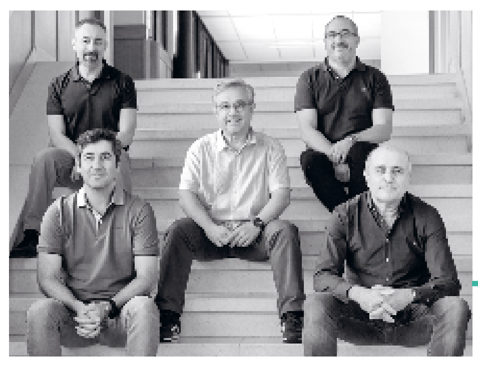
Arquitectura de ordenadores

Ejercicios prácticos de ARM/Thumb

Autores

Manuel José Fernández Iglesias
Martín Llamas Nistal
Luis Eulogio Anido Rifón
Juan Manuel Santos Gago
Fernando Ariel Mikic Fonte

Miscelánea
Serie de textos misceláneos

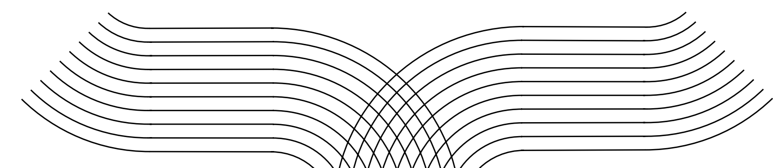


Manuel José Fernández Iglesias
Martín Llamas Nistal
Luis Eulogio Anido Rifón
Juan Manuel Santos Gago
Fernando Ariel Mikic Fonte

Luis Anido, Manolo Fernández, Martín Llamas, Juan Santos y Fernando Mikic son profesores del departamento de ingeniería telemática de la Universidade de Vigo.
Con diversas dedicaciones a lo largo de los años, y en ocasiones con ayuda de otros profesores del departamento, han sido los encargados de la puesta en marcha de las asignaturas dedicadas a la introducción a las arquitecturas de ordenadores en los diversos planes de estudio de la Escuela de Ingeniería de Telecomunicación de Vigo, desde su creación en 1985. Todos ellos pertenecen al grupo de investigación en ingeniería de sistemas telemáticos y, además de sus labores docentes, también comparten proyectos de investigación en el ámbito del e-learning.

Servizo de Publicacións

Universida de Vigo



Miscelánea
Serie de textos misceláneos

Edición
Universidade de Vigo
Servizo de Publicacións
Rúa de Leonardo da Vinci, s/n
36310 Vigo

Deseño gráfico
Área de Imaxe
Vicerreitoría de Comunicacións e Relacións Institucionais

Imaxe da portada
Adobe Stock

Maquetación
Manuel J. Fernández Iglesias, co sistema de procesado de textos LaTeX, desenvolvido orixinalmente por Leslie Lamport baseándose no sistema de maquetación TeX creado por Donald Knuth.

Impresión
Tórculo Comunicación Gráfica, S. A.

ISBN
978-84-8158-855-2

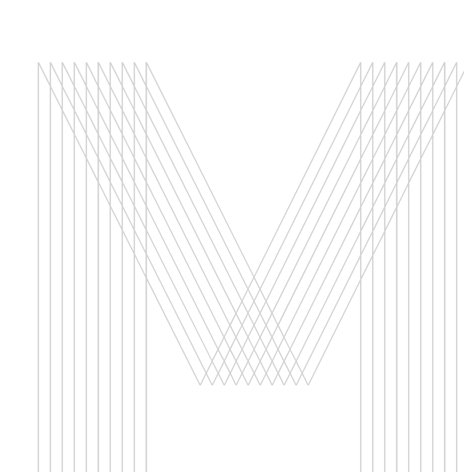
Depósito legal
VG 443-2020

© Servizo de Publicacións da Universidade de Vigo, 2020
© Manuel José Fernández Iglesias, Martín Llamas Nistal, Luis Eulogio Anido Rifón, Juan Manuel Santos Gago y Fernando Ariel Mikic Fonte

Sen o permiso escrito do Servizo de Publicacións da Universidade de Vigo, quedan prohibidas a reprodución ou a transmisión total e parcial deste libro a través de ningún procedemento electrónico ou mecánico, incluídos a fotocopia, a gravación magnética ou calquera almacenamento de información e sistema de recuperación.

Ao ser esta editorial membro da une, garántense a difusión e a comercialización das súas publicación no ámbito nacional e internacional.

Servizo de Publicacións

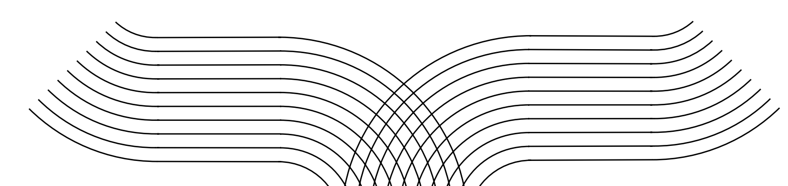


Arquitectura de ordenadores

Ejercicios prácticos de ARM/Thumb

Profesores

Manuel José Fernández Iglesias
Martín Llamas Nistal
Luis Eulogio Anido Rifón
Juan Manuel Santos Gago
Fernando Ariel Mikic Fonte



Prefacio

El presente libro está basado en las clases prácticas y exámenes prácticos que los autores han ido preparando a lo largo de más de veinte años de experiencia impartiendo las asignaturas de arquitectura de ordenadores en los diversos planes de estudio de la Escuela de Ingeniería de Telecomunicación de la Universidade de Vigo.

El libro se organiza en cuatro capítulos. El primer capítulo presenta, a modo de introducción, algunas de las alternativas disponibles para poner en práctica los diferentes ejercicios propuestos en este manual. El resto de los capítulos desarrolla, cada uno de ellos, un conjunto de ejercicios de complejidad creciente elegidos por su valor pedagógico.

Así, el capítulo 2 propone problemas orientados a la iniciación en la programación en ensamblador, ilustrando la programación de las estructuras básicas de control y el trasiego de información utilizando los registros y la memoria principal en una arquitectura *load/store* como la arquitectura ARM.

El siguiente capítulo está dedicado al procesado de información en ensamblador. Comenzamos con tareas básicas como contar, sumar y comparar números, para presentar a continuación el procesado de cadenas de caracteres. Más adelante, proponemos un conjunto de problemas centrados en la utilización combinada del repertorio de instrucciones aritméticas, lógicas y de desplazamiento.

Finalmente, el capítulo 4 introduce los conceptos de subprograma y de paso de parámetros. En primer lugar, se trata el paso de parámetros a subprogramas mediante registros. A continuación, presentamos el concepto de pila y su manejo, para luego recuperar el concepto de subprograma, esta vez para introducir el paso de parámetros a través de la pila, el anidamiento de llamadas a subprograma y la recursividad.

Cualquier manual práctico sobre programación en ensamblador necesita de una arquitectura de referencia, un lenguaje ensamblador y unas herramientas para poner en práctica los conocimientos adquiridos. Como veremos en el primer capítulo de este libro, utilizaremos la primera versión del repertorio

de instrucciones T32/Thumb de la arquitectura ARM, que incluye únicamente instrucciones de 16 bits, y el ensamblador para ARM de GNU. En cuanto a las herramientas, sugerimos dos opciones ampliamente validadas en entornos educativos. También utilizaremos en los enunciados de algunos ejercicios la sintaxis del lenguaje de programación C para describir los algoritmos utilizados.

ARM probablemente se ha convertido en la arquitectura de microprocesador más popular. Esta arquitectura, introducida hace más de 35 años, en la década de los 80 del siglo pasado, está presente en la actualidad en miles de millones de dispositivos, desde Fugaku, el superordenador japonés líder del ranking de los ordenadores más potentes del mundo en 2020, con cerca de 160.000 unidades del chip A64FX de 48 núcleos, hasta el Catweazle Mini, un femtocomputador de poco más de 0,25 cm^3 de volumen basado en el NXP LPC810 ARM Cortex M0+. Conocidos fabricantes como Apple, Broadcom, NVidia, Qualcomm o Samsung basan sus dispositivos más populares en esta arquitectura y los dispositivos ARM están presentes en sectores como la automoción, los vehículos espaciales, el hogar inteligente, los dispositivos vestibles (*wearable devices*), la telefonía móvil, el Internet de las Cosas (*Internet of Things*, IoT), o las ciudades inteligentes.

Thumb, conocido como T32 a partir de la versión 8 de la arquitectura ARM, es un repertorio de instrucciones y un estado de ejecución de la arquitectura ARM que, aunque se definió inicialmente a partir de las instrucciones más utilizadas, codificadas con 16 bits en vez de 32 bits, en la actualidad incluye las instrucciones originales, así como instrucciones adicionales de 16 y 32 bits. Está orientado fundamentalmente al desarrollo de sistemas embebidos, como por ejemplo los dispositivos para el control de sistemas de videovigilancia, routers, automóviles, cámaras digitales, electrodomésticos, y en general cualquier dispositivo del ámbito IoT. Al ocupar las instrucciones más utilizadas 16 bits en vez de 32, esta versión reducida de la arquitectura ARM permite incrementar de manera relevante la densidad de código, con lo que es posible realizar aplicaciones en dispositivos con menores requisitos de memoria, y por lo tanto con un menor coste de fabricación. Todos los núcleos de la arquitectura ARM soportan en mayor o menor medida el repertorio T32, e incluso hay micros ARM que sólo soportan instrucciones de este repertorio.

Este manual incluye en el apéndice B, a modo de referencia rápida, un resumen del modelo del programador Thumb, donde se recoge una descripción de los registros, de los indicadores de estado, de la organización de la memoria, y finalmente unas tablas con las instrucciones utilizadas en el libro.

El libro se completa con un apéndice sobre los juegos de caracteres UTF-8 e ISO Latin 9, que incluye a su vez información sobre el juego de caracteres ASCII y la manera en que éste se integra en el modelo UTF (apéndice C). Además, todos los ejercicios del libro están disponibles, junto con otra documentación y referencias interesantes, en la página `libroarm.webs.uvigo.gal`.

El texto ha sido revisado y corregido, pero en todo caso los autores asumen toda la responsabilidad por los gazapos que pudieran haberse escapado de este proceso de revisión. Agradecemos a nuestros lectores y lectoras que nos informen de cualquier error que pudieran localizar para proceder a corregirlo en futuras ediciones. Asimismo, agradecemos todo tipo de comentarios, críticas y sugerencias que puedan contribuir a mejorar tanto la presentación como el contenido de este libro. En la página del libro en Internet es posible encontrar nuestros datos de contacto.

Agradecimientos

Como indicamos anteriormente, el contenido de este manual surge de la experiencia docente de los autores, por lo que queremos mostrar nuestro reconocimiento a nuestro alumnado, fuente continua de aprendizaje para nosotros. Su presencia a lo largo de estos años nos ha hecho evolucionar y ha contribuido de manera determinante a mantener vivo nuestro afán por mejorar. Con ellos hemos ido creciendo a lo largo de estos años y gracias a ellos mantenemos viva nuestra curiosidad.

También queremos agradecer su apoyo y sus comentarios a los compañeros que a lo largo de estos años han impartido las asignaturas de arquitectura de ordenadores con nosotros, especialmente a Alberto Gil Solla, Manuel Caeiro Rodríguez y Luis M. Álvarez Sabucedo.

Finalmente, y no por ello menos importante, queremos dejar patente nuestro reconocimiento a nuestras familias por su apoyo, su comprensión y su generosidad.

Índice general

Índice de figuras 7

Índice de cuadros 9

1. Calentando motores 11
1.1. El simulador QtARMSim 12
1.2. Programación en Thumb con Raspberry Pi 17
1.3. Ejercicios introductorios 25

2. Soltando amarras 31
2.1. Estructuras básicas de control 31
2.2. Trasiego del contenido de la memoria 41

3. Velocidad de crucero 49
3.1. Contar y sumar 49
3.2. Comparar números 56
3.3. Operaciones con cadenas de caracteres 65
3.4. Operaciones aritmético-lógicas 76

4. A toda máquina 107
4.1. Subprogramas 107
4.2. La pila 121
4.3. Más sobre subprogramas 141

A. Ejercicios introductorios para Raspberry Pi 169

B. Modelo del programador 177
B.1. Registros del programador 177
B.2. Organización de la memoria 179
B.3. Modos de direccionamiento 180
B.3.1. Direccionamiento directo a registro 180

B.3.2. Direccionamiento inmediato 181
B.3.3. Direccionamiento indirecto a registro con desplazamiento 182
B.3.4. Direccionamiento relativo al contador de programa . . 183
B.3.5. Direccionamiento relativo al puntero de pila 185
B.3.6. Direccionamiento indirecto a registro con registro de desplazamiento 185
B.4. Repertorio de instrucciones 186
B.5. Selección de directivas del ensamblador 190

C. Tablas UTF-8 **191**

Índice de materias **199**

Índice de figuras

1.1. Ventana principal de QtARMSim. 13
1.2. Programa de ejemplo. 14
1.3. Desensamblado del programa de ejemplo. 15
1.4. Punto de ruptura en QtARMSim. 17
1.5. Raspberry Pi 4 Model B. 18
1.6. Programa de ejemplo en Raspbian 21
1.7. Analizando el programa de ejemplo. 24

2.1. Estructura de control `if-then`. 32
2.2. Ejemplo de estructura de control `if-then`. 33
2.3. Estructura de control `if-then-else`. 34
2.4. Ejemplo de estructura de control `if-then-else`. 35
2.5. Estructura de control `while`. 36
2.6. Ejemplo de estructura de control `while`. 37
2.7. Estructura de control `for`. 38
2.8. Ejemplo de estructura de control `for`. 39

4.1. Reorganización de una zona de memoria 108
4.2. Baraja española de 40 cartas. 159

B.1. Registros del programador T32/Thumb. 178
B.2. Registro de estado CPSR. 178
B.3. Codificación de la instrucción `add SP,#0x160`. 181
B.4. Codificación de la instrucción `bl 0x0670fd`. 185

Índice de cuadros

1.1. Atajos de teclado de QtARMSim. 13
1.2. Algunos comandos útiles de `gdb`. 23

B.1. Significado de los códigos de condición. 179
B.2. Modos de direccionamiento en el estado T32/Thumb. 180
B.3. Posibles valores de los datos inmediatos. 182
B.4. Instrucciones aritméticas. 187
B.5. Instrucciones lógicas y de desplazamiento. 188
B.6. Instrucciones de carga y almacenamiento. 188
B.7. Instrucciones de manejo de la pila. 189
B.8. Instrucciones de salto. 189
B.9. Directivas del ensamblador Thumb. 190

C.1. Caracteres UTF-8 de `0x00` a `0x3F`. 193
C.2. Caracteres UTF-8 de `0x40` a `0x7F`. 194
C.3. Caracteres UTF-8 de `0xC280` a `0xC2BF`. 195
C.4. Caracteres UTF-8 de `0xC380` a `0xC3BF`. 196
C.5. Caracteres UTF-8 alfabéticos utilizados en castellano, gallego y portugués (I). 197
C.6. Caracteres UTF-8 alfabéticos utilizados en castellano, gallego y portugués (II). 198

Capítulo 1
Calentando motores

Cualquier texto de ejercicios de programación necesita de un entorno donde poner en práctica los conocimientos adquiridos. Este primer capítulo tiene como objetivo presentar dos alternativas concretas para programar en lenguaje ensamblador utilizando el repertorio de instrucciones Thumb de 16 bits. En ambos casos hemos elegido soluciones populares en el entorno universitario, que además han demostrado su utilidad pedagógica. Se trata del simulador QtARMSim y del sistema Raspberry Pi / Raspbian / GNU toolchain. Estas dos alternativas se basan, respectivamente, en utilizar un simulador de la máquina o familia de máquinas que soportan el lenguaje ensamblador elegido (QtARMSim) y en utilizar una máquina real cuya arquitectura es compatible con el lenguaje ensamblador elegido (Raspberry Pi / Raspbian / GNU toolchain).

A la hora de escribir programas en ensamblador, ambas opciones utilizan el ensamblador de GNU. En el caso concreto de este manual, los ejercicios propuestos se han comprobado utilizando QtARMSim y el código ensamblador de las soluciones propuestas está listo para ser cargado directamente en dicho simulador. De todos modos, como veremos, con cambios menores podrían ser probadas también en cualquier otro entorno didáctico.

De las dos opciones presentadas, hemos elegido QtARMSim como herramienta concreta para presentar nuestros ejercicios por ser la más adecuada de las dos para el programador novel. Un simulador pedagógico como QtARMSim abstrae la mayor parte de los detalles de la máquina subyacente, lo que a su vez permite centrarnos en los retos de la programación. Por otra parte, la opción basada en la Raspberry Pi requiere conocimientos más profundos sobre arquitectura de ordenadores y sistemas operativos, pero permitirá al programador experimentado extraer todo el potencial de una máquina real.

En cualquier caso, el epígrafe dedicado a la programación en Thumb de la Raspberry Pi (cf. epígrafe 1.2) es autocontenido, con lo que se puede omitir

su lectura. El apéndice A presenta los ejercicios introductorios del epígrafe 1.3 para el caso de la Raspberry Pi.

1.1 El simulador QtARMSim

QtARMSim[1] es un simulador gráfico de la arquitectura ARM/Thumb sencillo y fácil de usar que fue diseñado para cursos de introducción a la arquitectura de ordenadores. Se distribuye bajo la licencia GPL v3+. Existen versiones para Microsoft Windows, macOS de Apple y Linux.

Una vez instalado de acuerdo con las instrucciones disponibles en la página Web del proyecto QtARMSim, podemos ejecutar el simulador como cualquier programa de nuestro ordenador. Al ejecutar la aplicación, aparecerá una ventana con cuatro zonas diferenciadas: (i) Registros; (ii) Editar/Desensamblar; (iii) Memoria, y (iv) Mensajes/Volcado de memoria/Pantalla LCD (cf. figura 1.1).

Si seleccionamos la pestaña Editor de la parte inferior del área de edición, entraremos en el editor. En esa zona podemos escribir nuestros programas utilizando el lenguaje ensamblador de GNU para la arquitectura ARM[2].

Vamos a utilizar el programa de la figura 1.2 como ejemplo para ilustrar el funcionamiento del simulador. Una vez editado, lo guardaríamos en un archivo utilizando la opción correspondiente de la barra de menús (🖫) o el atajo de teclado Ctrl + S (⌘ + S en un ordenador macOS). El cuadro 1.1 presenta un resumen de los atajos de teclado de QtARMSim.

Al seleccionar la pestaña ARMSim en la parte inferior de la zona de edición, el simulador intentará ensamblar automáticamente el programa. En caso de que haya errores, se notificarán en el área de Mensajes que está debajo de la zona de edición. Si es así, volveríamos a la pestaña del Editor, corregiríamos esos errores y lo intentaríamos nuevamente.

Una vez que el programa esté libre de errores, al seleccionar la pestaña ARMSim se ensamblará automáticamente el código fuente y QtARMSim pasará al modo de simulación. En este modo, cada línea se corresponde con una instrucción del nivel de máquina convencional almacenada en la memoria. La información que se muestra se obtiene mediante un proceso llamado desensamblado y consiste en lo siguiente (de izquierda a derecha):

---

[1] *Introducción a la arquitectura de computadores con QtARMSim y Arduino*, por Sergio Barrachina Mir, Maribel Castillo Catalán, Germán Fabregat Llueca, Juan Carlos Fernández Fernández, Germán León Navarro, José Vicente Martí Avilés, Rafael Mayo Gual y Raúl Montoliu Colás. (`http://lorca.act.uji.es/project/qtarmsim`).

[2] `https://sourceware.org/binutils/docs/as/index.html`.

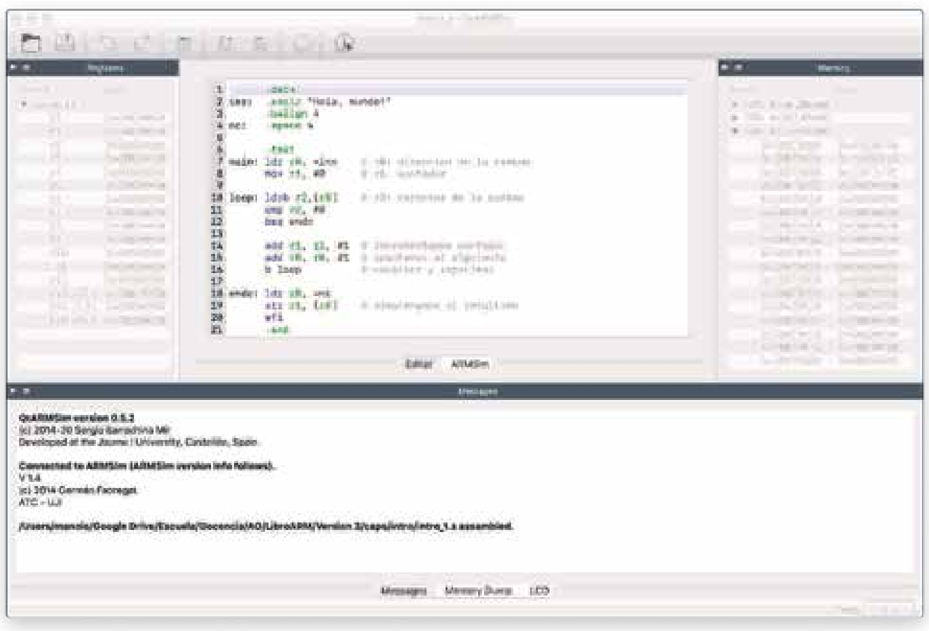


Figura 1.1. Ventana principal de QtARMSim.

Cuadro 1.1. Atajos de teclado de QtARMSim. En el caso de los ordenadores Apple, hay que reemplazar la tecla Ctrl por la tecla ⌘.

F1	Ayuda
Shift ⇑ + F1	¿Qué es ...?
F3	Restablecer la configuración predeterminada
F4	Restablecer la simulación
F5	*Step into* (paso a paso)
F6	*Step over* (saltar subprograma)
Ctrl + F11	Ejecutar
Ctrl + N	Nuevo archivo
Ctrl + O	Abrir archivo
Ctrl + S	Guardar archivo
Ctrl + Q	Salir de QtARMSim
Alt + P	Preferencias

intro_1.s

```
        .data
ins:    .asciz "Hola, mundo!"
        .balign 4
nc:     .space 4

        .text
main:   ldr r0, =ins     @ r0: dirección de la cadena
        mov r1, #0       @ r1: contador

loop:   ldrb r2,[r0]     @ r2: carácter de la cadena
        cmp r2, #0
        beq endc

        add r1, r1, #1   @ incrementamos contador
        add r0, r0, #1   @ apuntamos al siguiente
        b loop           @ carácter y repetimos

endc:   ldr r0, =nc
        str r1, [r0]     @ almacenamos el resultado
        wfi
        .end
```

Figura 1.2. Programa de ejemplo.

- La dirección de memoria en la que se almacena la instrucción.
- La instrucción en código máquina expresada en hexadecimal.
- La instrucción desensamblada, es decir, la instrucción en lenguaje ensamblador, con la sintaxis ARM completa.
- La línea de texto original del código fuente que produjo la instrucción durante el proceso de ensamblado.

Por ejemplo, la primera línea del programa anterior (cf. figura 1.3):

```
[0x00180000] 0x4805 ldr r0, [pc, #20]; 7 ldr r0, =ins @ r0: [...]
```

indica que:

- La instrucción está almacenada a partir de la dirección de memoria `0x00180000`.
- La instrucción en código máquina expresada en hexadecimal es `0x4805`.

- La instrucción desensamblada es `ldr r0, [pc, #20]`. Las instrucciones en este campo no se representan con la sintaxis específica Thumb, sino que se presentan con la sintaxis ARM completa.

- La instrucción se generó a partir de la línea número 7 del código fuente original, cuyo contenido era:

```
ldr r0, =ins @r0: dirección de la cadena
```

```
[0x00180000] 0x4805 ldr r0, [pc, #20]   ;  7        ldr r0, =ins
[0x00180002] 0x4049 eors r1, r1         ;  8        eor r1, r1, r1
[0x00180004] 0x7802 ldrb r2, [r0, #0]   ; 10 loop:  ldrb r2, [r0]
[0x00180006] 0x2A00 cmp r2, #0          ; 11        cmp r2, #0
[0x00180008] 0xD002 beq pc, #4          ; 12        beq endc
[0x0018000A] 0x3101 adds r1, #1         ; 14        add r1, r1, #1
[0x0018000C] 0x3001 adds r0, #1         ; 15        add r0, r0, #1
[0x0018000E] 0xE7F9 b pc, #-14          ; 16        b loop
[0x00180010] 0x4802 ldr r0, [pc, #8]    ; 18 endc:  ldr r0,=nc
[0x00180012] 0x6001 str r1, [r0, #0]    ; 19        str r1, [r0]
[0x00180014] 0xBF30 wfi                 ; 20        wfi
[0x00180016] 0x0000 movs r0, r0
[0x00180018] 0x0000 movs r0, r0          ins = 0x20070000
[0x0018001A] 0x2007 movs r0, #7
[0x0018001C] 0x0010 movs r0, r2
```

Figura 1.3. Desensamblado del programa de ejemplo. Obsérvese la traducción a una instrucción ARM de la pseudoinstrucción `ldr r0, =ins` y cómo se convierte `=ins` al modo de direccionamiento relativo a PC, actualizando al mismo tiempo una palabra (4 posiciones de memoria) después de la instrucción `wfi` con la dirección efectiva.

En el caso de que se trate de una pseudoinstrucción, el código desensamblado muestra la instrucción ARM que genera dicha pseudoinstrucción. En el caso de la pseudoinstrucción `ldr r0,=ins`, podemos observar que se sustituye por una instrucción `ldr` con direccionamiento relativo al contador de programa y por una palabra de memoria con la dirección efectiva depositada en la cercanía de la propia instrucción, en este caso 20 posiciones más adelante de donde apunta el contador de programa (`[pc, #20]`), en la palabra de dirección `0x00180018`. Dicha palabra contiene a su vez la dirección a partir de la cual está almacenada la cadena "Hola Mundo!" al principio de la zona de datos, etiquetada como `ins` (dirección `0x20070000`, figura 1.3).

Para ejecutar el programa, seleccionamos el icono correspondiente de la barra de menús (▶), la opción del menú Run, o el atajo de teclado Ctrl + F11 (⌘ + F11 en un Mac).

Cuando el programa finalice, veremos que los registros `r0`, `r1`, `r2` y `r15` y la posición de memoria `0x20070010` tienen fondo azul y están en negrita. Esto se debe a que el simulador resalta los registros y las posiciones de memoria que se modificaron como consecuencia de la ejecución del programa. `r15` es el contador de programa (PC), por lo que se modifica internamente para apuntar siempre a la siguiente instrucción a ejecutar.

Al trabajar con QtARMSim se sigue el convenio, un tanto artificioso, de que la última instrucción que se ejecuta en un programa es siempre **`wfi`**, es decir, cuando el simulador está ejecutando un programa y se encuentra una instrucción **`wfi`**, detiene la ejecución y devuelve el control al usuario[3]. Esto es un convenio específico de QtARMSim, que no tiene por qué ser el mismo en otras plataformas. Por ejemplo, en la Raspberry Pi terminaremos nuestros programas con una instrucción de salto **`bx`** `lr` que devuelve el control al sistema operativo (cf. epígrafe 1.2).

Una vez que se ejecuta la instrucción **`wfi`**, no se puede continuar con la ejecución del programa desde el simulador, o volver a ejecutarlo. Si queremos repetir la simulación, podemos volver a cargar el programa usando la opción de menú correspondiente (🗀), el atajo de teclado Ctrl + O , o simplemente podemos restablecer la simulación presionando F4 . También podemos modificar el registro `r15` (es decir, el contador de programa) para que apunte a una instrucción ejecutable (p. ej., a la dirección `0x00180000` donde está almacenada la primera instrucción de nuestro programa).

Para ejecutar un programa paso a paso, seleccionaríamos la opción del menú de ejecución paso a paso ({▾}), o presionaríamos F5 . Al ejecutar paso a paso, podemos monitorizar cómo cambian los valores de los registros y las posiciones de memoria durante la ejecución, y también los indicadores del procesador `N`, `Z`, `C` y `V` (signo, cero, acarreo y desbordamiento respectivamente).

Además de ejecutar paso a paso, es posible ejecutar las instrucciones en un programa de manera que, en caso de ejecutar una instrucción de llamada a subprograma, como por ejemplo la instrucción **`bl`**, se ejecute el subprograma completo de una vez antes de devolver el control a la ejecución paso a paso. Se trata de la opción *step over* ({}▾ o F6).

Un punto de ruptura o *breakpoint* es una indicación para que el simulador detenga la ejecución antes de ejecutar la instrucción donde se colocó dicho

[3]En entorno real, la instrucción **`wfi`** (wait for interrupt) detiene la ejecución a la espera de que se produzca una interrupción, cuya rutina de servicio retomará el control del procesador. En algunos microprocesadores, como los de la familia Cortex-M, la ejecución de **`wfi`** implica además entrar en modo de bajo consumo.

punto de ruptura. Al hacer clic en el margen de la ventana de simulación junto a una instrucción (por ejemplo, `ldr r0, =nc`), aparecerá un círculo rojo que indica que se definió un punto de ruptura (cf. figura 1.4). Al ejecutar un programa con Ctrl + F11 , el programa se detendrá justo antes de ejecutar la instrucción donde hemos colocado el punto de ruptura.

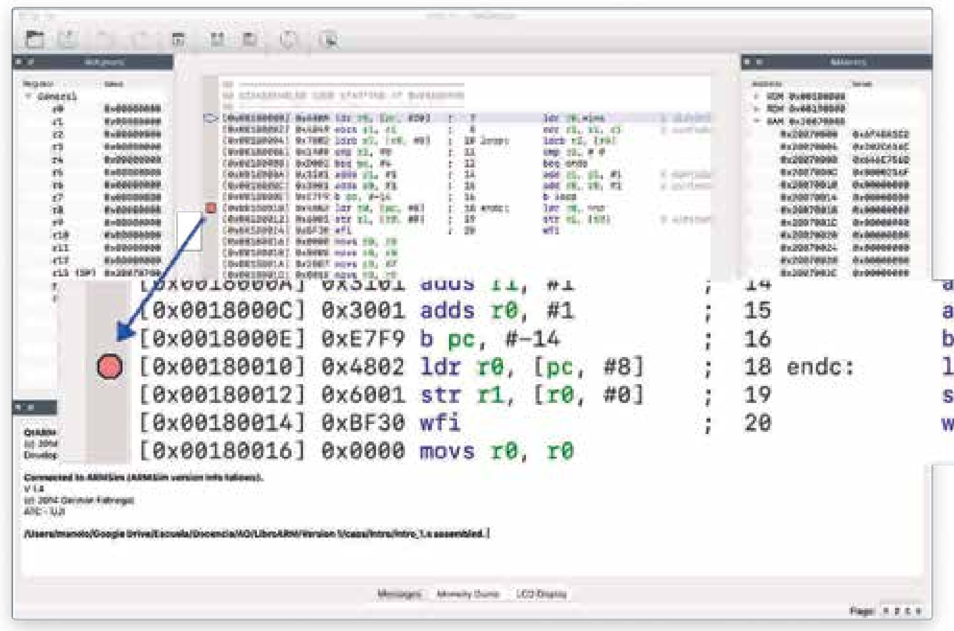


Figura 1.4. Punto de ruptura en QtARMSim.

Para eliminar un punto de ruptura ya definido, simplemente hacemos clic en su círculo rojo.

1.2 Programación en Thumb con Raspberry Pi

Raspberry Pi[4] es un ordenador de placa única (*single-board computer*, SBC) de bajo coste desarrollado con el objetivo de promover la enseñanza de la informática. Hasta el momento de escribir este manual, todos los modelos de Raspberry Pi se basan en un sistema en un chip (SoC) ARM de Broadcom. Dicho microprocesador incluye un núcleo ARM con soporte amplio para el repertorio de instrucciones T32 (y por tanto para las instrucciones Thumb de 16

[4] `https://raspberrypi.org`.

bits), una unidad de procesado de gráficos (GPU), la memoria RAM, un procesador de señal (DSP) y puertos USB. Los ordenadores Raspberry Pi disponen de un sistema operativo propio (RISC OS 5) y varios sistemas operativos basados en Linux para arquitectura ARM, como Raspbian (derivado de Debian), Arch Linux ARM (derivado de Arch Linux) y Pidora (derivado de Fedora). De ellos, el más popular es Raspbian. La figura 1.5 ilustra uno de los modelos más populares de este dispositivo.

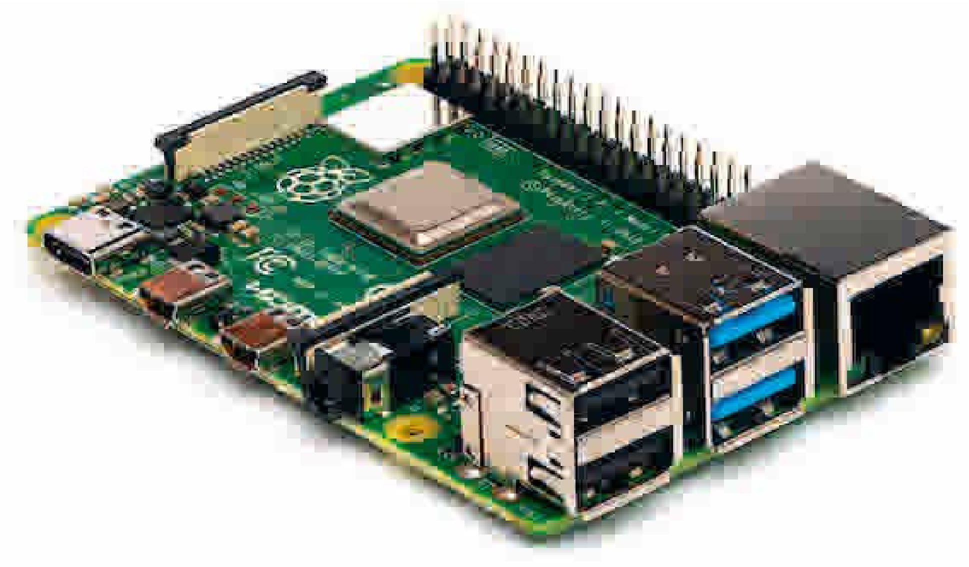


Figura 1.5. Raspberry Pi 4 Model B. El chip cuadrado brillante hacia el centro de la placa, a la derecha de la frambuesa, es el microprocesador, un SoC Broadcom BCM2711, con cuatro núcleos ARM Cortex-A72. (De Michael Henzler / Wikimedia Commons, CC BY-SA 4.0).

Como casi todas las distribuciones basadas en Linux, Raspbian incluye las herramientas de desarrollo de programas del proyecto GNU (*GNU Toolchain*). Entre ellas se encuentra el ensamblador de GNU GAS (`as`), el depurador GDB (`gdb`), el compilador GCC (`gcc`) y el montador de enlaces LN (`ln`). Como el microprocesador de la Raspberry Pi sobre el que se ejecuta Raspbian es un procesador ARM que soporta las instrucciones Thumb de 16 bits, con estas herramientas podremos ensamblar (con `as` y `ld` o `gcc`), depurar y probar (con `gdb`) programas escritos con el repertorio de instrucciones Thumb.

Aunque el procesador de la Raspberry Pi soporta Thumb, hay algunas características que Raspbian no soporta, por lo que no podremos programar exclusivamente en Thumb para la Raspberry Pi utilizando las herramientas

de GNU disponibles en Raspbian. Esto no quiere decir que no podamos escribir, ensamblar y probar programas en Thumb. Podemos escribir código en Thumb, pero tendremos que hacerlo de manera que Raspbian nos permita acceder a nuestro código y regresar desde él.

Una manera de superar este inconveniente se basa en aprovechar la posibilidad que ofrece ARM de mezclar código T32/Thumb con código ARM. Los microprocesadores ARM tienen un bit de estado T32 (`T`), de manera que cuando dicho bit toma el valor 1 (`T = 1`), el microprocesador funciona en *estado T32* y es capaz de decodificar las instrucciones del repertorio T32, y por lo tanto las instrucciones del repertorio original Thumb de 16 bits. Cuando toma el valor 0 (`T = 0`) se encuentra en *estado A32/ARM* y decodifica instrucciones ARM de 32 bits.

Para mezclar código ARM y código Thumb, es decir, para cambiar de estado A32/ARM a estado T32/Thumb y viceversa, debemos ejecutar una instrucción de *salto e intercambio* (*branch and exchange*). Se trata de instrucciones que realizan un salto y además cambian el valor del bit de estado `T`. Dichas instrucciones son **`bx`** en el caso del repertorio Thumb, y **`bx`** o **`blx`** en el caso del repertorio ARM completo. La instrucción **`blx`** realiza una llamada a subprograma (cf. página 107 del capítulo 4) y además intercambia el repertorio de instrucciones.

Utilizando las instrucciones anteriores, es posible *envolver* nuestro código Thumb con un envoltorio ARM para pasárselo al sistema operativo y para que el sistema operativo nos devuelva el control una vez ejecutado nuestro código. Nuestros programas escritos para ser analizados en la Raspberry Pi tendrán la estructura siguiente:

1. Un sencillo programa principal de entrada, escrito en ARM, que simplemente llama a un subprograma que contiene nuestro código Thumb utilizando la instrucción de llamada a subprograma **`blx`** (*branch, link and exchange*).

```
        .code   32       @ Indicamos que este código es ARM (A32).
        .align  4        @ Instrucciones alineadas a 32 bits.
        .global _start   @ o .global main
main:
        push {r4, lr}    @ Apilamos el registro de enlace lr.

        blx mi_programa  @ Utilizamos blx para conmutar a T32 al
                         @ al pasar el control a nuestro programa.
        pop {r4, lr}     @ Recuperamos el registro de enlace
        bx lr            @ y devolvemos el control al SO.
       .end
```

Podemos observar también que el programa principal salvaguarda en la pila el registro de enlace `lr` antes de ejecutar la instrucción **`blx`**, para así permitir el anidamiento de nuestro subprograma (cf. apartado 4.3)[5].

2. Nuestro código Thumb de 16 bits, escrito como un subprograma. Por ello, en vez de terminar nuestros programas con **`wfi`** como hacíamos en los programas para QtARMSim, los terminaremos con la instrucción de salto e intercambio **`bx`** `lr` del repertorio Thumb de 16 bits. Esta instrucción recupera la dirección de retorno al programa principal almacenada en el registro de enlace `lr`, donde la almacenó la instrucción **`blx`** del punto anterior, y cambia al microprocesador de vuelta al estado ARM (cf. apartado B.3.6 y tabla B.8).

El programa-envoltorio tendrá que estar declarado de manera que el montador de enlaces-cargador pueda encontrar el punto de entrada para pasarle el control cuando el sistema operativo lo cargue para ser ejecutado. Para ello, el convenio es etiquetar el punto de entrada deseado (en nuestro caso, la primera instrucción del programa-envoltorio) con la etiqueta `_start` declarada globalmente:

```
        .global _start
_start: @ aquí empezaría nuestro programa-envoltorio
```

Otra opción consiste en declarar nuestro programa-envoltorio como el programa principal de un programa en C y luego enlazar nuestro código con la librería estándar de C `libc` utilizando el compilador de C `gcc`. Con ello, tendremos a nuestra disposición todas las funcionalidades de dicha librería estándar, lo que nos permitirá leer y mostrar caracteres por pantalla entre otras muchas cosas. El único cambio será que ahora el punto de entrada estará etiquetado con la etiqueta `main` declarada globalmente:

```
        .global main
main: @ aquí empezaría nuestro programa-envoltorio
```

La figura 1.6 presenta un ejemplo de esta estrategia basada en el programa de la figura 1.2.

[5]Vemos que también se salvaguarda en la pila el registro `r4`. Esto es así porque el convenio de la arquitectura ARM para la llamada a procedimientos (AAPCS, *ARM Architecture Procedure Call Standard*) especifica que en los interfaces públicos (como los *procedimientos* principales `_start` o `main`) la pila tiene que estar siempre alineada a una dirección múltiplo de 8. En realidad, `r4` podría ser cualquier registro. Simplemente tenemos que garantizar que al introducir valores en la pila, el puntero de pila `sp` sigue siendo un múltiplo de 8 tal como nos lo encontramos.

intro_2a.s

```
      .data
ins: .asciz "¡Hola, mundo!"
      .balign 4
nc:   .space 4

      .text
/*
 Nuestros programas serán funciones invocadas desde el
 punto de entrada de un programa escrito para Raspbian.
-----------------------------------------------------------------*/
      .code  16       @ Esta directiva indica que usaremos Thumb
      .align 2        @ Las instrucciones de Thumb son de 16 bits
                      @ (alineadas en fronteras de 2 bytes)
mi_programa:
    ldr r0,=ins       @ dirección cadena de entrada
    eor r1, r1, r1    @ contador a 0
loop:
    ldrb r2, [r0]
    cmp r2, # 0
    beq endc

    add r1, r1, #1    @ contador de incrementos
    add r0, r0, #1    @ puntero de incremento
    b loop

endc:
    ldr r0, =nc
    str r1, [r0]      @ almacenamos el resultado
                      @ Volver a main.
    bx lr             @ Esta instrucción sustituye a wfi en QtARMSim
/*---------------------------------------------------------------
 Aquí estaría el punto de entrada desde el SO: una función llamada
 "main", declarada obligatoriamente como main
*/
    .code 32          @ Indicamos que nuestro código es ARM completo
    .align 4          @ Instrucciones alineadas a 32 bits
    .global main
main:
    push {r4, lr}     @ Apilamos el registro de enlace lr

    blx mi_programa   @ Utilizamos blx para conmutar de ARM a Thumb al
                      @ al pasar el control a nuestro programa
    pop {r4, lr}      @ Recuperamos el registro de enlace
    bx lr             @ y devolvemos el control al SO
   .end
```

Figura 1.6. Programa de la figura 1.2 adaptado para su ejecución bajo Raspbian tras ensamblarlo y montarlo con `as` y `gcc`.

Para escribir nuestros programas podemos utilizar cualquier editor de texto disponible para Raspbian, como `vim`, `nano`, `emacs`, el editor gráfico `gedit`, o cualquier otro. Asignaremos la extensión `.s` a los ficheros que creemos, igual que hace QtARMSim.

Para ensamblar un programa utilizaremos el comando `as` para invocar al ensamblador. En el caso de que nuestro código esté en un fichero llamado `mi_prog.s`, ejecutaríamos el comando:

```
$ as -g -o mi_prog.o mi_prog.s
```

Si no ha habido errores, el comando anterior generará un fichero objeto con la extensión `.o`, en nuestro ejemplo `mi_prog.o`, con información para el depurador. En caso de que el código fuente tuviera algún error, el ensamblador dará cuenta de dichos errores y no se generará el fichero anterior. Volveremos al editor para corregir los errores.

Ahora necesitamos enlazar el fichero objeto con las librerías del sistema para generar un programa ejecutable utilizando el compilador `gcc`, incluyendo información para el depurador:

```
$ gcc -g -o mi_prog mi_prog.o
```

En caso de que hubiéramos decidido prescindir de la librería estándar de C, habríamos declarado `_start` como punto de entrada y simplemente llamaríamos al enlazador-cargador con:

```
$ ld -o mi_prog mi_prog.o
```

En cualquiera de los dos casos, ahora tendríamos un programa que podemos ejecutar directamente:

```
$ ./mi_prog
```

Además, podemos realizar operaciones similares a las que podríamos realizar con un simulador como QtARMSim utilizando un depurador como GDB (GNU Debugger):

```
$ gdb mi_prog
```

El cuadro 1.2 recoge algunos comandos útiles de GDB, aunque las opciones disponibles son mucho más amplias y versátiles, sobre todo si extendemos el depurador con algún paquete como GEF. La figura 1.7 ilustra el interfaz de usuario de GFB con GEF.

