytes indicado por "num_bytes". Se inicializan a cero o a "fill_byte" si es especificado

En el apartado `.text`, se escriben las instrucciones que forman el programa. Cada línea está formada por:

- una etiqueta (opcional)
- un mnemónico (identifica la instrucción)
- operandos
- comentarios (si existen comienza por '@')

Por ejemplo:

```
main: mov r1, r2  @cargamos el contenido del registro r2 en r1
```

A continuación, tenemos el código completo de un programa que suma dos variables (dato1 y dato2) y el resultado lo almacena en otra variable (sum):

```
.data
dato1:   .word 2   @ Primer dato a comparar
dato2:   .word 5   @ Segundo dato a comparar
sum:     .word 0   @ Variable para almacenar la suma

.text
.global main              @ declaramos la etiqueta "main" como global
                          @ para que pueda ser llamada desde fuera de
                          @ este program (SO)
main: ldr r0, =dato1      @ cargamos la direccion del primer dato en r0
      ldr r1, [r0]        @ cargamos dato1 en r1 (dir indicada por r0)
      ldr r0, =dato2      @ cargamos la direccion del segundo dato en r0
      ldr r2, [r0]        @ cargamos dato2 en r2 (dir indicada por r0)
      add r0, r1, r2      @ sumamos los registros y el resultado va a r0
      ldr r3, =sum        @ cargamos la direccion del resultado en r3
      str r0, [r3]        @ almacenamos el resultado (r0) en sum
                          @ (direccion indicada por r3)
fin:  bx lr               @ terminamos el programa (saltamos de vuelta al
                          @ codigo que invoco "main")
```

4. Ejercicios de laboratorio

Ejercicio 1.

Carga el programa *suma.s* en el simulador *ARMSim#.* Inspecciona el segmento del texto y de los datos. Contesta a las siguientes cuestiones:

1. ¿En qué posición de memoria se ha cargado la primera instrucción de tu programa (la etiquetada con `main`)?

`main`	

2. ¿Qué posición de memoria se le ha asociado a las variables?

`dato1`	
`dato2`	
`sum`	

3. ¿Qué valor carga en *r0* la instrucción `ldr r0,=dato1`? ¿Qué representa ese valor?

Ejecuta el programa completo y comprueba que en la variable `sum` se ha cargado el correspondiente valor (suma de `dato1` y `dato2`).

Vuelve a ejecutar el programa, pero ahora paso a paso y ve comprobando la correcta ejecución de todas las instrucciones que lo componen (mirando cómo cambia el contenido de los registros y/o memoria implicados en cada una). Cambia el valor de las variables y ejecuta el código para ver los nuevos resultados.

Ejercicio 2.

Carga el programa *maximo.s* en el simulador. Este código va a escribir en la variable *max* el valor máximo contenido en las variables *dato1* y *dato2*. Para entender el código deberás repasar el significado y funcionamiento de las instrucciones *cmp, bgt* y *b.*

Ejecuta el programa paso a paso y observa cómo se comporta el código para conseguir su objetivo. Cambia el valor de los datos para forzar que se comporte de manera distinta.

El programa *maximo2.s* realiza la misma función que el anterior, pero utilizando instrucciones predicadas en vez de saltos (*movgt, movle*). Cárgalo en el simulador, ejecútalo paso a paso (cambiando los valores de los datos) y comprueba que el resultado es correcto. Presta especial atención al funcionamiento de las

instrucciones predicadas e intenta comprender porque no necesitamos utilizar saltos en esta versión.

Ejercicio 3.

Carga el programa *sumaVector.s* que como su nombre indica realiza la suma de todos los valores almacenados en un vector. La variable *tam* indica el tamaño del vector, *datos* contiene los valores del vector y *res* contendrá el resultado de la suma.

A continuación, aparece el código fuente. Comenta al lado de cada instrucción qué sentido tiene dicha instrucción en el código (para conseguir el objetivo de sumar todos los valores de un vector):

```
.data
tam:   .word 8
datos: .word 2, 4, 6, 8, -2 -4, -6 -7
res:   .word 0

.text
.global main
main: ldr r0, =tam
      ldr r1, [r0]
      ldr r2, =datos
      mov r3, #0
loop: cmp r1, #0
      beq sal
      ldr r4, [r2], #4
      add r3, r3, r4
      sub r1, r1, #1
      b loop
sal:  ldr r0, =res
      str r3, [r0]
      mov pc, lr
```

Ejecuta el código y comprueba su correcto funcionamiento. Cambia los valores del vector (tamaño incluido) y comprueba que sigue funcionando correctamente. Fíjate especialmente en aquellas instrucciones necesarias para el control del bucle.

En cuanto a la implementación del bucle, ¿Tú lo habrías hecho de la misma forma? ¿Se te ocurre otra forma de hacerlo?

Ejercicio 4.

Copia el código anterior con el nombre *maximoVector.s* y modifícalo para que en vez de sumar los elementos del vector *datos*, lo que almacene en *res* sea el valor máximo encontrado en dicho vector.

