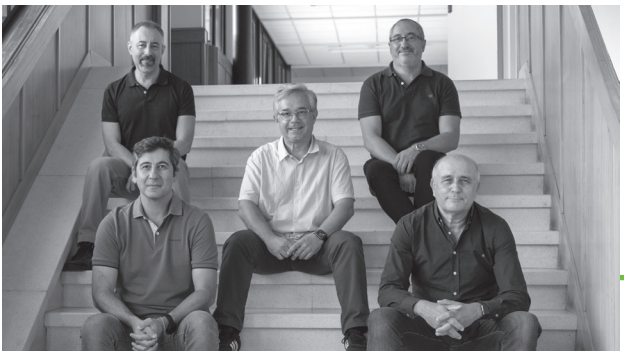
Manual de Programación en Ensamblador

Unha achega teórico-práctica

Manuais
Serie manuais didácticos

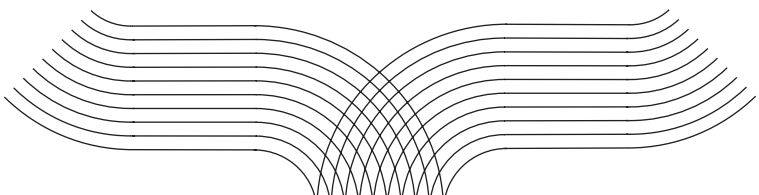
Manuel José Fernández Iglesias
Martín Llamas Nistal
Luis Eulogio Anido Rifón
Juan Manuel Santos Gago
Fernando Ariel Mikic Fonte

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 e Fernando Mikic son profesores do departamento de enxeñería telemática da Universidade de Vigo. Con diversas dedicacións ao longo dos anos -e as veces coa axuda doutro profesorado do departamento- encargáronse da posta en marcha das materías centradas na introdución ás arquitecturas dos ordenadores nos diversos planos de estudo da Escola de Enxeñería de Telecomunicación de Vigo dende a súa creación en 1985. Todos eles pertencen ao grupo de investigación en enxeñería de sistemas telemáticos e, ademáis das súas tarefas docentes, tamén comparten proxectos de investigación no eido do e-learning.

Servizo de Publicacións
Universidade Vigo



Manuais

Serie de manuais didácticos

n.º 078

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

Deseño da portada
Tania Sueiro Graña
Área de Imaxe
Vicerreitoría de Comunicacións e Relacións Institucionais

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

Fotografía da portada
Adobe Stock

Impresión
Tórculo Comunicación Gráfica, S. A.

ISBN (Libro impreso)
978-84-8158-908-5

Depósito legal
VG 500-2021

Servizo de Publicacións
Universidade Vigo

Manual de Programación en Ensamblador

Unha achega teórico-práctica

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

Prefacio

No mundo da programación, hoxe en día predominan de maneira clara as linguaxes de programación de alto nivel como JavaScript, Java, C# ou Python entre outras moitas. Devanditas linguaxes utilízanse para desenvolver a inmensa maioría das aplicacións e sistemas software que coñecemos: apps para dispositivos móbiles, skills para axentes intelixentes, programas de edición de textos, xogos de computador, portais Web ... probablemente todo o software que vaiamos utilizar un día calquera está escrito utilizando unha linguaxe de alto nivel. Isto é así porque as linguaxes de alto nivel abstraen os detalles particulares de cada dispositivo hardware, proporcionando un modelo común para programar calquera deles.

Por outra banda, a linguaxe ensambladora é unha linguaxe de programación de baixo nivel, é dicir, é unha linguaxe de programación que permite relacionarnos directamente cos dispositivos hardware do computador. Existe unha relación directa entre unha linguaxe ensambladora e as instrucións en código máquina dunha determinada arquitectura de computador. Por tanto, xa que as diferentes arquitecturas de computador caracterízanse entre outras cousas polo seu repertorio de instrucións en código máquina, cada unha de elas terá a súa propia linguaxe ensambladora. Para poder programar en ensamblador dispositivos de diferentes arquitecturas, teremos que aprender as diferentes linguaxes ensambladoras.