Cuadro 1.2. Algunos comandos útiles de gdb.

b <n>	Coloca un breakpoint en la línea <n>.
c	Continúa con la ejecución del programa a partir de la posición actual.
d <n>	Borra el breakpoint de la línea <n>.
h <c>	Muestra ayuda sobre el comando <c>.
i <s>	Muestra diversa información sobre el programa, en función de <s> (registers, args, breakpoints...) h i ofrece detalles sobre todas las opciones posibles.
l <n>	Lista 10 líneas del código fuente, centradas en la línea especificada por <n>.
p <e>	Evalúa la expresión <e> e imprime su valor. Si <e> es una etiqueta, muestra el contenido de la posición de memoria etiquetada.
r	Comienza la ejecución de un programa.
n	Ejecuta la siguiente línea de código (*step over*).
s	Ejecuta la siguiente instrucción (*step into*).
f	Ejecuta hasta el final del subprograma actual.
x/<n><f><t> <d>	Muestra <n> valores de memoria en formato <f>, de tamaño <t>, comenzando por la dirección <d>. h x ofrece detalles sobre todas las opciones posibles para <f> y <t>.
q	Sale del programa gdb.

```
[ Legend: Modified register | Code | Heap | Stack | String ]
──────────────────────────────────────────────────────────── registers ────
$r0   : 0x00021024  →  <ins+0> svcvs 0x0048a1c2
$r1   : 0xbefffcd4  →  0xbefffdf4  →  "/home/pi/hello"
$r2   : 0xbefffcdc  →  0xbefffe03  →  "SHELL=/bin/bash"
$r3   : 0x000103f0  →  <main+0> push {r4,  lr}
$r4   : 0x0
$r5   : 0x00010400  →  <__libc_csu_init+0> push {r4,  r5,  r6,  r7,  r8,  r9,  r10,  lr}
$r6   : 0x000102e0  →  <_start+0> mov r11,  #0
$r7   : 0x0
$r8   : 0x0
$r9   : 0x0
$r10  : 0xb6ffd000  →  0x00030f44
$r11  : 0x0
$r12  : 0xbefffc00  →  0x00000001
$sp   : 0xbefffb80  →  0x00000000
$lr   : 0x000103f8  →  <main+8> pop {r4,  lr}
$pc   : 0x000103d2  →  <hello+2> eors r1,  r1
$cpsr: [negative ZERO CARRY overflow interrupt fast THUMB]
──────────────────────────────────────────────────────────────── stack ────
0xbefffb80│+0x0000: 0x00000000   ← $sp
0xbefffb84│+0x0004: 0xb6e7f718  →  <__libc_start_main+268> bl 0xb6e96780 <__GI_exit>
0xbefffb88│+0x0008: 0xb6fb2000  →  0x00149f10
0xbefffb8c│+0x000c: 0xbefffcd4  →  0xbefffdf4  →  "/home/pi/hello"
0xbefffb90│+0x0010: 0x00000001
0xbefffb94│+0x0014: 0x000103f0  →  <main+0> push {r4,  lr}
0xbefffb98│+0x0018: 0x7936d734
0xbefffb9c│+0x001c: 0x712eda68
─────────────────────────────────────────────────────── code:arm:THUMB ────
      0x103cb <__do_global_dtors_aux+39> movs   r2,  r0
      0x103cd <frame_dummy+1>  ;      <UNDEFINED> instruction: 0xffe6eaff
      0x103d1 <hello+1>        ldr    r0,  [pc,  #44]   ; (0x10400 <main+16>)
 →    0x103d3 <hello+3>        eors   r1,  r1
      0x103d5 <loop+1>         ldrb   r2,  [r0,  #0]
      0x103d7 <loop+3>         cmp    r2,  #0
      0x103d9 <loop+5>         beq.n  0x103e0 <endc>
      0x103db <loop+7>         adds   r1,  #1
      0x103dd <loop+9>         adds   r0,  #1
────────────────────────────────────────────────────── source:hello.s+22 ────
     17  */
     18
     19  @---------------------------------------------------------------------
     20  hello:
     21      ldr r0,=ins      @ dirección cadena de entrada
 →   22      eor r1, r1, r1  @ contador a 0
     23  loop:
     24      ldrb r2, [r0]
     25      cmp r2, # 0
     26      beq endc
     27
────────────────────────────────────────────────────────────── threads ────
[#0] Id 1, Name: "hello", stopped 0x103d2 in hello (), reason: SINGLE STEP
──────────────────────────────────────────────────────────────── trace ────
[#0] 0x103d2 → hello()
[#1] 0x103f8 → main()
────────────────────────────────────────────────────────────────────────────
gef➤
```

Figura 1.7. Analizando el programa de ejemplo utilizando GDB con la extensión GEF.

1.3 Ejercicios introductorios

En este apartado proponemos una serie de ejercicios sencillos orientados a familiarizarse con QtARMSim y a repasar algunos conceptos básicos relacionados con la representación y almacenamiento de información en la arquitectura ARM T32/Thumb. El apéndice B incluye, a modo de referencia rápida, información sobre las instrucciones del repertorio Thumb de 16 bits (cf. cuadros B.4 a B.8) y sobre las directivas del ensamblador de GNU utilizadas en este libro (cf. cuadro B.9). El apéndice A presenta estos mismos ejercicios para el caso de la Raspberry Pi.

Ejercicio 1.1

El siguiente programa inicializa los registros `r0` a `r3` con 4 valores numéricos en decimal, hexadecimal, octal y binario, respectivamente. Edita el programa, ensámblalo y responde a las preguntas.

e_intro_1.s

```
      .text
main: mov r0, #30
      mov r1, #0x42
      mov r2, #0102
      mov r3, #0b1000010
stop: wfi
      .end
```

1. ¿Cómo se muestran los números anteriores al desensamblar el programa en la pestaña ARMSim de QtARMSim? ¿Están en la misma base que en el código original? ¿En qué base están representados?

2. Ejecuta el programa paso a paso. ¿Qué números se almacenan en los registros `r0` a `r3`?

Solución

Los datos inmediatos en el código desensamblado se representan en decimal, con lo que al observar el código en la ventana de simulación obtendríamos algo como:

```
[0x00180000] 0x201E movs r0, #30
[0x00180002] 0x2142 movs r1, #66
[0x00180004] 0x2242 movs r2, #66
[0x00180006] 0x2342 movs r3, #66
[0x00180008] 0xBF30 wfi
```

Podemos inferir la representación en hexadecimal de los datos inmediatos a partir del formato de instrucción, ya que los datos inmediatos se codifican en la propia instrucción. En el ejemplo anterior, se corresponden con el segundo byte de la instrucción `movs`: `0x1E`, `0x42`, `0x42` y `0x42`.

En cuanto a la representación del contenido de los registros, se representan en hexadecimal. Así, al ejecutar el programa el contenido de los registros aparece como sigue en el panel de registros de QtARMSim:

```
r0   0x0000001E
r1   0x00000042
r2   0x00000042
r3   0x00000042
```

Ejercicio 1.2

Edita el código presentado a continuación, ensámblalo, analízalo y responde las preguntas.

e_intro_2.s

```
        .data
word1:  .word 11
word2:  .word 0x11
word3:  .word 011
word4:  .word 0b11

        .text
stop:   wfi
        .end
```

1. Identifica las posiciones de memoria donde se almacenaron los datos definidos en el programa. Identifica los cuatro valores en hexadecimal.

2. ¿En qué direcciones se almacenan los cuatro valores? ¿Por qué estas direcciones de memoria son múltiplos de cuatro en lugar de ser direcciones consecutivas?

3. ¿Cuáles son los valores de las etiquetas `word1`, `word2`, `word3` y `word4`?

Solución

Las posiciones de memoria donde se almacenan las 4 palabras definidas en el código anterior son las siguientes:

```
[0x20070000]  0x0000000B
[0x20070004]  0x00000011
[0x20070008]  0x00000009
[0x2007000C]  0x00000003
```

Cada palabra ocupa 4 bytes y el convenio de almacenamiento o *endianness* es el extremista menor (*little endian*), es decir, el byte menos significativo de la palabra se almacena en la posición de memoria de dirección más baja.

Por tanto, los valores de las etiquetas `word1`, `word2`, `word3` y `word4` serán los indicados a continuación:

Etiqueta	Valor
`word1`	`0x20070000`
`word2`	`0x20070004`
`word3`	`0x20070008`
`word4`	`0x2007000C`

Ejercicio 1.3

Ensambla el siguiente código:

e_intro_3.s

```
      .data
wrds: .word 11, 0x11, 011, 0b11
      .text
stop: wfi
      .end
```

¿Hay algún cambio en los valores almacenados en la memoria con respecto a los almacenados en el caso del programa anterior? ¿Están en el mismo lugar?

Solución

No hay ningún cambio en lo que respecta al almacenamiento de los datos anteriores. El único cambio sería que ahora definimos una única etiqueta (`wrds`) con valor `0x20070000`.

Ejercicio 1.4

Ensambla el siguiente código:

e_intro_4.s

```
      .data
bys:  .byte 0x18, 0x19, 0x1a, 0x1b
      .text
```

```
stop: wfi
      .end
```

1. ¿Qué valores se almacenaron en la memoria? ¿En qué posiciones?

2. ¿Cuál es el valor de la etiqueta `bys`?

   Solución

   Se almacena un byte en cada posición de memoria, comenzando en la dirección `0x20070000`. La etiqueta `bys` tomará dicho valor.

Ejercicio 1.5

Ahora ensambla el siguiente código:

```
      .data                                       e_intro_5.s
strg: .ascii "abác"
byte: .byte  0xff
      .text
stop: wfi
      .end
```

1. ¿Qué rango de posiciones de memoria se reservó para la variable etiquetada como `strg`?

2. ¿Cómo se representa en memoria la cadena `abác`? ¿Cómo se representa el carácter `á`?

3. Averigua cuál es el sistema de codificación de caracteres de tu entorno de trabajo (UTF-8, ASCII, ISO Latin 9 . . .).

4. ¿Cuál es el valor de la etiqueta `byte`?

   Solución

   Se trata de una cadena de caracteres. QtARMSim se basa en el ensamblador de GNU, que utiliza UTF-8 como sistema de representación de caracteres en la mayoría de las plataformas, siendo la excepción más relevante Microsoft Windows, donde el sistema de representación utilizado es ISO Latin 9 (ISO/IEC 8859-15).

   Por lo tanto, para la cadena etiquetada como `streg` se reservarán 5 posiciones en el caso de la mayoría de las plataformas excepto Windows y la representación en memoria byte a byte será:

```
0x61 0x62 0xc3 0xa1 0x63
```

En el caso de QtARMSim bajo Microsoft Windows, la representación será:

```
0x61 0x62 0xe1 0x63
```

y se reservarán 4 posiciones de memoria.

Vemos que el carácter `á` se representa con el código de dos bytes (`0xc3`, `0xa1`) en UTF-8, de acuerdo con lo descrito en el apéndice C, y como `0xe1` en el caso de ISO Latin 9.

Finalmente, el valor de la etiqueta `byte` será `0x2007005` o `0x2007004` en QtARMSim, dependiendo de que el sistema de representación de caracteres sea UTF-8 o ISO Latin 9.

Ejercicio 1.6

Ensambla y analiza el siguiente código:

e_intro_6.s

```
        .data
b1:     .hword 0x11
gap:    .space 4
b2:     .byte  0x22
bign:   .word  0x33445566
        .text
stop:   wfi
        .end
```

1. ¿Cuántas posiciones de memoria se reservan para la variable `gap`?
2. ¿Podrían leerse o escribirse los cuatro bytes utilizados por la variable `gap` como si fueran una palabra? ¿Por qué?
3. ¿Cuál es el valor de la etiqueta `b1`? ¿Y de la etiqueta `b2`?
4. ¿Cuál es el valor de la etiqueta `bign`? ¿Podrían leerse o escribirse los cuatro bytes que comienzan en la dirección `bign` como si fueran una palabra? ¿Por qué?
5. Agrega una directiva `.balign` al código anterior para que la variable etiquetada como `bign` se alinee con un límite de palabra (es decir, con una dirección múltiplo de cuatro).

Solución

Para la variable `gap` se reservan 4 posiciones de memoria. No podremos acceder en lectura o escritura a los 4 bytes de `gap` como si fueran una palabra porque los accesos a palabra en ARM tienen que estar alineados a una dirección múltiplo de 4.

Para poder acceder a `bign` como una palabra tendríamos que añadir una directiva de alineación `.balign 4` o bien `.align 2` antes de la definición de dicha etiqueta. En ambos casos se alinearía el espacio etiquetado por `bign` a una dirección múltiplo de 4.

El código es el siguiente:

e_intro_7.s

```
      .data
b1:   .hword  0x11       @ esto ocupa dos bytes
gap:  .space  4          @ cuatro bytes adicionales
b2:   .byte   0x22       @ un byte adicional (total 7 bytes)
      .balign 4          @ alineamos a un múltiplo de 4
bign: .word   0x33445566 @ ahora podemos definir una palabra

      .text
stop: wfi
      .end
```

Capítulo 2
Soltando amarras

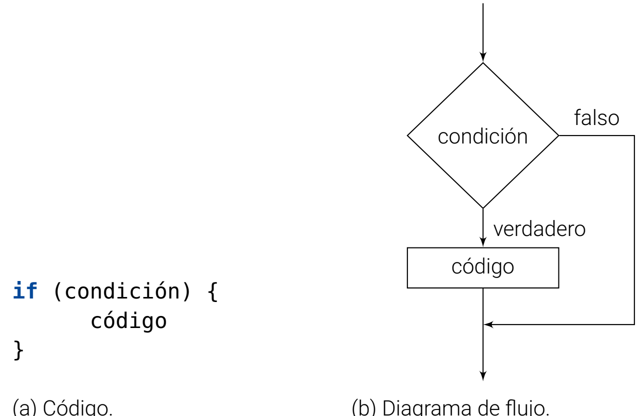
En este capítulo proponemos una serie de ejercicios orientados a la iniciación en la programación en ensamblador. Estos ejercicios están pensados para ayudar a comprender el funcionamiento de las instrucciones más básicas, así como la organización de los datos y su procesado en el nivel de máquina convencional, utilizando en nuestro caso como referencia la arquitectura ARM y el repertorio Thumb del núcleo ARM7TDMI.

Los primeros ejercicios del capítulo se dedican a presentar las estructuras básicas de programación `if-then`, `if-then-else`, `do-while` y `for`, y su programación en ensamblador. A continuación, se proponen algunos ejercicios donde se pide la codificación de programas que tienen como objetivo transferir información entre la memoria y los registros, y entre zonas de memoria, como por ejemplo intercambiar el contenido de una zona de memoria según diversos criterios, transferir de manera selectiva información de una zona de memoria a otra, mezclar los contenidos de varias zonas de memoria, o identificar contenidos en zonas de memoria.

2.1 Estructuras básicas de control

Los ejercicios siguientes tienen como objetivo presentar la programación en ensamblador de las cuatro estructuras de control más comunes en programación imperativa: `if-then`, `if-then-else`, `do-while` y `for`. Al menos una de estas estructuras aparecerá en prácticamente todos los ejercicios del libro.

Los cuatro ejercicios tienen una estructura similar. Primero se presenta la estructura de control, mediante pseudocódigo y mediante una descripción textual. A continuación, se propone un ejercicio concreto de programación en ensamblador, que también se ilustra mediante pseudocódigo. Finalmente, se presenta la solución al ejercicio mediante código Thumb de 16 bits.



(a) Código. (b) Diagrama de flujo.

Figura 2.1. Estructura de control `if-then`.

Ejercicio 2.1

En este ejercicio ilustramos la programación en ensamblador de la estructura de control `if - then` (cf. figura 2.1).

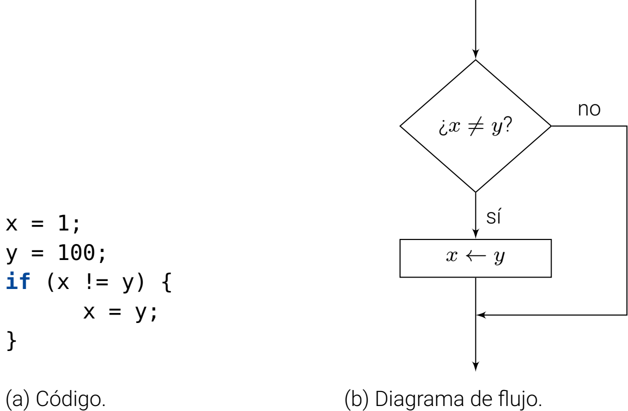
Esta estructura de control permite ejecutar cierto código según la evaluación de una condición simple, sea falsa o verdadera. Si la condición es verdadera, se ejecuta el bloque de código `código`; de lo contrario, no se ejecuta dicho código.

Para evaluar la condición en ensamblador, utilizamos instrucciones que actúan sobre los indicadores `Z, N, V, C`, y a continuación ejecutamos una instrucción de salto condicional que nos permita ignorar las instrucciones que forman parte de `código` en caso de que no se cumpla la condición.

Para ilustrar esta estructura de control, vamos a programar el ejemplo de la figura 2.2).

Solución

Básicamente, traducimos a lenguaje ensamblador el código de la figura 2.2). La comparación entre los dos valores de `x` e `y` la realizamos utilizando la instrucción **`cmp`**. Esta instrucción es equivalente a una instrucción de resta, solo que se modifican únicamente los indicadores de condición `N`, `Z`, `C` y `V` dependiendo del resultado de la operación (no se modifica ningún registro de datos). En caso de que `x` e `y` sean diferentes, es decir, en caso de que su resta sea distinta de cero, tras ejecutar la instrucción **`cmp`** el indicador `Z` tomará el valor 0 (es decir, el resultado de la operación no fue cero).



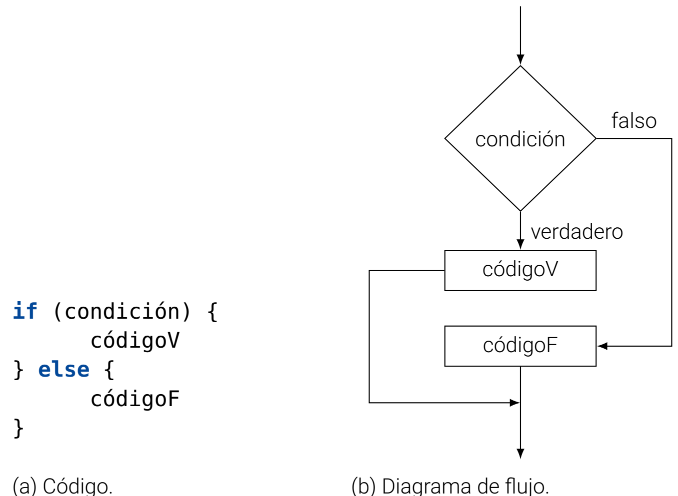
(a) Código. (b) Diagrama de flujo.

Figura 2.2. Ejemplo de estructura de control `if-then`.

Después, utilizamos la instrucción de salto condicional **beq** para elegir qué parte del código ejecutar en función de si se cumple la condición del `if` o no (cf. cuadro B.1 del apéndice B).

if-then.s

```
1         .data
2  x:     .word 1
3  y:     .word 100
4  
5         .text
6  
7  main:  ldr r0,=x
8         ldr r0,[r0]   @ r0 <- (x)
9         ldr r1,=y
10        ldr r1,[r1]   @ r1 <- (y)
11 
12        cmp r0,r1     @ condición: ¿x != y?
13        beq endif
14 @
15 @      código si se cumple la condición
16 @ {
17        ldr r3,=x
18        str r1,[r3]   @ (x) <- (y)
19 @ }
20 @      fin del código si se cumple la condición
21 @
22 endif:
23        wfi
24        .end
```




```
if (condición) {
        códigoV
} else {
        códigoF
}
```

(a) Código. (b) Diagrama de flujo.

Figura 2.3. Estructura de control `if-then-else`.

Ejercicio 2.2

En este ejercicio ilustramos la programación en ensamblador de la estructura de control `if - then - else` (cf. figura 2.3).

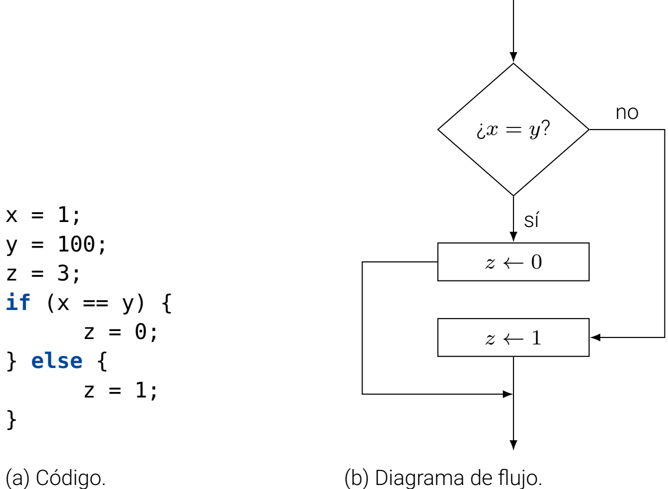
Esta estructura de control es una extensión de la estructura del ejercicio anterior. También permite ejecutar cierto código según la evaluación de una condición simple, sea falsa o verdadera. Si la condición es verdadera, se ejecuta el bloque de código `códigoV`; pero de lo contrario, se ejecuta el bloque de código `códigoF`.

Al igual que en el caso anterior, para evaluar la condición en ensamblador utilizamos instrucciones que actúan sobre los indicadores y a continuación ejecutamos una instrucción de salto condicional.

Para ilustrar esta estructura de control, vamos a programar el ejemplo de la figura 2.4).

Solución

Este ejercicio es muy similar al ejercicio anterior, solo que tendremos que ejecutar un bloque de código diferente en caso de que se cumpla la condición y en caso de que no se cumpla. Al igual que antes, la comparación entre los dos valores de `x` e `y` la realizamos utilizando la instrucción **`cmp`**. Después utilizamos la instrucción de salto condicional **`bne`** para elegir qué parte del código ejecutar en función de si se cumple la condición del `if` o no.




```
x = 1;
y = 100;
z = 3;
if (x == y) {
        z = 0;
} else {
        z = 1;
}
```

(a) Código.

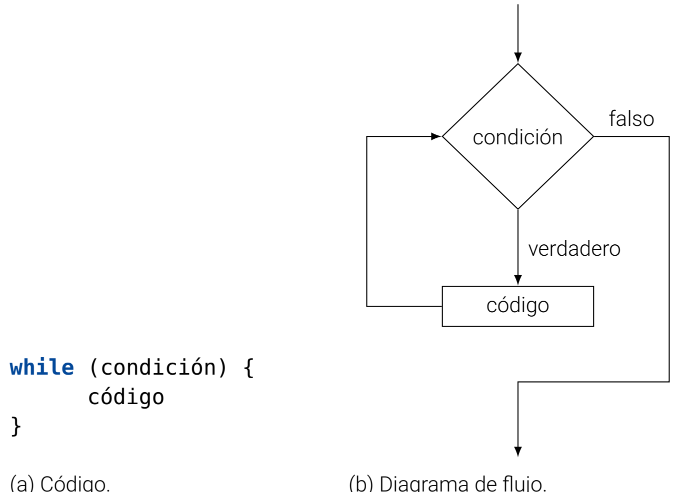
(b) Diagrama de flujo.

Figura 2.4. Ejemplo de estructura de control `if-then-else`.

Es decir, si los valores de `x` e `y` eran iguales, `z` tomará el valor 1 tras la ejecución de **`cmp`**, no se cumplirá la condición de salto **`bne`** y se ejecutarán las líneas de código que están a continuación de dicha instrucción de salto. Si no son iguales, `z` tomará el valor 0, se cumplirá la condición de salto y se ejecutará el código etiquetado como `else` (cf. cuadro B.1 del apéndice B).

`if-then-else.s`

```
1         .data
2   x:    .word 1
3   y:    .word 100
4   z:    .word 3
5
6         .text
7
8   main: ldr r0,=x
9         ldr r0,[r0]   @ r0 <- (x)
10        ldr r1,=y
11        ldr r1,[r1]   @ r1 <- (y)
12
13        cmp r0,r1     @ condición: ¿x = y?
14        bne else
15  @
16  @     código si se cumple la condición
17  @ {
18        mov r2,#0     @ r2 <- 0
19        b   endif
20  @ }
```



(a) Código. (b) Diagrama de flujo.

Figura 2.5. Estructura de control `while`.

```
21 @      código si no se cumple la condición
22 @ {
23 else:
24        mov r2,#1 @ r1 <- 1
25 @ }
26 @      código común a ambas opciones
27 @
28 endif:
29        ldr r3,=z
30        str r2,[r3] @ (z) <- r2 (un 0 o un 1)
31        wfi
32        .end
```

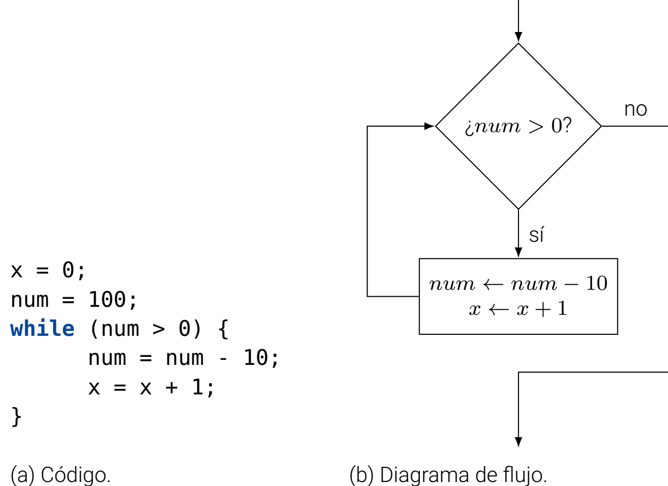
Ejercicio 2.3

En este ejercicio ilustramos la programación en ensamblador de la estructura de control `while` (cf. figura 2.5).

Esta estructura de control permite ejecutar cierto código mientras sea verdadera una condición simple. Si `condición` es verdadera, se ejecuta el bloque de código `código` y a continuación se vuelve a comprobar dicha condición. El bucle se repite hasta que `condición` sea falsa.

Como en los casos anteriores, para evaluar la condición en ensamblador utilizamos instrucciones que actúan sobre los indicadores `C`, `N`, `V` y `Z`, y a continuación ejecutamos una instrucción de salto condicional.

Para ilustrar esta estructura de control, vamos a programar el ejemplo de la figura 2.6.



(a) Código. (b) Diagrama de flujo.

Figura 2.6. Ejemplo de estructura de control `while`.

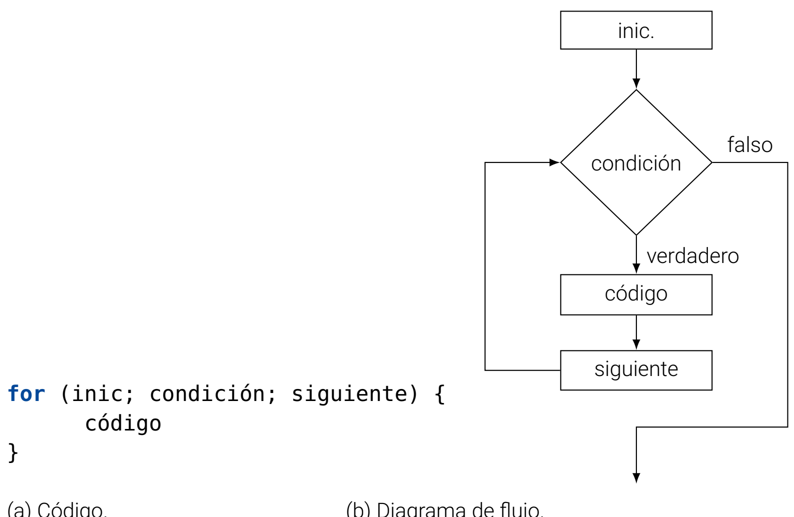
Solución

Al igual que en los ejercicios 2.1 y 2.2, utilizamos una instrucción **`cmp`** y una instrucción de salto condicional, en este caso **`ble`**, para tomar decisiones sobre el flujo de programa.

En el caso de esta estructura `while`, saldremos del bucle cuando `num` deje de ser mayor que 0. Para ello, comparamos `num` (almacenado en el registro `r1`) con cero. En el caso de que el resultado de la operación de resta asociada sea menor o igual que cero, o lo que es lo mismo, que el resultado no sea mayor que cero, la condición de salto de **`ble`** se hará efectiva y saldremos del bucle saltando a la posición etiquetada como `finb` (cf. cuadro B.1 del apéndice B).

while.s

```
1          .data
2  x:      .word 0
3  num:    .word 100
4
5          .text
6  main:
7          ldr r0,=x
8          ldr r0,[r0]   @ r0 <- (x)
9          ldr r1,=num
10         ldr r1,[r1]   @ r1 <- (num)
11         mov r2,#10
12
13 bucle:
14         cmp r1,#0     @ condición de fin
```



(a) Código. (b) Diagrama de flujo.

Figura 2.7. Estructura de control `for`.

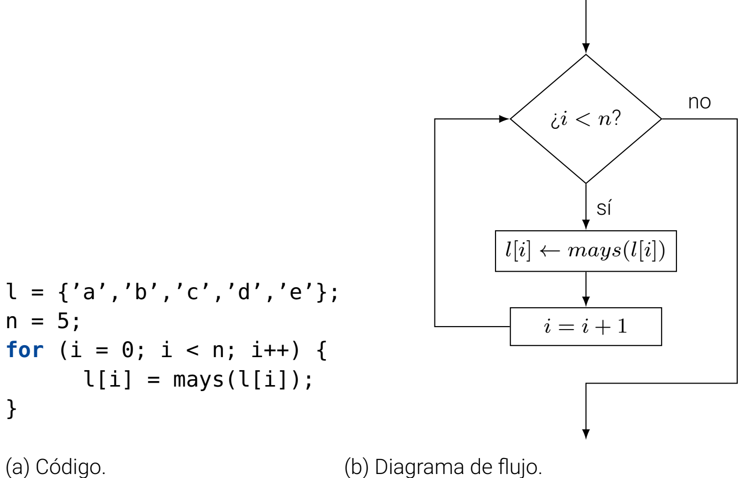
```
        ble finb     @ de bucle: ¿r1 <= 0?
@
@        contenido del bucle
@ {
        sub r1,r1,r2 @ r1 <- r1 - 10
        add r0,r0,#1 @ r0 <- r0 + 1
@ }
@       fin del contenido del bucle
@
        b bucle

finb:   ldr r2,=x    @ actualizamos los valores
        str r0,[r2]  @ de (x) y (num)
        ldr r2,=num  @ (x) <- r0
        str r1,[r2]  @ (num) <- r1
        wfi
        .end
```

Ejercicio 2.4

Finalmente, en este ejercicio ilustramos la programación en ensamblador de la estructura de control `for` (cf. figura 2.7).

Al igual que en el caso de la estructura `while`, la estructura `for` da lugar a un lazo o bucle, pero en este caso permite ejecutar un bloque de código un número determinado de veces.

1. Se evalúa la expresión `inic` dando como resultado un valor.



(a) Código.   (b) Diagrama de flujo.

Figura 2.8. Ejemplo de estructura de control `for`.

2. Se evalúa la expresión **`condición`**.

3. Si se cumple la condición evaluada, se ejecuta el bloque **`código`**. Si no, se termina el bucle.

4. Se evalúa la expresión **`siguiente`** y se vuelve al punto 2.

Como caso particular, esta estructura permite definir un bucle controlado por una variable desde un valor inicial hasta un valor final. Esta versión de la estructura es útil para procesar un conjunto indexado de valores, como por ejemplo un vector.

```
for (i = 0; i < límite; i = i + 1) {
      procesar elemento i-ésimo
}
```

En este caso, la expresión `i = 0` se corresponde con la expresión `inic` del caso general, la expresión `i < límite` con la expresión `condición` y la expresión `i = i + 1` con la expresión `siguiente`. El valor de la expresión `i < límite` nos va a servir de condición para seguir dentro del bucle o no. Se le da un valor inicial con `i = 0` (punto 1 anterior) y luego se va actualizando con `i = i + 1` (punto punto 4 anterior). Cuando se cumpla `i < límite` (punto 3 anteior), el bucle termina.

Para ilustrar esta estructura de control, vamos a realizar un programa que pasa a mayúsculas una lista de cinco caracteres ASCII (cf. figura 2.8).

Solución

Traducimos el código anterior a lenguaje ensamblador, utilizando una instrucción **cmp** y una instrucción de salto condicional **beq** para tomar decisiones sobre la terminación del bucle.

Las letras mayúsculas y minúsculas se diferencian únicamente en un bit. Para pasar a mayúsculas, ponemos a 0 ese bit mediante una máscara adecuada (`0xdf`) y una instrucción **and** (cf. ejercicio 3.14).

for.s

```
        .data
l:      .byte 'a', 'b', 'c', 'd', 'e'
        .balign 4
n:      .word 5

        .text
main:
        ldr r0,=l     @ r0 <- dir. de la lista
        ldr r1,=n
        ldr r1,[r1]   @ r1 <- (n) (tamaño de la lista)
        mov r2,#0     @ r2 <- 0 (índice del bucle for)
        mov r4,#0xdf  @ r4: máscara para pasar a mayúsculas

bucle:
        cmp r2,r1     @ condición de fin de
        beq finb      @ bucle: ¿índice = tamaño? ¿r2 = r1?
@
@       contenido del bucle
@ {
        ldrb r3,[r0,r2]  @ r3: cargamos un carácter
        and  r3,r3,r4    @ lo pasamos a mayúscula
        strb r3,[r0,r2]  @ lo almacenamos modificado.
@ }
@       fin del contenido del bucle
@
@       incrementamos el contador (índice = índice + 1)
@
        add r2,r2,#1
        b bucle

finb: wfi
        .end
```

2.2 Trasiego del contenido de la memoria

Una de las características esenciales de todos los procesadores RISC como los de la familia ARM es su arquitectura *load/store*. El repertorio de instrucciones se divide en dos grandes grupos:

- Acceso a la memoria, con instrucciones de carga (*load*) y almacenamiento (*store*) que se llevan a cabo entre la memoria y los registros.
- Operaciones aritméticas y lógicas, que únicamente son posibles entre valores almacenados en los registros.

Los ejercicios que se proponen a continuación tienen como objetivo practicar la transferencia de información desde la memoria principal del ordenador a los registros y viceversa, utilizando los modos de direccionamiento disponibles en el repertorio Thumb de 16 bits. Las instrucciones correspondientes a las operaciones aritméticas y lógicas se tratarán en el capítulo 3.

Ejercicio 2.5

Realizar un programa que intercambie una zona de memoria, utilizando el siguiente método:

> El método se basa en la comparación por cambio de elementos, ya que se van comparando de dos en dos los elementos (números enteros de 1 byte) de la zona de memoria. Dichos elementos se intercambiarán, si y solo si, tanto el primero como el segundo son menores o iguales que el contenido del registro `r7`.

Tras la ejecución del programa, la zona de memoria con los datos deberá continuar teniendo los mismos números enteros que antes de la ejecución del mismo. Obviamente, dichos números podrán estar en un orden diferente, pero no podrá faltar ni repetirse ninguno dentro de la citada zona.

Como ejemplo suponer que la zona de memoria contiene:

```
      .data
zona: .byte 2,1,7,3,5,6,4,9,8
```

Solución

Utilizamos el registro `r0` como registro índice a lo largo de la zona de memoria. En cada iteración, utilizamos el modo de direccionamiento indexado para obtener los dos números consecutivos a comparar, cuyas direcciones efectivas serán `r0` y (`r0 + 1`).

b_mem1.s

```
        .data
zona:   .byte 2,1,7,8,5,6,4,9,3
        .equ ult, 8          @ índice del último elemento
        .equ comp, 5         @ valor inicial de ejemplo para r7

        .text
main:   ldr r0, =zona        @ r0: puntero al principio
        mov r1, r0           @ de la zona de memoria
        add r1, r1, #ult     @ r1: puntero al final
        mov r7, #comp        @ inicializamos r7

buc:    ldrb r2, [r0]        @ vemos si este y el
        ldrb r3, [r0, #1]    @ siguiente son ambos
        cmp r7, r2           @ menores o iguales
        bmi sigo             @ que r7
        cmp r7, r3
        bmi sigo

        strb r2, [r0, #1]    @ si son, intercambiamos
        strb r3, [r0]

sigo:   add r0, r0, #1       @ apuntamos al siguiente
        cmp r0, r1           @ hasta terminar
        bne buc

        wfi
        .end
```

Ejercicio 2.6

Realizar un programa que intercambie una zona de memoria, utilizando el siguiente método:

> El método se basa en la comparación por cambio de elementos, ya que se van comparando de dos en dos los elementos (números enteros de 1 byte) de la zona de memoria. Dichos elementos se intercambiarán, si y solo si, el segundo de ellos es menor o igual que el elemento siguiente a sí mismo y en la anterior iteración no se había realizado un intercambio.

Tras la ejecución del programa, la zona de memoria con los datos deberá continuar teniendo los mismos números enteros que antes de la ejecución del mismo. Obviamente, dichos números podrán estar en un orden diferente, pero no podrá faltar ni repetirse ninguno dentro de la citada zona.

Como ejemplo suponer que la zona de memoria contiene:

```
        .data
zona: .byte 2, 1, 7, 8, 5, 6, 4, 9, 3
```

Solución

Utilizamos el registro `r1` como registro índice para navegar por la zona de memoria. En cada iteración, `r1` apuntará al primer número de los dos números a intercambiar, (`r1` + 1) al segundo de ellos y (`r1` + 2) al elemento siguiente al segundo, cuyo valor nos dirá si se ha de realizar el intercambio o no.

Además, utilizaremos el registro `r0` para indicar si en la anterior iteración realizamos un intercambio o no.

- `r0 = 0` indica que no se realizó un intercambio en la iteración anterior. Si en esta iteración realizamos un intercambio, pondremos además `r0 = 1`.
- Si `r0 = 1` significa que en la iteración anterior hicimos un intercambio, y por lo tanto en esta iteración simplemente haremos `r0 = 0` y no haremos nada más aparte de actualizar `r1` para apuntar al siguiente elemento.

b_mem2.s

```
1        .data
2  zona: .byte   2, 1, 7, 8, 5, 6, 4, 9, 3
3        .equ    penult, 7
4
5        .text
6  main: mov    r0, #0          @ r0: señala si hubo cambio
7        ldr    r1,=zona        @ r1: puntero a la zona de memoria
8        mov    r2, r1
9        add    r2, r2, #penult @ r2: puntero al penúltimo número
10                              @ (último a comprobar)
11 bucle:
12       cmp    r0, #0
13       bne    otro            @ si r0 es cero, no hubo cambio
14       ldrb   r7, [r1, #1]
15       ldrb   r6, [r1, #2]
16       cmp    r7, r6          @ comprobamos la condición
17       bgt    nocam           @ del enunciado
18
19       ldrb   r5, [r1, #0]    @ realizamos el intercambio
20       strb   r5, [r1, #1]
21       strb   r7, [r1, #0]
22       mov    r0, #1          @ y marcamos el cambio
23
```

```
24  nocam:
25        add   r1, #1          @ apuntamos al siguiente
26        cmp   r1, r2          @ y vemos si hemos terminado
27        bne   bucle
28        wfi
29
30  otro: mov   r0, #0          @ reseteamos el cambio
31        b     nocam
32        .end
```

Ejercicio 2.7

Realizar un programa que intercambie una zona de memoria, utilizando el siguiente método:

> El método se basa en la comparación por cambio de elementos (números enteros de 1 byte) de la zona de memoria. Supuestos los elementos ordenados $X1|X2|X3$. Se intercambiarán los elementos $X1$ y $X3$ si y solo si, $X1 < X2 < X3$.

En sucesivas iteraciones, los números que ocupan la posición X2, pasarán a ocupar la posición $X1$, los que ocupan la $X3$ pasarán a ocupar la $X2$, y por último los que ocupan la posición siguiente a la $X3$ pasarán a ocupar la $X3$.

Tras la ejecución del programa, la zona de memoria considerada deberá continuar teniendo los mismos números enteros que antes de la ejecución del mismo. Obviamente, dichos números podrán estar en un orden diferente, pero no podrá faltar ni repetirse ninguno dentro de la citada zona.

Como ejemplo suponer que la zona de memoria contiene:

```
      .data
zona: .byte 2,1,7,3,5,6,4,8,9
```

Solución

Al igual que en los ejercicios anteriores, utilizaremos el registro `r1` como puntero a los elementos en la zona de memoria. En cada iteración `r1` apuntará a $X1$, (`r1` + 1) apuntará a $X2$ y (`r1` + 2) apuntará a $X3$. Inicialmente, `r1` apuntará al primer elemento de la zona de memoria y se irá incrementando en cada iteración hasta llegar al final.

b_mem3.s

```
1        .data
2  zona: .byte      2, 1, 7, 3, 5, 6, 4, 8, 9
3        .equ       penult, 7
4
```

```
5         .text
6   main: ldr   r1,=zona          @ r1: puntero al primer dato
7         mov   r2, r1
8         add   r2,r2,#penult     @ r2: puntero al penúltimo elemento
9
10  bucle:
11        cmp   r1, r2            @ comprobamos si hemos terminado
12        beq   final
13
14        ldrb  r7, [r1, #0]      @ r7 <- X(i)
15        ldrb  r6, [r1, #1]      @ r6 <- X(i+1)
16        cmp   r7, r6            @ ¿X(i) < X(i+1)?
17        bge   esteno
18
19        ldrb  r5, [r1, #2]      @ r5 <- X(i+2)
20        cmp   r6, r5            @ ¿X(i+1) < X(i+2)?
21        bge   esteno
22
23        strb  r5, [r1, #0]      @ intercambiar
24        strb  r7, [r1, #2]      @ porque X(i) < X(i+1) < X(i+2)
25
26  esteno:
27        add   r1, #1            @ apuntamos al siguiente
28        b     bucle
29
30  final:
31        wfi
32        .end
```

Ejercicio 2.8

Considere una zona de memoria que contiene 10 números enteros de 1 byte. Realice un programa que, tomando dichos números agrupados de 3 en 3 ($X1$, $X2$, $X3$), intercambie los números $X1$ y $X3$ si y solo si $X1$ es mayor o igual que $X3$.

Utilice el siguiente código como ejemplo de la zona de memoria:

```
      .data
zona:
      .byte 2, 14, 1, 13, 6, 15, 8, 7, 16, 11
```

Solución

Este ejercicio tiene una estructura similar al ejercicio 2.7. Cambian las condiciones para realizar el intercambio de los números en cada iteración.

b_mem4.s

```
        .data
zona:  .byte 2, 14, 1, 13, 6, 15, 8, 7, 16, 11
        .equ  penult, 8

        .text
main:  ldr   r1,=zona      @ r1: puntero al primer dato
        mov   r2, r1
        add   r2,r2,#penult @ r2: puntero al penúltimo dato

bucle:
        ldrb  r7, [r1, #0]  @ r7 <- X(i)
        ldrb  r5, [r1, #2]  @ r5 <- X(i+2)
        cmp   r7, r5        @ si X(i) >= X(i+2) cambiar
        blt   esteno

        strb  r7, [r1, #2]  @ intercambio
        strb  r5, [r1, #0]

esteno:
        add   r1, #1        @ apuntamos al siguiente
        cmp   r1, r2        @ y vemos si acabamos
        bne   bucle

        wfi
        .end
```

Ejercicio 2.9

Considere una zona de memoria que contiene 10 números enteros positivos de 1 byte.

Realizar un programa que haga las siguientes operaciones:

- Tomando los números agrupados de 3 en 3 ($X1$, $X2$, $X3$), intercambiar los números $X1$ y $X3$ si y solo si $X1$ es menor que $X3$.
- Tras esto, guardar el último número, es decir, el que ocupa la posición 10 de la zona de memoria, en la dirección de memoria etiquetada como `ultimo`.