Ejecuta el código que has creado y comprueba su correcto funcionamiento. Haz varias pruebas cambiando el tamaño y los valores almacenados en el vector (por ejemplo, tamaño 1, todos los elementos negativos, …).

Ejercicio 5.

Escribe una función en ensamblador ARM que calcule el termino "n" de la sucesión de Fibonacci ("n" será un parámetro de entrada a la función).

Los números de Fibonacci $f_0, f_1, f_2, f_3, \ldots$, quedan definidos por las ecuaciones:

$$f_0 = 0$$

$$f_1 = 1$$

$$f_n = f_{n-1} + f_{n-2} \text{ para } n = 2, 3, 4, 5, \ldots$$

Requisitos:

- El fichero ensamblador se tiene que llamar "**fibonacci.s**"
- La función que calcula el término "**n**" de la sucesión se debe llamar "**fib**" y solo debe contener esta función (ni programa main, ni variables globales, ...)
- La función "**fib**" tomará un parámetro de entrada "**n**" que se le pasará por el registro **r0** (también llamado **a1**) y devolverá el resultado por el registro **r0**

Ejercicio 6.

Escribe una función en ensamblador ARM que dado un vector de números naturales devuelva el número de valores pares que hay almacenados en el vector.

Por ejemplo, si el vector que se le pasa a la función es **X**=[1, 2, 5, 7, 8], la función devolverá un 2 que es el número de valores pares que hay en el vector.

Requisitos:

- El fichero ensamblador se tiene que llamar "**eje6.s**"
- La función a crear debe llamarse "**evencount**"
- La función "**evencount**" debe aceptar como parámetros de entrada la dirección del vector que se le pasará por el registro **r0** y la longitud de dicho vector que se le pasará por el registro **r1.** Como resultado devolverá el número de valores pares del vector por el registro **r0**
- La función "**evencount**" debe llamar a una función "**iseven**", la cual acepta el número a chequear por el registro **r0** y devuelve por el registro **r0** un 1 si es par y un 0 si no lo es (en GUAC tenéis un *ejemplo de implementación* de una función "iseven")
- La función "**evencount**" **DEBERÁ** cumplir el convenio de llamada a funciones

Ejercicio Extra.

En este ejercicio se debe realizar una función **recursiva** que dado como parámetro de entrada un número natural "**n**", devuelva la suma de los cuadrados de los números desde **0** hasta **n**, es decir:

$$res = \sum_{i=0}^{n} i^2$$

La idea recursiva consiste en darse cuenta que dado un valor "n", basta con saber la suma de los cuadrados de "n-1" y a eso sumarle n^2.

Requisitos:

- El fichero ensamblador se tiene que llamar "**cuadrados.s**"
- La función que calcula la suma de los "**n**" cuadrados se debe llamar "**cuad**"
- La función "**cuad**" tomará un parámetro de entrada "**n**" que se le pasará por el registro **a1** y devolverá el resultado por el registro **a1**

Apéndices

Principales Características de la arquitectura ARM

Registros

En el modo usuario de la arquitectura ARM que vamos a utilizar en las prácticas, disponemos de 17 registros de 32 bits, cuyo nombre y uso aparecen en la siguiente tabla:

Registro	Sinónimo	Especial	Papel en el estándar de llamada a procedimiento
		CPSR	Current Program Status Register
r15		PC	Program Counter
r14		LR	Link Register
r13		SP	Stack Pointer
r12		IP	Intra-Procedure-call scratch register
r11	v8		Variable register 8
r10	v7		Variable register 7
r9	v6		Platform register (meaning defined by platform)
r8	v5		Variable register 5
r7	v4		Variable register 4
r6	v3		Variable register 3
r5	v2		Variable register 2
r4	v1		Variable register 1
r3	a4		Argument / scratch register 4
r2	a3		Argument / scratch register 3
r1	a2		Argument / result / scratch register 2
r0	a1		Argument / result / scratch register 1

Registros de uso general: Los registros de r0 a r11 se pueden usar de forma genérica para la programación en ensamblador, siempre siguiendo los criterios de llamada a función (AAPCS) que aparecen más adelante.

Registros especiales: Los siguientes registros tienen un uso especial:

- **PC (r15), Program counter**: contiene la dirección de memoria de la instrucción a ejecutar. Se incrementa en 4 para ejecutar la siguiente instrucción, excepto en las instrucciones de salto que escriben un valor distinto en él.
- **LR (r14), Link register**: en este registro la instrucción de llamada a función (*bl*) almacenará la dirección de retorno (la siguiente a la instrucción *bl*), y nos servirá para retornar de las funciones.
- **SP (r13), Stack pointer**: este registro contiene la dirección de memoria donde está el tope de la pila, usada para las llamadas a función.
- **CPSR (r16), Current program status register**: o registro de estado, que contiene el valor de los flags de estado N, Z, C y V.