Utilizar el siguiente código como ejemplo de la zona de memoria:

```
      .data
zona:
      .byte 2, 14, 1, 13, 6, 15, 8, 7, 16, 11
ultimo:
      .space      1
```

Solución

Ejercicio estructuralmente similar al ejercicio 2.8. Cambia la condición de intercambio (en este caso si $X1 < X3$ en vez de $X1 \geq X3$) y antes de terminar se almacena el último número en la posición indicada.

b_mem5.s

```
1        .data
2  zona: .byte      2, 14, 1, 13, 6, 15, 8, 7, 16, 11
3  ultimo:
4        .space  1
5        .equ    penult, 8
6
7        .text
8  main: ldr    r1,=zona      @ r1: puntero al primer dato
9        mov    r2, r1
10       add    r2,r2,#penult @ r2: puntero al penúltimo dato
11       ldr    r3,=ultimo
12
13 bucle:
14       ldrb   r7, [r1, #0]  @ r7 <- X(i)
15       ldrb   r5, [r1, #2]  @ r5 <- X(i+2)
16       cmp    r7, r5        @ si X(i) < X(i+2) cambiar
17       bge    esteno
18
19       strb   r7, [r1, #2]  @ intercambio
20       strb   r5, [r1, #0]
21
22 esteno:
23       add    r1, #1        @ apuntamos al siguiente
24       cmp    r1, r2        @ y vemos si acabamos
25       bne    bucle
26
27       ldrb   r6, [r1, #1]  @ antes de terminar, almacenamos
28       strb   r6, [r3]      @ el último elemento en la
29                            @ posición indicada.
30       wfi
31       .end
```

Capítulo 3
Velocidad de crucero

Una vez que conocemos la programación en ensamblador de las estructuras de control más habituales y las diferentes maneras de transferir información utilizando la memoria y los registros, dedicamos este capítulo al procesado de dicha información. Comenzamos con las operaciones más básicas, como contar y comparar números, para introducir a continuación el procesado de cadenas de caracteres. Finalmente, proponemos un conjunto de problemas centrados en la utilización combinada de las diferentes instrucciones aritméticas, lógicas y de desplazamiento.

3.1 Contar y sumar

Esta sección propone un conjunto de ejercicios para practicar la realización de operaciones básicas entre los valores almacenados en los registros. Los ejercicios se conciben de manera incremental, es decir, seguimos reforzando la experiencia adquirida en el capítulo anterior relativa a la carga y almacenamiento de valores en los registros y la complementamos con la realización de operaciones básicas.

Ejercicio 3.1

Realizar un programa que sume los números enteros situados en una zona de memoria determinada, excepto aquel que se encuentre en la posición de memoria etiquetada como `este_no`. El resultado de la suma se almacenará en el registro `r4`.

Tras la ejecución del programa, la zona de memoria con los datos deberá continuar teniendo los mismos números enteros que antes de la ejecución del mismo y en el mismo orden.

La zona de memoria contendrá 9 números enteros, y por lo tanto la posición de memoria etiquetada como `este_no` sólo podrá contener valores entre 1 y 9, ambos inclusive.

Como ejemplo, suponer que la zona de memoria contiene los valores siguientes:

```
         .data
zona:    .byte 2,1,7,8,5,6,4,9,3
este_no: .byte 3
```

Solución

El cometido del ejercicio es leer los bytes de la zona de memoria y sumarlos uno tras otro, omitiendo el indicado por la posición de memoria etiquetada como `este_no`.

Para ello, utilizamos el direccionamiento relativo a registro con desplazamiento, donde el registro `r0` apunta al principio de la zona y el registro `r1` contiene el desplazamiento o índice desde el principio de la zona. La dirección efectiva de cada dato será entonces `r0` + `r1`.

Además, utilizamos el registro `r2` para comprobar si hemos terminado de sumar todos los números y el registro `r7` para guardar el índice del elemento que tenemos que omitir.

b_cs1.s

```
         .data
zona:    .byte 2,1,7,8,5,6,4,9,3
este_no: .byte 3
         .equ tam, 9 @ tamaño de la zona

      .text
main: ldr  r0, =zona     @ r0: puntero a la zona
      mov  r1, #0        @ r1: índice en la zona
      mov  r2, #tam      @ r2: tamaño de la zona
      ldr  r7, =este_no
      ldrb r7, [r7]      @ r7: posición a omitir
      mov  r4, #0        @ r4: acumulador para las sumas

bucle:
      cmp  r1, r7        @ vemos si este se suma o no
      beq  esteno        @ si el índice en r1 es distinto
      ldrb r3, [r0, r1]  @ del valor en r7, sumamos
      add  r4, r4, r3    @ este número al resto

esteno:
      add  r1, r1, #1    @ apuntamos al siguiente
      cmp  r1, r2        @ vemos si hemos acabado
      bne  bucle
      wfi
      .end
```

Ejercicio 3.2

Realizar un programa que sume los números enteros de 1 byte que se encuentran en unas posiciones de memoria determinadas, excepto aquellos que sean menores que el contenido de la posición de memoria etiquetada como `lim`.

Los números enteros se encuentran en las posiciones de memoria indicadas por una tabla de direcciones.

El resultado final deberá guardarse en el registro `r4`.

Tras la ejecución del programa, la zona de memoria considerada deberá continuar teniendo los mismos números enteros que antes de la ejecución del mismo y en el mismo orden.

Como ejemplo suponer que la tabla de direcciones (etiquetada como `tabla`) donde están almacenados los números se define de la siguiente manera:

```
      .data
b1:   .byte 2
b2:   .byte 8
b3:   .byte 5
b4:   .byte 1
b5:   .byte 3
b6:   .byte 9
b7:   .byte 7
b8:   .byte 0
b9:   .byte 6
      .balign 4
tabla:
      .word b1, b2, b3, b4, b5, b6, b7, b8, b9
```

Solución

En este caso, la tabla de datos no contiene los números a sumar, sino las direcciones de dichos números. Para leer cada número a sumar, tendremos que acceder dos veces a la memoria.

- La primera vez, leeremos una dirección de la tabla de direcciones `tabla`. Para ello utilizaremos el registro `r0` como puntero. Lo inicializaremos con la dirección de comienzo de la tabla y lo iremos incrementando de 4 en 4 unidades en cada iteración (las direcciones en ARM ocupan 4 posiciones de memoria). Almacenaremos en `r1` la dirección del dato leída de la tabla.

- La segunda vez, utilizaremos la dirección obtenida en el paso anterior que tenemos en el registro `r1` para leer el dato de 1 byte en sí, que almacenaremos también en `r1`.

Luego, simplemente nos queda ir sumando los bytes leídos, siempre que su valor sea mayor que el indicado por la posición de memoria etiquetada como `lim`. Para ello, cargaremos dicho valor en el registro `r3` y lo compararemos con el dato en `r1`.

Utilizaremos también el registro `r2` para comprobar si hemos terminado de sumar todos los números. Para ello lo inicializaremos con el tamaño de la tabla, almacenado en la dirección de memoria etiquetada como `size`.

b_cs2.s

```
1         .data
2  b1:    .byte   4
3  b2:    .byte   8
4  b3:    .byte   5
5  b4:    .byte   1
6  b5:    .byte   3
7  b6:    .byte   9
8  b7:    .byte   7
9  b8:    .byte   0
10 b9:    .byte   6
11
12 lim:   .byte   4 @ límite para no sumar
13        .balign 4
14 size:  .word   9 @ tamaño de la tabla
15 tabla:
16        .word   b1, b2, b3, b4, b5, b6, b7, b8, b9
17
18
19     .text
20 main:
21     ldr  r3, =lim
22     ldrb r3, [r3]   @ r3: limite de las sumas
23
24     ldr  r2, =size
25     ldr  r2, [r2]   @ r2: tamaño de la zona de contador
26
27     ldr  r0, =tabla @ r0: puntero a la tabla de punteros
28     mov  r4, #0     @ r4: acumulador de sumas
29 bucle:
30     cmp  r2, #0     @ vemos si final de tabla
31     beq  final
32     sub  r2, r2, #1 @ actualizamos el contador
33
34     ldr  r1, [r0]   @ r1: puntero en la tabla de direcciones
35     ldrb r1, [r1]   @ leemos el byte al que apunta el puntero
36     cmp  r1, r3
37     blt  otro       @ dato < r3: saltamos la suma
38     add  r4, r4, r1 @ actualizamos la suma
```

```
39 otro:
40    add  r0, r0, #4 @ actualizamos el puntero a la tabla
41    b    bucle
42
43 final:
44    wfi
45    .end
```

Ejercicio 3.3

Realizar un programa que sume los números enteros que se encuentran en una zona de memoria determinada, excepto aquellos que:

- Sean mayores que el contenido de la posición de memoria etiquetada como `lim`.
- Se encuentren en posiciones de memoria mayores o iguales que la dirección almacenada en la posición de memoria etiquetada como `desde`.

Los números enteros de 1 byte se encuentran en las direcciones de memoria indicadas por una tabla de direcciones de memoria, etiquetada como `tabla`, y el resultado final deberá guardarse en la pila.

Tras la ejecución del programa, la zona de memoria considerada deberá continuar teniendo los mismos números enteros que antes de la ejecución del mismo y en el mismo orden.

Para realizar el ejercicio, utilice la información siguiente:

```
       .data
b1:    .byte   2
b2:    .byte   8
b3:    .byte   5
b4:    .byte   1
b5:    .byte   3
b6:    .byte   9
b7:    .byte   7
b8:    .byte   0
b9:    .byte   6

lim:   .byte   4
       .balign 4
tabla: .word   b1, b2, b3, b4, b5, b6, b7, b8, b9
desde: .word   b3
```

Solución

Este ejercicio es una combinación de los dos ejercicios anteriores.

b_cs3.s

```
        .data
b1:     .byte   2   @ tabla de valores
b2:     .byte   8
b3:     .byte   5
b4:     .byte   1
b5:     .byte   3
b6:     .byte   9
b7:     .byte   7
b8:     .byte   0
b9:     .byte   6

lim:    .byte   4   @ índice del dato a omitir
        .balign 4

tabla: .word   b1, b2, b3, b4, b5, b6, b7, b8, b9

desde: .word   b3  @ posición desde la cual simar
size:  .word   9   @ tamaño de la tabla

    .text
main:
   ldr  r3, =lim
   ldrb r3, [r3]    @ r3: limite para sumar el número
                    @ debe ser mayor estricto que r3
   ldr  r2, =size
   ldr  r2, [r2]    @ r2: tamaño de la zona contador

   ldr  r0, =tabla @ r0: puntero a la tabla de punteros
   mov  r5, #0      @ r5: suma


   ldr  r4, =desde @ r4: metemos la dirección a partir
   ldr  r4, [r4]    @ de la cual sumamos

bucle:
   cmp  r2, #0       @ vemos si recorrimos toda la tabla
   beq  final
   sub  r2, r2, #1   @ actualizamos el contador

   ldr  r1, [r0]     @ r1: puntero de la tabla
   cmp  r1, r4       @ comprobamos si dir. del dato >= r4
   blt  otro         @ (dirección a partir de la cual sumamos)

   ldrb r1, [r1]     @ byte al que apunta el puntero

   cmp  r1, r3
   bls  otro         @ si dato <= r3 saltamos la suma
   add  r5, r5, r1   @ actualizamos la suma
otro:
```

```
50     add  r0, r0, #4  @ actualizamos el puntero a la tabla
51     b    bucle
52
53  final:
54     push {r5}
55     wfi
56     .end
```

Ejercicio 3.4

Considere una zona de memoria etiquetada como `tabla`, que denominaremos *zona de punteros* y que contiene 3 *punteros*, es decir, las direcciones de memoria en las que se encuentran almacenados 3 números que consideraremos *datos*.

Realizar un programa que sume aquellos datos que se encuentren en direcciones de memoria mayores o iguales que el dato que contienen, guardando el resultado de la suma en el registro `r4`.

Para realizar el ejercicio, utilice la información siguiente:

```
      .data
d1:   .word 0x000023f0
d2:   .word 0x34234323
d3:   .word 0x0000dc0f

tabla:
      .word d1, d2, d3
```

Solución

En este ejercicio también partimos de una tabla de direcciones (punteros) que nos indican las posiciones de memoria donde están almacenados los datos. Utilizaremos el registro `r1` para navegar por la tabla, el registro `r2` para almacenar las direcciones de los datos leídas de la tabla y el registro `r3` para almacenar los datos en sí leídos a partir de la dirección en `r2`.

La condición para sumar números que aparece en el enunciado, teniendo en cuenta los registros utilizados, será entonces que `r2` >= `r3`, es decir, que la dirección de memoria sea mayor o igual que el dato contenido en ella.

b_cs4.s

```
1  .data
2
3  @ Almacenamos los datos en memoria
4
```

```
d1:    .word   0x000023f0 @ sumamos este
d2:    .word   0x34234323
d3:    .word   0x0000dc0f @ y este. Total 0x0000ffff

@ A continuación creamos la tabla de direcciones

tabla:
   .word   d1, d2, d3
   .equ    tam, 12      @ tabla de 3 x 4 = 12 posiciones

   .text
main:
   ldr    r1, =tabla  @ r1: puntero a la tabla de punteros
   mov    r0, #3      @ r0: contador
   mov    r4, #0      @ r4: acumulador de sumas

bucle:
   cmp    r0, #0      @ comprobamos si hemos terminado
   beq    final
   ldr    r2, [r1]    @ r2: puntero de la tabla de punteros
   ldr    r3, [r2]    @ r3: dato al que se apunta desde la tabla
   cmp    r2, r3      @ comparamos el dato con la dirección
   bls    otro        @ si dir < dato no sumamos
   add    r4, r4, r3  @ si dir >= dato lo sumamos
otro:
   add    r1, r1, #4  @ actualizamos puntero a la tabla
   sub    r0, r0, #1  @ actualizamos el contador
   b    bucle

final:
   wfi
   .end
```

3.2 Comparar números

De acuerdo con nuestro enfoque incremental, añadimos a nuestro repertorio de actividades un conjunto de ejercicios centrados en la comparación de valores en registros y en la toma de decisiones en función de los resultados de dicha comparación.

Ejercicio 3.5

Considere una zona de memoria que contiene números enteros positivos de un byte.

Realizar un programa que haga las siguientes operaciones, teniendo en cuenta que consideraremos los números de la zona de memoria en grupos de 3, que denominaremos X1, X2 y X3:

- Si X1 es igual a X2 y distinto de X3 entonces se sumará el valor de X1 al registro `r0`.
- Si X1 es distinto de X2 e igual a X3 entonces el valor de X1 se sumará al registro `r2`.
- Si X1 = X2 = X3 se terminará el programa.
- En cualquier otro caso se continúa con la iteración siguiente del programa.

En sucesivas iteraciones, los números que ocupan la posición X2, pasarán a ocupar la posición X1, los que ocupan la X3 pasarán a ocupar la X2, y por último los que ocupan la posición siguiente a la X3 pasarán a ocupar la X3. Los registros `r0` y `r2` deben inicializarse con el valor 0.

Utilizar el siguiente código como ejemplo de la zona de memoria (hay que tener en cuenta que la cantidad de números enteros en la zona de memoria no se conoce a priori, aunque obviamente dicha cantidad deberá ser mayor que 2):

```
        .data
zona:
        .byte 5, 5, 3, 5, 1, 5, 5, 5
```

Solución

Utilizaremos el registro `r1` como puntero para navegar por la zona de memoria. Para acceder a los tres valores identificados como X1, X2 y X3 en el enunciado, utilizaremos el direccionamiento indirecto a registro con desplazamiento utilizando como desplazamientos los valores 0 para X1, 1 para X2 y 2 para X3. En cada iteración, iremos incrementando el valor de `r1` en una unidad, ya que los datos almacenados en la zona de memoria son bytes.

Utilizaremos también los registros `r5`, `r6` y `r7` para almacenar los datos y hacer las comparaciones oportunas indicadas por el enunciado.

La zona de memoria no tiene un tamaño predeterminado, sino que el final de la zona se identifica porque hay tres valores iguales consecutivos.

b_cn1.s

```
      .data
zona: .byte   5, 5, 3, 5, 1, 5, 5, 5

      .text
main:
   ldr   r1,=zona  @ puntero a la zona de memoria
   mov   r0, #0    @ acumulador de la suma 1
   mov   r2, #0    @ acumulador de la suma 2

bucle:
   ldrb  r7, [r1, #0] @ r7 <- X(i)
   ldrb  r6, [r1, #1] @ r6 <- X(i+1)
   ldrb  r5, [r1, #2] @ r5 <- X(i+2)
   cmp   r7, r6
   bne   dist12

   cmp   r7, r5
   beq   fin

   add   r0, r0, r7   @ X(i) = X(i+1) y X(i) != X(i + 2)
itera:
   add   r1, r1, #1
   b     bucle

dist12:
   cmp   r7, r5
   bne   itera

   add   r2, r2, r7   @ X(i) != X(i+1) u X(i) = X(i+2)
   b     itera

fin:
   wfi                @ X(i) = X(i+1) = X(i + 2)
   .end
```

Ejercicio 3.6

Considere una zona de memoria que contiene una serie de números enteros positivos de un byte.

Realizar un programa que haga las siguientes operaciones, teniendo en cuenta que consideraremos los números de la zona de memoria en grupos de 3, que denominaremos X1, X2 y X3:

- Si X1 = X2 = X3 se sumará el valor de X1 al registro `r0` y se continúa con la iteración siguiente del programa.
- Si X1 es distinto de X2 y X3 se terminará el programa.

- En cualquier otro caso se sumará el valor de X2 y el de X3 al registro `r2` y se continúa con la iteración siguiente del programa.

En iteraciones sucesivas X1 será el que en la iteración anterior era X2, X2 el que en la iteración anterior era X3, y X3 el que en la iteración anterior era el siguiente a X3.

Utilizar el siguiente código como ejemplo de la zona de memoria (hay que tener en cuenta que la cantidad de números enteros en la zona de memoria no se conoce a priori, aunque obviamente dicha cantidad deberá ser mayor que 2):

```
      .data
zona: .byte 5, 5, 5, 3, 5, 5
```

Solución

En cuanto a su estructura, este ejercicio es similar al ejercicio anterior. Aumenta ligeramente la complejidad en cuanto a comprobar la condición de terminación y las tareas a realizar con los tres valores consecutivos X1, X2 y X3.

b_cn2.s

```
1      .data
2   zona:   .byte   5, 5, 5, 3, 5, 5
3
4      .text
5   main:
6      ldr    r1,=zona    @ puntero a la zona de memoria
7      mov    r0, #0      @ acumulador para la suma 1
8      mov    r2, #0      @ acumulador para la suma 2
9
10  bucle:
11     ldrb   r7, [r1, #0] @ r7 <- X(i)
12     ldrb   r6, [r1, #1] @ r6 <- X(i+1)
13     ldrb   r5, [r1, #2] @ r5 <- X(i+2)
14     cmp    r7, r6
15     beq    van2
16
17     cmp    r7, r5       @ X(i) != X(i+1)
18     bne    fin
19
20  otro:
21     add    r2, r2, r6   @ cualquier otro caso
22     add    r2, r2, r5
23     b      itera
24
25  van2:   cmp   r7, r5
```

```
26      bne   otro
27
28      add   r0, r0, r7   @ X(i) = X(i+1) = X(i + 2)
29   itera:
30      add   r1, r1, #1
31      b     bucle
32
33   fin:
34      wfi                @ X(i) != X(i+1) y  X(i) != X(i+2)
35      .end
```

Ejercicio 3.7

Considere una zona de memoria que contiene una serie de números enteros positivos de un byte.

Realizar un programa que haga las siguientes operaciones:

- Coger un número de la zona de memoria, y en las sucesivas iteraciones, en caso de haberlas, los siguientes.
- Si ese número (al cual llamaremos X) es menor que el siguiente (al cual llamaremos Y) el programa debe intercambiar ambos números y continuar con una nueva iteración, quedando guardado en el registro `r5` el número de intercambios realizados hasta el momento.
- Si X es igual a Y y al número de intercambios realizados hasta el momento, el programa debe terminar.
- Si no se cumplen ninguna de las condiciones anteriores el programa debe sumar X e Y al contenido del registro `r2` y continuar con una nueva iteración.

En cada nueva iteración, Y pasará a ser X, y el número siguiente a Y pasará a ser Y.

Utilizar el siguiente código como ejemplo de la zona de memoria (hay que tener en cuenta que la cantidad de números enteros en la zona de memoria no se conoce a priori, aunque obviamente dicha cantidad deberá ser mayor que 1):

```
      .data
zona: .byte 3, 5, 3, 2, 7, 2, 1, 5
```

Solución

En este caso, además de realizar operaciones con valores los consecutivos de una tabla de números, tendremos que intercambiar algunos de dichos valores.

Al igual que en los ejercicios anteriores, utilizaremos el direccionamiento indirecto a registro con desplazamiento para navegar por la tabla, con el registro `r0` como registro índice.

b_cn3.s

```
    .data
zona:   .byte   3, 5, 3, 2, 7, 2, 1, 5

    .text
main:
    ldr    r0,=zona     @ r0: puntero a la zona
    mov    r5, #0       @ r5: contador de intercambios
    mov    r2, #0       @ r2: acumulador de números

bucle:
    ldrb   r3, [r0]     @ cargamos el numero X en r3
    ldrb   r4, [r0, #1] @ y el siguiente (Y) en r4
    cmp    r3, r4       @ comprobamos si X = Y
    blt    inter        @ X < Y: intercambiar
    bne    seguir

    cmp    r3, r5       @ comprobamos la condición
    bne    seguir       @ de terminación (X = Y)

    wfi                 @ números iguales, e iguales
                        @ al número de intercambios
inter:
    add    r5, r5, #1   @ un intercambio más, ya que
    strb   r3, [r0, #1] @ es menor: intercambiamos
    strb   r4, [r0]

otro:   add r0,r0,#1 @ incrementamos el puntero
    b      bucle

seguir:
    add    r2, r2, r3   @ es mayor: sumamos X e Y
    add    r2, r2, r4
    b      otro
    .end
```

Ejercicio 3.8

Considere una zona de memoria que contiene una serie de números enteros de 16 bits mayores o iguales que cero.

Realizar un programa que haga las siguientes operaciones:

- Tomar un número de la zona de memoria. A ese número le denominaremos X.
- Si X = 0 el programa termina.
- Si X es mayor que el contenido del registro `r7` el programa debe guardar el contenido del registro `r7` en X (es decir, en la posición de memoria donde está X), contabilizar que se ha realizado un cambio y continuar con la siguiente iteración.
- Si X es menor o igual que el contenido del registro `r7` el programa debe guardar el contenido del registro `r2` en el registro `r7` y terminar.

En el registro `r2` se contabilizará el número de cambios realizados, es decir, el número de veces que se ha guardado el contenido del registro `r7` en la dirección de memoria donde se encontraba X.

En cada nueva iteración, el número siguiente a X pasará a ser X. El registro `r2` debe inicializarse con el valor 0 y el registro `r7` con el valor 3.

Utilizar el siguiente código como ejemplo de la zona de memoria. Hay que tener en cuenta que la cantidad de números enteros en la zona de memoria no se conoce a priori.

```
      .data
zona: .hword      7, 5, 3, 1, 2, 4, 0
```

Solución

Este ejercicio es semejante al anterior, solo que los datos ocupan 16 bits (medias palabras) y las operaciones a realizar son un poco más complejas.

En este caso, las operaciones de carga y almacenamiento se referirán a medias palabras (**`ldrh`**, **`strh`**) en vez de a bytes y tendremos que incrementar el puntero para navegar la tabla (el registro `r0` en este caso) en dos unidades en cada iteración.

b_cn4.s

```
1
2         .data
3   zona: .hword   7, 5, 3, 1, 2, 4, 0
4
5         .equ   v_comp, 2
6
7      .text
```

```
8  main:
9     ldr   r0, =zona   @ r0: puntero al inicio de la zona
10
11    mov   r7, #v_comp @ r7: valor para comparar
12    mov   r2, #0      @ r2: número de cambios a 0
13
14 bucle:
15    ldrh  r1, [r0]    @ cargamos X en r1
16
17    cmp   r1, #0      @ comprobamos condición de terminación
18
19    cmp   r1, r7      @ Comparamos X con r7
20    bls   termina     @ X menor o igual que r7
21
22 @ X es mayor que r7
23
24    strh  r7, [r0]
25    add   r2, r2, #1  @ contabilizamos el cambio
26    add   r0, r0, #2  @ incrementamos el puntero
27    b     bucle       @ saltamos al bucle
28
29 termina:
30    mov   r7, r5      @ movemos r2 a r7
31
32 final:
33    wfi
34    .end
```

Ejercicio 3.9

Considere una zona de memoria que contiene una serie de números de 32 bits enteros mayores o iguales que cero.

Realizar un programa que haga las siguientes operaciones:

- Coger un número de la zona de memoria. A ese número le denominaremos X y al siguiente Y.
- Si X es mayor que el contenido del registro `r7`:
  - Si X es mayor o igual que Y, se copia X a partir de la dirección etiquetada como `copia` (la primera vez, y en direcciones siguientes, tantas veces como sea necesario).
  - En caso contrario, se guarda Y en el registro `r6` .
- En caso contrario: Se acaba el programa.

En el registro `r2` se contabilizará el número de iteraciones del programa (sin incluir cuando se cumple la condición de salida). En caso de que en una iteración determinada el programa no acabe, se procederá a realizar una nueva iteración, en la cual Y pasará a ser X y el número siguiente a Y pasará a ser Y.

El registro `r7` debe inicializarse con el valor 5.

Utilizar el siguiente código como ejemplo de la zona de memoria. Hay que tener en cuenta que la cantidad de números enteros en la zona de memoria no se conoce a priori.

```
       .data
zona:  .word  7, 9, 8, 6, 1, 4
copia: .space 24
```

Solución

Además del ligero aumento en complejidad de las operaciones a realizar, en este ejercicio trabajamos con una tabla de palabras de 4 bytes. Las operaciones de carga y almacenamiento de los datos se referirán a palabras (**`ldr`**, **`str`**) en vez de a medias palabras y tendremos que incrementar el puntero para navegar por la tabla (el registro `r0`) en 4 unidades en cada iteración.

b_cn5.s

```
1
2        .data
3 zona:  .word   7, 9, 8, 6, 1, 4
4 copia: .space  24
5
6        .equ   v_comp, 5
7
8        .text
9 main:
10    ldr   r0, =zona   @ r0: puntero al inicio de la zona
11    ldr   r3, =copia  @ r3: puntero a la zona de copia
12    mov   r7, #v_comp @ r7: valor a comparar
13    mov   r2, #0      @ r2: número de iteraciones
14
15 bucle:
16    ldr   r1, [r0]    @ cargamos X en r1
17    cmp   r1, r7      @ vemos si terminamos
18    ble   final       @ si X <= r7
19
20    ldr   r4, [r0,#4] @ cargamos Y en r4
21    cmp   r1, r4      @ ¿X>= Y? ¿r1>=r4?
22    bge   copiar
23
```

```
24 @ X < Y : guardamos Y en r6
25
26    mov   r6, r4
27 otro:
28    add   r2, r2, #1  @ incrementamos el contador,
29    add   r0, r0, #4  @ incrementamos el puntero
30    b     bucle       @ e iteramos una vez más
31
32 @ X >= Y
33
34 copiar:
35    str   r1, [r3]    @ almacenamos el dato en
36    add   r3, r3, #4  @ la nueva tabla, e incrementamos
37    b     otro        @ el puntero a la nueva tabla
38
39 final:
40    wfi
41    .end
```

3.3 Operaciones con cadenas de caracteres

El conjunto de ejercicios propuesto a continuación se centra en la realización de operaciones con cadenas de caracteres: contar caracteres, pasar de minúsculas a mayúsculas, comprobar características simples de grupos de caracteres, etc.

Ejercicio 3.10

Realizar un programa que cuente el número de caracteres en una cadena terminada en 0. La cadena se compone de caracteres ASCII de 7 bits (puntos Unicode U+0000 a U+007F).

Para probar el programa, utilice la cadena:

```
        .data
cadena: .asciz "Hola Mundo!"
```

Solución

La solución consiste en ir recorriendo la cadena de caracteres a la vez que incrementamos un contador, hasta que detectemos un carácter `\0`. Utilizamos el registro `r0` para recorre la cadena y el registro `r2` como contador.

cuentacad_ascii.s

```
1
2    .data
```

```
cadena:
    .asciz   "Hola Mundo!"
    .balign  4
ncars:
    .space   4

    .text
main:
    ldr   r0, =cadena  @ r0: puntero a la cadena
    mov   r2, #0       @ r2: contador a 0

bucle:
    ldrb r1, [r0]      @ leemos un carácter
    cmp   r1, #0       @ comprobamos si llegamos al final
    beq   final

    add   r2, r2, #1   @ incrementamos contador,
    add   r0, r0, #1   @ incrementamos puntero
    b     bucle        @ e iteramos otra vez

final:
    ldr   r0,=ncars    @ r0: puntero a ncars
    str   r1, [r0]     @ guardamos resultado
    wfi
    .end
```

Ejercicio 3.11

Realizar un programa que cuente el número de dígitos `5` que hay en un número representado por una cadena de caracteres con sus caracteres ASCII, almacenada a partir de la dirección de memoria etiquetada como `cad`. El final de la cadena viene determinado por un carácter `\0`. El resultado (un entero) se dejará en la palabra de memoria etiquetada como `num`.

Para probar el programa, utilice la cadena:

```
     .data
cad: .asciz "12546579"
```

Solución

Similar al ejercicio 3.10, solo que únicamente contamos los caracteres `'5'`.

nde5.s

```

        .data
cad:    .asciz "12546579"
        .balign 4
```

```
5   num:   .space 4          @ dirección del resultado
6
7       .text
8   main:
9       mov r1,#0            @ r1: desplazamiento por la cadena
10      ldr r0,=cad          @ r0: puntero al principio de la cadena
11      mov r3,#0            @ r3: contador de coincidencias con '5'
12  bucle:
13      ldrb r2,[r0, r1]     @ leemos un carácter
14      cmp r2,#0            @ vemos si hemos terminado
15      beq termina          @ (la cadena termina con 0)
16
17      cmp r2,#'5'          @ vemos si es un '5'
18      bne next             @ si es un '5',
19      add r3,r3,#1         @ incrementamos el contador de '5's
20  next:
21      add r1,r1,#1         @ apuntamos al siguiente
22      b bucle              @ e iteramos otra vez
23
24  termina:
25      ldr r0,=num          @ ro:  dirección del resultado
26      str r3,[r0]          @ almacenamos el resultado
27      wfi
28      .end
```

Ejercicio 3.12

Realizar un programa que, dado un número natural representado por una cadena de caracteres ASCII que empieza en la dirección `cad` y termina con un `\0`, indique si el número es capicúa. Si lo es, se dejará un 1 en la posición `capi`, y si no lo es, se dejará un 0 en dicha posición de memoria.

Para probar el programa, utilice la cadena:

```
     .data
cad: .asciz "123454321"
```

Solución

Vamos recorriendo la cadena de caracteres numéricos simultáneamente desde el principio y desde el final, utilizando como punteros los registros `r1` y `r2`. `r1` se inicializa con la dirección de memoria etiquetada como `cad` (apunta al principio de la cadena). En el caso de `r2`, buscamos el final de la cadena desde el principio de ésta, marcado con un `\0`, y dejamos `r2` apuntando al carácter anterior a dicho `\0`.

Si al recorrer la cadena desde los dos extremos detectamos dos caracteres numéricos diferentes antes de que los punteros se encuentren

en el medio de la misma, el número no será capicúa y terminamos con "0". En el caso de que r1 y r2 se encuentren, habremos terminado con un número capicúa, por lo que el resultado será "1". En este último caso tenemos dos posibilidades, dependiendo de que el número de dígitos sea par o impar.

- Que r1 y r2 acaben apuntando al mismo dígito (p. ej. en "313"). Por lo tanto, la condición de terminación será que r1 y r2 tomen el mismo valor.
- Que r1 y r2 se crucen (p. ej. en "3113"). En este caso, la condición de terminación será que r1 pase a ser mayor que r2.

En conjunto, la condición de terminación será que r1 >= r2 o de manera equivalente que r2 <= r1.

ncap.s

```
      .data
cad:  .asciz   "123454321"

      .balign  4
capi: .space   4


   .text
main:
   ldr   r1, =cad    @ r1: puntero al principio
   mov   r2, r1      @ r2: puntero al final
   ldr   r5, =capi   @ r5: puntero a capi

buc1:
   ldrb  r3, [r2]
   cmp   r3, #0
   beq   llegue
   add   r2, r2,#1
   b    buc1

llegue:              @ restamos 1 a r2 para que quede
   sub   r2, r2, #1  @ apuntando al último


buc2:
   cmp   r2, r1      @ Comparamos los dos punteros
   bls   escapi      @ Si r2 es menor igual a r1 es que
                     @ han coincidido todos
                     @ y se cruzan los punteros,
```

```
32                          @ por tanto es capicua
33
34     ldrb  r3, [r1]       @ r3: número desde el inicio
35     ldrb  r4, [r2]       @ r4: número desde el final
36     cmp   r3, r4
37     bne   noescapi       @ si son distintos, no es capicúa
38
39  @ si son iguales actualizamos los punteros
40
41     add   r1, r1, #1
42     sub   r2, r2, #1
43     b     buc2
44
45  escapi:
46     mov   r0, #1
47     str   r0, [r5]
48     wfi
49
50  noescapi:
51     mov   r0, #0
52     str   r0, [r5]
53
54     wfi
55     .end
```

Ejercicio 3.13

Realizar un programa que cuente el número de caracteres de una cadena terminada en 0. La cadena se compone de caracteres UTF-8 de 1 o 2 bytes (puntos Unicode U+0000 a U+00FF, cf. apéndice C). Ten en cuenta que, por lo tanto, caracteres como “ñ” (`0xc2b1`) o “í” (`0xc3ad`) cuentan como 1 carácter.

Para probar el programa, utilice la cadena:

```
        .data
cadena: .asciz "¡Buenos días pequeño mundo!"
```

Solución

El ejercicio es similar al ejercicio 3.10, salvo que tenemos caracteres de 1 y 2 bytes. Los caracteres UTF-8 de 2 bytes tienen como primer byte `0xc2` o `0xc3` y su segundo byte tiene el bit más significativo a 1 (cf. apéndice C). Los caracteres de 1 byte tienen siempre el bit más significativo a 0.

Por tanto, a la hora de contar los caracteres de la cadena, en caso de que leamos un byte con el primer bit a 1 estaremos ante el primer o el segundo byte de un carácter de dos bytes, que tendremos que contar solo una vez. Utilizamos el registro `r3` para indicar si en la iteración

anterior leímos un carácter con el primer bit a 1. Al leer un carácter con el primer bit a 1 se pueden dar dos casos:

- Si `r3 = 0`, se trata del primer carácter con el primer bit a 1. Hacemos `r3 = 1` y leemos el siguiente byte.
- Si `r3 = 1`, se trata del segundo carácter con el primer bit a 1. Contamos un carácter más, hacemos `r3 = 0` y leemos el siguiente byte.

cuentacad_utf8.s

```
    .data
cadena:
    .asciz  "¡Buenos días pequeño mundo!"
    .balign 4
ncars:
    .space  4

    .text
main:
    ldr   r0,=cadena @ r0: puntero a la cadena
    mov r2, #0       @ r2: contador a 0
/*
  Los caracteres de dos bytes UTF-8 tienen el bit más
significativo a uno y están compuestos por dos bytes,
ambos con el bit más significativo a 1.
*/
    mov r3, #0       @ r3: para indicar si el carácter
                     @ anterior era auxiliar
    ldr r4,=0x80     @ r4: máscara para detectar especiales
                     @ (tienen el primer bit a 1)
bucle:
    ldrb r1, [r0]
    cmp   r1, #0     @ comprobamos si llegamos al final
    beq   final      @ de la cadena

    tst   r1, r4     @ Comprobamos el bit más significativo
    beq   escar      @ si es cero, es un carácter de 1 byte

@ el bit más significativo está a 1: carácter de 2 bytes

    cmp   r3, #0     @ comprobamos si en el anterior también
    bne   escar      @ si estaba, estamos ante un segundo byte UTF-8
    mov   r3, #1     @ en caso contrario es el primer byte de los dos
    b     otro       @ cambiamos el valor de r3 e iteramos de nuevo

escar:
    add   r2, r2, #1 @ incrementamos el contador
    mov   r3, #0     @ reseteamos el marcador de carácter especial
```

```
40
41 otro:
42    add  r0, r0, #1 @ incrementamos el puntero
43    b    bucle
44
45 final:
46    ldr  r0, =ncars @ almacenamos el resultado.
47    str  r2, [r0]
48    wfi
49    .end
```

Ejercicio 3.14

Realizar un programa que pase a mayúsculas una cadena terminada en `\0`. La cadena se compone de caracteres ASCII de 7 bits (puntos Unicode U+0000 a U+007F).

Para probar el programa, utilice la cadena:

```
     .data
cad: .asciz "Hola mundo!"
```

Solución

Vamos recorriendo la carrera utilizando el registro `r0` como puntero, hasta que encontremos el carácter de terminación `\0`, saltándonos los caracteres que no sean letras minúsculas, es decir, todos aquellos cuyo código ASCII sea menor que el código de la `'a'` o mayor que el de la `'z'`.

Los códigos ASCII de las letras mayúsculas y minúsculas se diferencian únicamente en su bit de peso 5. En el caso de que el carácter sea una letra minúscula, tendrá su bit de peso 5 a 1 (cf. cuadro C.2 del apéndice C). Para pasarla a mayúscula, simplemente ponemos ese bit a 0.

amayusc.s

```
1
2       .data
3  cad: .asciz "Hola mundo!"
4
5       .text
6  main:
7       ldr   r0, =cad   @ r0: puntero a la cadena
8
9  @ La diferencia entre mayúsculas y minúsculas es que
10 @ las primeras tienen el bit de peso 5 a 1
11 @ y las segundas a 0
12
13      mov   r4, #0x20  @ usamos r4 como mascara
```

```
14
15  buc: ldrb  r1, [r0]
16       cmp   r1, #0
17       beq   final
18
19       cmp   r1, #'a'   @ ¿es menor que 'a'?
20       blt   salta      @ si es así nos lo saltamos
21
22       cmp   r1, #'z'   @ ¿es mayor que 'z'?
23       bgt   salta      @ si es así nos lo saltamos
24                        @ si no, pasamos a mayúsculas
25       bic   r1, r4     @ ponemos a 0 el bit de peso 5
26       strb  r1,[r0]    @ y actualizamos en memoria
27
28  salta:
29       add   r0, r0, #1
30       b     buc
31
32  final:
33       wfi
34       .end
```

Ejercicio 3.15

Realizar un programa que pase a mayúsculas una cadena terminada en `\0`. La cadena se compone de caracteres UTF-8 de 1 o 2 bytes (puntos Unicode U+0000 a U+00FF, cf. apéndice C). Ten en cuenta que, por lo tanto, caracteres como "ñ" (`0xc2b1`) o "í" (`0xc3ad`) cuentan como 1 carácter.

Para probar el programa, utilice la cadena:

```
        .data
cad:    .asciz "¡Buenos días pequeño mundo!"
```

Solución

Este ejercicio es similar al 3.14, pero además tenemos que tener en cuenta las consideraciones sobre los caracteres UTF-8 de dos bytes del ejercicio 3.13. En cualquier caso, al igual que ocurre con los caracteres ASCII de 1 byte, la diferencia entre mayúsculas y minúsculas en el caso de los caracteres alfanuméricos de dos bytes también está únicamente en su bit de peso 5 del segundo byte (cf. cuadro C.4 del apéndice C).

amayusc_utf8.s

```
1        .data
2   cad: .asciz "¡Buenos días pequeño mundo!"
3
4
```

```
5          .text
6   main: ldr  r0, =cad   @ puntero al inicio de la cadena
7
8   @ La diferencia entre mayúsculas y minúsculas es que las
9   @ primeras tienen el bit de peso 5 a 1 y las segundas a 0
10
11         mov  r4, #0x20 @ usamos r4 como mascara
12  bucle:
13         ldrb r1, [r0]
14         cmp  r1, #0
15         beq  final
16                        @ Comparamos el primer byte con 128.
17         cmp  r1, #128  @ Los cars. de un byte serán menores.
18                        @ Si es mayor de 128, es el primero de
19         bcs  tienedos  @ dos bytes.
20         cmp  r1, #'a'  @ En caso de tener un byte
21         bcc  otro      @ vemos si es minúscula
22         cmp  r2, #'z'
23         bhi  otro
24
25         bic  r1, r4
26         strb r1, [r0]
27
28  otro: add  r0, r0, #1
29         b    bucle
30  tienedos:              @ Si el primer carácter es 0xC2
31         cmp  r1, #0xC2  @ es un carácter especial
32         beq  otro       @ de dos bytes.
33
34         add  r0, r0, #1 @ Leemos el segundo byte.
35         ldrb r1, [r0]
36
37         cmp  r1, #0xB7  @ Excepto el signo de dividir
38         beq  otro
39
40         cmp  r1, #0xBF  @ y el glifo y con diéresis
41         beq  otro
42                         @ en el resto de los casos son
43         bic  r1, r4     @ letras 'especiales' susceptibles de
44                         @ pasar a mayúsculas
45         strb r1, [r0]
46         b    otro
47
48  final:
49         wfi
50         .end
```

Ejercicio 3.16

Un palíndromo es una palabra o frase que se lee igual en un sentido que en otro, como por ejemplo *Amad a la dama*. Realizar un programa que, dada una cadena de caracteres ASCII que empieza en la posición de memoria etiquetada como `frase` y terminada por un punto "`.`", indique si dicha frase es un palíndromo o no. Si es un palíndromo dejará una "`S`" en la posición de memoria etiquetada como `pal` y si no lo es dejará una "`N`" en dicha posición de memoria. En caso de que la frase no contenga ninguna letra, el programa dejará una "`V`" en `pal`. La frase solamente contendrá, además del punto que indica el final de la misma, caracteres ASCII de letras mayúsculas o minúsculas y espacios en blanco.

Para probar el programa, utilice la cadena:

```
       .data
frase: .ascii "Amad a la dama."
```

Solución

Este ejercicio es muy similar al ejercicio 3.12, salvo que la cadena termina con un "." y puede tener espacios en blanco. En este caso, si la cadena no tiene caracteres alfanuméricos, el programa terminará con "`V`". Si al recorrer la frase desde los dos extremos detectamos dos caracteres diferentes antes de que los punteros se encuentren en el medio de la frase, no se tratará de un palíndromo y terminamos con "`N`". En el caso de que los punteros (`r0` y `r1` en este caso) se encuentren, habremos terminado con un palíndromo, por lo que el resultado será "`S`". La condición de terminación será que `r0` >= `r1`.

capicua.s

```
1        .data
2 
3  pal:   .byte      1
4  frase: .ascii "Amad a  la dama."
5  ese:   .byte 'S'     @ resultado si es palíndromo
6  ene:   .byte 'N'     @ resultado si no es palíndromo
7  uve:   .byte 'V'     @ resultado en caso de que no haya letras
8         .equ punto,46 @ código ASCII del "."
9 
10       .text
11 
12 main: ldr  r0,=frase @ r0: puntero desde el principio de la frase
13       mov  r1,r0     @ r1: puntero desde el final de la frase
14       mov  r4,#0     @ r4: indicador de si hay letras
15 bucle1:
16       ldrb r2,[r1]   @ leemos caracteres hasta llegar al punto
```

```
17         cmp   r2,#punto @ comprobamos si final de la cadena
18         beq   leerc1    @ ya llegamos al final
19         add   r1,r1,#1  @ actualizamos r2 si no llegamos al final
20         b     bucle1
21  leerc1:
22         cmp   r0,r1
23         bge   loes      @ si los punteros se cruzan acabamos
24         ldrb  r2,[r0]   @ leemos en r2 el carácter desde izquierda
25
26         cmp   r2,#punto @ si llega al final podría serlo
27         beq   loes
28
29         cmp   r2, #65   @ comparamos con A
30         bmi   leeotroc1 @ no es una letra
31         cmp   r2, #91   @ Z es 90
32         bmi   c1esmay
33
34         cmp   r2, #97   @ 'a' es 97
35         bmi   leeotroc1
36         cmp   r2, #123  @'z' es 122
37         bmi   leec2     @ c1 es minúscula
38  c1esmay:
39         add   r2,r2,#32 @ la pasamos a minúscula
40         b     leec2
41  leeotroc1:
42         add   r0,r0,#1
43         b     leerc1
44  leec2:
45         mov   r4,#1     @ hay al menos una letra
46         sub   r1,r1,#1  @ r1 apuntando al carácter anterior
47         ldrb  r3,[r1]
48
49         cmp   r3,#65    @ comparamos con A
50         bmi   leec2     @ no es una letra
51         cmp   r3,#91    @ 'Z' es 90
52         bmi   c2esmay   @ menor de 91 es mayúscula
53
54         cmp   r3, #97   @ 'a' es 97
55         bmi   leec2
56         cmp   r3, #123  @'z' es 122
57         bmi   compara   @ c2 es minúscula
58
59  c2esmay:
60         add   r3,r3,#32
61  compara:
62         cmp   r2,r3
63         bne   noloes    @ no es palíndromo
64         add   r0,r0,#1
65         b     leerc1    @ si son iguales, a por el siguiente
```

```
noloes:
        ldr  r0,=ene   @ almacenamos una 'N'
        ldrb r1,[r0]   @ en la dirección del resultado
        ldr  r0,=pal
        strb r1,[r0]

        wfi

loes:   cmp  r4, #0
        beq  vacia
        ldr  r0,=ese   @ almacenamos una 'S'
        ldrb r1,[r0]   @ en la dirección del resultado
        ldrb r0,=pal
        strb r1,[r0]

        wfi

vacia:
        ldr  r0, =uve  @ almacenamos una 'V'
        ldrb r1, [r0]  @ en la dirección del resultado
        ldrb r0, =pal
        strb r1, [r0]

        wfi
        .end
```

3.4 Operaciones aritmético-lógicas

A continuación, proponemos un conjunto de problemas centrados en la utilización de manera combinada de las instrucciones aritméticas, lógicas y de desplazamiento. Se incluyen problemas para manejar tablas de bits, convertir entre diferentes formatos de representación de información que requieren la realización de operaciones de desplazamiento y el trabajo con máscaras, la realización de operaciones aritméticas complejas a partir de operaciones simples, o el procesado de cadenas de caracteres.