Flags en el registro de estado	Significado
N	Negativo: 1 si el resultado de la operación ha sido negativo
Z	Cero: 1 si el resultado de la operación ha sido 0
C	Carry: 1 si la operación ha generado acarreo
V	Overflow: 1 si el resultado de la operación no se puede representar en complemento a 2 con 32 bits

Cuando se hace una llamada a función, el paso de parámetros sigue las siguientes reglas (estándar AAPCS):

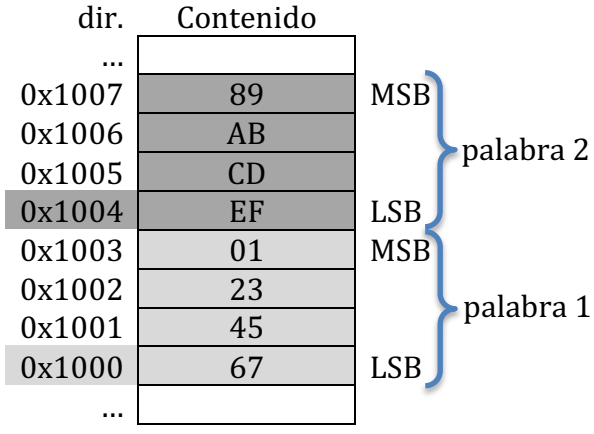
- Paso de parámetros: los 4 primeros parámetros se pasan por a1, a2, a3 y a4, el resto, si los hubiera por pila.
- Resultado del procedimiento: Se devuelve por registro a1 (si el resultado es de 32 bits) o por a1 y a2 (si el resultado es de 64 bits).
- No se preservan a1, a2, a3 y a4 en la llamada a función (se pueden sobrescribir libremente dentro de la función).
- Se preservan en la llamada a función de v1 a v8 (estos registros a la salida de la función deben conservar el mismo valor que tenían a la entrada).

Memoria

Organización de la memoria:

- Se organiza en palabras de 4 bytes (32 bits).
- Direccionamiento a nivel de byte (por defecto es Little-endian):
  - Dos palabras de 32 bits consecutivas en memoria están separadas por 4 posiciones. Así por ejemplo si la primera de dos instrucciones consecutivas (que ocupan cada una 1 palabra) estuviera en la posición de memoria $100_{(16}$ la siguiente instrucción estaría en la posición $104_{(16}$.
  - Todas las instrucciones ARM están en direcciones múltiplo de 4 (los dos bits menos significativos serán 00).
  - Las instrucciones ARM deben estar en posiciones alineadas: PC (32 bits) se incrementará de 4 en 4.
  - Los datos de 32 bits, en el modo Little-endian, el byte menos significativo (LSB, Least significant byte) se almacena en la posición más baja, y el byte más significativo (MSB, Most significant byte) en la más alta, como se aprecia en la siguiente figura.

Ejemplo: vista de la memoria donde se almacena dos palabras de 32 bits (.word). La primera palabra se almacena en la posición 0x1000 y contiene el valor 0x01234567, y la segunda en la posición 0x1004 y contiene el valor 0x89ABCDEF



AAPCS (Procedure Call Standard for the ARM Architecture)

Registro	Sinónimo	Especial	Preservado?	Papel en el estándar de llamada a procedimiento
r16		CPSR	No	Current Program Status Register
r15		PC	No	Program Counter
r14		LR	Si	Link Register
r13		SP	Si	Stack Pointer
r12		IP	No	Intra-Procedure-call scratch register
r11	v8		Si	Variable register 8
r10	v7		Si	Variable register 7
r9	v6		Si	Platform register (meaning defined by platform)
r8	v5		Si	Variable register 5
r7	v4		Si	Variable register 4
r6	v3		Si	Variable register 3
r5	v2		Si	Variable register 2
r4	v1		Si	Variable register 1
r3	a4		No	Argument / scratch register 4
r2	a3		No	Argument / scratch register 3
r1	a2		No	Argument / result / scratch register 2
r0	a1		No	Argument / result / scratch register 1

Registro de estado y ejecución condicional

Flags	Significado
N	Negativo: 1 si el resultado de la operación ha sido negativo
Z	Cero: 1 si el resultado de la operación ha sido 0
C	Carry: 1 si la operación ha generado acarreo
V	Overflow: 1 si el resultado de la operación ha producido desbordamiento

Condición{cond}	Ejecución condicional
EQ	(equal) When Z is enabled (Z is 1)
NE	(not equal). When Z is disabled. (Z is 0)
GE	(greater or equal than, in two's complement). When both V and N are enabled or disabled (V is N)
LT	(lower than, in two's complement). This is the opposite of GE, so when V and N are not both enabled or disabled (V is not N)
GT	(greather than, in two's complement). When Z is disabled and N and V are both enabled or disabled (Z is 0, N is V)
LE	(lower or equal than, in two's complement). When Z is enabled or if not that, N and V are both enabled or disabled (Z is 1. If Z is not 1 then N is V)
MI	(minus/negative) When N is enabled (N is 1)
PL	(plus/positive or zero) When N is disabled (N is 0)
VS	(overflow set) When V is enabled (V is 1)
VC	(overflow clear) When V is disabled (V is 0)
HI	(higher) When C is enabled and Z is disabled (C is 1 and Z is 0)
LS	(lower or same) When C is disabled or Z is enabled (C is 0 or Z is 1) CS/HS (carry set/higher or same) When C is enabled (C is 1)
CS/HS	(carry set/higher or same) When C is enabled (C is 1)
CC/LO	(carry clear/lower) When C is disabled (C is 0)