Ejercicio 3.17

Realizar un programa que muestre el factorial de un número N almacenado en la palabra de memoria etiquetada como `input`, con $0 \leq N \leq 7$, buscándolo en una tabla de factoriales. El programa almacenará el resultado en la palabra de memoria etiquetada como `output`.

Para probar el programa, utilice las definiciones siguientes:

```
        .data
factab:
        .word 1     @ 0! = 1
        .word 1     @ 1! = 1
        .word 2     @ 2! = 2
        .word 6     @ 3! = 6
        .word 24    @ 4! = 24
        .word 120   @ 5! = 120
        .word 720   @ 6! = 720
        .word 5040  @ 7! = 5040
input:
    .word   6
output:
    .space  4
```

Solución

El ejercicio consiste básicamente en acceder a una tabla de factoriales etiquetada como `factab` utilizando como índice el número del cual queremos calcular su factorial. Como cada palabra de memoria ocupa 4 bytes y en ARM la memoria se organiza en bytes (cada byte tiene su propia dirección), el factorial de `N` estará almacenado en la palabra cuya dirección es `factab + (N x 4)`. Para multiplicar por 4, desplazamos el número `N` dos veces hacia la izquierda con la instrucción **`lsl`**.

factorial.s

```
1
2       .data
3   factab:
4       .word   1    @ 0! = 1
5       .word   1    @ 1! = 1
6       .word   2    @ 2! = 2
7       .word   6    @ 3! = 6
8       .word   24   @ 4! = 24
9       .word   120  @ 5! = 120
10      .word   720  @ 6! = 720
11      .word   5040 @ 7! = 5040
12
13  input:
14      .word   6
15  output:
16      .space  4
17
18      .text
19  main:
20      ldr   r0, =factab  @  r0: comienzo de la tabla
21      ldr   r1, =input   @  r1: dirección del dato de entrada
22      ldr   r1, [r1]     @  r1: dato de entrada N
```

```
23      lsl   r1, #2        @  convertimos N en un desplazamiento (N x 4)
24      ldr   r2, [r0, r1] @  r2: factorial sacado de la tabla
25      ldr   r0, =output  @  r0: dirección del resultado
26      str   r2, [r0]     @  almacenamos el resultado
27
28      wfi
29      .end
```

Ejercicio 3.18

Realizar un programa que sume dos números enteros de 64 bits (2 palabras) almacenados en memoria en las posiciones etiquetadas como `num1` y `num2`, dejando el resultado en una zona de memoria etiquetada como `res`.

Para probar el programa, utilice las siguientes definiciones:

```
      .data
num1: .quad  0x123456789abcdef0
num2: .quad  0xedcba9876543210f
res:  .space 8
```

Solución

Este ejercicio ilustra la suma con acarreo. Primeramente, sumamos las palabras menos significativas del número utilizando la instrucción **add**, `0x9abcdef0` y `0x6543210f` según las definiciones que propone el enunciado. A continuación, sumamos las palabras más significativas junto con el acarreo de la primera suma (`0x12345678` y `0xedcba987`), utilizando la instrucción **adc**.

Como en todas las arquitecturas *load/store*, las operaciones aritméticas se realizan con valores almacenados en registros y el resultado se deposita en un registro. En nuestro caso, cargamos el primer número en los registros `r1` (parte menos significativa) y `r2` (parte más significativa), y el segundo número en los registros `r3` y `r4`. El resultado de la suma queda en los registros `r1` (parte menos significativa) y `r2` (parte más significativa).

suma64.s

```
1
2          .data
3   num1: .quad   0x123456789abcdef0   @ valores a sumar
4   num2: .quad   0xedcba9876543210f   @
5   res:  .space  8                    @ espacio para el resultado
6
7       .text
8   main:
9       ldr   r0, =num1    @ dir del primer valor
```

```
10      ldr    r1, [r0]      @ primera parte de n1 (menos sig.)
11      ldr    r2, [r0, #4]  @ segunda parte de n1 (mas sig.)
12      ldr    r0, =num2     @ dir del segundo valor
13      ldr    r3, [r0]      @ primera parte de n2
14      ldr    r4, [r0, #4]  @ segunda parte de n2
15
16      add    r1, r1, r3    @ suma parte menos significativa (set acarreo)
17      adc    r2, r2, r4    @ suma parte más significativa (con acarreo)
18
19      ldr    r0, =res      @ almacenamos el resultado
20      str    r1, [r0]
21      str    r2, [r0, #4]
22
23      wfi
24      .end
```

Ejercicio 3.19

Realizar un programa que divida un número entero de 32 bits entre un número entero de 16 bits, dejando el resultado en las palabras de memoria etiquetadas cono `coci` (valor del cociente de la división) y `resto` (valor del resto). Los datos se encuentran almacenados en las posiciones de memoria etiquetadas como `dendo` (dividendo) y `dsor` (divisor). Para probar el programa, utilice las siguientes definiciones:

```
       .data
dendo: .word   0x00070000
dsor:  .hword  0x3000
       .balign 4
coci:  .space  4
resto: .space  4
```

Solución

Para realizar la división, utilizaremos el método de las diferencias sucesivas o división euclídea:

```
int cociente;
int resto;

void division(int dividendo, int divisor) {
      while (dividendo > divisor) {
            dividendo = dividendo - divisor;
            cociente = cociente + 1;
      }
      resto = dividendo;
}
```

obteniéndose finalmente en `cociente` y `resto` los valores del cociente y del resto de la división. En nuestro caso, implementaremos el algoritmo anterior utilizando como variables los registros `r0` (dividendo / resto), `r1` (divisor) y `r3` (cociente).

Tendremos que tener en cuenta además que el dividendo es una palabra de 4 bytes y el divisor una media palabra.

dividir.s

```
1
2
3         .data
4  dendo: .word  0x00070000
5  dsor:  .hword 0x3000
6         .balign 4
7  coci:  .space 4
8  resto: .space 4
9
10     .text
11 main:
12     ldr    r0, =dendo    @ dividendo a r0
13     ldr    r0, [r0]
14     ldr    r1, =dsor     @ divisor a r1
15     ldrh   r1, [r1]
16     mov    r3, #0        @ init cociente a 0 (r3)
17
18     cmp    r1, #0        @ vemos si dividimos por cero
19     beq    error
20 bucle:
21     cmp    r0, r1        @ ¿divisor menor que dividendo?
22     blt    fin           @ si lo es, acabamos
23
24     add    r3, r3, #1    @ sumamos 1 al cociente
25     sub    r0, r0, r1    @ restamos el divisor del dividendo
26     b      bucle
27
28 error:
29     eor    r3, r3, r3    @ indicamos error
30     mvn    r3, r3
31
32 fin:
33     ldr    r7,=coci
34     str    r0, [r7, #4] @ almacenamos el resto
35     str    r3, [r7]     @ y el cociente
36
37     wfi
38     .end
```

Ejercicio 3.20

Realizar un programa que, dado un número positivo de 32 bits en octal representado por una cadena de caracteres ASCII terminada en cero y almacenada a partir de la dirección `numoc`, lo transforme a binario y lo almacene en la posición etiquetada como `resul`. Compruébelo con el siguiente ejemplo.

```
       .data
numoc: .asciz "3502570"
resul: .word  1
```

El programa deberá almacenar en la palabra etiquetada como `resul` el valor `03502570`.

Solución

El algoritmo para convertir el número en octal $0d_1d_2\ldots d_n$ de octal a decimal es:

```
int num = 0;

for (i = 0; i < n ; i++)
   num = 8 * num + di;
```

donde d_i/`di` son los dígitos de la representación en octal del número.

Por tanto, para resolver el problema convertimos los caracteres ASCII que representan los dígitos a números enteros para obtener los d_i/`di` y aplicamos el algoritmo anterior.

octobin.s

```
1        .data
2  numoc: .asciz "3502570"
3  resul: .word  1
4
5     .text
6  main:
7     ldr   r0,=numoc   @ r0: dirección de la cadena
8     mov   r2,#0       @ r2: resultado (num en el algoritmo)
9
10 otro:
11    ldrb  r1,[r0]     @ r1: cada dígito di
12    cmp   r1,#0       @ ¿hemos terminado?
13    beq   listo
14
15    sub   r1,r1,#'0'  @ pasamos di a binario
16    lsl   r2,#3       @ multiplicamos por ocho
17    add   r2,r2,r1    @ y sumamos al resultado (num)
18    add   r0,r0,#1    @ incrementamos el puntero
19    b     otro        @ y pasamos al siguiente.
20
```

```
listo:
   ldr   r0,=resul  @ almacenamos el resultado
   str   r2,[r0]

   wfi
   .end
```

Ejercicio 3.21

Realizar un programa que separe las dos mitades de un byte, almacenado en la posición de memoria etiquetada como `val` y deje cada mitad en cada uno de los bytes de una media palabra de 16 bits, etiquetada como `res`.

Para probar el programa, utilice las siguientes definiciones:

```
     .data
val: .byte   0x6c
     .balign 2
res: .space  2
```

Solución

Para resolver el problema, cargamos el byte almacenado en `val` en `r1` y pasamos sus 4 bits más significativos a `r3` con **`lsr`**. A continuación, dejamos esos 4 bits en su lugar definitivo en una media palabra con **`lsl`**. Finalmente, dejamos en `r1` los 4 bits menos significativos y le añadimos lo que tenemos en `r3`. Así, los 16 bits menos significativos de `r1` tendrán el resultado esperado, que almacenamos en `res`.

nibbles.s

```

     .data
val: .byte   0x6c
     .balign 2
res: .space  2

   .equ    mask,  0x000f  @ máscara para asilar un nibble
   .equ    nlsb,  0x04    @ contador para mover el primer nibble
   .equ    bdesp, 0x08    @ contador para mover el segundo nibble

   .text
main:
   ldr    r1, =val
   ldr    r1, [r1]        @ dato a separar
   lsr    r3, r1, #nlsb   @ pasamos el primer nibble a r3
   lsl    r3, r3, #bdesp  @ lo movemos un byte

   mov    r4, #mask       @ cargamos la máscara
```

```
19      and   r1, r1, r4      @ dejamos el segundo nibble
20      orr   r1, r1, r3      @ los añadimos a r4
21
22      ldr   r2, =res
23      str   r1, [r2]        @ almacenamos el resultado
24
25      wfi
26      .end
```

Ejercicio 3.22

Realizar un programa que, dado un número almacenado en la posición de memoria etiquetada como `num`, cuente la cantidad de ceros que tiene la representación binaria de ese número y deje el resultado en la palabra etiquetada como `ncero`. Compruébelo con el siguiente ejemplo.

```
       .data
num:   .word  0xabababab @ 2+1+2+1+2+1+2+1 ceros = 12 ceros
ncero: .space 4
```

Solución

Cargamos el número almacenado en `num` en el registro `r1` y lo desplazamos bit a bit a la derecha con **`asr`**, de manera que el bit menos significativo pasa al bit de acarreo `C`. Luego, utilizamos la instrucción de salto condicional **`bcs`** (saltar si el indicador `C` está activado) para incrementar o no un contador con el número de ceros (registro `r3`).

```
nceros.s
1          .data
2   num:   .word  0xabababab @ 2+1+2+1+2+1+2+1 ceros = 12 ceros
3   ncero: .space 4
4
5      .text
6   main:
7      ldr r0,=num
8      mov r2,#32    @ contador de desplazamientos
9      mov r3,#0     @ contador de ceros
10     ldr r1,[r0]
11  otro:
12     asr r1,#1     @ desp. a derechas (el bit que sale va a C)
13     bcs nocero    @ si salió un uno, pasamos
14     add r3,r3,#1  @ incrementamos el contador de ceros
15  nocero:
16     sub r2,r2,#1  @ decrementamos el contador de despl.
17     bne otro
18
19     ldr r0,=ncero @ almacenamos el resultado
```

```
20      str r3,[r0]
21
22      wfi
23      .end
```

Ejercicio 3.23

Realizar un programa que active un bit de una matriz de 8 x 8 bits serializada por filas. La matriz estará almacenada en una zona de memoria compuesta por 8 bytes a partir de la dirección de memoria etiquetada como `matriz`. El programa recibirá en dos bytes etiquetados como `col` y `fila` dos números comprendidos entre 0 y 7 que indican respectivamente las coordenadas (columna, fila) del bit a activar. Las filas y columnas se numeran del bit más significativo del byte al menos significativo (columnas) y de la dirección más baja a la más alta (filas).

Para probar el programa, utilice las siguientes definiciones:

```
        .data
col:    .byte   3
fila:   .byte   2
        .balign 4
matriz: .space  8
```

Solución

Una matriz es una estructura bidimensional formada por filas y columnas, mientras que la memoria es lineal y por lo tanto unidimensional. Por lo tanto, a la hora de almacenar una matriz en memoria tendremos que adoptar un convenio para hacerlo de manera consistente. Las dos opciones disponibles serían almacenar las filas de manera consecutiva en memoria, una detrás de la otra, o almacenar las columnas de manera consecutiva. En el primer caso decimos que la matriz está almacenada en memoria serializada por filas y en el segundo que está serializada por columnas.

Serializar una matriz bidimensional consiste en último término en convertir dicha matriz en un vector unidimensional. En el caso de una matriz serializada por filas, suponiendo que el primer elemento de la matriz es el de coordenadas $(0, 0)$ y que el primer elemento del vector es el de índice 0, el elemento de coordenadas (i, j) se corresponderá con el elemento del vector de índice $i \times d + j$, donde d es el número de columnas de la matriz. Por ejemplo, en una matriz 8 x 8 ($d = 8$), el índice del elemento de coordenadas $(0, 7)$ tomará el valor 7, el índice

del elemento de coordenadas $(1,7)$ tomará el valor 15 y el índice del elemento de coordenadas $(7,7)$ tomará el valor 63.

En este caso, las dimensiones de la matriz (8 x 8) coinciden con el tamaño de las posiciones de memoria del ordenador (8 bits), con lo que el elemento de coordenadas (i,j) se corresponderá con el bit j-ésimo del byte i-esimo de la matriz serializada a partir de la posición etiquetada como `matriz`.

tablabit.s

```
1          .data
2  col:      .byte   0
3  fila:     .byte   0
4            .balign 4
5  matriz:   .space  8
6
7      .text
8  main:
9      ldr  r0,=matriz @ ro: dirección de la tabla
10     ldr  r1,=fila   @ r1: dir. del número de fila
11     ldr  r2,=col    @ r2: dir. del número de columna
12     ldrb r1,[r1]    @ r1: número de fila
13     ldrb r2,[r2]    @ r2: número de columna
14
15     mov  r3,#0x80   @ r3: máscara para seleccionar
16     lsr  r3,r3,r2   @ la columna (0 = más significativo)
17     ldrb r4,[r0,r1] @ cargamos la fila
18     orr  r4,r4,r3   @ activamos el bit
19     strb r4,[r0,r1] @ y almacenamos el resultado
20
21     wfi
22     .end
```

Ejercicio 3.24

Existe una matriz de 4×16 bits almacenada a partir de la posición etiquetada como `array`. La matriz se almacena serializada por filas. Escriba un programa que, dada una fila (entre 0 y 3), una columna (entre 0 y 15) y un valor (0 o 1), actualice la matriz de manera acorde a esos datos.

Por ejemplo, dada la matriz

```
       .data
array: .byte 0b10000000, 0b00000000 @ fila 0
       .byte 0b01000000, 0b00000000 @ fila 1
       .byte 0b00100000, 0b00000000 @ fila 2
       .byte 0b00001000, 0b11111111 @ fila 3
@              ||||||||    ||||||||
@ columnas:    01234567    89abcdef
```

```
@
        .balign 4
fil:    .word 2       @ fila entre 0 y 3
col:    .word 12      @ columna entre 0 y 15
val:    .byte 1       @ nuevo valor, 0 o 1
```

El programa actualizará el sexto byte de la matriz (el que contiene el bit de la columna 12) con el nuevo valor `0b0000 1000` = `0x08`.

Solución

En el ejercicio 3.23 describimos el proceso de serialización de la matriz y la manera de calcular el índice sobre la matriz serializada, en este caso almacenada a partir de la dirección `array`. Para resolver el ejercicio, primero calculamos el índice del bit a modificar, de acuerdo con la fórmula del ejercicio 3.23:

$$indice = fila \times 16 + columna$$

Una vez que hemos calculado el índice, buscamos el byte donde se encuentra el bit a modificar. Para ello dividimos el índice por 8. El cociente de la división nos indicará el índice del byte en `array` y el resto el desplazamiento dentro de dicho byte, contando de izquierda a derecha.

matriz_4x16.s

```
1         .data
2  array: .byte 0b10000000,0b00000000 @ fila 0
3         .byte 0b01000000,0b00000000 @ fila 1
4         .byte 0b00100000,0b00000000 @ fila 2
5         .byte 0b00001000,0b11111111 @ fila 3
6  @               ||||||||    ||||||||
7  @ columnas:      01234567    89abcdef
8  @
9        .balign 4
10 fil:  .word 2       @ fila entre 0 y 3
11 col:  .word 12      @ columna entre 0 y 15
12 val:  .byte 1       @ nuevo valor, 0 o 1
13
14       .text
15
16       @ Calculamos el índice del bit a modificar.
17       @ Matriz serializada: index = (fila * 16) + columna
18
19 main: ldr r0,=fil
20       ldr r0,[r0]     @ r0 = fila
21       mov r1,#16
22       mul r0,r0,r1    @ r0 = fila * 16
23       ldr r1,=col
```

```
24        ldr r1,[r1]    @ r1 = columna
25        add r0,r1      @ r0 = índice del bit en la matriz serializada
26
27        @ Calcula el byte que tiene el bit a modificar (b_mod)
28        @ índice  = b_mod * 8 + offset
29        @ (contando de izquierda a derecha)
30
31        mov r1,r0        @ r1: índice
32        lsr r0,#3        @ r0: cociente de dividir por ocho
33        mov r2,#0b111
34        and r1,r2        @ r1: resto de la división
35
36        ldr r2,=#0x80    @ máscara inicial para activar o desactivar
37        lsr r2,r1        @ desplaza la máscara al bit objetivo
38
39        ldr r5,=array
40        ldrb r6,[r5,r0]  @ carga el byte objetivo (con el offset)
41        ldr r7,=val
42        ldr r7,[r7]
43        cmp r7,#0        @ ¿activar o desactivar?
44        beq set0
45
46  set1: orr r6,r2       @ activamos usando la máscara
47        b next
48
49  set0: mvn r2,r2
50        and r6,r2       @ desactivamos usando la máscara inversa
51
52  next: strb r6,[r5,r0] @ sustituimos el byte original
53
54        wfi
55       .end
```

Ejercicio 3.25

Realizar un programa que compare dos cadenas de caracteres terminadas con 0, almacenadas en memoria a partir de las posiciones etiquetadas como `cad1` y `cad2`, dejando el resultado de la comparación en la posición de memoria etiquetada como `res`. El programa ofrecerá como resultado el valor `0xff` en el caso de que las cadenas sean diferentes, y el valor `0x00` en el caso de que sean iguales.

Para probar el programa, utilice las cadenas siguientes:

```
      .data
cad1: .asciz "¡Hola Mundo!"
cad2: .asciz "¡Hola Mundo!"
res:  .space 1
```

Solución

Vamos recorriendo las cadenas utilizando el registro `r0` como puntero para recorrer la primera cadena y el registro `r1` para recorrer la segunda. Comparamos dos a dos los caracteres de ambas cadenas, almacenados respectivamente en los registros `r2` y `r3`.

Las condiciones de terminación son:

- Que los caracteres leídos de ambas cadenas sean diferentes. En este caso, las cadenas serán diferentes.
- Que una de las cadenas termine, es decir, que el byte leído tome el valor 0. En este caso, si la otra también termina, las cadenas serán iguales, y si no serán diferentes.

comp_cad.s

```
/*
 Comparamos dos cadenas terminadas en 0 a ver si son iguales

*/
      .data
cad1: .asciz "¡Hola Mundo!"
cad2: .asciz "¡Hola Mundo!"
res:  .space 1

   .text
main:
   ldr r0, =cad1    @ r0: puntero a la primera cadena
   ldr r1, =cad2    @ r1: puntero a la segunda cadena

buc:
   ldrb r2, [r0]    @ leemos un carácter de la primera cadena
   cmp r2, #0       @ si es 0, hemos terminado
   beq finc1

@ el carácter de la primera cadena no es a cero

   ldrb r3, [r1]    @ leemos un carácter de la segunda cadena
   cmp r3,#0        @ si es cero, hemos terminado
   beq dif          @ y las cadenas son diferentes

@ quedan caracteres en ambas cadenas

   cmp r2, r3       @ si son diferentes, hemos terminado
   bne dif          @ y las cadenas son diferentes

@ los caracteres siguen siendo iguales,
@ seguimos con los siguientes caracteres de ambas cadenas
```

```
33
34      add r0, r0, #1
35      add r1, r1, #1
36      b buc
37
38   @ primera cadena finalizada
39
40   fincl:
41      ldrb r3, [r1]     @ leemos un carácter de la segunda cadena
42      cmp r3, #0        @ si es 0, hemos terminado
43      bne dif           @ y las dos cadenas son iguales
44
45   gres:
46      ldr r1, =res      @ almacenamos el resultado de la
47      strb r2, [r1]     @ comparación
48      wfi
49
50   @ las cadenas son diferentes
51
52   dif:
53      mov r2, #0xff     @ resultado para cadenas diferentes
54      b gres
55
56   @ las cadenas son iguales
57
58   igual:
59      mov r2, #0        @ resultado para cadenas iguales
60      b gres
61      .end
```

Ejercicio 3.26

Realizar un programa para calcular la representación en código ASCII del valor en hexadecimal de un byte almacenado en la posición de memoria etiquetada como `val`. Tenga en cuenta de que el resultado se representará con dos dígitos hexadecimales (0 a 9 y letras de la A a la F). El resultado se almacenará en 2 bytes a partir de la posición etiquetada como `res`.

Para probar el programa, utilice las siguientes definiciones:

```
     .data
val: .byte  0xa7
res: .space 2
```

Solución

Cada uno de los dos dígitos hexadecimales de un byte se codifica con 4 bits, es decir, con un *nibble*. Convertimos a los caracteres ASCII co-

rrespondientes cada uno de los nibbles, primero el menos significativo y luego el más significativo.

Para convertir un valor de 4 bits al carácter ASCII que representa su valor en hexadecimal, primero comprobamos si el número está entre `0x0` y `0x9` o entre `0xA` y `0xF` (cf. cuadro C.2 del apéndice C):

- En el primer caso, para convertir el número al carácter que lo representa tendremos que sumarle el código ASCII del carácter `'0'`, es decir, tendremos que sumar el valor `0x30`.
- En el segundo caso, para realizar la conversión tendremos que sumar `0x37`. Por ejemplo, `0xA + 0x37 = 0x41`, que es el código ASCII del carácter `'A'`.

hextoascii.s

```
      .data
val: .byte  0xa7 @ dato de ejemplo
res: .space 2     @ resultado

    .text
main:
    mov   r2, #0x0f     @ enmascaramos todo menos el nibble
    ldr   r0, =val      @ menos significativo
    ldrb  r1, [r0]      @ cargamos el byte en r0
    and   r1, r1, r2
    cmp   r1, #0x09     @ vemos si '0' - '9' o 'A' - 'Z'
    bhi   mayor1

    add   r1, r1, #'0'  @ menor que 9
    b     dig2

mayor1:
    add   r1, r1, #0x37 @ mayor que 9 : 0x0A + 0x37 = 0x41 = 'A'

dig2:
    ldr   r3, =res
    strb  r1, [r3, #1]  @ lo almacenamos en su sitio
@
@ hacemos lo mismo con el nibble (grupo de 4 bits)
@ más significativo.
@
    ldrb  r1, [r0]
    lsr   r1, r1, #4    @ eliminamos la parte ya convertida
    cmp   r1, #0x09
    bhi   mayor2

```

```
33     add   r1, r1, #0x30
34     b     fin
35
36  mayor2:
37     add   r1, r1, #0x37
38
39  fin:
40     strb  r1, [r3]     @ lo almacenamos en su sitio
41
42     wfi
43     .end
```

Ejercicio 3.27

Realizar un programa que multiplique dos números de 32 bits almacenados en las palabras de memoria etiquetadas como `num1` y `num2`, dejando el resultado (64 bits) en la doble palabra etiquetada como `res`.

Para probar el programa, utilice las siguientes definiciones:

```
      .data
num1: .word  0x11223344
num2: .word  0x55667788
res:  .space 8
```

Solución

Cargamos los números a multiplicar en registros, separando sus mitades altas y bajas, para poder utilizar la instrucción **`mul`** de Thumb que multiplica medias palabras de 16 bits:

- `num1` se carga en (los 16 bits menos significativos) de `r5` y `r0`.
- `num2` se carga en (los 16 bits menos significativos) de `r3` y `r1`.

Hacemos además una copia de los 16 bits menos significativos de `num2` en `r7` para facilitar la multiplicación.

A continuación, aplicamos directamente el método de multiplicación siguiente (`num1` = `{r5r0}` y `num2` = `{r3r1}` o `{r3r7}`):

```
                     r5r0
                  r3r1/r7
 -------------------------
           r5xr1    r7xr0
r3xr5      r3xr0
---------------------------
r3xr5 (r5xr1+r3xr0) r7xr0
  r3        r0        r7
```

Quedando el resultado en los registros `{r3r0r7}`. Al realizar la suma `(r5xr1+r3xr0)` tenemos que tener en cuenta los posibles acarreos que se produzcan.

Para los datos del enunciado (`0x1122 3344` x `0x5566 7788`) el resultado es `0x0005 b736 a6011 7d820`

mul32.s

```
        .data
num1:   .word   0x11223344
num2:   .word   0x55667788
res:    .space  8
        .equ    carry, 0x00010000

    .text
main:
    ldr    r0, =num1      @ cargamos los números
    ldr    r0, [r0]       @ a multiplicar en r0 y r1
    ldr    r1, =num2
    ldr    r1, [r1]

    lsr    r5, r0, #16    @ mitad alta de num1 a r5
    lsr    r3, r1, #16    @ mitad alta de num1 a r3

    lsl    r6, r5, #16
    lsl    r7, r3, #16
    bic    r0, r0, r6     @ mitad baja de num1 queda en r0
    bic    r1, r1, r7     @ mitad baja de num2 queda en r1

    mov    r7, r1         @ r7: copia de la mitad baja de num1

/*

Aplicamos el algoritmo de multiplicación indicado, con
(num 1 = {r5r0} y num2 = {r3r1} o {r3r7})

El resultado queda en {r3r0r7}

*/


    mul    r7, r0, r7   @ parcial r7xr0
    mul    r0, r3, r0   @ parcial r3xr0
    mul    r1, r5, r1   @ parcial r5xr1
    mul    r3, r5, r3   @ parcial r3xr5

    add    r0, r1, r0   @ sumamos r5xr1 + r3xr0
    bcc    saltar       @ vemos si tenemos que trasladar
                        @ un acarreo
```

```
43      ldr   r6, =carry   @ y si es así,
44      add   r3, r3, r6   @ lo añadimos en el lugar correcto
45
46   saltar:
47      lsl   r4, r0, #16  @ colocamos en su sitio las
48      lsr   r0, r0, #16  @ distintas partes (r3 - r0 - r7)
49      add   r7, r7, r4   @ 32 bits menos significativos
50      adc   r3, r3, r0   @ 32 bits más significativos
51
52      ldr   r6, =res     @ dirección del resultado
53      str   r7, [r6]     @ almacenamos los 32 bits menos sig.
54      str   r3, [r6, #4] @ y los 32 bits más significativos
55
56      wfi
57      .end
```

Ejercicio 3.28

Realizar un programa que transforme un número entero de 32 bits almacenado a partir de la posición de memoria etiquetada como `num` en su representación en punto flotante normalizada, con base 2, normalización entera, con una mantisa de 31 bits, 1 bit para el signo (bit más significativo), representación con signo y magnitud para la mantisa para los números negativos y una característica de 8 bits en exceso de 128, es decir $C = E + 128$ donde C es la característica y E el exponente. El número 0 se representa con una mantisa y una característica de valor 0.

Por ejemplo, el número `0x1c00abba` se representará como `0x7002aee8 x 2exp-2` (signo y mantisa SM = `0x7002aee8` y característica C = -2 + 128 = 126 = `0x7e`) y el número entero `0x0000abba` como `0x55dd0000 x 2exp-15` (signo y mantisa SM = `0x55dd0000` y característica C = -15 + 128 = 113 = `0x71`), donde `exp` representa el operador exponenciación.

En el caso del número negativo `-3000`, cuya representación en complemento a 2 es `0xfffff448` y cuyo valor absoluto en hexadecimal es `0xbb8`, se representará como `0x-5dc00000 x 2exp-19` (signo y mantisa SM = `0xddc00000` y característica C = -19 + 128 = 109 = `0x6d`)

El programa almacenará el signo y la mantisa a partir de la posición de memoria etiquetada como `mant` y la característica en la posición etiquetada como `car`.

Para probar el programa, utilice las siguientes definiciones:

```
      .data
num:  .word  0x07bbaacc
mant: .space 4
car:  .space 1
```

Solución

Si el número original es positivo, lo cargamos en el registro `r0` y lo vamos desplazando a la izquierda hasta que el bit de signo cambia a `1`, a la vez que incrementamos un contador en `r1`. Cada vez que desplazamos una unidad a la izquierda habremos multiplicado dicho número por dos, con lo que el registro `r1` representa el número de multiplicaciones realizadas hasta que el bit más significativo de la representación binaria del número pasa a ser `1`.

En el momento en que el bit de signo pase de `0` a `1`, tendremos:

- En `r1`, el valor del exponente de la representación en punto flotante normalizada cambiado de signo (p. ej. 3 desplazamientos significan que tenemos que multiplicar el resultado final por 2^{-3} para recuperar el entero positivo original).
- En `r0`, el valor de la mantisa, sin el bit más significativo, que pasó al bit de signo.

En este punto sólo nos queda:

- Calcular la característica C como `-r1 + 128` (es decir, $C = E + 128$. Para ello restamos de 128 del valor del exponente cambiado de signo que tenemos en `r1` utilizando la instrucción **`sub`**, dejando el resultado en `r1`.
- Recuperar el bit perdido que pasó al bit de signo y el bit de signo original (que debe de ser `0`, ya que suponemos que el número era positivo). Para ello utilizamos la instrucción **`lsr`**.

Si el número original es negativo, tomamos nota del signo en el registro `r2`, calculamos su valor absoluto o magnitud utilizando la instrucción **`neg`** y normalizamos según los pasos anteriores. Al completar la normalización, añadimos el bit de signo correcto a la mantisa para obtener la representación requerida de signo y magnitud.

Con los datos de las definiciones del ejercicio, el signo S es `0`, la mantisa M es `0x7bbaacc0` y la característica C es `0x7c`. Podemos probar también con el resto de los ejemplos del enunciado.

normaliza.s

```
1
2        .data
3  num:  .word    0x07bbaacc       @ número de ejemplo
4  mant: .space   4                @ mantisa y signo
5  car:  .space   1                @ característica
```

```
6          .equ signo, 0x80000000  @ signo negativo
7
8       .text
9   main:
10      ldr   r0,=num
11      ldr   r0, [r0]      @ r0: número a normalizar
12      mov   r1, #0        @ r1: para acumular el exponente
13      mov   r2, #0        @ r2: signo (0 = positivo)
14      cmp   r0, #0        @ comprobamos si el número es cero
15      beq   nada          @ si lo es, hemos terminado
16      bgt   bucle         @ comprobamos si es positivo
17
18      ldr   r2, =signo    @ si es negativo, guardamos el signo
19      neg   r0, r0        @ y normalizamos el valor absoluto
20
21  bucle:
22      lsl   r0, r0, #1    @ movemos los bits a la izquierda
23      blt   final         @ salió un 1 por Z -> acabamos
24      add   r1, r1, #1    @ incrementamos en 1 el exp.
25      b     bucle         @ y seguimos buscando
26
27  final:
28      lsr   r0, r0, #1    @ recuperamos el 1 perdido por Z
29      orr   r0, r0, r2    @ ponemos el signo correcto
30      mov   r2, #128
31      sub   r1, r2, r1    @ característica: -r1 + 128
32
33  nada:
34      ldr   r3, =mant     @ almacenamos signo y mantisa
35      str   r0, [r3]      @ y característica
36      strb  r1, [r3, #4]  @ (0's si originalmente num = 0)
37
38      wfi
39      .end
```

Ejercicio 3.29

Dada una lista de caracteres ASCII almacenados en un bloque de memoria etiquetado como `cad`, realizar un programa que convierta todas las palabras (entendidas como secuencias de letras separadas por caracteres ASCII no alfabéticos) de tal modo que la primera letra se asegure que sea mayúscula y el resto de letras consecutivas sean minúsculas, dejando el resultado en el bloque de memoria etiquetado como `conv`. El final de la cadena se delimita con un cero.

Por ejemplo, dadas las definiciones siguientes, el resultado almacenado a partir de la posición `conv` será la cadena `Hola Mundo Cruel: []Vaya Par De2Gemelos{}`.

```
      .data
cad:  .asciz "hOlA mUndo CRUEL: []vaya PAR de2gEMELOS{}"
      .balign 4
conv: .space  64
```

Solución

Vamos recorriendo la cadena de caracteres utilizando el registro `r0` como puntero a lo largo de la misma. En cada iteración, leemos un carácter en `r1`.

- En el caso de que sea un carácter alfabético al principio de una secuencia de caracteres alfabéticos (es decir, que el carácter anterior no era alfabético), lo pasamos a mayúscula.
- En el caso de que sea un carácter alfabético dentro de una secuencia de caracteres alfabéticos (es decir, que el carácter anterior era alfabético), lo pasamos a minúscula.
- En el resto de los casos, se tratará de un carácter no alfabético y por lo tanto lo dejamos como está.

Utilizamos el registro `r2` para indicar si en esta iteración estamos al principio de una secuencia de caracteres alfabéticos. Si (`r2` = 0) indicará que estamos al principio de una secuencia de caracteres alfabéticos o palabra.

- Si (`r2` = 0) y este carácter es alfabético, hacemos (`r2` = 1). Ya no estamos a principio de palabra.
- Si (`r2` = 0) y este carácter no es alfabético, no hacemos nada y dejamos `r2` como está.
- Si (`r2` = 1) y este carácter es alfabético, seguimos dentro de una palabra, por lo que dejamos `r2` como está.
- Si (`r2` = 1) y este carácter no es alfabético, hacemos (`r2` = 0). Se terminó la palabra y el siguiente carácter alfabético que aparezca será principio de palabra.

capita.s

```
1       .data
2 cad:  .asciz "hOlA mUndo CRUEL: []vaya PAR de2gEMELOS{}"
3       .balign 4
4 conv: .space  64 @ espacio para almacenar la cadena convertida
```

```
5
6      .text
7   main:
8      mov  r2,#0    @ r2: indica si principio de palabra (r2 = 0)
9      mov  r4,#0xdf @ r4: máscara para pasar a mayúsculas
10     mov  r5,#0x20 @ r5: máscara para pasar a minúsculas
11     ldr  r3,=conv @ r3: puntero al destino
12     ldr  r0,=cad  @ r0: puntero al principio de la cadena
13  bucle:
14     ldrb r1,[r0]  @ cargamos un carácter en r1
15     cmp  r1,#0    @ vemos si hemos terminado
16     beq  listo    @ (la cadena termina con un 0)
17
18     cmp  r1,#'A'  @ filtramos los caracteres no alfanuméricos
19     bmi  initp    @ los menores que 'A' no son alfanuméricos
20     cmp  r1,#'z'  @ los mayores que 'z' tampoco
21     bpl  initp    @ ahora sólo quedan entre 'A' y 'z'
22     cmp  r1,#'Z'  @ los menores que 'Z' son letras mayúsculas
23     ble  eslet    @ los de en medio no son alfanuméricos
24     cmp  r1,#'a'  @ los mayores que 'a' son letras minúsculas
25     bmi  initp
26
27  eslet:
28     cmp  r2,#0    @ se trata de una letra
29     bne  aminus   @ vemos si principio de palabra (r2 = 0)
30
31     and  r1,r1,r4 @ convertimos a mayúscula
32     mov  r2,#1    @ ya no es principio de palabra (r2 = 1)
33     b    otro
34
35  aminus:
36     orr  r1,r1,r5 @ no estamos a principio de palabra
37     b    otro     @ convertimos a minúscula
38
39  initp:
40     mov  r2,#0    @ venimos de un carácter no alfabético
41                   @ sigu. letra: principio de palabra (r2 = 0)
42  otro:
43     strb r1,[r3]  @ almacenamos el carácter
44     add  r0,r0,#1 @ y apuntamos al siguiente
45     add  r3,r3,#1 @ en origen y destino
46     b    bucle    @ siguiente carácter
47
48  listo:
49     strb r1,[r3]  @ almacenamos el cero final
50     wfi
51     .end
```

Ejercicio 3.30

El objetivo de este ejercicio es realizar un programa que divida un número entero de 32 bits entre 10 . Ya conocemos el método de la división euclídea (cf. ejercicio 3.19). En este ejercicio vamos a aprovechar el hecho de que

$$\frac{8}{10} \simeq 0,1100110011001100\ldots$$

y por lo tanto

$$\frac{N}{10} \simeq \frac{1}{2^3}(\frac{N}{2^1} + \frac{N}{2^2} + 0 + 0 + \frac{N}{2^5} + \frac{N}{2^6} + 0 + 0 + \frac{N}{2^9} + \frac{N}{2^{10}} + \ldots)$$

Si hacemos $q = N/2^1 + N/2^2$, tenemos que:

$$\frac{N}{10} \simeq \frac{1}{2^3}(q + q\frac{1}{2^4} + q\frac{1}{2^8} + q\frac{1}{2^{16}} + \ldots)$$

Como en un ordenador la precisión de N es limitada (p. ej., un número de 32 bits), si tomamos suficientes términos de la aproximación conseguiremos un valor lo suficientemente preciso como para no cometer errores en el resultado. En el caso de un número de 32 bits, con los términos indicados en la última expresión es suficiente para cometer un error de como máximo una unidad.

Por lo tanto, podemos utilizar el algoritmo siguiente para dividir un entero por 10, considerando que cada desplazamiento a la derecha de un entero divide dicho entero por 2 (el operador >> es el desplazamiento a la derecha):

```
unsigned int coci; /* cociente */
unsigned int resto;

void div10(unsigned int num) {
    unsigned int coci, resto;
    coci = (num >> 1) + (num >> 2); // calculamos q
    coci = coci + (coci >> 4);      // sumamos q + q/2^4
    coci = coci + (coci >> 8);      // sumamos q + q/2^4 + q/2^8
    coci = coci + (coci >> 16);     // sumamos q + q/2^4 + q/2^8 + q/2^16
    coci = coci >> 3;               // multiplicamos por 1/2^3

// calculamos el resto como num - coci x 10 = num - coci x (4 + 1) x 2

    resto = num - (((coci << 2) + coci) << 1);

// ajustamos cociente y resto si el resto es mayor que 9
```

```
    if (resto > 9) {
     coci = coci + 1;
     resto = resto - 10;
    }

}
```

Realice un programa que divida un entero de 32 bits entre 10, utilizando el algoritmo descrito y las definiciones siguientes:

```
       .data
num:   .word   235452
coci:  .space  4
res:   .space  4
```

Solución

La solución consiste en programar directamente el algoritmo anterior. Suponiendo que vamos acumulando en `r1` el valor de `coci`, para calcular la suma con desplazamiento

```
coci = coci + (coci >> d);
```

utilizamos el código

```
   lsr r3,r1,#d  @ r3: (coci >> d)
   add r1,r3     @ r1: coci + (coci >> d)
```

utilizando `r3` como almacenamiento temporal.

div10.s

```
1        .data
2  num:  .word  235452
3  coc:  .space 1
4  res:  .space 1
5
6        .text
7  main: ldr r1,=num
8        ldr r1,[r1]     @ r1: coci, inicialmente num
9        mov r2,r1       @ r2: resto, inicialmente num
10       lsr r3,r1,#2    @ r3: coci >> 2
11       sub r1,r3       @ r1: coci - (coci>>2) = (coci>>1 + coci>>2)
12       lsr r3,r1,#4
13       add r1,r3       @ r1: coci + (coci >> 4)
14       lsr r3,r1,#8
15       add r1,r3       @ r1: coci + (coci >> 8)
16       lsr r3,r1,#16
17       add r1,r3       @ r1: coci + (coci >> 16)
18       lsr r1,#3       @ r1: coci >> 3  = cociente final
19
```

```
20        lsl r3,r1,#2
21        add r3,r1       @ r3: (((coci << 2) + coci) << 1 =
22        lsl r3,#1       @ = coci * ((4 + 1) * 2) = coci * 10
23        sub r2,r3       @ r2: num - r3 = resto
24
25        cmp r2,#10      @ ¿resto > 9?
26        blt fin
27        add r1,#1       @ coci = coci +1
28        sub r2,#10      @ resto = resto - 10
29
30  fin:  ldr r3,=coc     @ almacenamos el resultado
31        str r1,[r3]     @ cociente
32        str r2,[r3,#4]  @ resto
33
34        wfi
35       .end
```

Ejercicio 3.31

El objetivo de este ejercicio es realizar un programa que cuente el número de bits a 1 que tiene una palabra de 32 bits utilizando el método *divide y vencerás*. El método consiste en sustituir primero cada campo de 2 bits con la suma de los dos bits individuales que estaban originalmente en el campo y luego sumar los campos adyacentes de 2 bits, colocando los resultados en cada campo de 4 bits, y así sucesivamente.

El método se ilustra a continuación. En la primera fila vemos una palabra de 32 bits cuyos bits a 1 se deben sumar (`0x23475fc1`), las filas sucesivas muestran los agrupamientos y sumas de bits descritas y la última fila muestra el resultado (16 bits a uno).

```
|0 0 1 0 0 0 1 1 0 1 0 0 0 1 1 1 0 1 0 1 1 1 1 1 1 1 0 0 0 0 0 1|

|0 0|0 1|0 0|1 0|0 1|0 0|0 1|1 0|0 1|0 1|1 0|1 0|1 0|0 0|0 0|0 1|

|0 0 0 1|0 0 1 0|0 0 0 1|0 0 1 1|0 0 1 0|0 1 0 0|0 0 1 0|0 0 0 1|

|0 0 0 0 0 0 1 1|0 0 0 0 0 1 0 0|0 0 0 0 0 1 1 0|0 0 0 0 0 0 1 1|

|0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1|0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1|

|0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0|
```

Este método se puede aplicar de manera sencilla mediante máscaras, desplazamientos y sumas. Con una máscara adecuada y desplazamientos,

podemos filtrar y alinear los bits a sumar en cada una de las cinco etapas. Un posible algoritmo basado en este método sería el siguiente

```
unsigned int num1s(unsigned int n) {
      n = (n & 0x55555555) + ((n >> 1)  & 0x55555555);
      n = (n & 0x33333333) + ((n >> 2)  & 0x33333333);
      n = (n & 0x0f0f0f0f) + ((n >> 4)  & 0x0f0f0f0f);
      n = (n & 0x00ff00ff) + ((n >> 8)  & 0x00ff00ff);
      n = (n & 0x0000ffff) + ((n >> 16) & 0x0000ffff);

      return n;
}
```

En vez de realizar un bucle con 32 iteraciones, podemos resolver el problema en 5 pasos.