Escribir registro estado	cmp inst{s}: add**s**, sub**s**, and**s**, …
Salto (b) condicional	b{cond}: b**eq**, b**ne**, b**gt**, b**le** …
Ejecución condicional	inst{cond}: add**eq**, sub**ne**, ldr**gt**, …

Instrucciones ensamblador ARM

Instrucciones aritméticas y lógicas:

`Op{cond}{s} Rd, Rn, Oper2`

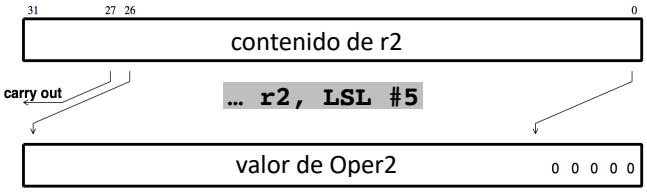
Op	Operación	Significado
add	Suma	Rd ← Rn + Oper2
sub	Resta	Rd ← Rn – Oper2
rsb	Resta inversa	Rd ← Oper2 – Rn
adc	Suma con acarreo	Rd ← Rn + Oper2 + C
sbc	Resta con acarreo	Rd ← Rn – Oper2 + C – 1
rsc	Resta inversa con acarreo	Rd ← Oper2 – Rn + C – 1
and	And lógico bit a bit	Rd ← Rn AND Oper2
orr	Or lógico bit a bit	Rd ← Rn OR Oper2
eor	Or exclusivo bit a bit	Rd ← Rn EOR Oper2
bic	Bit clear	Rd ← Rn AND NOT Oper2

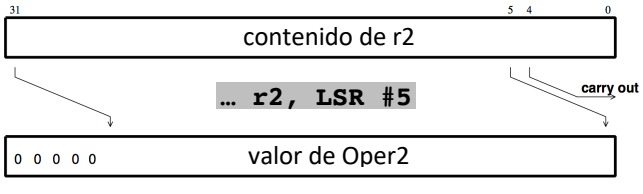
Ejemplos		
`add r0, r1, r2`	`r0 ← r1 + r2`	
`add r0, r1, #4`	`r0 ← r1 + 4`	
`add r0, r1, r2, LSL #2`	`r0 ← r1 + r2 << 2`	r1 + el resultado de desplazar a la izquierda 2 bits r2 (r2 x 4)
`add r0, r1, r2, LSR r3`	`r0 ← r1 + r2 >> r3`	r1 + el resultado de desplazar a la derecha r2, los bits indicados en r3

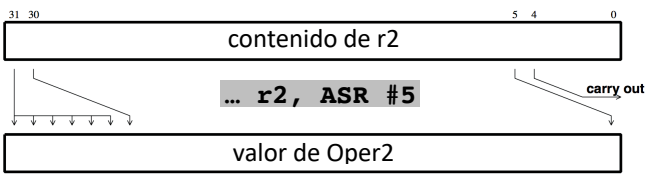
{cond}		
cualquiera de las condiciones del cuadro de ejecución condicional (eq, ne, ge, …)	`addeq r1, r2, r3` `addne r1, r2, r3`	si Z = 1 entonces r1 ← r2 + r3 si Z = 0 entonces r1 ← r2 + r3

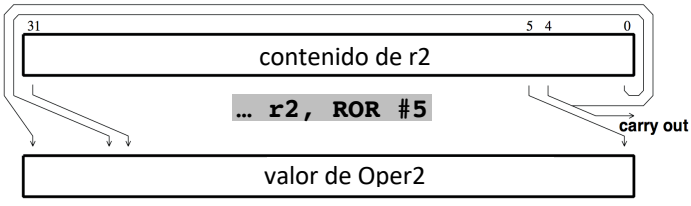
{s}		
guarda flags (N,Z,C,V) de la operación en CPSR	`adds r1, r2, r3`	r1 ← r2 + r3; CPSR ← flags NZCV de la suma

Desplazamientos Oper2		
LSL	Desplazamiento lógico a la izquierda (Logical Shift Left)	… r2, LSL #5
LSR	Desplazamiento lógico a la derecha (Logical Shift Right)	… r2, LSR #5
ASR	Desplazamiento aritmético a la derecha (Arithmetic Shift Right)	… r2, ASR #5
ROR	Rotación a la derecha (Rotate Right)	… r2, ROR #5









Instrucciones de transferencia:

Op{cond}{s} Rd, Oper2

Op	Operación	Significado
mov	Mover	Rd ← Oper2
mvn	Mover negado	Rd ← NOT Oper2

Ejemplos		
mov r0, r1	r0 ← r1	
mov r0, #4	r0 ← 4	
mov r0, r1, LSL #3	r0 ← r1 << 2	r1 desplazado a la izquierda 2 bits (r1 x 8)
mov r0, r1, LSR r3	r0 ← r1 >> r3	r1 desplazado a la derecha los bits indicados en r3

Nota: Para **{cond}, {s}** y deplazamientos del operando **Oper2**, ver las tablas presentadas en las instrucciones aritméticas y lógicas.