Realice el programa solicitado de acuerdo con el algoritmo anterior. Utilice las definiciones siguientes para probar el programa:

```
       .data
num:   .word   0x23475fc1 @ tiene 16 bits a uno
res:   .byte   1
```

Solución

La solución es una implementación directa del algoritmo anterior. Utilizamos los registros `r0`, `r1` y `r2` para programar la etapa

```
r0 = (r0 & v) + ((r0 >> d) & v)
```

de la siguiente manera:

```
    mov r1,r0     @ r1: copia de r0
    ldr r2,=v     @ r2: v
    and r1,r1,r2 @ (r0 & v)
    lsr r0,r0,#d @ (r0 >> d)
    and r0,r0,r2 @ ((r0 >> d) & v)
    add r0,r0,r1 @ r0: (r0 & v) + ((r0 >> d) & v)
```

num1s.s

```
1        .data
2  num:   .word 0x23652fe1
3  res:   .space 1
4
5        .text
6
7  main: ldr r0,=num
8        ldr r0,[r0]
9
10 @ r0 = (r0 & 0x55555555) + ((r0 >> 1) & 0x55555555)
```

```
11
12          mov r1,r0
13          ldr r2,=0x55555555
14          and r1,r1,r2
15          lsr r0,r0,#1
16          and r0,r0,r2
17          add r0,r0,r1
18
19  @ r0 = (r0 & 0x33333333) + ((r0 >> 2) & 0x33333333)
20
21          mov r1,r0
22          ldr r2,=0x33333333
23          and r1,r1,r2
24          lsr r0,r0,#2
25          and r0,r0,r2
26          add r0,r0,r1
27
28  @ r0 = (r0 & 0x0f0f0f0f) + ((r0 >> 4) & 0x0f0f0f0f)
29
30          mov r1,r0
31          ldr r2,=0x0f0f0f0f
32          and r1,r1,r2
33          lsr r0,r0,#4
34          and r0,r0,r2
35          add r0,r0,r1
36
37  @ r0 = (r0 & 0x00ff00ff) + ((r0 >> 8) & 0x00ff00ff)
38
39          mov r1,r0
40          ldr r2,=0x00ff00ff
41          and r1,r1,r2
42          lsr r0,r0,#8
43          and r0,r0,r2
44          add r0,r0,r1
45
46  @ r0 = (r0 & 0x0000ffff) + ((r0 >> 16) & 0x0000ffff)
47
48          mov r1,r0
49          ldr r2,=0x0000ffff
50          and r1,r1,r2
51          lsr r0,r0,#16
52          and r0,r0,r2
53          add r0,r0,r1
54
55          ldr r1,=res
56          strb r0,[r1]
57
58          wfi
59         .end
```

Ejercicio 3.32

Realizar un programa que sume dos números BCD empaquetados de 8 dígitos almacenados a partir de las posiciones de memoria etiquetadas como `num1` y `num2` y deje el resultado, también en formato BCD empaquetado, a partir de la posición de memoria etiquetada como `res`.

Para probar el programa, utilice las siguientes definiciones:

```
      .data
num1: .byte  0x35, 0x64, 0x28, 0x01
num2: .byte  0x46, 0x58, 0x90, 0x01
res:  .space 4
```

Solución

Cada sumando ocupa cuatro bytes consecutivos y cada byte almacena dos dígitos BCD, uno en cada uno de los dos *nibbles* (grupos de 4 bits) de cada byte.

Para realizar la suma, vamos leyendo los bytes de cada sumando, comenzando por el menos significativo, y los sumamos, teniendo en cuenta si hay algún acarreo de la suma de los dígitos en el byte anterior, que también habrá que sumar.

Para sumar los dos dígitos de un byte, filtramos los *nibbles* correspondientes, dejando en `r3` y `r4` los *nibbles* / dígitos menos significativos y en `r1` y `r2` los *nibbles* / dígitos más significativos. Sumamos los dígitos por separado.

Los dígitos en BCD toman los valores de 0 a 9. Si al sumar dos dígitos obtenemos un valor mayor que 9, habrá que realizar un ajuste. En ese caso, restamos 10 al resultado para obtener el dígito correcto e indicamos que se ha producido un acarreo. Por ejemplo: `0x8` + `0x7` = `0xf`. El resultado correcto será `0xf` - `0xa` = `0x5` y nos llevamos una (indicamos un acarreo de 1, y el resultado en BCD será `0x15`).

El resultado de sumar los números 35.642.801 y 46.589.001 del enunciado, cuyas representaciones en BCD empaquetado son respectivamente `0x35 64 28 01` y `0x35 64 28 01`, sería:

```
  0x35 64 28 01
+ 0x46 58 90 01
----------------
  0x82 23 18 02
```

Es decir, el número 82.231.802, cuya representación en BCD empaquetado es `0x0x82 23 18 02`.

sumabcd.s

```
/*
 Suma dos números BCD empaquetados de 8 dígitos y deja el resultado
 en BCD empaquetado.

*/

      .data
num1: .byte  0x35, 0x64, 0x28, 0x01
num2: .byte  0x46, 0x58, 0x90, 0x01
cont: .space 4          @ contador de bytes
dres: .space 4          @ dirección del byte sumado
res:  .space 4          @ dirección del resultado

   .equ   lon, 4        @ longitud del número en bytes
   .equ   mask, 0x0f    @ mascara
   .equ   offset, 0x03 @ desp. al digito menos signif.

   .text
main:
   ldr   r1, =num1     @ cargamos en r1 el byte menos
   add   r1, r1, #offset
   ldrb  r1, [r1]      @ significativo del primer numero

   ldr   r2, =num2     @ cargamos en r2 el byte menos
   add   r2, r2, #offset
   ldrb  r2, [r2]      @ significativo del segundo numero

   mov   r3, #lon      @ inicializamos el contador
   ldr   r4, =cont     @ con el tamaño en bytes de los sumandos
   str   r3, [r4]

   ldr   r3, =res      @ inicializamos un puntero al resultado
   add   r3, r3, #offset
   ldr   r4, = dres    @ empezando por el byte menos significativo
   str   r3, [r4]

   mov   r5, #0        @ r5: acarreo entre dígitos, inicialmente a 0

bucle:
   mov   r3, r1        @ copiamos el byte en curso
   mov   r4, r2        @ de ambos números a r3 y r4
   mov   r0, #mask     @ r0: máscara para filtrar 4 bits

   and   r3, r3, r0    @ nos quedamos con el nibble menos
   and   r4, r4, r0    @ significativo de ambos números (r3 y r4)
   lsr   r1, r1, #4    @ y dejamos preparado el nibble
   lsr   r2, r2, #4    @ mas significativo (r1 y r2)

   add   r6, r3, r4    @ sumamos los dígitos menos significativos
```

```
50      add   r6, r6, r5    @ y el acarreo (de haberlo) en r6
51      cmp   r6, #0x0a     @ ¿da mas de 10?
52      blt   rc1           @ si no, el acarreo es 0
53      mov   r5, #1        @ si da mas de 10, el acarreo es 1
54      sub   r6, r6, #0x0a @ y restamos 10 del resultado
55      b     sigu
56
57  rc1:
58      mov   r5, #0        @ acarreo a 0
59
60  @ tenemos sumados los dos dígitos menos significativos de
61  @ un byte en r6, y en r5 el acarreo correspondiente
62
63  sigu:
64      mov   r3, r1        @ seguimos con el segundo dígito del byte
65      mov   r4, r2        @ hacemos lo mismo que con el primer dígito
66      and   r3, r3, r0    @ nos quedamos con el dígito a sumar
67      and   r4, r4, r0    @ de ambos bytes
68      lsr   r1, r1, #4    @ desplazamos a la derecha
69      lsr   r2, r2, #4    @ ambos nibbles y
70      add   r7, r3, r4    @ sumamos ambos nibbles con el acarreo
71      add   r7, r7, r5    @ del menos significativo al mas significativo
72      cmp   r7, #0x0a     @ vemos si hay acarreo, y ajustamos
73      blt   rc2           @ el resultado restando 10 si es necesario
74      mov   r5, #1
75      sub   r7, r7, #0x0a
76      b     fin
77
78  rc2:
79      mov   r5, #0        @ acarreo a 0
80
81  @ tenemos sumados los dos dígitos más significativos de
82  @ un byte en r7, y en r5 el acarreo correspondiente
83
84  fin:
85      lsl   r7, r7, #4    @ montamos los dos nibbles
86      orr   r6, r6, r7    @ en el byte menos significativo de r6
87      ldr   r0, =dres     @ recuperamos la dir. de destino
88      ldr   r0, [r0]
89      strb  r6, [r0]      @ almacenamos el resultado
90      sub   r0, r0, #1    @ y apuntamos al siguiente dígito
91
92      ldr   r3, =cont     @ recuperamos el contador
93      ldr   r3, [r3]
94      sub   r3, r3, #1    @ lo decrementamos
95      beq   acabo         @ y vemos si acabamos
96
97      ldr   r1, =cont     @ no acabamos: actualizamos
98      str   r3, [r1]      @ la dirección de destino
```

```
99      ldr    r1, =dres       @ y el contador
100     str    r0, [r1]
101
102     ldr    r1, =num1       @ cargamos los siguientes dígitos
103     add    r1, r1, r3      @ en r1 y r2
104     sub    r1, r1, #1      @ utilizamos el contador como
105     ldrb   r1, [r1]        @ puntero a la zona de destino
106                            @ (el desplazamiento es uno menos
107     ldr    r2, =num2       @ de lo que marca el contador)
108     add    r2, r2, r3      @ diri = numi + cont - 1
109     sub    r2, r2, #1
110     ldrb   r2, [r2]
111
112     b      bucle           @ nueva iteración
113
114 acabo:
115     wfi
116     .end
```

Capítulo 4
A toda máquina

En este capítulo comenzamos introduciendo el concepto de subrutina o subprograma y la programación de subprogramas tomando como referencia la arquitectura ARM T32/Thumb. En definitiva, ilustramos el uso de la instrucción `bl` para invocar un subprograma y el uso de `mov pc,lr` para retornar al programa principal.

Más adelante trabajamos el concepto de pila implementada en la memoria de acceso aleatorio del ordenador.

Una vez que nos hemos familiarizado con el uso de la pila, retomamos el concepto de subprograma con el objetivo de introducir el paso de parámetros a través de la pila y la salvaguarda de registros en la pila. También estudiaremos las llamadas anidadas a subprogramas y la recursividad.

4.1 Subprogramas

En este primer contacto con el concepto de subprograma nos centramos en el paso de parámetros a través de registros, e ilustramos este concepto a través de su utilización en una serie de problemas elegidos por su idoneidad pedagógica. Seguimos el convenio de llamada a subprogramas del AAPCS (*ARM Architecture Procedure Call Standard*), que propone el uso de los registros `r0` a `r3` para el paso de argumentos a subprogramas y para el paso de los resultados de la ejecución de un subprograma al programa llamante. Además, las variables locales se deben almacenar también en los registros `r0` a `r3` de ser posible. El programa llamante no puede suponer que el subprograma va a preservar los valores de ninguno de los registros `r0` – `r3`. Más adelante, en el apartado 4.3, veremos como la pila nos proporciona un mecanismo adecuado para permitir que los subprogramas usen todos los registros sin riesgo de destruir datos de otros subprogramas o programas llamantes.

Aprovechamos además para introducir la codificación de caracteres UTF-8 (cf. apéndice C) mediante un ejercicio de conversión de ASCII extendido a

dicho sistema de codificación, y un concepto importante en la programación de sistemas como es el concepto de paridad.

Ejercicio 4.1

Realizar un programa que reorganice una zona de memoria que contiene números enteros positivos de dos bytes (medias palabras). El programa deberá rotar los números de la zona de memoria hacia dos posiciones más bajas (4 direcciones de memoria) que la que ocupan actualmente, de manera circular, es decir, el que ocupa la posición 3 deberá ocupar la posición 1, el que ocupa la 4 deberá almacenarse en la 2 y así sucesivamente. Finalmente, el elemento que ocupa la posición de memoria 1 deberá pasarse a la penúltima posición y el que ocupa la posición 2 la última (cf. figura 4.1).

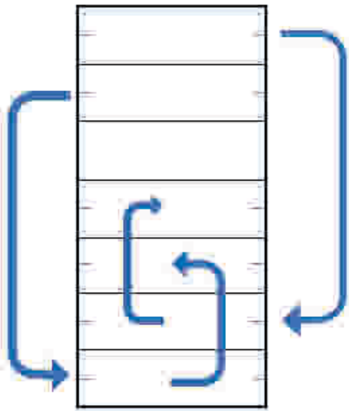


Figura 4.1. Reorganización de una zona de memoria propuesta en el enunciado. La primera media palabra pasará a ocupar la penúltima posición de la zona; la última palabra se almacenará dos posiciones (4 direcciones de memoria) más arriba.

Para ello, realice un subprograma de apoyo que lleve a cabo la rotación de los números (medias palabras) almacenados en una zona de memoria. Dicho subprograma recibirá como argumentos:

- `r0`: dirección de comienzo de la zona de memoria.
- `r1`: número de elementos en la zona de memoria (número de medias palabras).

Tras la ejecución del programa principal, la zona de memoria considerada deberá continuar teniendo los mismos números enteros que antes de la ejecución del mismo. Obviamente, dichos números podrán estar en un orden diferente, pero no podrá faltar ni repetirse ninguno dentro de la citada zona).

Para realizar el ejercicio, utilice la zona de memoria siguiente:

```
zona: .hword      2, 1, 7, 8, 5, 6, 4, 9, 3
```

Solución

El subprograma `rota` será el subprograma de apoyo encargado de realizar una rotación de la zona de memoria. Para ello, primero toma el primer elemento de la misma y lo deposita en el último lugar, guardando el valor original de la última posición en `r2`. A continuación, vamos desplazando los elementos de la zona hacia direcciones decrecientes desde el segundo elemento:

```
ldrh r3,[r0,#2]  @ leemos un dato
strh r3,[r0]     @ lo almacenamos en la posición anterior
```

Finalmente, almacenamos el valor que va en penúltimo lugar, que estaba originalmente en el último y que habíamos reservado en `r2`.

El programa principal simplemente llamará al subprograma de apoyo dos veces.

rot_num.s

```
1        .data
2  zona: .hword 2, 1, 7, 8, 5, 6, 4, 9, 3
3        .equ   tam, 9
4
5        .text
6
7  /* Tenemos que rotar la zona dos veces */
8
9  main: ldr   r0,=zona     @ r0: dirección de la zona
10       mov   r1,#tam      @ r1: tamaño
11       bl    rota         @ rotamos
12
13       ldr   r0,=zona
14       mov   r1,#tam
15       bl    rota         @ rotamos otra vez
16 nomov:
17       wfi
18
19 /*
20  rota: Subprograma para rotar las medias palabras
21  almacenadas en una zona de memoria.
```

```
22   Argumentos:
23        r0: dirección de comienzo de la zona.
24        r1: tamaño (en medias palabras) de la zona.
25  */
26  rota:
27  /*
28  Primero comprobamos si hay medias palabras que rotar
29  */
30
31        cmp  r1,#1       @ si la zona no tiene al menos dos
32        ble  done        @ medias palabras, hemos terminado.
33
34        sub  r1,r1,#1    @ calculamos el fin de la zona
35        lsl  r1,#1
36        add  r1,r1,r0    @ r1: apunta ahora al último elemento
37
38        ldrh r2,[r1]     @ movemos el primero al
39        ldrh r3,[r0]     @ último a través de r3
40        strh r3,[r1]
41
42  otro: ldrh r3,[r0,#2]  @ movemos hacia abajo
43        strh r3,[r0]     @ desde el segundo
44        add  r0,#2
45        cmp  r0, r1
46        bne  otro
47
48        sub  r0,#2
49        strh r2,[r0]     @ movemos el último al penúltimo
50
51  done: mov  pc,lr       @ y terminamos
52        .end
```

Ejercicio 4.2

Realizar un programa que compare dos cadenas de caracteres terminadas con 0, almacenadas en memoria a partir de las posiciones etiquetadas como `cad1` y `cad2`, dejando el resultado de la comparación en la posición de memoria etiquetada como `res`. El programa ofrecerá como resultado el valor `0xff` en el caso de que las cadenas sean diferentes y el valor `0x00` en el caso de que sean iguales.

Utilice dos subprogramas de soporte, uno para calcular la longitud de una cadena y otro para comparar dos cadenas de igual longitud.

Para probar el programa, utilice las cadenas siguientes:

```
      .data
cad1: .asciz "¡Hola Mundo!"
cad2: .asciz "¡Hola Mundo!"
```

```
res:  .space      1
```

Solución

El primer subprograma, etiquetado como `tamc`, va leyendo caracteres de la cadena hasta que se encuentra el carácter 0, a la vez que va contando el número de iteraciones. El número de iteraciones será el tamaño de la cadena.

```
i = 0;
while (cad[i] != 0) {
    i = i + 1;
}
tamaño = i;
```

El segundo subprograma, etiquetado como `eqc`, compara los caracteres de dos cadenas dos a dos, suponiendo que ambas son de igual longitud, hasta que encontramos caracteres diferentes o llegamos al final de las cadenas:

```
i = 0;
while (cad1[i] == cad2[i]) {
    if (cad1[i] == 0) break;
    i = i + 1;
}

if (cad1[i] = 0)
      result = 0;     /* iguales */
else
      result = 0xff; /* diferentes */
```

El programa principal llama a los dos subprogramas en secuencia.

1. Primero se comprueba el tamaño de ambas con `tamc`. Si las cadenas son de distinto tamaño, son diferentes.
2. Si son del mismo tamaño, se comprueba si son iguales con `eqc`.

comp_cad_srt.s

```
/*
 Comparamos dos cadenas terminadas en 0

*/
      .data
cad1: .asciz "¡Hola Mundo!"
cad2: .asciz "¡Hola Mundo!"
res:  .space 1

```

```
10          .text
11  main:   ldr     r5, =cad1       @ direcciones de las cadenas
12          ldr     r6, =cad2
13          mov     r7, #0xff       @ suponemos que son distintas
14
15          mov     r0, r5
16          bl      tamc            @ calculamos el tamaño de la primera
17          mov     r4, r0
18
19          mov     r0,r6
20          bl      tamc            @ y el de la segunda
21          mov     r3, r0
22
23          cmp     r3, r4
24          bne     fin             @ si los tamaños son diferentes
25                                  @ las cadenas no pueden ser iguales
26          mov     r0, r5
27          mov     r1, r6
28          bl      eqc             @ vemos si son iguales
29          mov     r7, r0          @ recuperamos el resultado
30
31  fin:    ldr     r1, =res
32          strb    r7, [r1]        @ almacenamos el resultado
33          wfi
34
35  @
36  @ subrutina de cálculo del tamaño de una cadena de caracteres
37  @ terminada en 0
38  @
39  @       entrada: en r0: dir. de comienzo de la cadena (1 word)
40  @       salida:  en r0: tamaño (1 word)
41  @
42  tamc:
43          mov     r2, #0      @ contador a cero
44  buctc:
45          ldrb    r1, [r0]    @ cargamos byte de primera cadena
46          cmp     r1, #0      @ vemos si hemos terminado
47          beq     tcd
48          add     r2, r2, #1  @ incrementamos contador
49          add     r0, r0, #1  @ incrementamos el puntero
50          b       buctc
51
52  tcd:    mov     r0, r2      @ almacenamos el resultado
53          mov     pc,lr       @ volvemos
54  @
55  @ subrutina que determina si dos cadenas terminadas en 0 de la misma
56  @ longitud son iguales
57  @
58  @       entrada: r0, r1: direcciones de comienzo de las cadenas
```

```
59  @       salida:   r0: 0x00000000 = son iguales, 0x000000ff son
60  @                 diferentes.
61  @
62  eqc:  ldrb    r2, [r0]       @ cargamos un carácter
63        cmp     r2, #0         @ si llegamos al final
64        beq     fieqc          @ es que son iguales
65        ldrb    r3, [r1]       @ cargamos uno de la otra cadena
66        cmp     r2, r3         @ los comparamos
67        bne     disteqc        @ si no, cadenas distintas
68        add     r0, r0, #1     @ actualizamos punteros
69        add     r1, r1, #1
70        b       eqc            @ otro carácter
71
72  fieqc:
73        mov     r0, #0x00      @ son iguales
74        mov     pc,lr
75
76  disteqc:                     @ son diferentes
77        mov     r0, #0xff
78        mov     pc, lr
79        .end
```

Ejercicio 4.3

Realizar un programa que convierta una secuencia de bytes ISO Latin 1 (ASCII extendido) en el rango `0x00` a `0xff`, almacenados a partir de la posición de memoria etiquetada como `cad`, en caracteres UTF-8 de uno o dos bytes, según la codificación descrita en el Apéndice C. El resultado se almacenará a partir de la dirección etiquetada como `res`.

Para probar el programa, utilice la cadena de ejemplo siguiente:

```
      .data
cad:  .byte      0xa1, 'H', 'o', 'l', 'a', ' ', 'p', 'e', 'q', 'u', 'e'
      .byte      0xf1, 'o', ' ', 'm', 'u', 'n', 'd', 'o', ' ', 'm'
      .byte   0xed, 'o', '!', 0x00
res:  .space     36
```

Solución

Hacemos uso de un subprograma de soporte (`btoutf8`) que recibe en `r0` un carácter ISO Latin 1 y devuelve en `r0` su representación en UTF-8 (uno o dos bytes). Recorremos la secuencia de bytes y vamos convirtiendo cada byte en dicha secuencia utilizando el programa de soporte.

btoutf8.s

```
      .data
cad: .byte  0xa1, 'H', 'o', 'l', 'a', ' ', 'p', 'e', 'q', 'u', 'e'
      .byte  0xf1, 'o', ' ', 'm', 'u', 'n', 'd', 'o', ' ', 'm'
      .byte  0xed, 'o', '!', 0x00
res: .space 36

    .text
main:
    ldr    r4, =cad     @ r4: dir. de la cadena de partida
    ldr    r5, =res     @ r5: dir. del resultado
buc:
    ldrb   r0, [r4]     @ r0: cargamos un carácter
    cmp    r0, #0       @ vemos si fin de cadena
    beq    acab

    bl     btoutf8      @ convertimos el carácter utilizando
    cmp    r0, #0x80    @ el subprograma; vemos si es de 2 bytes
    blt    solo1

    strb   r0, [r5, #1] @ almacenamos el byte menos signific. (de segundo)
    lsr    r0, r0, #8   @ preparamos el byte más significativo (0xCX)
    strb   r0, [r5, #0] @ para almacenar el 0xCX en su sitio
    add    r5, r5, #2   @ incrementamos el puntero
    b      otro

solo1:
    strb   r0, [r5]     @ almacenamos tal cual
    add    r5, r5, #1
otro:
    add    r4, r4, #1
    b      buc

acab:    wfi


/*

 btoutf8: subprograma para convertir caracteres en el rango
 0x00 a 0xff en caracteres UTF8 de 1 o 2 bytes.

   entrada: r0, byte menos significativo, carácter de entrada
   salida:  r0, conversión de 1 o 2 bytes. resto a cero.
*/
btoutf8:
    mov    r1,#ff
    and    r0,  r0, r1 @ nos quedamos con el byte menos significativo
    cmp    r0, #0x80
    blt    yaconv      @ carácter de 1 byte. No nay que convertirlo.
```

```
50
51      ldr   r2, =0xC000 @ máscara para el byte más significativo
52      ldr   r3, =0xff00 @ máscara para borrar el byte más sign.
53
54      lsl   r1, r0, #2  @ byte de r1: el byte más significativo
55      orr   r1, r1, r2  @ del resultado
56      and   r1, r1, r3
57
58      mov   r2,#0x3f    @ cambiamos los dos bits más signif.
59      and   r0, r0, r2  @ de r0 por 0b10
60      mov   r2,#0x80
61      orr   r0, r0, r2
62
63      orr   r0, r0, r1  @ y montamos todo en r0
64
65   yaconv:
66      mov   pc,lr
67      .end
```

Ejercicio 4.4

Realizar un programa para calcular la representación en código ASCII del valor en hexadecimal de un número entero no negativo de 4 bytes etiquetado como `val`. El resultado se almacenará en a partir de la posición de memoria etiquetada como `res`. Para ello se utilizará un subprograma, etiquetado como `hextoa`, que reciba en los cuatro bits menos significativos de `r0` un número entre 0 y 15 y devuelva, también en `r0`, el código ASCII de su representación en hexadecimal.

Por ejemplo, si el número recibido es `0xa7b5ff90`, el programa deberá almacenar a partir de la posición de memoria etiquetada como `res` la cadena de caracteres “`0xA7B5FF90`”.

Para probar el programa, utilice las siguientes definiciones:

```
     .data
val: .word  0xa7b5ff90
res: .space 10
```

Solución

Primero generamos los caracteres “`0x`”. A continuación, vamos tomando los bits del número de entrada de cuatro en cuatro, depositándolos en registro `r0`, convirtiéndolos a ASCII utilizando el subprograma `hextoa` y almacenando los códigos ASCII obtenidos en su lugar de la cadena resultado `res`.

hextoascii_srt.s

```
1
2        .data
3   val: .word  0xa7b5ff90  @ dato de ejemplo
4   res: .space 10          @ resultado
5
6      .text
7   main:
8      ldr   r4, =val     @ dirección del dato
9      ldr   r5, =res     @ dirección del resultado
10  @
11  @ ponemos el marcador hex ’0x’
12  @
13     mov   r3, #’0’
14     strb  r3, [r5]
15     mov   r3, #’x’
16     strb  r3, [r5, #1]
17     add   r5, r5, #2
18  @
19  @ convertimos y almacenamos
20  @
21     mov   r6, #7       @ contador de dígitos
22     ldr   r4, [r4]     @ cargamos el número en r4
23
24  otro:
25     mov   r0, r4
26     bl    hextoa       @ convertimos el nibble menos significativo
27     strb  r0, [r5, r6] @ lo almacenamos en su sitio
28     cmp   r6, #0       @ ¿hemos terminado?
29     beq   final
30
31     lsr   r4, r4, #4   @ lo desplazamos un nibble
32     sub   r6, r6, #1   @ restamos 1 del contador / puntero
33     b     otro         @ y otra vez
34
35  final:
36     wfi
37
38  /* subrutina para pasar un dígito en binario a ASCII en hexadecimal
39     entrada: r0 dígito en binario (4 bits menos significativos).
40        Se ignora el resto.
41     salida:  r0: carácter ASCII del número (8 bits menos
42        significativos). Resto a cero.
43  */
44
45  hextoa:
46     mov   r1, #0x0f     @ enmascaramos todo menos el nibble menos
47     and   r0, r0, r1    @ significativo
48     cmp   r0, #0x09     @ vemos si ’0’ - ’9’ o ’A’ - ’Z’
49     bhi   msr1
```

```
50
51      add    r0, r0, #'0'  @ menor que 9
52      b      dsr1
53
54  msr1:
55      add    r0, r0, #0x37   @ mayor que 9 : 0x0A + 0x37 = 0x41 = 'A'
56  dsr1:
57      mov    pc,lr
58      .end
```

Ejercicio 4.5

Realizar un programa que tome una secuencia de caracteres ASCII (rango `0x00` a `0x7f`) y convierta el bit más significativo de cada carácter en el bit de paridad correspondiente, de manera que el numero total de bits a 1 en el carácter sea siempre par.

Por ejemplo, este programa convertiría `0x34` en `0xb4` (para que tenga un número par de bits) y dejaría `0x33` como está (ya tiene un número par de bits).

La secuencia a convertir comienza en la dirección de memoria etiquetada como `strm` y termina con el carácter `0x04` (carácter de control de fin de texto `eot`).

Para probar el programa, utilice la cadena de ejemplo siguiente:

```
      .data
strm: .byte 0x00, 0x01, 0x02, 0x70, 0x71, 0x72, 0x0f, 0x04
```

Solución

Utilizamos un programa de apoyo, etiquetado como `setpar`, que cuenta el numero de bits a 1 en el byte menos significativo de `r0`, y que devuelve en el mismo registro `r0` el byte de entrada, pero con la paridad correcta. Para contar los bits a 1, utilizamos la instrucción **`lsl`** y luego incrementamos un contador en función del valor desplazado al bit de signo, tomando la decisión con la instrucción **`bpl`**.

El programa principal analizará la cadena de caracteres hasta encontrar el carácter `eot`, modificando cada uno de ellos utilizando el subprograma `setpar`.

parity.s

```
1
2          .data
3   strm: .byte   0x00, 0x01, 0x02, 0x70, 0x71, 0x72, 0x0f, 0x04
4
5       .text
6   main:
```

```
7      ldr   r4, =strm  @ r4: para navegar la lista de números
8   bucle:
9      ldrb  r0, [r4]
10     cmp   r0, #0x04
11     beq   fin        @ si es EOT, terminamos (sin convertirlo, es par)
12
13     bl    setppar    @ llamamos al subprograma para
14     strb  r0, [r4]   @ convertir este carácter.
15     add   r4, r4, #1
16     b     bucle
17
18  fin:   wfi
19
20  /*
21   setppar: subprograma que cuenta el numero de bits a 1
22   en el byte menos significativo de r0,
23   y devuelve en r0 el byte con la paridad correcta.
24
25  Entrada: r0: byte a comprobar.
26  Salida:  r0: con paridad par.
27
28  */
29
30  setppar:
31     mov   r3, r0      @ guardamos una copia de r0
32     lsl   r3, r3, #24 @ desplazamos el byte hasta que quede al borde
33     mov   r2, #0      @ contador de bits a 1
34     mov   r1, #7      @ contador de desplazamientos
35
36  pbuc:
37     lsl   r3, r3, #1  @ el bit menos significativo va al signo
38     bpl   nosuma      @ si no es uno, pasamos al siguiente
39
40     add   r2, r2, #1  @ incrementamos el contador de unos
41  nosuma:
42     sub   r1, r1, #1
43     bne   pbuc        @ al siguiente si no hemos acabado
44
45     lsr   r2, #1      @ vemos si la cuenta es par o impar
46     bcc   espar       @ si es par, acabamos
47
48     mov   r1, #0x80   @ máscara para activar el bit de paridad
49     orr   r0, r0, r1  @ lo activamos
50
51  espar:
52     mov   pc,lr
53     .end
```

Ejercicio 4.6

Realizar un programa que calcule la representación en decimal de un número entero no negativo de 16 bits utilizando caracteres ASCII. El programa almacenará la representación como una cadena, poniendo el carácter '0' en las posiciones más significativas si es necesario. El número original estará almacenado en la media palabra etiquetada como `num` y el resultado se almacenará a partir de la dirección etiquetada como `res`.

Por ejemplo, el número 22 se representaría mediante la cadena de caracteres "00022" y el número 65321 mediante la cadena "65321". El número 0 se representaría con la cadena "00000". Tenga en cuenta que el número máximo de caracteres necesarios para representar un número de 16 bits es de 5.

Utilice un subprograma para dividir por 10. Dicho subprograma deberá recibir en `r0` el número a dividir por 10, y devolverá el cociente de la división en `r0` y el resto en `r1`.

Para probar el programa, utilice las definiciones siguientes:

```
      .data
num:  .hword      457
res:  .space      5
```

Solución

Nuestro programa irá dividiendo el número original por 10, hasta que el número sea menor que 10. En cada iteración, el resto de la división será uno de los dígitos del número original, comenzando por el dígito menos significativo. Sumamos a dicho resto el valor del código ASCII que representa el '0' para convertirlo en el código ASCII de un número.

```
i = 4;
while (i >= 0) {
   n = n div 10;
   r = n res 10;
   res[i] = r + '0';
   i = i - 1;
}
```

Por ejemplo, los dígitos del número 567 se obtienen como 567 / 10 = 56 con resto/dígito 7; 56 / 10 = 5, con resto/dígito 6 y 5 / 10 = 0, con resto/dígito final 5.

Como conocemos el tamaño máximo de la representación ASCII del número (5 dígitos), no es necesario considerar como caso especial la posibilidad de añadir caracteres '0' en las posiciones más significativas.

Por ejemplo, si el número original fuera 0, obtendríamos “00000” ejecutando el algoritmo 5 veces consecutivas, ya que 0 / 10 = 0, con resto 0.

El subprograma de dividir por 10, etiquetado como **div10**, realiza la división mediante el algoritmo de las restas sucesivas o algoritmo de Euclides (cf. ejercicio 3.19).

bin2dec.s

```
/*
 Representación de un número en decimal con caracteres ASCII.

      Se almacena la representación como cadena, poniendo 0's
      en las posiciones más significativas si es necesario.

*/
      .data
num:  .hword      457
res:  .space      5

      .text
main: ldr       r4, =res      @ dirección del resultado
      mov       r3, #5        @ contador de caracteres
      ldr       r0, =num
      ldr       r0, [r0]      @ numero a convertir

otro:                         @ pasamos (lo que queda d)el número
      bl        div10         @ dividimos por 10
                              @ el resto en r1 será el nuevo dígito.
      add       r1, #'0'      @ pasamos a ascii
      sub       r3, r3, #1
      strb      r1, [r4, r3]  @ almacenamos en su posición

      cmp       r0, #0        @ ¿hemos terminado?
      beq       escero

      cmp       r3, #0        @ ¿el último dígito era el de orden 5?
      beq       fin

escero:
      mov       r0, #'0'      @ rellenamos con ceros, es decir,

otrop:
      sub       r3, r3, #1    @ ponemos 0's en los caracteres que
      strb      r0, [r4, r3]  @ faltan
      cmp       r3, #0
      bne       otrop

fin:  wfi

```

```
42  /*
43   Subprograma para dividir por 10.
44  
45        Entrada: en r0 una palabra con el dividendo
46        Salida:  en r0 el cociente, y en r1 el resto
47  
48        En este ejemplo, realizamos la división por restas sucesivas
49        (algoritmo de Euclides de la división).
50  */
51  div10:
52        mov       r2, #0          @ init cociente a 0 (r2)
53  
54  bucle:
55        cmp       r0, #10         @ ¿divisor menor que 10?
56        blt       fins            @ si lo es, acabamos
57  
58        add       r2, r2, #1      @ sumamos 1 al cociente
59        sub       r0, r0, #10     @ restamos 10 del dividendo
60        b         bucle
61  
62  fins:
63        mov       r1, r0          @ almacenamos el resto
64        mov       r0, r2          @ y el cociente
65        mov       pc, lr
66        .end
```

4.2 La pila

Los ejercicios incluidos en este apartado trabajan el concepto de pila. Se introduce en primer lugar su estructura, funcionamiento y cómo se opera en una *máquina de pila*. A continuación, se proponen varios ejercicios en los que se ilustra el procesado y modificación de datos almacenados en la pila, incluyendo casos en los que el número de datos resultado de las operaciones realizadas es diferente del número de datos almacenados inicialmente en la pila.

Ejercicio 4.7

Dada una cadena de caracteres terminada en 0 almacenada a partir de la dirección `cad`, realice un programa que, usando la pila, la almacene en orden inverso. Por ejemplo, dadas las definiciones siguientes, el resultado sería la cadena `ritrevni a seretcarac ed anedaC`.

```
     .data
cad: .asciz "Cadena de caracteres a invertir"
```

Solución

Utilizamos el hecho de que una memoria de pila es una memoria LIFO (*last in, first out* o *último en entrar, primero en salir*), es decir, que el elemento que extraemos de la pila con una instrucción **pop** es el último que hemos introducido con una instrucción **push**. En otras palabras, si apilamos los caracteres de la cadena en el orden en que están, al desapilarlos los obtendremos en orden inverso.

invcad.s

```
1
2        .data
3  cad: .asciz "Cadena de caracteres a invertir"
4
5      .text
6  main:
7     ldr   r0,=cad    @ dir. comienzo de la cadena
8     mov   r1,#0      @ índice sobre la cadena
9  otro:
10    ldrb r2,[r0,r1] @ leemos un carácter
11    cmp   r2,#0      @ vemos si terminamos de leer
12    beq   leida
13
14    push {r2}       @ apilamos este carácter
15    add   r1,r1,#1   @ incrementamos el índice
16    b     otro
17 leida:
18    cmp   r1,#0      @ vemos si hemos terminado
19    beq   hecho      @ r1 tiene el tamaño de la cadena
20
21    pop   {r2}       @ sacamos un carácter
22    strb r2,[r0]    @ lo almacenamos
23    sub   r1,r1,#1   @ actualizamos contador
24    add   r0,r0,#1   @ y puntero
25    b     leida
26 hecho:
27    wfi
28    .end
```

Ejercicio 4.8

Una máquina de pila es un ordenador en el cual la memoria se comporta como una pila. En estas máquinas, las operaciones se realizan mediante un conjunto de instrucciones que operan implícitamente con los valores en la pila y reemplazan dichos valores por el resultado de dichas operaciones.

El objetivo de este ejercicio es simular una máquina de pila en un ordenador ARM/Thumb. La pila simulada almacenará palabras de 32 bits, tendrá

un tamaño de 64 palabras de memoria y se simula en memoria principal de acuerdo con las siguientes definiciones:

```
        .data
pila:   .space 63*4   @ pila es la dirección del comienzo de la pila
bpila:  .space 4      @ bpila es la dirección de la base (fondo) de la pila
ppila:  .word 1       @ ppila almacena en cada momento el puntero de pila
```

El puntero de pila (`pp`) estará siempre disponible en la dirección de memoria etiquetada como `ppila` y apuntará siempre al último dato introducido en la pila. La pila crece hacia direcciones decrecientes de memoria. Cuando `pp` tome el valor `bpila` querrá decir que la pila está vacía y cuando tome el valor `tpila` querrá decir que la pila está llena.

Para simular la pila, realice los siguientes subprogramas:

1. `initp`: Inicializa la pila simulada. Inicializa el puntero de pila a `bpila+4`, de manera que cuando se apile el primer elemento este quede apilado en el fondo de la pila (en la dirección de memoria `bpila`).

2. `apila`: Apila la palabra de memoria almacenada en `r0` y actualiza de manera adecuada el puntero de pila almacenado en `ppila`. En caso de que no sea posible porque la pila está llena, se indicará mediante alguno de los indicadores del registro de estado.

3. `desap`: Saca la palabra de la cima de la pila y la deja en `r0` y actualiza de manera adecuada el puntero de pila almacenado en `ppila`. En caso de que no sea posible porque la pila está vacía, se indicará mediante el indicador Z del registro de estado.

4. `sumap`: Suma las dos palabras de la cima de la pila y las sustituye por el resultado de la suma. Los indicadores del registro de estado se modifican en función del resultado. Si la pila está vacía o si solo hay un dato, la subrutina no hace nada.

5. `restap`: Resta las dos palabras de la cima de la pila y las sustituye por el resultado de la resta. La resta se realiza sustrayendo de la palabra que está en la cima de la pila la palabra que está apilada debajo de ella. Los indicadores del registro de estado se modifican en función del resultado. Si la pila está vacía o si solo hay un dato, la subrutina no hace nada.

6. `multp`: Multiplica los 16 bits menos significativos de las dos palabras de la cima de la pila y las sustituye por el resultado de la multiplicación. Los indicadores del registro de estado se modifican en función del resultado. Si la pila está vacía o si solo hay un dato, el subprograma no hace nada.

7. `swapp`: Intercambia las dos palabras que están en la cima de la pila. Si la pila está vacía o si solo hay un dato, no hace nada.

Para comprobar la máquina de pila simulada, escriba un programa que calcule el resultado de la operación:

$$0x20 + ((0x40 + 0x35) * 0x100) - 0x200$$

dejando el resultado (`0x0000 7320`) en el registro `r0`.

Solución

Las operaciones a realizar para obtener el resultado `0x0000 7320` son: `apila 0x20`, `apila 0x40`, `apila 0x35`, `suma`, `apila 0x100`, `multiplica`, `suma`, `apila 0x200`, `swap`, `resta`, `desapila`.

Como curiosidad, podemos ver que la secuencia de operaciones anteriores se corresponde con la operación original escrita en notación inversa polaca.

maq_pila1.s

```
        .data
pila:  .space 63*4  @ Cima de la pila: pila
bpila: .space 4     @ Base (fondo) de la pila: bpila
ppila: .word 1      @ Puntero de pila

/*
    Operación de ejemplo:
                0x20 + ((0x40 + 0x35) * 0x100) - 0x200 -> r0
*/

   .text
main:
   bl   initp
   mov  r0,#0x20
   bl   apila     @ apila 0x20
   mov  r0,#0x40
   bl   apila     @ apila 0x40
   mov  r0,#0x35
   bl   apila     @ apila 0x35

   bl   sumap     @ suma
   ldr  r0,=#0x100
   bl   apila     @ apila 0x100

   bl   multp     @ multiplica
   bl   sumap

   ldr  r0,=#0x200
```

```
29      bl    apila      @ apila 0x200
30
31      bl    swapp      @ swap
32      bl    restap     @ resta
33
34      bl    desap      @ desapila: resultado a r0
35
36      wfi
37
38
39   /*
40    initp: Subprograma para inicializar la pila
41   */
42   initp:
43      ldr    r0,=bpila    @ dirección del fondo de la pila
44      add    r0,r0,#4     @ una palabra más (bpila+4)
45      ldr    r1,=ppila
46      str    r0,[r1]      @ inicializamos el puntero de pila
47      mov    pc,lr        @ volvemos
48
49   /*
50   apila: Subprograma que almacena en la pila el contenido
51      de r0. En caso de que no sea posible porque la pila
52      está llena, se activa el indicador de cero (Z = 1)
53      Si todo OK, Z = 0.
54
55      El indicador Z para marcar pila llena resulta conveniente
56      porque es el indicador que usamos en el código para
57      completar o no la operación.
58
59   */
60   apila:
61      ldr    r1,=ppila
62      ldr    r2,=pila
63      ldr    r3,[r1]    @ r3: puntero de pila
64      cmp    r3,r2      @ si iguales, pila llena y Z = 1
65      beq    fina
66
67      sub    r3,r3,#4   @ apilamos (y Z = 0)
68      str    r0,[r3]
69      str    r3,[r1]    @ actualizamos puntero de pila
70
71   fina:
72      mov    pc,lr
73
74   /*
75   desap: Subprograma que saca la palabra de la cima de la pila y la
76      deja en r0. En caso de que no sea posible porque la pila
77      está vacía, se activa el indicador de signo (N=1)
```

```
78       Si todo OK, N=0.
79
80       Elegimos el indicador N para marcar pila vacía porque resulta
81       conveniente (es el indicador que usamos en el código para
82       completar o no la operación).
83    */
84    desap:
85       ldr    r1,=ppila
86       ldr    r2,=bpila
87       ldr    r3,[r1]      @ r3: puntero de pila
88       cmp    r2,r3        @ si r3 > r2, pila vacía y N = 1
89       bmi    find
90
91       ldr    r0,[r3]      @ desapilamos (y N = 0)
92       add    r3,r3,#4
93       str    r3,[r1]      @ actualizamos el puntero de pila
94
95    find:
96       mov    pc,lr
97
98    /*
99    sumap:   Suma las dos palabras en la cima de la pila y las
100      sustituye por el resultado de la suma. Los indicadores
101      se activan en función del resultado de la suma.
102
103      Si la pila está vacía o si solo hay un dato, no hace nada.
104   */
105   sumap:
106      ldr    r1,=ppila
107      ldr    r2,=bpila
108      ldr    r3,[r1]   @ r3: puntero de pila
109      cmp    r2,r3
110      bmi    fins      @ si r3 > r2, pila vacía
111      beq    fins      @ si r3 = r2, solo un dato
112
113      ldr    r0,[r3]   @ r0: suma de las dos palabras en la
114      add    r3,r3,#4  @ cima de la pila,
115      ldr    r2,[r3]   @ y eliminamos el dato de sobra
116      add    r0,r0,r2  @
117      str    r0,[r3]   @ guardamos en la pila
118
119      str    r3,[r1]   @ actualizamos el puntero de pila
120
121   fins:
122      mov    pc,lr
123
124
125   /*
126   restap:   Resta las dos palabras en la cima de la pila y
```

```
127    las sustituye por el resultado de la suma. Los indica-
128    dores se activan de acuerdo con el resultado de la resta
129
130    Si la pila está vacía o si solo hay un dato, no hace nada.
131 */
132 restap:
133    ldr   r1,=ppila
134    ldr   r2,=bpila
135    ldr   r3,[r1]    @ r3: puntero de pila
136    cmp   r2,r3
137    bmi   finr       @ si r3 > r2, pila vacía
138    beq   finr       @ si r3 = r2, solo un dato
139
140    ldr   r0,[r3]    @ r0: resta de las dos palabras en la
141    add   r3,r3,#4   @ cima de la pila,
142    ldr   r2,[r3]    @ y eliminamos el dato de sobra
143    sub   r0,r0,r2   @
144    str   r0,[r3]    @ guardamos en la pila
145    str   r3,[r1]    @ actualizamos el puntero de pila
146
147 finr:
148    mov    pc,lr
149
150 /*
151 multp:   Multiplica las dos palabras en la cima de la pila
152    y las sustituye por el resultado. Los indicadores
153    se activan de acuerdo con el resultado de la multiplicación.
154
155    Si la pila está vacía o si solo hay un dato, no hace nada.
156 */
157 multp:
158    ldr   r1,=ppila
159    ldr   r2,=bpila
160    ldr   r3,[r1]    @ r3: puntero de pila
161    cmp   r2,r3
162    bmi   finm       @ si r3 > r2, pila vacía
163    beq   finm       @ si r3 = r2, solo un dato
164
165    ldr   r0,[r3]    @ r0: producto de las dos palabras en la
166    add   r3,r3,#4   @ cima de la pila,
167    ldr   r2,[r3]    @ y eliminamos el dato de sobra
168    mul   r0,r0,r2   @
169    str   r0,[r3]    @ guardamos en la pila
170    str   r3,[r1]    @ actualizamos el puntero de pila
171
172 finm:
173    mov   pc,lr
174
175 /*
```

```
176 swapp:   Intercambia las dos palabras en la cima de la pila.
177
178    Si la pila está vacía o si solo hay un dato, no hace nada.
179 */
180 swapp:
181    ldr   r1,=ppila
182    ldr   r2,=bpila
183    ldr   r3,[r1]    @ r3: puntero de pila
184    cmp   r2,r3
185    bmi   finw       @ si r3 > r2, pila vacía
186    beq   finw       @ si r3 = r2, solo un dato
187
188    ldr   r0,[r3]    @ r0, r2: para el intercambio
189    ldr   r2,[r3,#4] @
190    str   r0,[r3,#4] @ Intercambiamos
191    str   r2,[r3]    @
192
193 finw:
194    mov   pc,lr
195    .end
```

Ejercicio 4.9

Al igual que el ejemplo anterior, el objetivo de este ejercicio es simular una máquina de pila en un ordenador ARM/Thumb, pero en esta ocasión utilizando la pila del sistema. La pila, que permite albergar palabras de 32 bits, no tiene definido un límite o capacidad máxima a priori y su localización viene especificada por el contenido del registro `sp` (Stack Pointer), que apunta siempre al último dato introducido en la pila. Igual que en el caso anterior, la pila crece hacia direcciones decrecientes de memoria.