Instrucciones de salto:

b{cond} label
bl{cond} label
bx{cond} Rm

Op	Operación	Significado
b	Salto	PC ← label
bl	Salto y enlace	LR ← PC + 4; PC ← label
bx	Salto a registro	PC ← Rm

Ejemplos		
b label	PC ← label	La siguiente instrucción a ejecutar después de ésta, será la almacenada en la posición indicada por la etiqueta "*label*"
bl funcion	LR ← PC + 4 PC ← funcion	Se almacena en el registro LR la dirección de la instrucción siguiente a este salto. La siguiente instrucción a ejecutar después del salto, será la almacenada en la posición indicada por la etiqueta "*funcion*"
bx lr	PC ← LR	La posición de memoria de la siguiente instrucción a ejecutar después del salto, será la almacenada en el registro LR

Nota: Para **{cond}** ver las tablas presentadas en las instrucciones aritméticas y lógicas.

Instrucciones de comparación:

Op{cond} Rn, Oper2

Op	Operación	Significado
tst	Test	CPSR ← flags NZCV de Rn AND Oper2
teq	Test equivalencia	CPSR ← flags NZCV de Rn EOR Oper2
cmp	Comparar	CPSR ← flags NZCV de Rn - Oper2
cmn	Comparar negativo	CPSR ← flags NZCV de Rn + Oper2

Nota: Para **{cond}** y deplazamientos del operando **Oper2**, ver las tablas presentadas en las instrucciones aritméticas y lógicas. Estas instrucciones no tienen campo {s} ya que almacenan los flags por defecto (lo que no almacenan es el resultado de la operación)

Instrucciones de carga y almacenamiento (load / store):

- Offset cero: **Op{cond}{b} Rd, [Rn]**
- Relativa a PC: **Op{cond}{b} Rd, label**
- Offset postindexado: **Op{cond}{b} Rd, [Rn], FlexOffset**
- Offset preindexado: **Op{cond}{b} Rd, [Rn, FlexOffset]{!}**

Op	Operación	Significado
ldr	Carga (Load)	Rd ← MEM(dir)
str	Almacenamiento (Store)	MEM(dir) ← Rd

Nota: La dirección de memoria **dir** se calcula, dependiendo del tipo de instrucción, a partir de **Rn**, **label** y/o **FlexOffset**.

Ejemplos		
ldr r0, [r1]	r0 ← MEM(r1)	Carga en r0 el valor almacenado en la posición de memoria indicada en el registro r1
ldr r0, label	r0 ← MEM(label)	Carga en r0 el valor almacenado en la posición de memoria asociada a "*label*" ("*label*" debe aparecer en el segmento de código)
ldr r0, [r1], #4	r0 ← MEM(r1); r1 ← r1 + 4	Carga en r0 el valor almacenado en la posición de memoria indicada en el registro r1. Posteriormente incrementa en 4 el valor de r1.
ldr r0, [r1, #-4]	r0 ← MEM(r1-4)	Carga en r0 el valor almacenado en la posición de memoria calculada restando 4 al contenido del registro r1.
ldr r0, [r1, #-4]!	r0 ← MEM(r1-4) r1 ← r1 – 4	Carga en r0 el valor almacenado en la posición de memoria calculada restando 4 al contenido del registro r1. Actualiza r1 con el valor r1 - 4
ldr r0, [r1, r2]	r0 ← MEM(r1+r2)	Carga en r0 el valor almacenado en la posición de memoria calculada sumando al registro r1 el registro r2.
ldr r0, =0x12345678 ldr r0, =label	r0 ← 0x12345678 r0 ← label	Carga en r0 el valor de una constante de 32 bits (0x12345678). Si se especifica una etiqueta que representa la dirección de memoria de un dato, se cargaría en r0 la dirección del dato (no el dato)

{b}		
Carga o almacena un byte en vez de un word.	ldrb r1, [r2]	Carga en el byte menos significativo de r1 el byte de la posición de memoria indicada en r2. El resto de bytes de r1 se ponen a 0.

{!}		
La dirección calculada se escribe en el registro Rn (sólo para preindexadas)	ldr r1, [r2, r3]!	Carga en r1 ←MEM(r2+r3) y además actualiza el valor de r2 ← r2 + r3
	ldr r1, [r2, r3]	Sólo carga en r1 ←MEM(r2+r3), no modifica r2

Nota: Para **{cond}** ver las tablas presentadas en las instrucciones aritméticas y lógicas. Para una completa explicación de **FlexOffset** ver documentación complementaria de las instrucciones ARM.

Ejemplo de ejecución de varias instrucciones ldr con distintos modos de direccionamiento

Pos. Mem.	Valor almacenado
	.data
0x00001000	dato1: .word 0x12345678
0x00001004	.word 0xABCDEF01

	.text	Valor cargado	Justificación
0x00002000	ldr r0, [r1]	**r0 ←0x12345678**	MEM(r1) = MEM(0x00001000)
0x00002004	ldr r0, [r1], #4	**r0 ←0x12345678** **r1 ←0x00001004**	MEM(r1) = MEM(0x00001000) r1+4
0x00002008	ldr r0, [r2, #-4]	**r0 ←0x12345678**	MEM(r2-4) = MEM(0x00001004-4) = MEM(0x00001000)
0x0000200C	ldr r0, [r3, #-4]!	**r0 ←0xABCDEF01** **r3 ←0x00001004**	MEM(r3-4) = MEM(0x00001008-4) = MEM(0x00001004) r3-4
0x00002010	ldr r0, [r4, r5]	**r0 ←0x12345678**	MEM(r4+r5) = MEM(0x0000100C+0xFFFFFFF4)= MEM(0x00001000)
0x00002014	ldr r0, =0xCAFECAFE	**r0 ← 0xCAFECAFE**	Valor de la constante
0x00002018	ldr r0, =dato1	**r0 ← 0x00001000**	Valor de la etiqueta dato1

Registros	Valor almacenado
r1	0x00001000
r2	0x00001004
r3	0x00001008
r4	0x0000100C
r5	0xFFFFFFF4 (-12)

Comandos del Shell de Linux

Descripción	Comando
Conectar por red a la Raspberry Pi (usuario: tc, password: tc)	`ssh –X tc@nombreRaspPi`
Copiar fichero por red a la Raspberry Pi (password: tc)	`scp fichero tc@nombreRaspPi:`
Copiar fichero desde la Raspberry Pi (password: tc)	`scp tc@nombreRaspPi:fichero .`
Mostrar contenido directorio	`ls`
Cambiar de directorio	`cd nombreDir`
Crear nuevo directorio	`mkdir nombreDir`
Borrar fichero	`rm nombreFichero`
Editar fichero ensamblador	`geany programa.s &`

Compilar fichero ensamblador
Enlazar
Enlazar con librería wiringPi
	`as –o programa.o programa.s`
	`gcc –o programa programa.o`
	`gcc –o programa programa.o -lwiringPi`

Compilar con Makefile	`make programa`
Ejecutar programa compilado Ejecutar programa que usa wiringPi (pide password: tc)	`./programa` `sudo ./programa`
Apagar Raspberry Pi y borrar ficheros	`./borraYapaga.sh`

Copiar ficheros desde/hacia la Raspberry Pi de forma gráfica

1. Abrir explorador de ficheros de Linux
2. En la barra de dirección poner:
   sftp://tc@nombreRaspPi
3. Cambiar de directorio a:
   /home/tc/practicas
4. Arrastrar y soltar ficheros desde/hacia esta ventana.

Usando GDB (supuesto se ha compilado con "-g")

Descripción	Comando
Lanzar el gdb	`gdb [-tui] executable_name`
Listar el código fuente	**l**`ist`
Poner breakpoint	**b**`reak line-number`
Listar nros. breakpoints	**i**`nfo` **break**
Borrar un breakpoint	**d**`el num_breakpoint`
Borrar todos los breakpoints	**cl**`ear`
Ejecutar programa	**r**`un`
Ver contenidos registros CPU	**i**`nfo registers`
Desensamblar código máquina	**disas**`semble [starting_address ending_address]`
Ejecución por pasos de instrucciones máquina	**s**`tep i [number_steps]`
Ejecución por pasos (entra en funciones que estén en el fuente)	**n**`ext i`
Continuar ejecución	**c**`ontinue`
Ver contenidos memoria	**x**`/nfs starting_address` **Options** / **Possible values** n: Number of ítems — any number f: Format — **o**ctal, he**x**, **d**ecimal,**u**nsigned decimal, bi**t**, **f**loat, **a**ddress, **i**nstruction, **c**har, and **s**tring s: Size — **b**yte, **h**alfword, **w**ord, **g**iant(8B) Ej.: ver en hexadecimal las 12 palabras en tope pila: `x/12xw $sp`
Imprimir pos_mem/registro	**p**`rint/f label/reg`
Salir	**q**`uit`

Comandos TUI (Text Uslayouter Interface for gdb)

Descripción	Comando
Activar/desactivar TUI	`C-x A`
Cambiar ventana activa	`focus next/prev src/asm/split/regs, C-x o`
Cambiar ventanas mostradas	`layout next/prev/src/asm/split/regs, C-x 1, C-x 2`
Refrescar pantalla	`refresh, C-L`

Librería *wiringPi*

Librería para utilización de la placa de expansión (leds, pulsadores, altavoz).
Llamadas a función desde ensamblador:

1. Cargar los parámetros de la función en los registros (hasta cuatro parámetros de entrada se pasan por registro: 1er parámetro en r0, 2º en r1, …)
2. Llamar a la función:
   bl *nombre_función*
3. Recoger el valor de retorno de la función (si lo hay) del registro r0