Las instrucciones básicas para acceder a la pila son:

- `push {regs}`: Apila los registros indicados, en orden de número, de mayor a menor. En otras palabras, el registro de la lista con menor número quedará en la cima de la pila.

- `pop {regs}`: Saca de la pila tantos valores de 32 bits como registros aparezcan en la lista de registros y los deposita en dichos registros, en orden de número, de menor a mayor. Es decir, el valor que estaba en la cima de la pila se deposita en el registro con menor número.

Para simular la pila, realice los siguientes subprogramas:

1. `sumap`: Suma las dos palabras de la cima de la pila y las sustituye por el resultado de la suma. Los indicadores del registro de estado se modifican en función del resultado.

2. `restap`: Resta las dos palabras de la cima de la pila y las sustituye por el resultado de la resta. La resta se realiza sustrayendo de la palabra que está en la cima de la pila la palabra que está apilada debajo de ella. Los indicadores del registro de estado se modifican en función del resultado.

3. `multp`: Multiplica los 16 bits menos significativos de las dos palabras de la cima de la pila y las sustituye por el resultado de la multiplicación. Los indicadores del registro de estado se modifican en función del resultado.

4. `swapp`: Intercambia las dos palabras que están en la cima de la pila.

Para comprobar la máquina de pila simulada, escriba un programa que calcule el resultado de la operación:

$$0x20 + ((0x40 + 0x35) * 0x100) - 0x200$$

dejando el resultado (`0x0000 7320`) en el registro `r0`.

Solución

De ejercicio anterior, sabemos que las operaciones para realizar la operación $0x20+((0x40+0x35)*0x100)-0x200$, con resultado `0x0000 7320`, son: `apila 0x20`, `apila 0x40`, `apila 0x35`, `suma`, `apila 0x100`, `multiplica`, `suma`, `apila 0x200`, `swap`, `resta`, `desapila`.

Recuerda que `pop {r0, r1}` primero desapila `r0` y después `r1`, y da igual el orden en que los registros aparecen en la instrucción, ya que se codifican utilizando una máscara de registros. En otras palabras, `pop {r0, r1}` es lo mismo que `pop {r1, r0}` o `pop {r0-r1}`.

maq_pila2.s

```
/*
    Operación de ejemplo:
              0x20 + ((0x40 + 0x35) * 0x100) - 0x200 -> r0
*/
   .text
main:
   mov  r2,#0x20
   mov  r1,#0x40
   mov  r0,#0x35
   push {r0,r1,r2} @ apila, en orden, 0x20, 0x40, y finalmente 0x35

   bl   sumap      @ suma
   ldr  r0,=#0x100
   push {r0}       @  apila 0x100

   bl   multp      @ multiplica
```

```
18	bl   sumap
19
20	ldr  r0,=#0x200
21	push {r0}       @  apila 0x200
22
23	bl   swapp      @ swap
24	bl   restap     @ resta
25
26	pop  {r0}       @  desapila: resultado a r0
27
28	wfi
29
30
31 /*
32  sumap:   Suma las dos palabras en la cima de la pila
33  y las sustituye por el resultado de la suma. Los
34  indicadores se activan en función del resultado de la suma.
35
36 */
37 sumap:
38	pop   {r0, r1}   @ cargamos los datos
39	add   r0,r0,r1   @ hacemos la suma, modificando flags
40	push  {r0}       @ almacenamos el resultado (no modifica flags)
41	mov   pc,lr      @ volvemos
42
43
44 /*
45 restap:   Resta las dos palabras en la cima de la pila y
46	las sustituye por el resultado de la suma. Los
47	indicadores se activan de acuerdo con el resultado de la resta.
48
49 */
50 restap:
51	pop   {r0, r1}   @ cargamos los datos
52	sub   r0,r0,r1   @ hacemos la resta, modificando flags
53	push  {r0}       @ almacenamos el resultado (no modifica flags)
54	mov   pc,lr      @ volvemos
55
56 /*
57 multp:   Multiplica las dos medias palabras menos significativas
58	de las dos palabras en la cima de la pila, y las
59	sustituye por el resultado (una palabra). Los indicadores se activan
60	de acuerdo con el resultado de la multiplicación.
61
62 */
63 multp:
64	pop   {r0, r1}   @ cargamos los datos
65	mul   r0,r0,r1   @ hacemos la multiplicación, modificando flags
66	push  {r0}       @ almacenamos el resultado (no modifica flags)
```

```
67      mov   pc,lr       @ volvemos
68
69   /*
70   swapp:   Intercambia las dos palabras en la cima de la pila.
71
72      Si la pila está vacía o si solo hay un dato, no hace nada.
73   */
74   swapp:
75      ldr   r0,[sp,#0]    @ cargamos el primer dato
76      mov   r1,r0
77      ldr   r0,[sp,#4]    @ cargamos el segundo dato
78      str   r0,[sp,#0]    @ lo ponemos en el lugar del primero
79      str   r1,[sp,#4]
80      mov   pc,lr
81      .end
```

Ejercicio 4.10

Realizar un subprograma, etiquetado como `letras`, que dada una secuencia de 24 caracteres ASCII en la pila los sustituye por un número entero de 32 bits de valor 1 si todos los caracteres son letras y por el valor 0 en caso contrario (alguno de los caracteres no es una letra). La subrutina no modificará globalmente los registros `r4`-`r12`.

Pruebe su subprograma con el programa siguiente:

```
       .data
frase: .ascii "abcdfEFGHijkaMNdfrdsfcdf"

     .text
     ldr r4, =frase
     mov r5, #20
buc: ldr r6, [r4, r5]  @ carga 4 caracteres de la secuencia en r6
     push {r6}         @ y los introduce en la pila
     cmp r5,#0         @ comprobamos si ya hemos apilado todos
     beq subp
     sub r5,#4         @ si no, actualizamos el contador
     b buc             @ y repetimos con los 4 siguientes
subp:
     bl letras
     pop {r0}
     wfi
```

que devuelve `r0` = `1`.

Solución

letras.s

```
1         .data
```

```
frase: .ascii "abcdfEFGHijkaMNdfrdsfcdf"

     .text
main:
     ldr r4, =frase
     mov r5, #20
buc: ldr r6, [r4, r5] @ carga 4 caracteres de la secuencia en r6
     push {r6}        @ y los introduce en la pila
     cmp r5,#0        @ comprobamos si ya hemos apilado todos
     beq subp
     sub r5,#4        @ si no, actualizamos el contador
     b buc            @ y repetimos con los 4 siguientes
subp:
     bl letras
     pop {r0}
     wfi
/*
  letras: dados 24 caracteres en la pila, los sustituye por
  el valor 1 si todos los caracteres son letras, y por el valor
  0 en caso contrario (alguno de los caracteres no es una letra).
*/
letras:
     mov r0,sp       @ ro: puntero a los datos
     mov r1,#23      @ contador e índice
bucsr:
     cmp r1,#0       @ vemos si hemos terminado
     beq fins        @ por aquí, eran todos letras
     ldrb r2,[r0,r1] @ r2: carácter a comprobar
     cmp  r2,#'A'    @ vemos si no es una letra
     blt  noes
     cmp  r2,#'Z'
     blt  otro
     cmp  r2,#'a'
     blt  noes
     cmp  r2,#'z'
     blt  otro

noes:
     mov  r0,#0      @ por aquí, alguno no era letra
     b salir

otro:
     sub r1,r1,#1
     b   bucsr

fins: mov  r0,#1     @ todos eran letras
salir:
     add  sp,#20
     str  r0,[sp]
```

```
51        mov   pc,lr
52        .end
```

Ejercicio 4.11

Realizar un subprograma, etiquetado como `conmuta`, que dados 24 caracteres en la pila, los devuelva, también por pila, habiendo transformado las letras mayúsculas en minúsculas y la minúsculas en mayúsculas, dejando el resto de caracteres igual. La subrutina no modificará globalmente los registros `r4`-`r12`.

Puede usar el siguiente programa para comprobar el correcto funcionamiento de `conmuta`:

```
        .data
frase: .ascii "Piensa antes de hablar!!"

      .text
      ldr r4, =frase
      mov r5, #20
buc: ldr r6, [r4, r5]  @ carga 4 caracteres de la secuencia en r6
      push {r6}         @ y los introduce en la pila
      cmp r5,#0         @ comprobamos si ya hemos apilado todos
      beq subp
      sub r5,#4         @ si no, actualizamos el contador
      b buc             @ y repetimos con los 4 siguientes
subp:
      bl conmuta
      pop {r0-r5}
      wfi
```

Después de ejecutar el programa anterior, los valores de los registros quedarían como `r0` = `0x4e454970`, `r1` = `0x41204153`, `r2` = `0x5345544e`, `r3` = `0x2045420`, `r4` = `0x4c424148` y `r5` = `0x21215241`, es decir, la frase resultante es *pIENSA ANTES DE HABLAR!!*.

Solución

El subprograma `conmuta` analiza los 24 caracteres en la frase de entrada, uno a uno. Primero comprueba si se trata de una letra (es decir, si su código ASCII está entre 'A' y 'Z' o entre 'a' y 'z'). Si es así, conmuta dicho carácter de mayúscula a minúscula o viceversa. Para ello tiene en cuenta que la diferencia entre los códigos ASCII de las letras mayúsculas y minúsculas está en un único bit (cf. cuadro C.5 del apéndice C). Utilizando la máscara adecuada (`0x20`) y la operación lógica o exclusiva (**`eor`**) conmutamos únicamente el bit necesario.

conmuta.s

```
      .data
frase: .ascii "Piensa antes de hablar!!"

     .text
main:
     ldr r4, =frase
     mov r5, #20
buc: ldr r6, [r4, r5] @ carga 4 caracteres de la secuencia en r6
     push {r6}        @ y los introduce en la pila
     cmp r5,#0        @ comprobamos si ya hemos apilado todos
     beq subp
     sub r5,#4        @ si no, actualizamos el contador
     b buc            @ y repetimos con los 4 siguientes
subp:
     bl conmuta
     pop {r0-r5}
     wfi

/*
 conmuta: cambia mayúsculas por minúsculas,
  y viceversa. El resto de los caracteres quedan
  inalterados
*/
conmuta:
     mov r0,sp       @ r0: puntero a los datos en la pila
     mov r1,#24      @ r1: contador de caracteres
     mov r2,#0x20    @ r2: máscara para convertir (conmutar bit 6)

bucs:
     ldrb r3,[r0]    @ r3: carácter de la cadena
     cmp r3,#'A'     @ comprobamos si es una letra
     blt nocam       @ y si es así la cambiamos
     cmp r3,#'Z'
     blt cam
     cmp r3,#'a'
     blt nocam
     cmp r3,#'z'
     blt cam

/* nocam: código si no hay que modificar el carácter en r3 */

nocam:
     sub r1,r1,#1   @ decrementamos el contador
     beq fins       @ si es cero, hemos terminado
     add r0,r0,#1   @ e incrementamos el puntero
     b bucs

/* cam: código si hay que modificar el carácter en r3 */
```

```
50
51  cam: eor r3,r3,r2    @ o exclusiva: 1 eor 1 = 0, y 1 eor 0 = 1
52       strb r3,[r0]    @ almacenamos el carácter cambiado
53       b nocam         @ actualizamos puntero y contador
54
55  fins: mov pc,lr      @ volvemos del subprograma.
56       .end
```

Ejercicio 4.12

Realizar un subprograma, etiquetado como `mul16`, que dados 12 números de 16 bits en la pila, los devuelva, también por pila, habiendo sustituido aquellos que son múltiplo de 16 por un 0. La subrutina no modificará globalmente los registros `r4`-`r12`.

Utiliza el programa siguiente para probar tu solución:

```
      .data
nums: .hword 1, 2, -3, 16, -5, 6, 7, -32, 9, 10, 11, 64

      .text
      ldr r4,=nums     @ r4: dirección de comienzo de la zona de números
      sub sp,#24       @ reservamos espacio en la pila para los números
      mov r5,sp        @ r5: dirección de la cima dela pila
      mov r6,#22       @ r6: contador (inicializado a (12 - 1) * 2 bytes)
buc:  ldrh r7,[r4,r6]  @ r7: cargamos un número de la zona
      strh r7,[r5,r6]  @ y lo dejamos en su sitio en la pila
      cmp  r6,#0       @ vemos si hemos terminado
      beq  csp
      sub  r6,#2       @ si no, actualizamos el contador
      b buc            @ y repetimos con la siguiente media palabra
csp:  bl mul16
      pop {r0-r5}
      wfi
```

Después de ejecutar el programa anterior, los valores de los registros quedarían como `r0` = `0x00020001`, `r1` = `0x0000FFFD`, `r2` = `0x0006FFFB`, `r3` = `0x00000007`, `r4` = `0x000a0009` y `r5` = `0x0000000b`, es decir, los números resultantes serán 1, 2, -3, 0, -5, 6, 7, 0, 9, 10, 11 y 0.

Solución

Para realizar el subprograma `mul16` recorremos la tabla de números comprobando si son múltiplos de 16. Para ello enmascaramos los 4 bits menos significativos de cada número y comprobamos si son 0.

Este procedimiento funciona también para los números negativos, ya que el complemento a 2 de un número positivo múltiplo de 16 tam-

bién tiene sus 4 bits menos significativos iguales a 0. Por ejemplo, el complemento a 2 de `0x12345670` es `0xEDCBA990`.

mul16.s

```
      .data
nums: .hword 1, 2, -3, 16, -5, 6, 7, -32, 9, 10, 11, 64

      .text
      ldr r4,=nums     @ r4: dirección de comienzo de la zona de números
      sub sp,#24       @ reservamos espacio en la pila para los números
      mov r5,sp        @ r5: dirección de la cima dela pila
      mov r6,#22       @ r6: contador (inicializado a (12 - 1) * 2 bytes)
buc:  ldrh r7,[r4,r6]  @ r7: cargamos un número de la zona
      strh r7,[r5,r6]  @ y lo dejamos en su sitio en la pila
      cmp  r6,#0       @ vemos si hemos terminado
      beq  csp
      sub  r6,#2       @ si no, actualizamos el contador
      b buc            @ y repetimos con la siguiente media palabra
csp:  bl mul16
      pop {r0-r5}
      wfi

/*
 mul16: a partir de una lista de 12 números de 16 bits
   en la pila, sustituye por un cero los números que
   son múltiplo de 16.

*/
mul16:
      mov r0,sp           @ r0: puntero a los números
      ldr r1,=12          @ r1: contador
      mov r2,#0x0000000f  @ r2: máscara para comprobar

bucs: ldrh r3,[r0]        @ r3: número a comprobar (16 bits)
      and r3,r3,r2        @ dejamos los 4 bits menos significativos
      bne nomul           @ si son 0, era múltiplo de 16
      strh r3,[r0]        @ reemplazamos por un cero

nomul:
      sub r1,r1,#1        @ actualizamos puntero y contador
      beq fins
      add r0,r0,#2
      b bucs

fins: mov pc,lr
      .end
```

Ejercicio 4.13

Realizar un subprograma, etiquetado como `altdec`, que dados 8 caracteres ASCII numéricos almacenados en la pila que representan a un número decimal, los sustituye por un número entero de 32 bits de valor 1 si sus dígitos alternan sus valores (p. ej., `76767676`) y por el valor 0 en caso contrario. En caso de que los 8 caracteres ASCII sean iguales, el subprograma devolverá el valor 1. La subrutina no modificará globalmente los registros `r4`-`r12`.

Suponga que los caracteres almacenados en la pila son efectivamente caracteres numéricos y pruebe su subprograma con el programa siguiente:

```
        .data
numero: .ascii "76767676"

      .text
      ldr r4,=numero
      ldr r5,[r4,#0]
      ldr r6,[r4,#4]
      push {r5-r6}
      bl altdec
      pop {r0}
      wfi
```

que devuelve `r0` = `1`.

Solución

El problema de comprobar si los caracteres alternan sus valores es equivalente al problema de comprobar si los caracteres de la cadena tomados dos a dos se repiten. Por ejemplo, en el caso de `76767676`, los caracteres alternan porque se repite el patrón `76`. Aprovechamos esta propiedad para realizar el ejercicio.

altdec.s

```
1          .data
2  numero: .ascii "76767676"
3  
4        .text
5  main: ldr r4,=numero
6        ldr r5,[r4,#0]
7        ldr r6,[r4,#4]
8        push {r5-r6}
9        bl altdec
10       pop {r0}
11       wfi
12 
13 /*
14  altdec: dados 8 caracteres ASCII numéricos almacenados
```

```
15    en la pila, los sustituye por un 1 si sus dígitos
16    alternan sus valores, y por un 0 en caso contrario.
17 */
18 altdec:
19         mov  r0,sp       @ r0: puntero a los dígitos
20         mov  r1,#6       @ r1: contador / puntero
21
22         ldrh r2,[r0,r1] @ r2: primer par de dígitos
23
24 bucsr:
25         sub  r1,r1,#2    @ r3: par de dígitos siguiente
26         ldrh r3,[r0,r1]
27         cmp  r2,r3       @ vemos si son iguales
28         bne  noes        @ si no lo son, no hay alternancia
29
30         cmp  r1,#0       @ comprobamos si hemos acabado
31         bne  bucsr
32
33         mov  r0,#1       @ dígitos alternantes
34
35 fins:  add  sp,#4       @ ajustamos la pila
36         str  r0,[sp]     @ y devolvemos un 1
37         mov  pc,lr
38
39 noes:  mov  r0,#0       @ no lo eran. Devolvemos
40         b    fins        @ un cero.
41         .end
```

Ejercicio 4.14

Realizar un subprograma, etiquetado como `lowdigit`, que dados 8 caracteres ASCII numéricos almacenados en la pila que representan a un número decimal, devuelve en la pila una palabra de 32 bits, que sustituye a los caracteres originales, cuyo valor es el valor numérico (un entero entre 0 y 9) del dígito de menos valor del número original. La subrutina no modificará globalmente los registros `r4`-`r12`.

Suponga que los caracteres almacenados en la pila son efectivamente caracteres numéricos y pruebe su subprograma con el programa siguiente:

```
        .data
numero: .ascii "23450879"

      .text
      ldr r4,=numero
      ldr r5,[r4,#0]
      ldr r6,[r4,#4]
      push {r5-r6}
```

```
bl lowdigit
pop {r0}
wfi
```

que deja `r0 = 0`.

Solución

lowdigit.s

```
        .data
numero: .ascii "23450879"

      .text
main: ldr r4,=numero
      ldr r5,[r4,#0]
      ldr r6,[r4,#4]
      push {r5-r6}
      bl lowdigit
      pop {r0}
      wfi

/*
  lowdigit: dados 8 caracteres ASCII numéricos almacenados
    en la pila, devuelve en la pila una palabra de 32 bits,
    que sustituye a los caracteres originales, cuyo valor es
    el valor numérico (un entero entre 0 y 9) del dígito
    de menos valor del número original.
*/
lowdigit:
      mov r0,sp       @ r0: puntero a los dígitos
      mov r1,#7       @ r1: contador / índice
      ldrb r2,[r0,r1] @ r2: dígito a comprobar

bucsr:
      sub r1,r1,#1    @ buscamos el menor
      ldrb r3,[r0,r1] @ y lo mantenemos en r2
      cmp r2,r3
      ble  nocambie
      mov r2,r3

nocambie:
      cmp r1,#0       @ vemos si hemos terminado
      bne bucsr

      sub r2,r2,#'0'  @ calculamos el valor del
      add sp,#4       @ menor dígito (cod. ASCII - cod. '0')
      str r2,[sp]     @ ajustamos la pila, y almacenamos
      mov pc,lr
      .end
```

Ejercicio 4.15

Realizar un subprograma, etiquetado como `sumamat4`, que dados 16 números de 16 bits en la pila organizados como una matriz 4x4 serializada por filas, los sustituye por 4 números de 32 bits cada uno de ellos representando la suma de cada una de las 4 filas. En la cima de la pila quedará la suma de los elementos de la primera fila, debajo de ella la suma de los elementos de la segunda fila y así sucesivamente. La subrutina no modificará globalmente los registros `r4`-`r12`.

Prueba el subprograma utilizando el programa siguiente:

```
        .data
matriz: .hword 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 ,13, 14 ,15 ,16

      .text
      ldr r4,=matriz
      sub sp,#32
      mov r5,sp
      mov r6,#30
buc:  ldrh r7,[r4,r6]
      strh r7,[r5,r6]
      cmp  r6,#0
      beq  csp
      sub  r6,#2
      b buc
csp:  bl sumamat4
      pop {r0-r3}
      wfi
```

que deja `r0` = `10`, `r1` = `26`, `r2` = `42` y `r3` = `58`.

Solución

sumamat4.s

```
1          .data
2  matriz: .hword 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 ,13, 14 ,15 ,16
3
4        .text
5  main: ldr r4,=matriz
6        sub sp,#32
7        mov r5,sp
8        mov r6,#30
9  buc:  ldrh r7,[r4,r6]
10       strh r7,[r5,r6]
11       cmp  r6,#0
12       beq  csp
13       sub  r6,#2
14       b buc
```

```
15  csp:  bl sumamat4
16        pop {r0-r3}
17        wfi
18
19  /*
20    sumamat4: dados 16 números de 16 bits en la pila
21      organizados como una matriz 4x4 serializada por filas,
22      los sustituye por 4 números de 32 bits cada uno de
23      ellos representando la suma de cada una de las 4 filas.
24  */
25  sumamat4:
26        push {r4}       @ vamos a necesitar usar r4
27        mov r0,sp       @ r0: puntero a los sumandos
28        add r0,r0,#28   @ empezamos por el final
29        mov r1,sp       @ r1: puntero para los resultados
30        add r1,r1,#32   @ aquí quedará la suma de la última fila
31        mov r2,#4       @ r2: contador de iteraciones
32                        @ (4 filas)
33  bucs: ldrh r3,[r0]    @ r3: acumulador de sumas de una file
34        ldrh r4,[r0,#2] @ primer y segundo elemento
35        add r3,r3,r4    @ de la fila
36        ldrh r4,[r0,#4] @ tercer elemento
37        add r3,r3,r4
38        ldrh r4,[r0,#6] @ cuarto elemento
39        add r3,r3,r4
40        str r3,[r1]     @ almacenamos el resultado
41        sub r2,r2,#1    @ actualizamos contador, y comprobamos
42        beq fin         @ si final
43
44        sub r0,r0,#8    @ movemos r0 a la fila anterior
45        sub r1,r1,#4    @ y r1 al destino de la siguiente suma
46        b bucs
47
48  fin:  pop {r4}        @ recuperamos r4
49        add sp,sp,#16   @ eliminamos lo que sobra de la pila
50        mov pc,lr       @ y regresamos
51        .end
```

4.3 Más sobre subprogramas

Los ejercicios siguientes retoman el concepto de subprograma, esta vez con paso de parámetros y salvaguarda del contexto utilizando la pila, tal como se introdujo en el apartado 4.2. También estudiaremos las llamadas anidadas a subprogramas y la recursividad.

Al igual que en el apartado 4.1, seguimos el convenio de llamadas a subprogramas del *ARM Architecture Procedure Call Standard* (AAPCS). Así, si un

subprograma necesita más de cuatro argumentos, se prefiere el paso de los cuatro primeros en los registros `r0` a `r3` y el resto en la pila. En el caso de modificarse alguno de los registros `r4` a `r7`, el subprograma debe preservar su valor para restaurarlo antes de devolver el control al programa invocante. Además, en el caso de producirse llamadas anidadas a subprogramas, será necesario preservar también el valor del registro `lr`, utilizando para ello la pareja de instrucciones `push {regs,lr}` y `pop {regs,pc}` al principio y al final de un suprograma respectivamente. Las llamadas recursivas son un caso particular de esta situación.

Proponemos una selección de ejercicios de una complejidad mayor que los presentados en el capítulo 4.1, que además hacen uso de las nuevas características citadas.

Ejercicio 4.16

Realizar un subprograma, etiquetado como `hswap`, que intercambie dos medias palabras de memoria. Los parámetros de entrada serán las direcciones de las medias palabras en los registros `r0` y `r1` respectivamente. La subrutina dejará inalterados `r0` y `r1`, y no modificará globalmente ningún registro `r4` – `r12`.

Puede usar el siguiente programa para comprobar el correcto funcionamiento de `hswap`:

```
      .data
zona: .hword 0x11FF
      .hword 0x1111
      .hword 0xAA22
      .hword 0x2222

      .text
main: ldr r0, =zona
      add r1, r0, #2
      bl hswap
      add r0, r1, #2
      add r1, r0, #2
      bl hswap
      wfi

hswap: @ el código del subprograma iría aquí

      .end
```

Después de ejecutar el programa anterior, la zona de memoria debería quedar así:

```
       .data
zona:  .hword 0x1111
       .hword 0x11FF
       .hword 0x2222
       .hword 0xAA22
```

Ahora, haciendo uso de la anterior subrutina `hswap`, programe otra subrutina, etiquetada como `ordena`, que dados 12 números enteros de 16 bits en la pila (parámetros de entrada) devuelva esos mismos enteros en la pila, pero ordenados de menor a mayor. Use el algoritmo de la burbuja que describimos a continuación, con `TAM = 12` en este caso.

```
for (i=1; i < TAM; i++)
     for j=0 ; j < TAM - 1; j++)
         if (lista[j] > lista[j+1])
                    hswap(r0 = &lista[j], r1 = &lista[j+1]);
```

La subrutina no modificará ningún registro `r4` – `r12`. Use el siguiente programa para comprobar el correcto funcionamiento de `ordena`:

```
       .data
zona:  .hword 1234, 26, -453, 83, -7000, 0
       .hword -15000, 1, 777, 2020, 987, 10

       .text
main:  ldr r4, =zona
       mov r5, #20
loop:
       ldr r6, [r4, r5]
       push {r6}
       cmp r5, #0
       beq next
       sub r5, #4
       b loop
next:  bl ordena
       ldr r4, =zone
       mov r5, #0
loopf:pop {r6}
       str r6, [r4, r5]
       cmp r5, #20
       beq endf
       add r5, #4
       b loopf
endf:  wfi

ordena: @ aquí iría el código del subprograma
       .end
```

Después de ejecutar el programa anterior, la zona de memoria debería quedar ordenada

```
      .data
zona: .hword -15000, -7000, -453, 0, 1, 10
      .hword 26, 83, 777, 987, 1234, 2020
```

Solución

El subprograma `hswap` intercambia las medias palabras cuyas direcciones están almacenadas en los registros `r0` y `r1` utilizando como almacenamiento intermedio los registros `r2` y `r3`.

Para programar la subrutina `ordena`, básicamente traducimos a ensamblador el pseudocódigo del algoritmo de la burbuja indicado en el enunciado. Utilizamos el subprograma `hswap` para realizar los intercambios.

También será necesario extender los enteros de 16 bits a palabras, cargándolos con la instrucción **`ldsh`**, para poder realizar comparaciones enteras.

burbuja.s

```
1
2          .data
3  zona:   .hword 1234, 26, -453, 83, -7000, 0
4          .hword -15000, 1, 777, 2020, 987, 10
5
6          .text
7  /*
8    Programa de prueba del enunciado
9  */
10 main:   ldr r4, =zona
11         mov r5, #20
12 loop:
13         ldr r6, [r4, r5]
14         push {r6}
15         cmp r5, #0
16         beq next
17         sub r5, #4
18         b loop
19 next:   bl ordena
20         ldr r4, =zona
21         mov r5, #0
22 loopf:  pop {r6}
23         str r6, [r4, r5]
24         cmp r5, #20
25         beq endf
26         add r5, #4
```

```
27          b loopf
28  endf:   wfi
29
30  /*
31    hswap: subprograma de la primera parte.
32
33    Recibe las direcciones de las dos medias palabras
34    a intercambiar en r0 y r1.
35
36  */
37
38  hswap:  ldrh r2, [r0]
39          ldrh r3, [r1]
40          strh r3, [r0]
41          strh r2, [r1]
42          mov pc, lr
43
44  /*
45   ordena: subprograma de la segunda parte.
46
47   Dadas 12 medias palabras en la pila
48   devuelva esas mismas medias palabras en la pila ordenadas,
49   utilizando el algoritmo de la burbuja.
50  */
51  ordena: push {r4-r7,lr}    @ salvaguardamos los registros r4-r7
52          mov r7, sp
53          add r7, #20        @ r7: puntero al principio de la lista
54          mov r4, #1         @ r4: índice i = 1
55
56  for1:   cmp r4, #12        @ ¿i = 12?
57          beq fin1
58          mov r5, #0         @ r5: índice j = 0
59          mov r6, #0         @ r6: desplazamiento en la tabla de lista[j]
60
61  for2:   cmp r5, #11        @ ¿j = 11?
62          beq fin2
63
64          ldsh r0,[r7,r6]    @ r0:lista[j] (extensión de signo)
65          add r5, #1         @ j = j + 1
66          add r6, #2
67          ldsh r1, [r7,r6]   @ r1=lista[j+1] (extensión de signo)
68
69          cmp r0, r1
70          ble for2           @ si lista[j] <= lista[j + 1], iteramos
71
72          push {r0,r1}       @ lista[j] > lista[j + 1: intercambiar
73          add r1, r7,r6      @ r1: dirección de lista[j+1]
74          sub r0, r1,#2      @ r0: dirección de lista[j]
75          bl hswap           @ llamada al subprograma del primer
```

```
76          pop {r0,r1}         @ ejercicio
77          b for2              @ y volvemos a iterar
78
79  fin2:   add r4, #1          @ i = i + 1
80          b for1
81
82  fin1:   pop {r4-r7,pc}
83         .end
```

Ejercicio 4.17

Realizar un programa que cuente la cantidad de apariciones de cada uno de los dígitos de '0' a '9' que hay almacenados en una zona de memoria, cuyas direcciones de inicio y final están almacenadas en las posiciones de memoria etiquetadas respectivamente como `i_zona` y `f_zona`.

El número total de apariciones de cada dígito se almacenará en una tabla de 10 elementos que comienza en la dirección etiquetada como `cuenta`, de manera que el número de apariciones del dígito '0' se almacenaría en `cuenta` y el número de apariciones del dígito '9' en la dirección `(cuenta + 9)`.

Para realizar el programa, utilice un subprograma, etiquetado como `cuentad`, que cuente el número de apariciones de determinado carácter en una zona de memoria. Los parámetros de entrada de dicho subprograma son los siguientes:

- `r0`: dirección de comienzo de la zona de memoria.
- `r1`: dirección de comienzo del último elemento en la zona de memoria.
- `r2`: carácter a buscar.

El subprograma devolverá en `r0` el número de apariciones del carácter indicado dentro de la zona.

Utilice las definiciones siguientes:

```
        .data
zona:  .byte   '2', '6', '7', '3', '4', '2', '8', '5', '2', '3', '3', '2'
d_ult: .space 0
       .balign  4
i_zona: .word   zona    @ comienzo de la zona
f_zona: .word   d_ult   @ fin de la zona
cuenta: .space  10      @ tabla con el número de apariciones
```

Solución

El programa principal llamará 10 veces al subprograma `cuentad` sobre la misma zona, es decir, con los mismos parámetros de entrada en `r0` y `r1`, pero con valores consecutivos en el registro `r2`, comenzando por `r2` = `0x30` (código ASCII del carácter '0') y terminando con `r2` = `0x39` (código ASCII del carácter '9'). Después de cada invocación de `cuentad`, el programa principal almacenará el resultado en `r0` en la posición correspondiente de la tabla de resultados almacenada a partir de la dirección `cuenta`.

El subprograma `cuentad` analiza las posiciones de memoria de la zona de datos una a una comparando su contenido con el carácter modelo en `r0`, incrementando un contador de apariciones en caso de que coincida:

```
cont = 0;
for (i = *i_zona; i >= *f_zona; i++) {
      if (*i == r0) cont = cont + 1;
}
r0 = cont;
```

cuentadig.s

```
1          .data
2  zona:   .byte   '2', '6', '7', '3', '4', '2', '8', '5', '2', '3', '3'
3  d_ult:  .byte   '2'
4          .balign  4
5  i_zona: .word   zona     @ comienzo de la zona
6  f_zona: .word   d_ult    @ fin de la zona
7  cuenta: .space  10       @ tabla con el número de apariciones.
8
9
10     .text
11 main:
12    ldr    r4,=cuenta    @ tabla de resultados
13    mov    r5, #0        @ índice de dígitos
14    ldr    r0,=i_zona
15    ldr    r0,[r0]       @ puntero al principio de la zona
16    ldr    r1,=f_zona
17    ldr    r1,[r1]       @ puntero al final de la zona
18    push   {r0, r1}      @ salvaguardamos los punteros
19 otro:
20    mov    r2,#'0'
21    add    r2,r5         @ dígito a contar
22    bl     cuentad
23    strb   r0,[r4,r5]    @ almacenamos el resultado
24    cmp    r5,#9
25    beq    final
26    add    r5,r5,#1      @ incrementamos el índice
```

```
27      ldr   r0,[sp]       @ recuperamos los punteros al
28      ldr   r1,[sp,#4]    @ principio y final de la zona
29      b     otro
30
31  final:
32      add   sp,sp,#8      @ balanceamos la pila
33      wfi
34
35  /*
36  cuentad: subprograma para contar cuántos caracteres hay en una
37    zona de memoria.
38    Parámetros:
39      - r0: dirección de comienzo de la zona de memoria.
40      - r1: dirección de comienzo del último elemento en la zona de memoria.
41      - r2: carácter a buscar.
42    Resultado:
43      - r0: número de ocurrencias del carácter pasado como parámetro.
44  */
45  cuentad:
46      push  {r4}        @ salvaguardamos r4
47      mov   r4,#0       @ r4: contador de apariciones
48  buc:
49      cmp   r0,r1       @ ¿final de la tabla?
50      bgt   finsr
51
52      ldrb  r3,[r0]     @ cargamos un carácter
53      cmp   r3,r2       @ vemos si es el carácter buscado
54      bne   noes
55
56      add   r4,r4,#1    @ si es, incrementamos el contador
57  noes:
58      add   r0,r0,#1    @ incrementamos el puntero
59      b     buc
60
61  finsr:
62      mov   r0,r4       @ pasamos el contador al registro de resultado
63      pop   {r4}        @ recuperamos r4
64      mov   pc,lr       @ retornamos
65      .end
```

Ejercicio 4.18

Realizar un subprograma, etiquetado como `setbit`, que active o desactive determinado bit en una matriz $M \times N$ de bits almacenada en memoria de forma serializada por filas. El subprograma recibirá los siguientes argumentos:

- En `r0`, el número M de filas en la matriz.

- En `r1`, el número N de columnas en la matriz.
- En `r2`, la fila i donde se encuentra el bit a modificar.
- En `r3`, la columna j donde se encuentra el bit a modificar.
- En la pila, la dirección a partir de la cual está almacenada la matriz.
- También en la pila, apilado encima del valor anterior, una palabra que indica si hay que activar (valor 1) o desactivar (valor 0) el bit indicado por los registros (`r3`, `r4`).

Además de modificar el bit indicado, el subprograma devolverá como resultado los siguientes valores:

- En `r0`, el valor 1 si el bit se modificó correctamente y 0 si hubo un error.
- En `r1`, el código de error si hubo error (si `r0` = 0) y 0 en caso contrario (si `r0` = 1)

La pila no será modificada por el subprograma.

Para probar el subprograma, utilice la matriz 4×16 del ejercicio 3.24 para activar el bit indicado en dicho ejercicio:

```
       .data
array: .byte 0b10000000, 0b00000000 @ fila 0
       .byte 0b01000000, 0b00000000 @ fila 1
       .byte 0b00100000, 0b00000000}@ fila 2
       .byte 0b00001000, 0b11111111 @ fila 3
@             ||||||||    ||||||||
@ columnas:    01234567    89abcdef
@
       .balign 4
fil:   .word 2      @ fila entre 0 y 3
col:   .word 12     @ columna entre 0 y 15
```

Solución

El programa principal simplemente pasará al subprograma `setbit` los parámetros necesarios para activar el bit indicado en la matriz del ejemplo:

- En `r0`, el número de filas (4).
- En `r1`, el número de columnas (16).

- En `r2`, la fila del bit a activar (2).
- En `r3`, la columna del bit a activar (12).
- En la pila, la dirección `array`.
- También en la pila, una palabra con valor 1.

El subprograma `setbit` comprobará primero si hay errores en los parámetros de entrada, básicamente, si las coordenadas del bit a modificar referencian una posición dentro de la matriz.

A continuación, calcularemos el byte y el offset en ese byte donde está el bit a activar. Primero calculamos el índice absoluto del bit en la matriz serializada ($Indx$):

$$Indx = i \times N + j,\ 0 \leq Indx \leq (N \times M) - 1$$

Luego obtenemos el byte (en `r0`) y el desplazamiento dentro de dicho byte (en `r1`) a partir del índice $Indx$, como el cociente y el resto de dividir $Indx$ por 8.

Finalmente nos queda activar o desactivar mediante la máscara adecuada el bit indicado por `r1` dentro del byte de dirección (`array` + `[r0]`).

matriz_gen.s

```
@
@ Ejemplo extraído del ejercicio indicado
@

       .data
array: .byte 0b10000000,0b00000000 @ fila 0
       .byte 0b01000000,0b00000000 @ fila 1
       .byte 0b00100000,0b00000000 @ fila 2
       .byte 0b00001000,0b11111101 @ fila 3
@               ||||||||   ||||||||
@ columnas:     01234567   89abcdef
@
      .balign 4
row:  .word 2       @ fila de ejemplo
col:  .word 12      @ columna de ejemplo

      .text

@
@ definimos códigos de error
@
.equ  nif,1         @ error de numero de filas
.equ  nic,2         @ error de numero de columnas
```

```
24  .equ  ffr,3        @ error de fila fuera de rango
25  .equ  cfr,4        @ error de columna fuera de rango
26
27  @
28  @ Programa principal
29  @
30  @ Parámetros: activar (1) el bit de coordenadas (2, 12)
31  @ de una matriz 4 x 16:
32  @
33  @     - r0 = 4
34  @     - r1 = 16
35  @     - r2 = 2   [row]
36  @     - r3 = 12  [col]
37  @
38  main: mov r0,#4     @ 4 filas
39        mov r1,#16    @ 16 columnas
40        ldr r2,=row   @ fila a activar
41        ldr r2,[r2]
42        ldr r3,=col   @ columna a activar
43        ldr r3,[r3]
44        ldr r5,=array @ dirección de la matriz
45        mov r4,#1     @ activar
46
47        push {r4,r5}  @ estos parámetros se pasan en la pila
48        bl  setbit
49        add  sp,#8    @ balanceamos la pila
50
51        wfi
52
53  /*
54   setbit: subrutina para activar o desactivar
55     el bit en la posición (i,j) en una matriz de
56     tamaño (m x n) serializada por filas y almacenada
57     a partir de la dirección dirmat.
58
59     Parámetros:
60      r0: m (número de filas de la matriz)
61      r1: n (número de columnas de la matriz)
62      r2: i (fila del bit a activar, de 0 a m-1)
63      r3: j (columna del bit a activar, de 0 a n-1)
64      En la pila (dos palabras, apiladas en el orden indicado:
65         - dirmat (dirección de la matiz).
66         - 1 para indicar activación y 0 para indicar desactivación.
67
68     Resultado: r0: 1 si el bit se activó, y 0 si hubo error
69                r1: código de error si hubo error, 0 en caso contrario
70   */
71  setbit:
72  @
```

```
73  @ Comprobamos errores en los parámetros de entrada
74  @
75        cmp r0,#0
76        ble e_nif         @ error de numero de filas (M <= 0)
77        cmp r1,#0
78        ble e_nic         @ error de número de columnas (N <= 0)
79        cmp r2,r0
80        bge e_ffr         @ error de fila fuera de rango (i >= M)
81        cmp r3,r1
82        bge e_cfr         @ error de columna fuera de rango (j >= N)
83        cmp r2,#0
84        blt e_ffr         @ error de fila fuera de rango (f < 0)
85        cmp r3,#0
86        blt e_cfr         @ error de columna fuera de rango (c < 0)
87  @
88  @ Calculamos el byte y el offset en ese byte
89  @ donde está el bit a activar. Primero calculamos el índice
90  @ absoluto del bit en la matriz serializada y lo dejamos en r2:
91  @
92  @ index = i * N + j (valor entre 0 y (N * M) - 1)
93  @
94        mul r2,r2,r1      @ r2: i * N
95        add r2,r2,r3      @ r2: index
96  @
97  @ ahora calculamos byte y offset a partir de index (a r0 y r1)
98  @
99  @ r0: byte = index div 8; r1: offset: index mod 8
100
101       mov r1,#0x07      @ máscara para calcular el resto
102       and r1,r1,r2      @ resto a r1
103       lsr r0,r2,#3      @ cociente a r0
104 @
105 @ activamos o desactivamos el bit de destino (bit r1 del byte
106 @ en [dirmat + r0]). Utilizamos como máscara de partida
107 @ 0x80, y desplazamos a la derecha el offset almacenado en r1
108 @
109       ldr r2,=#0x80     @ máscara de partida para cambiar
110       lsr r2,r1         @ máscara final (con el bit en offset)
111       ldr r3,[sp,#4]    @ dirección de la matriz
112       add r3,r3,r0      @ r3: dir del byte a modificar
113       ldrb r1,[r3]      @ r1: byte a modificar
114       ldr r0,[sp]       @ lsb de r0: activar o desactivar
115
116       cmp r0,#0         @ ver si activar o desactivar
117       beq set0
118
119 set1: orr r1,r2         @ activa el bit objetivo
120       b sigue
121
```

```
122  set0: mvn r2,r2
123        and r1,r2          @ desactiva bir objetivo
124
125  sigue:
126        strb r1,[r3]       @ sustituye el byte
127        mov r0,#1          @ original con el bit objetivo
128        b back             @ y salimos sin errores
129  @
130  @ resultados con errores
131  @
132  e_nif:
133        mov r1,#nif        @ número de filas incorrecto
134        b   backe
135  e_nic:
136        mov r1,#nic        @ número de columnas incorrecto
137        b   backe
138  e_ffr:
139        mov r1,#ffr        @ fila fuera de rango
140        b   backe
141  e_cfr:
142        mov r1,#cfr        @ fila fuera de rango
143  backe:
144        mov r0,#0
145  back: mov pc,lr          @ volvemos
146        .end
```

Ejercicio 4.19

Utilizando el subprograma del ejercicio 4.18, realizar un subprograma que active o desactive una X en una matriz de 8 x 8 leds monocolor. Las aspas de la X se corresponden con las dos diagonales de la matriz.

La matriz de leds está representada en memoria mediante una matriz de bits serializada por filas. Cada bit de la matriz en memoria representa un led de la matriz de leds.

El subprograma recibirá como parámetros:

- En `r0`, la dirección de comienzo de la matriz de leds.
- En `r1`, el valor 1 para activar la X y el valor 0 para desactivarla.

Para probar el subprograma, realice un programa que primero active y luego desactive una X en una matriz de leds almacenada a partir de la dirección `m_leds`.

```
        .data
m_leds: .space 8 @ memoria utilizada por la matriz de leds (8 bytes)
```

Solución

El programa principal simplemente llama dos veces al suprograma de activación (`act_lx`), la primera con `r1 = 1` y la segunda con `r1 = 0`.

El subprograma `act_lx` hace uso del subprograma del ejercicio 4.18 para activar o desactivar los elementos (i, j) de la matriz que están en las diagonales principal y secundaria:

```
for (r5 = 0, r4 = 7; r4 >= 0; r5++, r4--) {
      setbit(8, 8, r5, r5, r1);
      setbit(8, 8, r4, r5, r1);
}
```

matriz_led.s

```
1
2         .data
3  m_leds: .space 8       @ memoria utilizada por la matriz
4
5         .text
6
7  @
8  @ Programa principal
9  @
10 .equ  activar,1
11 .equ  desactivar,0
12
13 main: ldr r0,=m_leds   @ activamos la X
14       mov r1,#activar
15       bl  act_lx
16
17       ldr r0,=m_leds   @ desactivamos la X
18       mov r1,#desactivar
19       bl  act_lx
20
21       wfi
22
23 /*
24  act_lx: subrutina que activa una X (las dos diagonales) d
25  e una matriz de bits 8 x 8.
26
27 Parámetros:
28     r0: dirección de comienzo de la matriz
29     r1: 1 para indicar activar y 0 para desactivar
30 */
31 act_lx:
32     push {r4,r5,lr} @ preservamos la dir. de retorno y
33                     @ registros por encima de r3
34     push {r0}       @ apilamos dir. matriz y acción
35     push {r1}       @ común para todas las llamadas a
```

```
36                              @ setbit
37      mov   r4,#7             @ contador de iteraciones y offsets
38      mov   r5,#0             @ de las diagonales
39  otro:
40      mov   r0,#8             @ activamos el bit i-ésimo
41      mov   r1,#8             @ de la diagonal principal
42      mov   r2,r5
43      mov   r3,r5
44      bl    setbit
45
46      mov   r0,#8             @ activamos el bit i-ésimo
47      mov   r1,#8             @ de la diagonal secundaria
48      mov   r2,r5
49      mov   r3,r4
50      bl    setbit
51
52      cmp   r4,#0             @ comprobamos si hemos terminado
53      beq   fin
54
55      add   r5,r5,#1          @ preparamos los dos siguientes bits
56      sub   r4,r4,#1          @ de la siguiente fila
57      b     otro
58
59  fin:
60      add sp,#8               @ balanceamos la pila
61      pop {r4,r5,pc}          @ recuperamos los registros y volvemos
62      .end
```

Ejercicio 4.20

Realizar un programa que compruebe si dos cadenas, almacenadas respectivamente a partir de las direcciones `cad1` y `cad2`, tienen el mismo número de apariciones de cada carácter que está representado en la cadena.

Para ello, el programa deberá:

- Crear una tabla para cada cadena, donde se almacene la cuenta de cada carácter que tiene la cadena.
- Una vez construidas las tablas, comparar si ambas tablas son iguales.

No se consideran caracteres que no sean 'A'-'Z' o 'a'-'z' y cuentan como iguales los caracteres en mayúsculas y minúsculas.

Para probar el programa, utilice la cadena de ejemplo siguiente:

```
      .data
cad1: .asciz      "AbcDeFg  ZZZ.......ZZZ ., AbCdEFF AA,,,,,,,"
cad2: .asciz      "AbcDeFg  ZZZZCdEFF AAZZ ., Ab"
```

Solución homologas.s

```
/*
 Comprueba si dos cadenas tienen el mismo numero de caracteres
 de cada carácter que tienen en la cadena. No se consideran
 caracteres que no sean 'A'-'Z' o 'a'-'z', y cuentan como
 iguales mayúsculas y minúsculas

 Para ello:

   - Creamos una tabla para cada cadena con la cuenta
     de cada carácter en la posición de dicho carácter:

        num. de caracteres tipo i = tabla[i]

     con a = carácter 0 y z = carácter 25

   - Comparamos si ambas tablas son iguales.

*/
      .data
cad1: .asciz "AbcDeFg  ZZZ.......ZZZ ., AbCdEFF AA,,,,,,,"
cad2: .asciz "AbcDeFg  ZZZZCdEFF AAZZ ., Ab"
tab1: .space 26
tab2: .space 26

      .text
main: ldr     r0, =cad1
      ldr     r1, =tab1
      bl      tabcar

      ldr     r0, =cad2
      ldr     r1, =tab2
      bl      tabcar

      ldr     r0, =tab1
      ldr     r1, =tab2
      mov     r2, #26
      bl      compvec

      wfi
/*
 tabcar:
 Subprograma para contar el numero de ejemplares de cada letra
 que tiene una cadena.

      entrada: r0: dirección de la cadena terminada en 0
               r1: dirección de la tabla para el resultado
      salida: tabla actualizada.
```

```
48  */
49
50          .equ      mascara,  0b11011111    @ para pasar a mayúsculas
51          .equ      lpost, 25               @ la tabla tiene 26 slots
52                                            @ (de 0 a 25)
53  tabcar:
54  @
55  @ Inicializamos la tabla a cero
56  @
57          mov      r2, #0
58          mov      r3, #lpost
59  bini:
60          strb     r2, [r1, r3]
61          sub      r3, r3, #1
62          cmp      r3, #0
63          bne      bini
64          strb     r2, [r1]
65  @
66  @ contamos el numero de apariciones de cada letra que aparece
67  @
68          mov      r3, #mascara  @ para pasar a mayúsculas
69  bucles:
70          ldrb     r2, [r0]      @ cargamos un carácter
71          cmp      r2, #0
72          beq      terms
73
74          and      r2, r2, r3    @ pasamos todo a mayúsculas y
75          cmp      r2, #'A'      @ solo consideramos letras
76          blt      noletra
77
78          cmp      r2, #'Z'
79          bgt      noletra
80  @
81  @ si llegamos aquí tenemos letras
82  @
83          sub      r2, #'A'      @ dejamos las letras en el rango 0-25
84          ldrb     r3, [r1, r2]  @ y lo utilizamos de índice en la tabla
85          add      r3, r3, #1    @ para actualizar la cuenta del "slot"
86          strb     r3, [r1, r2]  @ correspondiente
87  noletra:
88          add      r0, r0, #1    @ actualizamos el puntero a la cadena
89          b        bucles
90
91  terms:
92          mov      pc, lr
93  @
94  @ Fin de tabcar
95  @
96  /*
```

```
 compvec : subprograma para comparar dos vectores del mismo tamaño

      entrada: r0, dir. del primer vector;
      r1 = dir. del segundo vector,
      r2 = tamaño de los vectores
      salida: r0 = 1 si son iguales, 0 en caso contrario.

*/
compvec:
      push     {r4, lr}      @ preservamos r4, y de paso lr

bucc: cmp      r2, #0        @ vemos si hemos terminado
      beq      igual

      sub      r2, r2, #1    @ movemos el índice
      ldrb     r3, [r0, r2]  @ cargamos valores
      ldrb     r4, [r1, r2]
      cmp      r3, r4        @ y comparamos
      bne      dist
      b        bucc          @ el siguiente

igual:
      mov      r0, #1        @ indicador a "son iguales"
      b        fsr
dist:
      mov      r0, #0        @ indicador a "son distintos"
fsr:  pop      {r4, pc}      @ recuperamos r4, y de paso pc
      .end
```

Ejercicio 4.21

MEDIO es una versión simplificada del juego de cartas UNO. En este nuevo juego, cada jugador recibe 10 cartas de una baraja española de 40 cartas (cf. figura 4.2) y una puntuación aleatoria inicial entre 1 y 12 puntos indicada por la denominación de la primera carta que recibe. Gana el jugador que alcanza la mayor puntuación, de acuerdo con la denominación de las 9 cartas restantes, de acuerdo con la tabla siguiente (el palo de la carta es indiferente y las figuras no puntúan):

Carta	Nueva punt. (P)	Carta	Nueva punt. (P)
1	$P = P \times 5$	5	$P = P - 5$
2	$P = P - 10$	6	$P = P \times 3$
3	$P = P + 10$	7	$P = (P + 10) \div 2$
4	$P = P \div 2$	10 - 12	P no varía

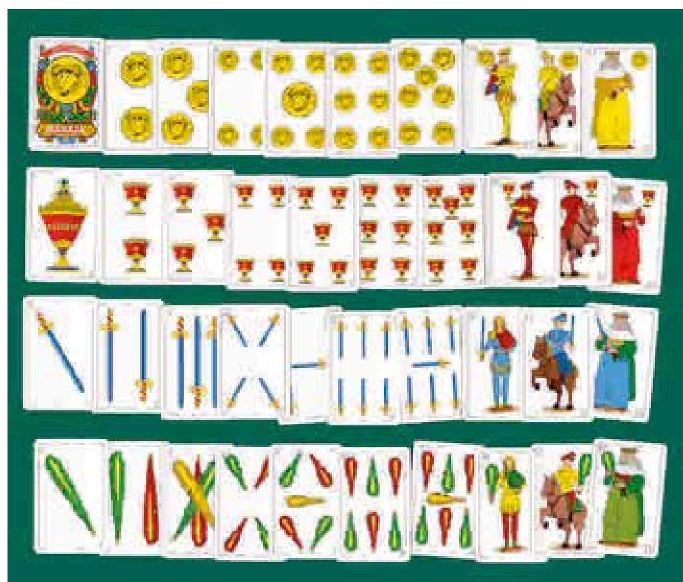


Figura 4.2. Baraja española de 40 cartas. Los palos se denominan oros, copas, espadas y bastos, y los naipes de numeración 10, 11 y 12 se conocen respectivamente como sota, caballo y rey (De Basquetteur - Trabajo propio, CC BY-SA 3.0, Wikipedia).

Realizar un subprograma que resuelva una jugada del juego MEDIO pasada como parámetro. El subprograma recibirá en el byte menos significativo de `r0` la denominación de una carta (sin el palo) y en `r1` la puntuación acumulada hasta el momento. El subprograma devolverá en `r1` la nueva puntuación tras jugar la carta.

Para probar el subprograma, utilice las definiciones y el programa propuestos:

```
      .data
/*
  Secuencia de cartas obtenidas
*/
mano:
      .byte 10, 1, 2, 3, 4, 10, 5, 6, 11, 7, 0

      .text
/*
  Programa de prueba. Partimos de una puntuación
  (en r1) de 10 indicado por la primera carta,
  y jugamos las cartas 1 oros,
  2 copas, 3 espadas, 4 oros, sota de oros,
```

```
  5 copas, 6 bastos, caballo de oros, 7 de oros
  en ese orden. La puntuación final en
  r1 es r1 = 55 = 0x37
*/
main: ldr  r4,=mano     @ r4: dir. lista de cartas
      ldrb r1,[r4]      @ puntuación inicial
      add  r4,r4,#1     @ apuntamos a la siguiente carta
buc:  ldrb r0,[r4]      @ r0: valor de carta
      cmp  r0,#0        @ vemos si hemos terminado
      beq  done
      bl   comp_score   @ jugamos la carta
      add  r4,r4,#1     @ apuntamos a la siguiente carta
      b  buc            @ repetimos

done: wfi
```

Solución

Vamos a utilizar este ejercicio para ilustrar el uso de las tablas de saltos. Una tabla de saltos es básicamente una tabla ordenada de instrucciones de salto o de direcciones. Sirve para transferir el control del programa (bifurcación, salto) a un segmento de código o a un subprograma dependiendo del valor de una variable entera utilizado como índice sobre la tabla. A partir del valor de dicha variable, se calcula un desplazamiento desde el principio de la tabla multiplicando el índice por la longitud de la instrucción o de la dirección de salto (el número de bytes en memoria ocupados por cada instrucción de bifurcación o por cada dirección).

Las tablas de saltos se basan en el hecho de que las instrucciones de salto pueden ejecutarse de manera extremadamente eficiente, y es más útil cuando partimos de un parámetro que puede convertirse fácilmente en un valor de índice secuencial.

En nuestro caso, utilizaremos como índice la numeración de una carta, por lo que la tabla de saltos contendrá las direcciones de los segmentos de código que calculan la nueva puntuación dependiendo de dicha numeración.

Por tanto, la dirección del segmento de código que resuelve la carta de denominación i será:

$$Dir_i = Dir_{TS} + ((i - 1) \times 4)$$

donde Dir_{TS} es la dirección de comienzo de la tabla de saltos, ya que las direcciones de memoria en ARM/Thumb ocupan cuatro bytes.

Mediante una tabla de saltos traducimos a ensamblador una estructura de control del tipo `switch`:

```
switch (carta) {
case 1:
   r1 = r1 * 5;
   break;
case 2:
   r1 = r1 - 10;
   break;
...
}
```

jump_table.s

```
1        .data
2
3  @  Secuencia de cartas obtenidas
4
5  mano:
6        .byte 1, 2, 3, 4, 10, 5, 6, 11, 7, 0
7
8  @  Tabla de saltos
9
10       .balign 4
11 jump_tab:
12 carta1: .word s_carta1
13 carta2: .word s_carta2
14 carta3: .word s_carta3
15 carta4: .word s_carta4
16 carta5: .word s_carta5
17 carta6: .word s_carta6
18 carta7: .word s_carta7
19
20       .text
21 /*
22   Programa de prueba. Partimos de una puntuación
23   (en r1) de 10 y jugamos las cartas 1 oros,
24   2 copas, 3 espadas, 4 oros, sota de oros,
25   5 copas, 6 bastos, caballo de oros, 7 de oros
26   en ese orden. La puntuación final en
27   r1 es r1 = 35 = 0x23
28 */
29 main: mov  r1,#10        @ inicializamos r1 a 10
30       ldr  r4,=mano      @ r4: dir. lista de cartas
31 buc:  ldrb r0,[r4]       @ r0: valor de carta
32       cmp  r0,#0         @ vemos si hemos terminado
33       beq  done
34       bl   comp_score    @ jugamos la carta
35       add  r4,r4,#1      @ apuntamos a la siguiente carta
36       b  buc             @ repetimos
```

```
37
38  done: wfi
39
40  /*
41    comp_score. Juega una carta.
42    Parámetros:
43        - r0: valor de la carta en el byte menos significativo
44        - r1: valor inicial de la puntuación
45    Salida:
46        - r1: nuevo valor de la puntuación.
47  */
48  comp_score:
49        cmp r0,#0          @ si no es una carta válida
50        ble ocs            @ hemos terminado
51        cmp r0,#7
52        bgt ocs
53
54  /* tenemos un carta válida */
55
56        sub r0,r0,#1       @ calculamos la entrada en la tabla
57        lsl r0,r0,#2       @ de saltos para este carta:
58        ldr r2,=jump_tab   @ dir seg. código a r2:
59        ldr r2,[r2,r0]     @ r2 = jump_tab + (ro - 1) * 4
60        push {r2}          @ pasamos la dir. del código en r2
61        pop {pc}           @ al pc
62
63  ocs:  mov pc, lr         @ terminamos. Volvemos al programa
64                           @ principal
65  /*
66    Carta 1: r1 <- r1 * 5
67  */
68  s_carta1:
69        mov r2,r1
70        lsl r1,r1,#2
71        add r1,r1,r2
72        b   ocs
73
74  /*
75    Carta 2: r1 <- r1 - 10
76  */
77  s_carta2:
78        sub r1,r1,#10
79        b   ocs
80
81  /*
82    Carta 3: r1 <- r1 + 10
83  */
84  s_carta3:
85        add r1,r1,#10
```

```
86          b   ocs
87
88  /*
89    Carta 4: r1 <- r1 / 2
90  */
91  s_carta4:
92          lsr r1,r1,#1
93          b   ocs
94
95  /*
96    Carta 5: r1 <- r1 - 5
97  */
98  s_carta5:
99          sub r1,r1,#5
100         b   ocs
101
102 /*
103   Carta 6: r1 <- r1 * 3
104 */
105 s_carta6:
106         mov r2,r1
107         lsl r1,r1,#1
108         add r1,r1,r2
109         b   ocs
110
111 /*
112   Carta 7: r1 <- (r1 + 10) / 2
113 */
114 s_carta7:
115         add r1,r1,#10
116         lsr r1,r1,#1
117         b   ocs
118         .end
```

Ejercicio 4.22

Realizar un programa que calcule el factorial de un número entero de 1 byte almacenado en la posición de memoria etiquetada como `num` de manera recursiva mediante un subprograma. El resultado (palabra de 4 bytes) quedará almacenado a partir de la dirección de memoria etiquetada como `res`.

Para probar el programa, utilice las siguientes definiciones:

```
      .data
num:  .byte 5
      .balign     4
res:  .space      4
```

Solución

Como pide el enunciado, resolvemos el problema teniendo en cuenta que el factorial de un número se puede calcular de manera recursiva:

```
int fact(int n) {
   if (n < 2) return 1;
   return (n * fact (n - 1));
}
```

factorial_srt.s

```
1
2        .data
3  num:  .byte   5
4        .balign 4
5  res:  .space  4
6
7      .text
8  main:
9      ldr    r0, =num   @ tomamos el numero
10     ldrb   r0, [r0]
11     bl     factor     @ llamamos al subprograma
12     ldr    r1, =res
13     strb   r0, [r1]   @ guardamos el resultado
14
15     wfi
16
17
18 /*
19     factor:
20     subprograma de cálculo del factorial de manera recursiva
21
22     entrada: r0 = número para calcular su factorial
23     salida:  r0 = factorial del numero
24
25     p. ej, si llamamos con r0 = 5 el subprograma devuelve
26     r0 = 120 = 0x78
27
28 */
29
30 factor:
31     cmp    r0, #2         @ comprobamos si el cálculo es trivial
32     bge    mayor1
33
34     mov    r0, #1         @ 0! = 1! = 1
35     mov    pc, lr         @ hemos terminado
36 @
37 @ fact (n) = fact(n-1) * n
38 @
39 mayor1:
```

```
40      push  {r1, lr}
41      mov   r1, r0
42      sub   r0, #1
43      bl    factor         @ llamamos con n - 1
44      mul   r0, r1         @ multiplicamos el resultado por n
45      pop   {r1, pc}       @ y volvemos
46      .end
```

Ejercicio 4.23

Realizar un programa que resuelva el problema de las torres de Hanoi. Se trata de un rompecabezas consistente en tres postes o columnas y varios discos ensartados en dichos postes. Los discos tienen diferente tamaño y están ordenados de manera que cada disco puede estar apilado únicamente sobre un disco de mayor tamaño.

El juego se inicia con todos los discos apilados, en orden, en el primer poste. El objetivo del juego es traspasar todos los discos de la primera columna a la tercera, manteniendo el orden (de mayor a menor tamaño, empezando por abajo), tal como estaban al principio, utilizando el poste central como columna auxiliar.

Sólo están permitidos los movimientos siguientes:

1. Solo se puede mover un disco de cada vez.
2. Un disco de mayor tamaño no puede apilarse sobre uno más pequeño.
3. Solo se puede desplazar de una columna a otra el disco situado en la cima.

El algoritmo recursivo para resolver el problema es el siguiente:

```
void hanoi(int n, char de, char a, char aux) {
   if (n==1)
      mover(n, de, a);
   else {
      hanoi(n-1, de, aux, a)
      mover(n, de, a);
      hanoi(n-1 ,aux ,a ,de)
     }
}
```

Para resolver el rompecabezas, realice un subprograma recursivo que reciba los siguientes parámetros:

- `r0`: número de discos en el poste origen.

- `r1`: poste de origen (1 carácter alfanumérico).
- `r2`: poste de destino (1 carácter alfanumérico).
- `r3`: poste auxiliar (1 carácter alfanumérico).

Los movimientos se registrarán en una zona de memoria, donde se almacenarán dichos movimientos en orden creciente, un movimiento en cada palabra de memoria, con el formato siguiente:

`0xNNNNDDAA`

Donde `NNNN` son 16 bits que representan el número de disco que se mueve, entre 1 y n; `DD` representa el carácter ASCII del poste origen y `AA` el carácter ASCII del poste destino.

Para probar el programa, resuelva el problema para cuatro discos y tres postes, utilizando las definiciones siguientes:

```
        .data
point:  .word  table   @ puntero a la tabla de movimientos
table:  .space 64      @ tabla de movimientos (suficiente para 4 discos)
        .equ   ORI,'A' @ nombre del poste origen
        .equ   DES,'C' @ nombre del poste destino
        .equ   AUX,'B' @ nombre del poste auxiliar
```

Solución

Básicamente, nuestro objetivo es traducir a código ensamblador el pseudocódigo del algoritmo recursivo del enunciado. La llamada inicial al algoritmo será:

```
mov    r0,#4     @ 4 discos
mov    r1,#ORI   @ poste de origen es 'A'
mov    r2,#DES   @ poste de destino es 'C'
mov    r3,#AUX   @ poste auxiliar es 'B'
bl     hanoi     @ llamamos al subprograma
```

A partir de ese momento, se irán produciendo llamadas recursivas al mismo subprograma (**`bl`** `hanoi`) hasta el momento que el movimiento a realizar sea un movimiento trivial del disco más pequeño (`r0` = 1). A partir de ahí, se irán recuperando hacia atrás los contextos de cada llamada anidada, con el resultado de mover un solo disco en cada paso de resolución de la recursividad, hasta recuperar el contexto de la llamada inicial (hanoi(4, 'A', 'C', 'B')).

Para implementar con éxito la recursividad necesitaremos almacenar todo el contexto del subprograma en la pila (los registros y la dirección de retorno), de manera que se pueda recuperar de manera ordenada el contexto de cada invocación anidada. La pila es una estructura de almacenamiento LIFO (last-in, first-out; último en entrar, primero en salir), que se corresponde con el orden de recuperación del contexto de las llamadas anidadas en un subprograma que implementa un algoritmo recursivo.

hanoi.s

```
        .data
point:  .word  table   @ puntero a la tabla de movimientos
table:  .space 64      @ tabla de movimientos (para 4 discos)
        .equ   ORI,'A' @ nombre del poste origen
        .equ   DES,'C' @ nombre del poste destino
        .equ   AUX,'B' @ nombre del poste auxiliar

   .text

@ llamada inicial al subprograma:  hanoi(4, 'A', 'C', 'B')

main:
   mov    r0,#4     @ 4 discos
   mov    r1,#ORI   @ poste de origen es 'A'
   mov    r2,#DES   @ poste de destino es 'C'
   mov    r3,#AUX   @ poste auxiliar es 'B'
   bl     hanoi     @ llamamos al subprograma
   wfi

/* Subprograma hanoi. Parámetros:

   r0 -> n (número de discos)
   r1 -> poste origen  (char)
   r2 -> poste destino (char)
   r3 -> poste auxiliar (char)
*/
hanoi:
   push   {r0-r6,lr} @ algoritmo recursivo: todo el contexto
   cmp    r0,#1      @ a la pila
   bne    again      @ if (n<>1) seguimos con la recursividad

@ n == 1: mueve (1, de, a)

   lsl    r0,#8      @ en r0 tenemos el número de disco
   orr    r0,r1      @ montamos el poste origen sobre r0
   lsl    r0,#8
```

```
39      orr    r0,r2      @ y también el poste destino
40
41      ldr    r4,=point  @ almacena el último movimiento
42      ldr    r5,[r4]
43      str    r0,[r5]
44      add    r5,#4      @ actualizamos el puntero a la tabla
45      str    r5,[r4]
46      b      finh
47
48  @ n <> 1 : hanoi(n-1, de, aux, a); mueve(n, de, a);;
49  @          hanoi(n-1 ,aux ,a ,de)
50
51  again:
52      sub    r0,#1      @ llamada recursiva
53      mov    r4,r3      @ hanoi(n-1, de, aux, a)
54      mov    r3,r2
55      mov    r2,r4
56      bl     hanoi
57
58      mov    r6,r0      @ mueve (n, de, a)
59      add    r6,#1      @ recuperamos el valor de n
60      lsl    r6,#8      @ y montamos sobre r6
61      orr    r6,r1      @ n, poste origen y poste destino
62      lsl    r6,#8
63      orr    r6,r3
64
65      ldr    r4,=point
66      ldr    r5,[r4]
67      str    r6,[r5]   @ almacena este movimiento
68      add    r5,#4     @ actualizamos el puntero
69      str    r5,[r4]
70
71      mov    r4,r1     @ llamada recursiva
72      mov    r1,r2     @ hanoi(n-1, aux, a, de)
73      mov    r2,r3
74      mov    r3,r4
75      bl     hanoi
76
77  @ recuperamos el contexto de la pila
78
79  finh:
80      pop    {r0-r6,pc}
81      .end
```

Apéndice A
Ejercicios introductorios para Raspberry Pi

En este apéndice reelaboramos los ejercicios introductorios del capítulo 1 para el caso de la Raspberry Pi. Proponemos una serie de ejercicios sencillos orientados a familiarizarse con las herramientas de programación de GNU disponibles en la Raspberry Pi. Además, repasamos algunos conceptos básicos relacionados con la representación y almacenamiento de información en la arquitectura ARM T32/Thumb.

El apéndice B incluye, a modo de referencia rápida, información sobre las instrucciones (cf. cuadros B.4 a B.8) y las directivas del ensamblador de GNU utilizadas en este libro (cf. cuadro B.9).

Ejercicio A.1

El siguiente programa inicializa los registros `r0` a `r3` con 4 valores numéricos en decimal, hexadecimal, octal y binario, respectivamente. Edita el programa utilizando el editor de tu preferencia, ensámblalo y enlázalo tal como se describe en el apartado 1.2, carga tu programa en GDB y responde a las preguntas.

e_intro_1.s

```
        .text
        .code  16
        .align 2

ej1:    mov r0, #30
        mov r1, #0x42
        mov r2, #0102
        mov r3, #0b1000010
fin:    bx lr

        .code 32
        .align 4
        .global main
main:
        push {r4, lr}
        blx ej1
```

```
        pop {r4, lr}
        bx lr
       .end
```

1. ¿Cómo se muestran los números anteriores al listar el programa en GDB? ¿Están en la misma base que en el código original? ¿En qué base están representados?

2. Ejecuta el programa paso a paso. ¿Qué números se almacenan en los registros `r0` a `r3`?

   Solución

   Los datos inmediatos en el código desensamblado se representan en decimal, con lo que al observar el código en GDB obtendríamos algo como:

```
0x103d2 <ej1>     movs   r0, #30
0x103d2 <ej1+2>   movs   r1, #66
0x103d4 <ej1+4>   movs   r2, #66
0x103d6 <ej1+6>   movs   r3, #66
0x103d8 <fin>     bx     lr
```

   En cuanto a la representación del contenido de los registros, en general se representa en hexadecimal, salvo que se solicite expresamente un formato diferente (por ejemplo, con el comando `x` de GDB). Así, al ejecutar el programa el contenido de los registros se representa como sigue tras ejecutar el comando `info registers` de GDB:

```
r0   0x1e
r1   0x42
r2   0x42
r3   0x42
```

Ejercicio A.2

Edita el código presentado a continuación, ensámblalo, analízalo con GDB y responde las preguntas.

e_intro_2.s

```
        .data
word1: .word 11
word2: .word 0x11
word3: .word 011
word4: .word 0b11
```

```
       .text
       .code  16
       .align 2

nada:   bx lr

       .code 32
       .align 4
       .global main
main:
        push {r4, lr}
        blx nada
        pop {r4, lr}
        bx lr
       .end
```

1. Identifica las posiciones de memoria donde se almacenaron los datos definidos en el programa. Identifica los cuatro valores en hexadecimal.

2. ¿En qué direcciones se almacenan los cuatro valores? ¿Por qué estas direcciones de memoria son múltiplos de cuatro en lugar de ser direcciones consecutivas?

3. ¿Cuáles son los valores de las etiquetas `word1`, `word2`, `word3` y `word4`?

Solución

El resultado obtenido con GDB es el siguiente:

```
[0x00021024]  0x0000000B
[0x00021028]  0x00000011
[0x0002102C]  0x00000009
[0x00021030]  0x00000003
```

Las direcciones, en el caso de la Raspberry Pi, son direcciones de memoria virtual asignadas por el sistema operativo en el momento de cargar el programa, por lo que pueden variar de un ordenador a otro, o incluso de una sesión a otra. En cualquier caso, lo importante es darnos cuenta de que cada palabra ocupa 4 bytes y de que el convenio de almacenamiento o *endianness* es el extremista menor (*little endian*), es decir, el byte menos significativo de la palabra se almacena en la posición de memoria de dirección más baja.

Por tanto, los valores de las etiquetas `word1`, `word2`, `word3` y `word4` serán los indicados a continuación:

Etiqueta	Valor
`word1`	`0x00021024`
`word2`	`0x00021028`
`word3`	`0x0002102C`
`word4`	`0x00021030`

Ejercicio A.3

Ensambla el siguiente código:

e_intro_3.s

```
      .data
wrds: .word 11, 0x11, 011, 0b11

      .text
      .code  16
      .align 2

nada:  bx lr

      .code 32
      .align 4
      .global main
main:
       push {r4, lr}
       blx nada
       pop {r4, lr}
       bx lr
      .end
```

¿Hay algún cambio en los valores almacenados en la memoria con respecto a los almacenados en el caso del programa anterior? ¿Están en el mismo lugar?

Solución

No hay ningún cambio en lo que respecta al almacenamiento de los datos anteriores. El único cambio sería que ahora definimos una única etiqueta (`wrds`) con valor `0x00021024`.

Ejercicio A.4

Ensambla el siguiente código:

e_intro_4.s

```
       .data
bys:   .byte 0x18, 0x19, 0x1a, 0x1b

      .text
```

```
        .code  16
        .align 2

nada:   bx lr

        .code 32
        .align 4
        .global main
main:
        push {r4, lr}
        blx nada
        pop {r4, lr}
        bx lr
        .end
```

1. ¿Qué valores se almacenaron en la memoria? ¿En qué posiciones?
2. ¿Cuál es el valor de la etiqueta `bys`?

   Solución

   Se almacena un byte en cada posición de memoria, comenzando en la dirección `0x00021024`. La etiqueta `bys` tomará dicho valor.

Ejercicio A.5

Ahora ensambla el siguiente código:

e_intro_5.s

```
        .data
strg:   .ascii "abác"
byte:   .byte  0xff

        .text
        .code  16
        .align 2

nada:   bx lr

        .code 32
        .align 4
        .global main
main:
        push {r4, lr}
        blx nada
        pop {r4, lr}
        bx lr
        .end
```

1. ¿Qué rango de posiciones de memoria se reservó para la variable etiquetada como `strg`?

2. ¿Cómo se representa en memoria la cadena `abác`? ¿Cómo se representa el carácter `á`?

3. Averigua cuál es el sistema de codificación de caracteres de tu entorno de trabajo (UTF-8, ASCII, ISO Latin 9 ...).

4. ¿Cuál es el valor de la etiqueta `byte`?

   Solución

   Se trata de una cadena de caracteres. Raspberry Pi se basa en el ensamblador de GNU, que utiliza UTF-8 como sistema de representación de caracteres en la mayoría de las plataformas, incluida la propia Raspberry Pi.

   Por lo tanto, para la cadena etiquetada como `streg` se reservarán 5 posiciones y la representación en memoria byte a byte será:

   ```
   0x61 0x62 0xc3 0xa1 0x63
   ```

   Vemos que el carácter `á` se representa con el código de dos bytes (`0xc3`, `0xa1`) en UTF-8, de acuerdo con lo descrito en el apéndice C.

   Finalmente, el valor de la etiqueta `byte` será `0x00021029`.

Ejercicio A.6

Ensambla y analiza el siguiente código:

e_intro_6.s

```
        .data
b1:     .hword 0x11
gap:    .space 4
b2:     .byte  0x22
bign:   .word  0x33445566

        .text
        .code  16
        .align 2

nada:    bx lr

        .code 32
        .align 4
        .global main
main:
```

```
    push {r4, lr}
    blx nada
    pop {r4, lr}
    bx lr
   .end
```

1. ¿Cuántas posiciones de memoria se reservan para la variable `gap`?
2. ¿Podrían leerse o escribirse los cuatro bytes utilizados por la variable `gap` como si fueran una palabra? ¿Por qué?
3. ¿Cuál es el valor de la etiqueta `b1`? ¿Y de la etiqueta `b2`?
4. ¿Cuál es el valor de la etiqueta `bign`? ¿Podrían leerse o escribirse los cuatro bytes que comienzan en la dirección `bign` como si fueran una palabra? ¿Por qué?
5. Agrega una directiva `.balign` al código anterior para que la variable etiquetada como `bign` se alinee con un límite de palabra (es decir, con una dirección múltiplo de cuatro).

Solución

Para la variable `gap` se reservan 4 posiciones de memoria. No podremos acceder en lectura o escritura a los 4 bytes de `gap` como si fueran una palabra porque los accesos a palabra en ARM tienen que estar alineados a una dirección múltiplo de 4.

Para poder acceder a `bign` como una palabra tendríamos que añadir una directiva de alineación `.balign 4` o bien `.align 2` antes de la definición de dicha etiqueta. En ambos casos se alinearía el espacio etiquetado por `bign` a una dirección múltiplo de 4.

El código es el siguiente:

e_intro_7.s

```
1        .data
2  b1:    .hword   0x11       @ esto ocupa dos bytes
3  gap:   .space   4          @ cuatro bytes adicionales
4  b2:    .byte    0x22       @ un byte adicional (total 7 bytes)
5         .balign 4           @ alineamos a un múltiplo de 4
6  bign:  .word    0x33445566 @ ahora podemos definir una palabra
7
8         .text
9         .code  16
10        .align 2
11
12 nada:  bx lr
```

```
13
14        .code 32
15        .align 4
16        .global main
17 main:
18         push {r4, lr}
19         blx nada
20         pop {r4, lr}
21         bx lr
22        .end
```

Apéndice B
Modelo del programador

Thumb, conocido como T32 a partir de la versión 8 de la arquitectura ARM, es un estado de los microprocesadores de dicha arquitectura que proporciona una versión compacta del repertorio de instrucciones, donde se combinan instrucciones de 16 bits con instrucciones de 32 bits. T32/Thumb dispone de versiones de 16 bits de las instrucciones que se usan con más frecuencia, y en el caso de aplicaciones que no necesiten toda la potencia expresiva original de ARM, los programas escritos con el repertorio Thumb ocuparán menos que los programas equivalentes escritos en la versión completa del repertorio de ARM (repertorio A32 de 32 bits).

En este apéndice presentamos una breve introducción al modelo del programador Thumb de 16 bits, a modo de referencia y herramienta de consulta rápida a la hora de afrontar los ejercicios y problemas propuestos en el libro. En primer lugar describimos de manera sucinta la organización de la memoria y los registros cuando el microprocesador funciona en estado T32/Thumb. A continuación, enumeramos los modos de direccionamiento disponibles, así como las instrucciones utilizadas en el libro. Finalmente, identificamos las directivas del ensamblador de GNU utilizadas en los programas recogidos en este manual.

B.1 Registros del programador

La arquitectura ARM tiene 16 registros de propósito general de 32 bits, de los cuales se suelen asignar funciones especiales al registro `r13`, que funciona como puntero de pila, y al registro `r14`, que funciona como registro de enlace para almacenar la dirección de retorno tras una llamada a subprograma. Además, `r15` hace las funciones de contador de programa.

En el caso del estado T32/Thumb, solo se dispone de los ocho primeros registros como registros de propósito general (cf. figura B.1) y, como veremos más adelante, determinadas instrucciones como **`push`** en el caso de `r13` o **`bl`**

en el caso de `r14` utilizan implícitamente estos dos registros como puntero de pila y registro de enlace respectivamente.

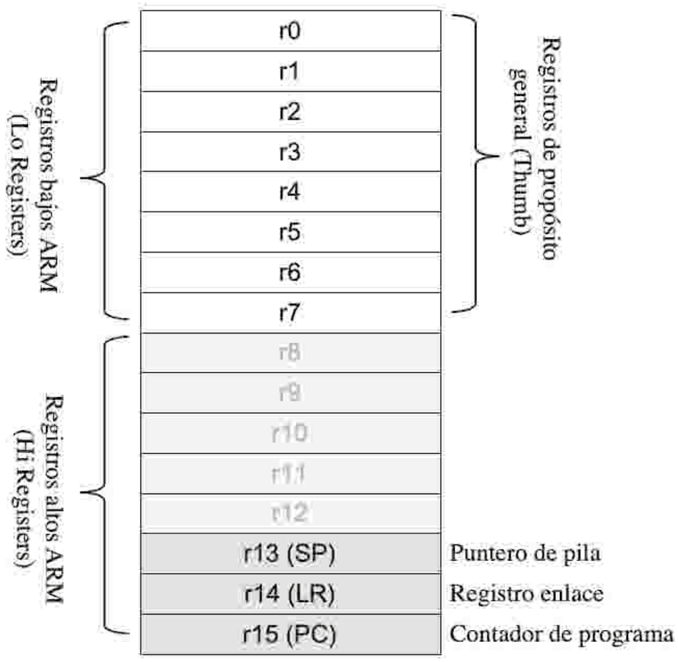


Figura B.1. Registros del programador T32/Thumb: ocho registros de propósito general (`r0` – `r7`), más el puntero de pila, el registro de enlace y el contador de programa (`r13` – `r15`).

La limitación en el uso de los registros a los registros `r0` a `r7`, junto con la interpretación implícita de la función de los registros `r13` y `r14`, que por tanto no necesita codificación adicional más allá del propio código de operación, permite codificar una amplia variedad de instrucciones con tan solo 16 bits por instrucción.

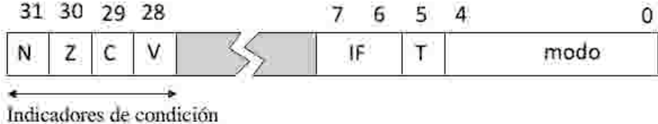


Figura B.2. Registro de estado (CPSR - Current Processor Status Register). Los cuatro bits más significativos son los indicadores de condición.

En cuanto al registro de estado (cf. figura B.2), los cuatro bits de mayor peso almacenan los indicadores de condición (*condition flags*), mientras que los ocho bits de menor peso contienen información del sistema relacionada

con el estado del procesador y los mecanismos de tratamiento de las interrupciones. Los indicadores de condición se utilizan, entre otros cometidos, para tomar decisiones sobre el flujo de programa mediante las instrucciones de salto condicional. Los códigos de condición de las instrucciones de salto tienen el significado recogido en el cuadro B.1 .

Cuadro B.1. Significado de los códigos de condición (Cond.) en función de los indicadores `N`, `Z`, `C` y `V`.

Cond.	Ind.	Significado
`mi`	N	Minus - Negativo
`eq`	Z	Equal - Igual / Cero
`cs`	C	Carry set - Acarreo
`vs`	V	oVerflow set - Desbordamiento
`hi`	$C\overline{Z}$	Higher - Mayor sin signo
`gt`	$\overline{Z}(N = V)$	Greater - Mayor que
`ge`	$N = V$	Greater or equal - Mayor o igual
`pl`	$\overline{N}$	Plus - Positivo
`ne`	$\overline{Z}$	Not equal - Distinto (de cero)
`cc`	$\overline{C}$	Carry clear - No acarreo
`vc`	$\overline{V}$	oVerflow clear - No desbordamiento
`ls`	$\overline{C} + Z$	Lower or same - Menor o igual sin signo
`le`	$Z + (N \neq V)$	Less or equal - Menor o igual que
`lt`	$N \neq V$	Less than - Menor que

B.2 Organización de la memoria

La memoria se organiza en palabras de 4 bytes y es direccionable a nivel de byte. Esto implica que hay una dirección de memoria distinta para cada byte que forma parte de la memoria del computador. Las instrucciones que acceden a palabras en memoria, necesitan que estas estén alineadas en direcciones de memoria múltiplo de 4.

El convenio para el orden de almacenamiento de las palabras en memoria por defecto es el extremista menor (*little-endian* en inglés), es decir, el byte de menor peso de la palabra se almacena en la dirección de memoria más baja.

Además de trabajar con bytes y palabras de 4 bytes, algunas instrucciones trabajan con medias palabras (16 bits, 2 bytes). En el caso de las instrucciones que utilizan medias palabras almacenadas en memoria, sus direcciones deben ser múltiplo de 2.

Las direcciones de memoria son de 32 bits, por lo que el rango de direccionamiento directo es de 0 a $2^{32} - 1 =$ 4.294.967.295 bytes, es decir, el tamaño máximo de la memoria direccionable es de 4 GB.

B.3 Modos de direccionamiento

Una instrucción en ensamblador codifica qué operación se debe realizar, con qué operandos fuente hay que realizar dicha operación y dónde se debe guardar el resultado. En el ámbito de la arquitectura de ordenadores, se conoce como modos de direccionamiento a las distintas formas en las que puede indicarse la dirección efectiva de los operandos y del resultado de las instrucciones de un procesador.

El cuadro B.2 recoge los modos de direccionamiento soportados en el estado T32/Thumb.

Cuadro B.2. Modos de direccionamiento en el estado T32/Thumb.

Modo	Dirección efectiva
Directo a registro	DE = Registro (dato en registro)
Inmediato	DE = Instrucción (dato en la instrucción)
Indirecto con desp.	DE = R + desplazamiento (5 bits)
Relativo a PC	DE = PC + desplazamiento (8 bits)
Relativo a SP	DE = SP + desplazamiento (8 bits)
Indirecto con RD	DE = R + RD

Modo	Ejemplo
Directo a registro	`add r0, r1, r2`
Inmediato	`add r0, r1, #4`
Indirecto con desp.	`ldr r0, [r7, #4]`
Relativo a PC	`ldr r0, [pc, #20]`
Relativo a SP	`ldr r0, [sp, #20]`
Indirecto con RD	`ldr r0, [r1, r3]`

B.3.1 Direccionamiento directo a registro

El direccionamiento directo a registro se utiliza en la mayor parte de las instrucciones, tanto de transferencia, como de transformación de datos, para algunos o todos sus operandos. En Thumb se utiliza este modo, por ejemplo,

para especificar los dos operandos fuente y el destino de las instrucciones de suma y resta. Así, la instrucción `add r0, r2, r3` sumaría los contenidos de los registros `r2` y `r3` dejando el resultado en el registro `r0` y la instrucción `sub r0, r1, r2` restaría el contenido del registro `r2` del registro `r1` dejando el resultado en el registro `r0`.