Función	Argumentos	Descripción
`int` **`wiringPiSetup`** `(void)`	Ninguno	Función que inicializa la librería. Debe ser llamada antes de utilizar cualquiera de las funciones de la librería.
`void` **`pinMode`** `(int pin, int mode)`	**pin:** id del pin a configurar **mode:** modo de operación del pin (INPUT, OUTPUT)	Configura el modo de funcionamiento (mode) del pin especificado por el parámetro "pin". El modo puede ser de entrada (INPUT, 0) o salida (OUTPUT, 1).
`void` **`digitalWrite`** `(int pin, int value)`	**pin:** id del pin en el que escribir **value:** valor a escribir en el pin	Genera la salida especificada por el parámetro "value" (0, 1) en el pin especificado por el parámetro "pin". Dicho pin ha debido ser inicializado como de salida (OUTPUT) con la función pinMode.
`int` **`digitalRead`** `(int pin)`	**pin:** id del pin cuyo estado (0, 1) se quiere leer	Lee el estado del pin (0, 1) especificado por el parámetro "pin". El valor leído lo devuelve por el registro r0.
`void` **`delay`** `(unsigned int howLong)`	**howLong:** número de milisegundos	Suspende la ejecución del programa durante los milisegundos especificados en el parámetro "howLong".
`void` **`delayMicroseconds`** `(unsigned int howLong)`	**howLong:** número de microsegundos	Suspende la ejecución del programa durante los microsegundos especificados en el parámetro "howLong".

Ejemplo de lectura del estado de un pin, correspondiente a un pulsador de la placa conectada a la Raspberry Pi.

```
.include "wiringPiPins.s"      @ fichero con definiciones de los pines
                               @ de la placa #BUTTON1, #RLED1, …
                               @ y modos para los pines #INPUT, #OUTPUT

      bl wiringPiSetup         @ inicializamos la librería wiringPi

                               @ Vamos a configurar el pin asociado al
                               @ pulsador 1 como de entrada (pinMode)
      mov r0, #BUTTON1         @ indicamos el pin del pulsador en r0
                               @ (1er parámetro de la función)
      mov r1, #INPUT           @ indicamos el modo en el registro r1
                               @ (2º parámetro de la función)
      bl pinMode               @ llamamos a la función pinMode

                               @ Vamos a leer el estado del pulsador 1
                               @ con la función digitalRead
      mov r0, #BUTTON1         @ indicamos el pin del pulsador en r0
                               @ (1er parámetro de la función)
      bl digitalRead           @ llamamos a la función digitalRead
      cmp r0, #0               @ En el registro r0 está el valor leído
      …                        @ del pin asociado al pulsador (0, 1)
```

Librería *UsefulFunctions / UsefulFunctionsRpi*

Librería básica de entrada/salida por terminal (*UsefulFunctions.s* para usar con ARMSim# y *UsefulFunctionsRpi* para usar en la Raspberry Pi).

- **Para usar en ARMsim#**: En el menú "*File→Load Multiple*" añadir junto al fichero fuente de vuestro código, el fichero "*UsefulFunctions.s*"
- **Para usar en Rpi**: A parte de ensamblar vuestro fichero fuente:
  - ensamblar también el fichero "*UsefulFunctionsRpi.s*"
    - `as UsefulFunctionsRpi.s –o UsefulFunctionsRpi.o`
  - luego enlazarlo con gcc a vuestro fichero objeto
    - `gcc fuente.o UsefulFunctionsRpi.o –o ejecutable`

Llamadas a función desde ensamblador:

1. Cargar los parámetros de la función en los registros (r0, r1, …)
2. Llamar a la función: bl *nombre_función*
3. Recoger el valor de retorno de la función (si lo hay) del registro r0