B.3.2 Direccionamiento inmediato

En el caso del direccionamiento inmediato, uno de los operandos se codifica en la propia instrucción. El rango de valores de dicho operando puede variar dependiendo de cada instrucción concreta, utilizándose para su codificación entre 3 y 8 bits (cf. cuadro B.3). Como ejemplo de instrucciones que usen este modo de direccionamiento están las instrucciones aritméticas (p. ej. `add r2,r1,#Inm3`, `sub r2,#Inm8`, `sub SP,#Inm9`, o `add Rd,SP,#Imm10`).

Debido a que las palabras de 32 bits tienen que estar alineadas a direcciones múltiplo de cuatro y las medias palabras tienen que estar alineadas a direcciones pares, no es necesario codificar todos los bits del dato inmediato si se conoce de antemano que dicho dato es una palabra o una media palabra. Por ejemplo, la instrucción `add SP, #Inm9`, suma la constante `#Inm9` al puntero de pila (cf. figura B.3). Como el puntero de pila almacena una dirección, el resultado de la operación tiene que ser también una dirección. Por ello, para codificar los nueve bits de `#Inm9` solo necesitamos almacenar los siete más significativos, ya que los dos bits menos significativos tienen que ser necesariamente 0.

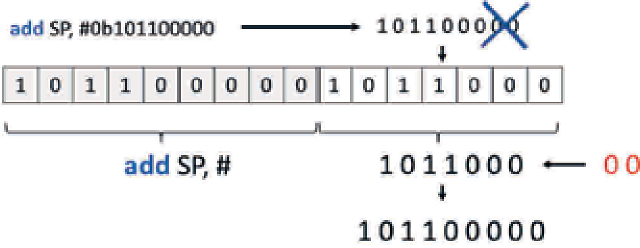


Figura B.3. Codificación de la instrucción `add SP,#0x160`. La instrucción codifica sólo los siete bits más significativos del dato inmediato (los dos bits menos significativos toman siempre el valor 0).

Lo mismo ocurre con `#Inm10` en `add Rd,SP,#Imm10`. Para codificar el dato inmediato necesitamos almacenar únicamente ocho bits, siendo los dos menos significativos siempre cero.

En definitiva, la codificación y el rango de los datos inmediatos difieren en aquellas instrucciones que utilizan el dato inmediato para calcular la dirección

de una palabra o de una media palabra, es decir, para sumar o restar una constante al puntero de pila (cf. cuadro B.4), o bien para calcular la dirección efectiva en una instrucción de carga o almacenamiento de palabras o medias palabras (cf. cuadro B.6) con direccionamiento indirecto con desplazamiento.

Estas situaciones en las que el rango de valores del dato inmediato (p. ej., de 0 a 510) no coincide con el número de elementos codificados se representan en negrita en la tabla B.3.

Cuadro B.3. Posibles valores de los datos inmediatos. Aparecen en negrita aquellos casos en los que el rango de valores codificados es mayor que el representable con el número de bits reservados en la instrucción. En esos casos, uno o dos de los bits que codifican el dato inmediato toman el valor 0 de manera implícita, en función de si codifican palabras o medias palabras. Por ejemplo Inm5 se codifica con 5 bits y **Imn9** se codifica con 7 bits (los dos bits menos significativos son cero). x2 y x4 significan respectivamente que el dato inmediato es múltiplo de 2 o múltiplo de 4.

Rango de los operandos inmediatos					
Inm3	0 – 7	**Inm6**	0 – 62, x2	**Inm10**	0 – 1020, x4
Inm5	0 – 31	**Inm7**	0 – 124, x4		
Inm8	0 – 255	**Inm9**	0 – 508, x4		

B.3.3 Direccionamiento indirecto a registro con desplazamiento

Este modo se utiliza en las instrucciones `ldr`/`ldrh`/`ldrb` y `str`/`strh`/`strb` de carga y almacenamiento, para el operando fuente y el destino respectivamente. La dirección efectiva se calcula sumando una constante al contenido de un registro de dirección, cuyo rango dependerá del tamaño del operando o el destino. En todos los casos, se utilizan 5 bits para codificar un desplazamiento.

- En el caso de `ldr` y `str`, cuyos operandos están alineados a palabra, la instrucción define un desplazamiento de 7 bits, donde los 5 bits más significativos del desplazamiento están codificados en dicha instrucción y los 2 bits menos significativos toman valor 0. De esta manera, el rango del desplazamiento es de (0 – 124) bytes o (0 – 31) palabras.

- En el caso de `ldrh` y `strh`, cuyos operandos están alineados a medias palabras, la instrucción define un desplazamiento de 6 bits, donde los 5 bits más significativos están codificados en la instrucción y el bit menos significativo toma valor 0. Así, el rango del desplazamiento es de (0 – 62) bytes o (0 – 31) medias palabras.

- En el caso de **ldrb** y **strb**, con operandos alineados a byte, se define un desplazamiento de 5 bits codificados en la instrucción, con un rango de desplazamiento de (0 – 31) bytes.

Por ejemplo, la instrucción de carga `ldr r0, [r7, #0b1010100]` realiza la operación R0 ← [R7 + 0b1010100], pero en la instrucción solamente se codifican los 5 bits más significativos del desplazamiento (10101).

Este modo se conoce como direccionamiento indirecto con desplazamiento porque, si en el direccionamiento directo a registro el dato estaba en el registro codificado en la instrucción, en este modo el dato está en la dirección indicada por dicho registro, y por tanto se trata de un direccionamiento indirecto.

La instrucción **bx** de salto e intercambio (cf. apartado 1.2 del capítulo 1 y apéndice A) supone un caso particular de este modo de direccionamiento. En este caso, la dirección efectiva se define mediante los 31 bits más significativos del registro codificado en la instrucción (el bit menos significativo de la dirección efectiva tiene que ser 0, ya que las instrucciones tienen que comenzar siempre en direcciones pares). Se trataría por tanto de un modo de direccionamiento indirecto a registro (sin desplazamiento). Por ejemplo, el siguiente código:

```
ldr r0,=0x28000000
bx  r0
```

provocaría un salto a la dirección `0x28000000` (y además el procesador pasaría al estado ARM completo).

B.3.4 Direccionamiento relativo al contador de programa

El direccionamiento relativo al contador de programa es una variante del direccionamiento indirecto con desplazamiento en el que el registro de dirección es el contador de programa (`pc`, `r15`). Este modo especifica la dirección efectiva del operando o de un salto como la suma del contenido del contador de programa y un desplazamiento codificado en la propia instrucción.

En el caso de las instrucciones de carga y almacenamiento, este modo de direccionamiento solo está disponible con operandos de 32 bits (instrucciones **ldr** y **str** únicamente), y el desplazamiento tiene un rango de 10 bits de los cuales se codifican en la instrucción 8 (los 2 bits menos significativos tienen valor 0). De esta manera, el rango del desplazamiento es de (0 – 1020) bytes o (0 – 255) palabras. En el caso de las pseudoinstrucciones `ldr r0,=etiqueta` o análogas, donde `etiqueta` es una etiqueta que referencia una dirección de memoria en un programa, el ensamblador sustituye

dicha pseudoinstrucción por una instrucción equivalente con direccionamiento relativo a PC, como vimos en el capítulo 1 (cf. figura 1.3).

El modo de direccionamiento relativo a contador de programa es especialmente útil para acceder a posiciones de memoria que se encuentran en las inmediaciones de la instrucción que se está ejecutando. En Thumb de 16 bits, para que las instrucciones de salto se puedan codificar en únicamente dos bytes y el código sea directamente reubicable en su mayor parte, en lugar de saltos absolutos se recurre a utilizar saltos relativos al PC. En el caso de la instrucción de salto incondicional `b`, el desplazamiento es un número de 12 bits en complemento a 2 que proporciona un rango de desplazamiento de $\pm$ 2048 bytes. De esos 12 bits, se codifican los 11 bits más significativos, ya que el bit menos significativo tiene que ser 0 al estar las instrucciones alineadas a direcciones pares. Las instrucciones de salto condicional tienen un rango de desplazamiento de 9 bits ($\pm$ 256 bytes), de los cuales se codifican 8 en la instrucción.

En el caso de la instrucción de llamada a subprograma `bl` el desplazamiento se codifica de una manera especial. En este caso, el desplazamiento es un número de 23 bits en complemento a dos, de los cuales se codifican 22 bits, proporcionando un rango de $\pm$ 4 Mbytes. Obviamente, es imposible codificar los 22 bits del desplazamiento utilizando una sola instrucción de 16 bits, por lo que al ensamblarse dicha instrucción se generan dos instrucciones `bl` consecutivas, donde la primera de ellas codifica los 11 bits más significativos del desplazamiento y la segunda los 11 bits menos significativos. En otras palabras, la instrucción siempre se codifica como un par de instrucciones, y así es posible codificar un *salto lejano* con un desplazamiento de 23 bits. Por ejemplo, la instrucción `bl eti`, donde `eti` apunta a una dirección situada `0x0670fd` posiciones más adelante, se codifica como dos instrucciones `bl`, donde la primera codifica la parte alta del offset como `0x06d`, y la segunda la parte baja del offset como `0x07f` (cf. figura B.4).

Como indicamos antes para las instrucciones de carga y almacenamiento, al programar en ensamblador no es necesario calcular manualmente el desplazamiento a sumar al contador de programa para obtener la dirección efectiva de un salto. Para ello utilizaremos en las instrucciones de salto una etiqueta que identifica la localización de la dirección efectiva (e.g., `bcc eti`). El ensamblador calculará el desplazamiento correcto teniendo en cuenta la localización de la instrucción y la dirección de destino.

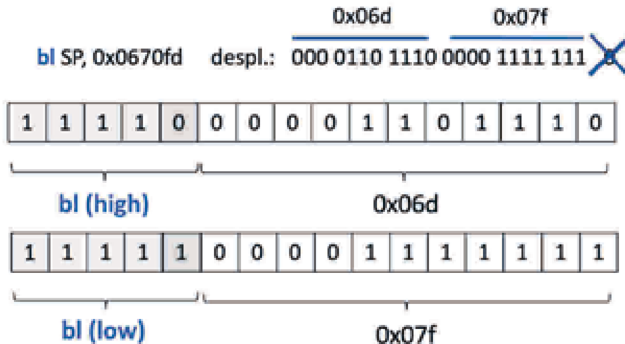


Figura B.4. Codificación de la instrucción **bl** `0x0670fd`. La instrucción se codifica como dos instrucciones **bl** consecutivas, donde cada una de ellas codifica la mitad de los bits del desplazamiento.

B.3.5 Direccionamiento relativo al puntero de pila

El direccionamiento relativo al puntero de pila es similar al relativo al contador de programa, pero utilizando como registro de direccionamiento el puntero de pila (`sp`, `r14`). La dirección efectiva es la suma del contenido del puntero de pila y un desplazamiento de 10 bits codificado en la propia instrucción con 8 bits, con un rango del desplazamiento de (0 – 1020) bytes o (0 – 255) palabras.

El direccionamiento relativo al puntero de pila se utiliza únicamente en las instrucciones **`ldr`** y **`str`**, y de manera implícita en las instrucciones **`push`** y **`pop`** de acceso a la pila.

B.3.6 Direccionamiento indirecto a registro con registro de desplazamiento

El modo de direccionamiento indirecto con registro de desplazamiento es muy similar al indirecto con desplazamiento. La diferencia radica en que el desplazamiento se obtiene de un registro en lugar de ser un dato inmediato. Un ejemplo sería la instrucción **`str`** `r0, [r1, r3]`, que cargaría en el registro `r0` la palabra cuya dirección de memoria es la suma de los contenidos de `r1` y `r3` ($R0 \leftarrow [R1 + R3]$).

Este modo está disponible para las instrucciones de carga y almacenamiento de bytes, palabras y medias palabras, y además para las instrucciones de carga de bytes y palabras con extensión de signo **`ldsb`** y **`ldsh`** (cf. tabla B.6).

B.4 Repertorio de instrucciones

Las tablas siguientes recogen las instrucciones utilizadas en el libro. Se trata básicamente del repertorio T32/Thumb del núcleo ARM7TDMI, la primera versión del repertorio, que incluía únicamente las instrucciones de 16 bits. Para una descripción detallada de estas instrucciones consúltese el capítulo 5 del manual de referencia de dicho núcleo[1] *Thumb Instruction Set, ARM7TDMI Data Sheet,* de Advanced RISC Machines Ltd. (ARM), un documento fácilmente localizable en Internet.

Existen tres instrucciones en la referencia anterior que no hemos utilizado en este manual introductorio. Se trata de **ldmia**, **stmia** y **swi**. Las instrucciones **ldmia** y **stmia** tienen un comportamiento similar a **pop** y **push** respectivamente, solo que el registro utilizado como puntero puede ser cualquier registro de datos `r0` – `r7`. Por ejemplo, la instrucción `ldmia r1!,{r0,r3-r5}` lee de la memoria la palabra indicada por el registro `r1` y las tres siguientes, almacenando su contenido en los registros `r0`, `r3`, `r4` y `r5`, e incrementa `r1` en 16 unidades (4 palabras), y `stmia r14!,{r0,r3-r5}` es equivalente a `push {r0,r3-r5}`. Utilizar estas instrucciones añadiría una complejidad innecesaria a los ejercicios de un texto introductorio como este, que por otra parte ya ilustra el uso de las instrucciones **push** y **pop**.

La instrucción **swi** dispara una interrupción software. Al tratarse de un texto introductorio sobre programación en ensamblador, no hemos tratado las interrupciones, por lo que la instrucción **swi** no tenía cabida en nuestros programas.

Como vimos antes, la instrucción **bx** de la tabla B.8 se utiliza únicamente en el apartado 1.2 del capítulo 1 y en el apéndice A para devolver el control al sistema operativo desde nuestros programas en Thumb para la Raspberry Pi.

Además de las instrucciones Thumb del núcleo ARM7TDMI recogidas en las tablas siguientes, en los ejercicios propuestos en este libro utilizamos la instrucción del repertorio Thumb-2 **wfi** para marcar la terminación de nuestro código al utilizar el simulador QtARMSim. En un entorno multitarea real, la instrucción **wfi** (wait for interrupt) pone al procesador en modo de espera hasta que se produce una interrupción, y en algunos microprocesadores para aplicaciones embebidas además se pasa al modo de bajo consumo. En el

[1] El núcleo ARM7TDMI (ARM7 + **T**humb de 16 bits + **D**epuración JTAG + **M**ultiplicador rápido + emulador **I**n-circuit mejorado) es uno de los núcleos ARM más usados hasta el año 2009. Lo podemos en contrar en miles de sistemas embebidos. Entre los usos notables del núcleo ARM7 tenemos el móvil Nokia 6110, las consolas Dreamcast, Game Boy Advance y Nintendo DS, el reproductor Zune HD, el iPod, y la aspiradora Roomba.

momento de producirse la interrupción, la rutina de servicio correspondiente tomará el control del procesador para ceder dicho control a una nueva tarea.

Cuadro B.4. Instrucciones aritméticas. Obsérvese que el registro puntero de pila `sp` tiene un tratamiento especial, con instrucciones específicas de suma y resta.

Mover			
Inmediato 8 bits	`mov Rd, #Inm8`	Rd ← Inm8	NZ · ·
Registros	`mov Rd, Ro`	Rd ← Ro	NZ · ·
Sumar			
Inmediato 3 bits	`add Rd, Ro, #Inm3`	Rd ← Ro + Inm3	NZCV
Inmediato 8 bits	`add Rd, Rd, #Inm8`	Rd ← Rd + Inm8	NZCV
Registros	`add Rd, Ro1, Ro2`	Rd ← Ro1 + Ro2	NZCV
Al SP (`r13`)	`add SP, #Inm9`	SP ← SP + Inm9	· · · ·
SP a Reg.	`add Rd, SP, #Inm10`	Rd ← SP + Inm10	· · · ·
Con acarreo	`adc Rd, Rd, Ro`	Rd ← Rd + Ro + C	NZCV
Restar			
Inmediato 3 bits	`sub Rd, Ro, #Inm3`	Rd ← Ro − Inm3	NZCV
Inmediato 8 bits	`sub Rd, Rd, #Inm8`	Rd ← Rd − Inm8	NZCV
Registros	`sub Rd, Ro1, Ro2`	Rd ← Ro1 − Ro2	NZCV
Al SP (`r13`)	`sub SP, #Inm9`	SP ← SP − Inm9	· · · ·
Con acarreo	`sbc Rd, Rd, Ro`	Rd ← Rd − Ro − $\overline{C}$	NZCV
Negar	`neg Rd, Ro`	Rd ← − Ro	NZCV
Multiplicar			
Registros	`mul Rd, Ro, Rd`	Rd ← Ro ∗ Rd	NZCV
Comparar			
Registros	`cmp Rn, Rm`	Rn − Rm	NZCV
Registros negado	`cmn Rn, Rm`	Rn + Rm	NZCV
Inmediato 8 bits	`cmp Rn, #Inm8`	Rn − Inm8	NZCV

Cuadro B.5. Instrucciones lógicas y de desplazamiento.

Lógicas			
AND	`and Rd, Rd, Ro`	$Rd \leftarrow Rd\ AND\ Ro$	NZ · ·
Borrar bits	`bic Rd, Rd, Ro`	$Rd \leftarrow Rd\ AND\ NOT(Ro)$	NZ · ·
OR	`orr Rd, Rd, Ro`	$Rd \leftarrow Rd\ OR\ Ro$	NZ · ·
XOR	`eor Rd, Rd, Ro`	$Rd \leftarrow Rd\ XOR\ Ro$	NZ · ·
NOT	`mvn Rd, Ro`	$Rd \leftarrow NOT(Ro)$	NZ · ·
Test	`tst Rn, Rm`	$Rn\ AND\ Rm$	NZ · ·
Desplazamiento			
Lógico	`lsl Rd, Ro, #Inm5`	$Rd \leftarrow Ro << Inm5$	NZC ·
a la izquierda	`lsl Rd, Rd, Rs`	$Rd \leftarrow Rd << [Rs]_{7:0}$	NZC ·
Lógico	`lsr Rd, Ro, #Inm5`	$Rd \leftarrow Ro >>_l Inm5$	NZC ·
a la derecha	`lsr Rd, Rd, Rs`	$Rd \leftarrow Rd >>_l [Rs]_{7:0}$	NZC ·
Aritmético	`asr Rd, Ro, #Inm5`	$Rd \leftarrow Ro >>_a Inm5$	NZC ·
a la derecha	`asr Rd, Rd, Rs`	$Rd \leftarrow Rd >>_a [Rs]_{7:0}$	NZC ·
Rotación a la derecha	`ror Rd, Rd, Rs`	$Rd \leftarrow Rd\ ROR\ [Rs]_{7:0}$	NZC ·

Cuadro B.6. Instrucciones de carga y almacenamiento.

Cargar		
Ind. Desp., palabra	`ldr Rd, [Ri, #Inm7]`	$Rd \leftarrow [Rn + Inm7]$
– media palabra	`ldrh Rd, [Ri, #Inm6]`	$Rd \leftarrow Ext0([Rn + Inm6]_{15:0})$
– byte	`ldrb Rd, [Ri, #Inm5]`	$Rd \leftarrow Ext0([Rn + Inm5]_{7:0})$
Ind. Reg., palabra	`ldr Rd, [Ri, Rm]`	$Rd \leftarrow [Rn + Rm]$
– media pal.	`ldrh Rd, [Ri, Rm]`	$Rd \leftarrow Ext0([Rn + Rm]_{15:0})$
– media pal., signo	`ldsh Rd, [Ri, Rm]`	$Rd \leftarrow ExtS([Rn + Rm]_{15:0})$
– byte	`ldrb Rd, [Ri, Rm]`	$Rd \leftarrow Ext0([Rn + Rm]_{7:0})$
– byte, signo	`ldsb Rd, [Ri, Rm]`	$Rd \leftarrow ExtS([Rn + Rm]_{7:0})$
Relativo a PC	`ldr Rd, [PC, #Inm10]`	$Rd \leftarrow [PC + Inm10]$
– a SP	`ldr Rd, [SP, #Inm10]`	$Rd \leftarrow [SP + Inm10]$
Almacenar		
Ind. Desp., palabra	`str Ro, [Ri, #Inm7]`	$[Rn + Inm7] \leftarrow Ro$
– media palabra	`strh Ro, [Ri, #Inm6]`	$[Rn + Inm6]_{15:0} \leftarrow Ro_{15:0}$
– byte	`strb Ro, [Ri, #Inm5]`	$[Rn + Inm5]_{7:0} \leftarrow Ro_{7:0}$
Ind. Reg., palabra	`str Rd, [Ri, Rm]`	$[Rn + Rm] \leftarrow Ro$
– media pal.	`strh Ro, [Ri, Rm`	$[Rn + Rm]_{15:0} \leftarrow Ro_{15:0}$
– byte	`strb Ro, [Ri, Rm]`	$[Rn + Rm]_{7:0} \leftarrow Ro_{7:0}$
Relativo a SP	`str Ro, [SP, #Inm10]`	$[SP + Inm10] \leftarrow Ro$

Ext0: relleno con ceros a la izquierda; ExtS: relleno con el signo a la izquierda.

Cuadro B.7. Instrucciones de manejo de la pila.

Gestión de la pila		
Apilar	`push {Regs}`	SP ← SP − Tam(Regs) [SP] ← Regs
Enlazar y apilar	`push {Regs, lr}`	SP ← SP − Tam(Regs, LR) [SP] ← Regs,LR
Desapilar	`pop {Regs}`	Regs ← [SP] SP ← SP + Tam(Regs)
Desapilar y retorno	`pop {Regs, pc}`	Regs, PC ← [SP] SP ← SP + Tam(Regs, PC)

Cuadro B.8. Instrucciones de salto. Los códigos de condición `cond` de la instrucción de salto condicional son los recogidos en el cuadro B.1.

Saltos			Rango
Incondicional	`b eti`	PC ← eti	± 2048 B
Condicional	`b{xx} eti`	Si {xx}, PC ← eti	± 256 B
A subprograma	`bl eti`	LR ← dir. sig. instrucción PC ← eti	± 4 MB
E intercambiar	`bx Ra`	PC ← (Ra AND `0xFFFFFFFE`)^{i}	4 GB

i: Si Ra_0 = 0, cambio de estado T32/Thumb a estado ARM. Ra contiene una dirección absoluta.

B.5 Selección de directivas del ensamblador

En este trabajo seguimos la notación y los convenios del ensamblador de GNU. Todas las directivas tienen nombres que comienzan con un punto ('.') y el resto de la directiva está compuesto por letras minúsculas.

El cuadro B.9 recoge las directivas utilizadas en este manual.

Cuadro B.9. Directivas del ensamblador Thumb.

Directivas del ensamblador	
`.align N`	Siguiente dato empieza en dir. múltiplo de 2^N.
`.ascii “cadena”`	Inicializa una zona de memoria con los caracteres UTF-8 de `cadena`.
`.asciz “cadena”`	Inicializa una zona de memoria con los caracteres UTF-8 de `cadena`, termina con 0.
`.balign N`	Siguiente dato empieza en dir. múltiplo de N.
`.byte valor`	Inicializa un byte a `valor`.
`.text`	Ensambla lo que sigue en la zona de código.
`.data`	Ensambla lo que sigue en la zona de datos.
`.end`	No hay más instrucciones.
`.equ símbolo, expr`	Asigna el valor de `expr` a `símbolo`.
`.equiv símbolo, expr`	Como `.equ`, pero da error si existe `símbolo`.
`.eqv`	Equivalente a `.equiv`.
`.hword valor`	Inicializa una media palabra a `valor`.
`.quad valor`	Inicializa una doble palabra a `valor`.
`símbolo .req rd`	Define `símbolo` como un alias para `rd`.
`.set`	Equivalente a `.equ`.
`.space N`	Reserva N bytes de memoria a 0.
`símbolo .unreq simbolo`	Cancela el alias `símbolo`.
`.word valor`	inicializa una palabra a `valor`.

Apéndice C
Tablas UTF-8

UTF-8 (8-bit Unicode Transformation Format) es un formato de codificación de caracteres que utiliza secuencias de longitud variable. Se corresponde con el estándar RFC 3629 de la Internet Engineering Task Force (IETF) y entre sus ventajas está la de incluir directamente los caracteres ASCII tradicionales de 7 bits, por lo que cualquier mensaje ASCII se representa sin cambios.

UTF-8 también sirve para codificar con 2, 3 o 4 bits la práctica totalidad de símbolos de uso común, como los caracteres de cualquier alfabeto (latino, cirílico, chino, japonés, coreano, etc.) o los símbolos matemáticos. En el caso concreto del castellano y de otras lenguas romances, los caracteres acentuados (á, é, ó, etc.) y los que utilizan otros símbolos diacríticos (ç, ñ, ü, etc.) se codifican con dos bytes.

A continuación, presentamos como referencia las tablas de caracteres UTF-8 correspondientes a los puntos Unicode `U+0000` – `U+00FF`. La codificación de estos 256 primeros caracteres Unicode en UTF-8 se lleva a cabo de acuerdo con las siguientes reglas:

Rango Unicode	UTF-8	Comentarios
`U+0000-U+007F`	`xxxxxxx`	Caracteres ASCII de 7 bits.
`U+0080-U+00FF`	`110yyyyy 10xxxxxx`	Caracteres acentuados y otros símbolos de uso común.

Los caracteres del primer grupo (cf. cuadros C.1 y C.2) se corresponden con los caracteres ASCII originales de 7 bits. Se trata pues de símbolos de un único byte cuyo bit más significativo es 0. Se agrupan a su vez de la siguiente manera:

Codificación	Grupo
`0b000X XXXX`	Caracteres de control.
`0b001X XXXX`	Dígitos y signos de puntuación.
`0b010X XXXX`	Mayúsculas y caracteres especiales.
`0b011X XXXX`	Minúsculas y caracteres especiales.

Los caracteres del segundo grupo (cf. cuadros C.3 y C.4) son caracteres que solían definirse como caracteres ASCII extendidos, codificados en un byte con el bit más significativo a 1 (i.e., valores de `0x80` a `0xFF`) y que, como vimos, se corresponden con los puntos unicode entre `U+0080` y `U+00FF`. La codificación UTF-8 en 2 bytes se realiza de la siguiente manera:

`0b0000 0000 XXXX YYYY` → `0b1100 00XX 10XX YYYY`

Por ejemplo, la letra "ñ" cuyo punto Unicode es `U+00F1` se codificaría como (cf. cuadro C.4 en la página 196):

`0b0000 0000 1111 0001` → `0b1100 0011 1011 0001`

Es decir, como `0xC3B1`. La razón de esta transformación es para garantizar otra de las propiedades del sistema de codificación UTF-8 llamada *no superposición*, que consiste en que los conjuntos de valores que puede tomar cada byte de un carácter multibyte son disjuntos, por lo que no es posible confundirlos entre sí. En otras palabras, no es posible por ejemplo confundir ningún símbolo UTF-8 de un byte con el primer o el segundo byte de un símbolo de dos bytes. Además, en una transmisión de símbolos UTF-8 es posible determinar el inicio de cada símbolo sin reiniciar la lectura desde el principio de la transmisión.

En las tablas C.3 y C.4 se sustituye el valor en decimal por el valor del punto Unicode en hexadecimal, ya que en este caso este dato es mucho más informativo que dicho valor en decimal.

Finalmente, los cuadros C.5 y C.6 recogen los caracteres alfabéticos utilizados en castellano, gallego y portugués.

Cuadro C.1. Caracteres UTF-8 de `0x00` a `0x3F`. Caracteres de control, dígitos y signos de puntuación. La codificación CP437 utilizada por el PC de IBM original, utilizaba los caracteres de control de (soh) a (us) para definir los caracteres imprimibles de la tabla. El carácter correspondiente a (nul) no formaba parte del CP437 original.

Dec	Hex	Sim	Ctl	Dec	Hex	Char
0	0x00	␀	(nul)	32	0x20	␣
1	0x01	☺	(soh)	33	0x21	!
2	0x02	☻	(stx)	34	0x22	"
3	0x03	♥	(etx)	35	0x23	#
4	0x04	♦	(eot)	36	0x24	$
5	0x05	♣	(enq)	37	0x25	%
6	0x06	♠	(ack)	38	0x26	&
7	0x07	•	(bel)	39	0x27	'
8	0x08	◘	(bs)	40	0x28	(
9	0x09		(tab)	41	0x29	)
10	0x0A	◙	(lf)	42	0x2A	*
11	0x0B	♂	(vt)	43	0x2B	+
12	0x0C		(ff)	44	0x2C	,
13	0x0D	♪	(cr)	45	0x2D	-
14	0x0E	♫	(so)	46	0x2E	.
15	0x0F	☼	(si)	47	0x2F	/
16	0x10	►	(dle)	48	0x30	0
17	0x11	◄	(dc1)	49	0x31	1
18	0x12	↕	(dc2)	50	0x32	2
19	0x13	‼	(dc3)	51	0x33	3
20	0x14	¶	(dc4)	52	0x34	4
21	0x15	§	(nak)	53	0x35	5
22	0x16	▬	(syn)	54	0x36	6
23	0x17	↨	(etb)	55	0x37	7
24	0x18	↑	(can)	56	0x38	8
25	0x19	↓	(em)	57	0x39	9
26	0x1A		(eof)	58	0x3A	:
27	0x1B	←	(esc)	59	0x3B	;
28	0x1C	∟	(fs)	60	0x3C	<
29	0x1D	↔	(gs)	61	0x3D	=
30	0x1E	▲	(rs)	62	0x3E	>
31	0x1F	▼	(us)	63	0x3F	?

Cuadro C.2. Caracteres UTF-8 de `0x40` a `0x7F`. Mayúsculas, minúsculas y caracteres especiales.

Dec	Hex	Char	Dec	Hex	Char
64	0x40	@	96	0x60	‘
65	0x41	A	97	0x61	a
66	0x42	B	98	0x62	b
67	0x43	C	99	0x63	c
68	0x44	D	100	0x64	d
69	0x45	E	101	0x65	e
70	0x46	F	102	0x66	f
71	0x47	G	103	0x67	g
72	0x48	H	104	0x68	h
73	0x49	I	105	0x69	i
74	0x4A	J	106	0x6A	j
75	0x4B	K	107	0x6B	k
76	0x4C	L	108	0x6C	l
77	0x4D	M	109	0x6D	m
78	0x4E	N	110	0x6E	n
79	0x4F	O	111	0x6F	o
80	0x50	P	112	0x70	p
81	0x51	Q	113	0x71	q
82	0x52	R	114	0x72	r
83	0x53	S	115	0x73	s
84	0x54	T	116	0x74	t
85	0x55	U	117	0x75	u
86	0x56	V	118	0x76	v
87	0x57	W	119	0x77	w
88	0x58	X	120	0x78	x
89	0x59	Y	121	0x79	y
90	0x5A	Z	122	0x7A	z
91	0x5B	[	123	0x7B	{
92	0x5C	\	124	0x7C	|
93	0x5D	]	125	0x7D	}
94	0x5E	ˆ	126	0x7E	˜
95	0x5F	_	127	0x7F	⌂

Cuadro C.3. Caracteres UTF-8 de `0xC280` a `0xC2BF`. Símbolos varios. Se corresponden con el segundo bloque Unicode, también conocido como Controles C1 y Suplemento Latin-1. Los primeros 32 caracteres son caracteres de control no imprimibles. La columna Dif representa las diferencias entre ISO Latin 1 (columna Sim) e ISO Latin 9.

Uni	UTF-8	Ctl	Uni	UTF-8	Sim	Dif
0x80	0xC280	(pad)	0xA0	0xC2A0	NBSP	
0x81	0xC281	(hop)	0xA1	0xC2A1	¡	
0x82	0xC282	(bph)	0xA2	0xC2A2	¢	
0x83	0xC283	(nbh)	0xA3	0xC2A3	£	
0x84	0xC284	(ind)	0xA4	0xC2A4	¤	€
0x85	0xC285	(nel)	0xA5	0xC2A5	¥	
0x86	0xC286	(ssa)	0xA6	0xC2A6	¦	Š
0x87	0xC287	(esa)	0xA7	0xC2A7	§	
0x88	0xC288	(hts	0xA8	0xC2A8	¨	š
0x89	0xC289	(htj)	0xA9	0xC2A9	©	
0x8A	0xC28A	(lts)	0xAA	0xC2AA	ª	
0x8B	0xC28B	(pld)	0xAB	0xC2AB	«	
0x8C	0xC28C	(plu)	0xAC	0xC2AC	¬	
0x8D	0xC28D	(ri)	0xAD	0xC2AD		
0x8E	0xC28E	(ss2)	0xAE	0xC2AE	®	
0x8F	0xC28F	(ss3)	0xAF	0xC2AF	¯	
0x90	0xC290	(dcs)	0xB0	0xC2B0	°	
0x91	0xC291	(pu1)	0xB1	0xC2B1	±	
0x92	0xC292	(pu2)	0xB2	0xC2B2	²	
0x93	0xC293	(sts)	0xB3	0xC2B3	³	
0x94	0xC294	(cch)	0xB4	0xC2B4	´	Ž
0x95	0xC295	(mw)	0xB5	0xC2B5	µ	
0x96	0xC296	(spa)	0xB6	0xC2B6	¶	
0x97	0xC297	(epa)	0xB7	0xC2B7	·	
0x98	0xC298	(sos)	0xB8	0xC2B8	¸	ž
0x99	0xC299	(sgci)	0xB9	0xC2B9	¹	
0x9A	0xC29A	(sci)	0xBA	0xC2BA	º	
0x9B	0xC29B	(csi)	0xBB	0xC2BB	»	
0x9C	0xC29C	(st)	0xBC	0xC2BC	¼	Œ
0x9D	0xC29D	(osc)	0xBD	0xC2BD	½	œ
0x9E	0xC29E	(pm)	0xBE	0xC2BE	¾	Ÿ
0x9F	0xC29F	(apc)	0xBF	0xC2BF	¿	

Cuadro C.4. Caracteres UTF-8 de `0xC380` a `0xC3BF`. Caracteres acentuados. Se corresponden con el segundo bloque Unicode, también conocido como Suplemento Latin-1. Podemos observar que las mayúsculas y minúsculas se diferencian en un solo bit, el mismo que en el caso de los caracteres ASCII (bloque `0x40` a `0x7F`).

Uni	UTF-8	Char	Uni	UTF-8	Char
0xC0	0xC380	À	0xE0	0xC3A0	à
0xC1	0xC381	Á	0xE1	0xC3A1	á
0xC2	0xC382	Â	0xE2	0xC3A2	â
0xC3	0xC383	Ã	0xE3	0xC3A3	ã
0xC4	0xC384	Ä	0xE4	0xC3A4	ä
0xC5	0xC385	Å	0xE5	0xC3A5	å
0xC6	0xC386	Æ	0xE6	0xC3A6	æ
0xC7	0xC387	Ç	0xE7	0xC3A7	ç
0xC8	0xC388	È	0xE8	0xC3A8	è
0xC9	0xC389	É	0xE9	0xC3A9	é
0xCA	0xC38A	Ê	0xEA	0xC3AA	ê
0xCB	0xC38B	Ë	0xEB	0xC3AB	ë
0xCC	0xC38C	Ì	0xEC	0xC3AC	ì
0xCD	0xC38D	Í	0xED	0xC3AD	í
0xCE	0xC38E	Î	0xEE	0xC3AE	î
0xCF	0xC38F	Ï	0xEF	0xC3AF	ï
0xD0	0xC390	Ð	0xF0	0xC3B0	ð
0xD1	0xC391	Ñ	0xF1	0xC3B1	ñ
0xD2	0xC392	Ò	0xF2	0xC3B2	ò
0xD3	0xC393	Ó	0xF3	0xC3B3	ó
0xD4	0xC394	Ô	0xF4	0xC3B4	ô
0xD5	0xC395	Õ	0xF5	0xC3B5	õ
0xD6	0xC396	Ö	0xF6	0xC3B6	ö
0xD7	0xC397	×	0xF7	0xC3B7	÷
0xD8	0xC398	Ø	0xF8	0xC3B8	ø
0xD9	0xC399	Ù	0xF9	0xC3B9	ù
0xDA	0xC39A	Ú	0xFA	0xC3BA	ú
0xDB	0xC39B	Û	0xFB	0xC3BB	û
0xDC	0xC39C	Ü	0xFC	0xC3BC	ü
0xDD	0xC39D	Ý	0xFD	0xC3BD	ý
0xDE	0xC39E	Þ	0xFE	0xC3BE	þ
0xDF	0xC39F	ß	0xFF	0xC3BF	ÿ

Cuadro C.5. Caracteres UTF-8 alfabéticos utilizados en castellano, gallego y portugués (I).

Dec	Hex	Char	Dec	Hex	Char
65	0x41	A	97	0x61	a
66	0x42	B	98	0x62	b
67	0x43	C	99	0x63	c
68	0x44	D	100	0x64	d
69	0x45	E	101	0x65	e
70	0x46	F	102	0x66	f
71	0x47	G	103	0x67	g
72	0x48	H	104	0x68	h
73	0x49	I	105	0x69	i
74	0x4A	J	106	0x6A	j
75	0x4B	K	107	0x6B	k
76	0x4C	L	108	0x6C	l
77	0x4D	M	109	0x6D	m
78	0x4E	N	110	0x6E	n
79	0x4F	O	111	0x6F	o
80	0x50	P	112	0x70	p
81	0x51	Q	113	0x71	q
82	0x52	R	114	0x72	r
83	0x53	S	115	0x73	s
84	0x54	T	116	0x74	t
85	0x55	U	117	0x75	u
86	0x56	V	118	0x76	v
87	0x57	W	119	0x77	w
88	0x58	X	120	0x78	x
89	0x59	Y	121	0x79	y
90	0x5A	Z	122	0x7A	z

Cuadro C.6. Caracteres UTF-8 alfabéticos utilizados en castellano, gallego y portugués (II).

Uni	UTF-8	Char	Uni	UTF-8	Char
0xC0	0xC380	À	0xE0	0xC3A0	à
0xC1	0xC381	Á	0xE1	0xC3A1	á
0xC2	0xC382	Â	0xE2	0xC3A2	â
0xC3	0xC383	Ã	0xE3	0xC3A3	ã
0xC7	0xC387	Ç	0xE7	0xC3A7	ç
0xC9	0xC389	É	0xE9	0xC3A9	é
0xCA	0xC38A	Ê	0xEA	0xC3AA	ê
0xCD	0xC38D	Í	0xED	0xC3AD	í
0xD1	0xC391	Ñ	0xF1	0xC3B1	ñ
0xD3	0xC393	Ó	0xF3	0xC3B3	ó
0xD4	0xC394	Ô	0xF4	0xC3B4	ô
0xD5	0xC395	Õ	0xF5	0xC3B5	õ
0xDA	0xC39A	Ú	0xFA	0xC3BA	ú
0xDC	0xC39C	Ü	0xFC	0xC3BC	ü

Índice de materias

`.align`, 30, 175
`.balign`, 30, 175
`_start`, 20
`as`, 22
`blx`, 19
`bx`, 19
`gcc`, 22
`gdb`, 22
    comandos útiles, 23
`ldmia`, 186
`main`, 20
`stmia`, 186
`swi`, 186
`wfi`, 16, 186

A32, 19, 177
AAPCS, 20, 107, 141
Arch Linux ARM, 18
ARM7TDMI, 186
ASCII, 28, 81, 95, 115, 119, 174
    tablas, 192
atajos de teclado, 12

BCD, 103
bit T32/Thumb, 19
breakpoint, 16
Broadcom, 17
Broadcom BCM2711, 18

capicúa, 67
característica, 93
comparar cadenas, 87, 110
contador de programa, 177
contar
    bits a 1, 100
    caracteres, 65, 69, 146, 155
    ceros, 83
    dígitos, 146
    números, 66
convertir
    a cadena, 119
    a mayúsculas, 71, 72
    caracteres, 113, 117
    números, 89, 115
    palabras, 95
Cortex-A72, 18
Cortex-M, 16
códigos de condición, 179

desensamblado, 12
directivas, 190
divide y vencerás, 100
división, 79
    por 10, 98, 119

ejecución paso a paso, 16
endianness, 27, 171

little endian, 27, 171, 179
ensamblador, 12, 22
directivas, 190
estructuras
`for`, 38
`if - then - else`, 34
`if - then`, 32
`switch`, 159
`while`, 36
exponente, 93
extremista menor, 27, 171

factorial, 76, 163

GEF, 23
GNU toolchain, 11, 18

indicadores de condición, 178
instrucciones
almacenamiento, 188
aritméticas, 187
carga, 188
desplazamientos, 188
directivas, 190
lógicas, 188
manejo de la pila, 189
saltos, 189
intercambio de zonas de memoria, 41, 42, 44, 46
invertir cadena, 121
ISO Latin 1, 113
ISO Latin 9, 28, 174

little endian, 27, 171, 179
longitud de una cadena, 110
look-up table, 76

manejo de la pila, 121, 122, 128
mantisa, 93
matriz, 84, 85, 148, 153
modos de direccionamiento, 180
directo a registro, 180
indirecto con desplazamiento, 182
indirecto con registro de desplazamiento, 185
inmediato, 181
relativo a PC, 183
relativo a SP, 185
multiplicar, 91
máquina de pila, 122, 128

nibble, 82, 89, 103
normalizar, 93

octal, 81

palíndromo, 74
paridad, 117
Pidora, 18
procesar
cadenas, 65–67, 69, 71, 72, 74, 110, 121, 146, 155
tabla de enteros, 49, 51, 53, 57, 58, 60, 61, 63
pseudoinstrucción, 15, 184
puntero de pila, 177
punteros, 55
punto de ruptura, 16
punto flotante, 93

QtARMSim, 12
atajos de teclado, 13

Raspberry Pi, 17, 169
Raspbian, 18
recursividad, 141, 163, 165
registro de enlace, 177
registro de estado, 178
registros, 177
RISC OS 5, 18

rotación de una zona de memoria, 108

salto lejano, 184
SBC, 17
SoC, 17
step over, 16
subprograma, 107, 141
    anidadamiento, 20, 142
sumar enteros, 49, 51, 53, 55, 78, 103

T32, 2, 17, 19, 177
tabla de saltos, 159, 160
Thumb, 177
torres de Hanoi, 165

UTF-8, 28, 69, 72, 113, 174, 191
    tablas, 192

Miscelánea

Serie de textos misceláneos

Últimas publicacions na colección

A transformación feminina da realidade. Unha cuestión de (des)memoria (2020)
Xulio Pardo de Neyra

Libro-Obxecto e Xénero: Estudos ao redor do libro infantil como artefacto (2019)
Isabel Mociño Gónzalez

VHDL sintetizable para estudiantes de Ingeniería (2018)
Carlos Castro Miguens e José B. Castro Miguens

Calidad de la energía eléctrica (2016)
Manuel Pérez Donsión

El camino de los sentidos (2014)
Inmaculada Báez Montero, Eva Freijeiro Ocampo e María Suárez Rodríguez

Arquitectura de ordenadores

Ejercicios prácticos de ARM/Thumb

La arquitectura ARM, desarrollada hace más de 35 años, está presente en la actualidad en miles de millones de dispositivos, desde Fugaku, el superordenador japonés líder del ranking de los ordenadores más potentes del mundo en 2020, hasta el Catweazle Mini, un femtocomputador de poco más de 0,25 centímetros cúbicos de volumen. Thumb es una versión reducida de ARM en la que la mayoría de las instrucciones ocupan 16 bits, y está orientada fundamentalmente al desarrollo de dispositivos embebidos, como por ejemplo los dispositivos para controlar un ascensor, un electrodoméstico, determinadas funciones de un automóvil, y en general cualquier dispositivo del ámbito del Internet de las Cosas.

El presente libro está basado en las clases prácticas y exámenes prácticos que los autores han ido preparando a lo largo de más de veinte años de experiencia impartiendo las asignaturas de arquitectura de ordenadores en los diversos planes de estudio de la Escuela de Ingeniería de Telecomunicación de la Universidade de Vigo.

Servizo de Publicacións

UniversidadeVigo