Función	Argumentos (r0, r1, …)	Descripción
`void` **`tcprints`** `(char *str)`	**str:** dirección de la cadena ASCII terminada con byte 0.	Imprime por la salida estándar la cadena de texto cuya dirección está en "str".
`void` **`tcprintsln`** `(char *str)`	**str:** dirección de la cadena ASCII terminada con byte 0.	Imprime por la salida estándar la cadena de texto cuya dirección está en "str" y a continuación pasa a la siguiente línea.
`void` **`tcprinti`** `(int n)`	**n:** número entero a imprimir.	Imprime por la salida estándar el entero "n" en decimal.
`void` **`tcprintiln`** `(int n)`	**n:** número entero a imprimir.	Imprime por la salida estándar el entero "n" en decimal y a continuación pasa a la siguiente línea.
`void` **`tcprintu`** `(unsigned int n)`	**n:** número entero sin signo a imprimir.	Imprime por la salida estándar el entero sin signo "n" en decimal.
`void` **`tcprintuln`** `(unsigned int n)`	**n:** número entero sin signo a imprimir.	Imprime por la salida estándar el entero sin signo "n" en decimal y a continuación pasa a la siguiente línea.
`void` **`tcprinthex`** `(int n)`	**n:** número entero a imprimir en hexadecimal.	Imprime por la salida estándar el entero "n" en hexadecimal.
`void` **`tcprinthexln`** `(int n)`	**n:** número entero a imprimir en hexadecimal.	Imprime por la salida estándar el entero "n" en hexadecimal y a continuación pasa a la siguiente línea.
`int` **`tcgets`** `(char *buff, int size)`	**buff:** dirección de memoria donde almacenar la cadena leída (debe tener tamaño suficiente para la cadena + byte 0). **size:** El tamaño del buffer.	Lee una cadena de texto ASCII de la entrada estándar y la almacena en "buff". Devuelve el número de caracteres leídos (0 si lee EOF o ha habído un error).
`int` **`tcstrlen`** `(char *str)`	**str:** dirección de la cadena ASCII terminada con byte 0.	Devuelve la longitud (número de caracteres) de la cadena "str" (sin contar el byte 0 final).
`char *`**`tcitoa`** `(int n, char *buff)`	**n:** número entero a convertir a cadena. **buff:** dirección de memoria donde almacenar la cadena que representa el número (debe tener tamaño suficiente para la cadena + byte 0).	Convierte el entero "n" en una cadena de texto ASCII terminada en byte 0 que almacena en la dirección indicada en "buff" (debe haber espacio suficiente). Devuelve la dirección del buffer.
`char *`**`tcutoa`** `(unsigned int n, char *buff)`	**n:** número entero sin signo a convertir a cadena. **buff:** dirección de memoria donde almacenar la cadena que representa el número (debe tener tamaño suficiente para la cadena + byte 0).	Convierte el entero sin signo "n" en una cadena de texto ASCII terminada en byte 0 que almacena en la dirección indicada en "buff" (debe haber espacio suficiente). Devuelve la dirección del buffer.
`int` **`tcatoi`** `(char *str)`	**str:** dirección de la cadena ASCII terminada con byte 0 que representa un número entero.	Convierte la cadena de texto cuya dirección está en "str" en un entero y lo devuelve como salida de la función.

Ejemplo de uso de algunas funciones de la librería *UsefulFunctions*

Codigo fuente:

```
.data
num:   .word 0xffffffff
cad0:  .asciz "Comenzando con las pruebas de la libreria UsefulFunctions ..."
cad1:  .asciz "Este es el numero con signo: "
cad2:  .asciz " y este es el numero sin signo: "
cad3:  .asciz "Escribe un numero: "
cad4:  .asciz "El numero que has escrito menos uno es: "
buff:  .space 256

.text
.global main
main:
       push {lr}

       ldr r0, =cad0   @ cargamos r0 con la direccion de cad0
       bl tcprintsln   @ imprimimos cad0 con newline al final

       ldr r0, =cad1   @ cargamos r0 con la direccion de cad1
       bl tcprints     @ la imprimimos sin newline al final
       ldr r0, =num    @ cargamos la direccion de la variable entera num
       ldr r0, [r0]    @ cargamos en r0 el valor de la variable (0xffffffff)
       bl tcprinti     @ la imprimimos como un numero con signo (-1) (sin
                       @ newline)
       ldr r0, =cad2   @ cargamos en r0 la direccion de cad2
       bl tcprints     @ la imprimimos sin newline al final
       ldr r0, =num    @ volvemos a cargar la direccion de la variable num
       ldr r0, [r0]    @ y volvemos a cargar en r0 el valor de la variable
                       @ (0xffffffff)
       bl tcprintuln   @ pero ahora lo imprimimos como un numero sin signo y con
                       @ newline

       ldr r0, =cad3
       bl tcprints     @ imprimimos cad3 sin newline

       ldr r0, =buff   @ en r0 cargamos la direccion del buffer donde almacenar
                       @respuesta @ leida por teclado
       mov r1, #256    @ en r1 ponemos el tamaño del buffer
       bl tcgets       @ leemos una cadena de la entrada estandar (en buff)

       ldr r0, =buff   @ tomamos la direccion del buffer con la cadena leida
       bl tcatoi       @ la transformamos a un entero
       sub r1, r0, #1  @ le restamos 1

       ldr r0, =cad4
       bl tcprints     @ imprimimos cad4 sin newline

       mov r0, r1      @ ponemos en r0 el numero leido al que le hemos restado 1
       bl tcprintiln   @ y lo imprimimos como entero y terminando la linea

       pop {pc}
```

Ensamblado, enlazado y ejecución del programa ejemplo en una Rpi:

```
tc@raspberrypi:~/practicas $ as pru_UsefulFunctions.s -o pru_UsefulFunctions.o
tc@raspberrypi:~/practicas $ as UsefulFunctionsRpi.s -o UsefulFunctionsRpi.o
tc@raspberrypi:~/practicas $ gcc pru_UsefulFunctions.o UsefulFunctionsRpi.o -o pru_UsefulFunctions
tc@raspberrypi:~/practicas $ ./pru_UsefulFunctions
Comenzando con las pruebas de la libreria UsefulFunctions ...
Este es el numero con signo: -1 y este es el numero sin signo: 4294967295
Escribe un numero: -15
El numero que has escrito menos uno es: -16
tc@raspberrypi:~/practicas $
```