De todos os xeitos, e a pesar da prevalencia das linguaxes de programación de alto nivel, non podemos subestimar a importancia do ensamblador. Como diciamos antes, o ensamblador permítenos manipular o hardware directamente, e ademais acceder a toda a riqueza expresiva da linguaxe máquina, e con iso acometer proxectos que pola súa dependencia directa do hardware serían dificilmente realizables cunha linguaxe de alto nivel, como a programación de manexadores de dispositivos (*drivers*), sistemas en tempo real, núcleos de sistemas operativos, ou o código de arranque de sistemas embebidos.

Ademais, adquirir destreza cunha linguaxe ensambladora permítenos coñecer mellor como está organizado e como funciona un procesador, o que pode resultarnos beneficioso mesmo cando escribimos programas nunha linguaxe de alto nivel. Por exemplo, se estamos a escribir un programa que necesita utilizar os dispositivos dun computador de maneira eficiente, comprender en detalle como funciona o procesador e como este comunícase con eles a través dos portos de entrada e saída, pode ser moi vantaxoso. En moitos casos unha boa opción é escribir parte do código en ensamblador e ver como se comporta directamente o procesador, a memoria, e os portos de entrada e saída no noso caso particular e, unha vez que comprobamos que a nosa solución é axeitada, podemos resolver o noso problema cunha linguaxe de alto nivel, ou mesmo embeber o noso código en ensamblador dentro do programa de alto nivel.

En definitiva, a ensambladora é a única linguaxe coa que nos podemos comunicar directamente coa máquina, sen tradutores ou intermediarios. É a linguaxe que recoñece un determinado microprocesador, e cada arquitectura de microprocesador recoñece o seu propio ensamblador.

Con todo, os microprocesadores realizan tarefas individuais moi sinxelas, como transferir valores de e cara á memoria, realizar operacións aritméticas, ou realizar saltos ou bifurcacións a outras zonas do programa en función do resultado dunha operación realizada anteriormente. Polo xeral, cada unha desas tarefas sinxelas tradúcese nunha instrución en código máquina, que á súa vez correspóndese cunha instrución en ensamblador. En consecuencia, a linguaxe ensambladora e os programas escritos con ela adoitan ser fáciles de comprender.

Calquera manual práctico sobre programación en ensamblador precisa dunha arquitectura de referencia, dunha linguaxe ensambladora concreta no que realizar os programas, e dunhas ferramentas para pór en práctica os coñecementos adquiridos. Como veremos no primeiro capítulo deste libro, utilizaremos a primeira versión do repertorio de instrucións T32/Thumb da arquitectura ARM, que inclúe unicamente instrucións de 16 bits, e o ensamblador para ARM de GNU. En canto ás ferramentas, neste manual utilizaremos dúas opcións amplamente validadas en contornas educativas. Tamén utilizaremos nos enunciados dalgúns exercicios a sintaxe da linguaxe de programación C para describir os algoritmos utilizados.

ARM é na actualidade a arquitectura de microprocesador máis popular. Esta arquitectura, introducida hai máis de 35 anos, na década dos 80 do século pasado, está presente en miles de millóns de dispositivos, desde Fuga-

ku (Sato y col., 2020), o supercomputador xaponés líder da clasificación dos computadores máis potentes do mundo en 2021, con preto de 160.000 unidades do chip A64FX de 48 núcleos, até o Kinetis KL02 (NXP Semiconductors, 2017), un femtocomputador de 1,9 x 2 x 0,5 mm baseado no núcleo ARM Cortex-M0+, con 32kB de memoria flash e 4 KB de RAM, pensado para produtos que se poidan tragar. Coñecidos fabricantes como Apple, Broadcom, NVidia, Qualcomm ou Samsung basean os seus dispositivos máis populares nesta arquitectura e os dispositivos ARM están presentes en sectores como a automoción, os vehículos espaciais, o fogar intelixente, os dispositivos vestibles (*wearable devices*), a telefonía móbil, a Internet das Cousas (*Internet of Things*, IoT), ou as cidades intelixentes .

Thumb, coñecido como T32 a partir da versión 8 da arquitectura ARM, é un repertorio de instrucións e un estado de execución dos microprocesadores que seguen a arquitectura ARM que, aínda que se definiu inicialmente a partir das instrucións máis utilizadas, codificadas con 16 bits no canto de 32 bits, na actualidade inclúe as instrucións orixinais, así como instrucións adicionais de 16 e 32 bits. Está orientado fundamentalmente ao desenvolvemento de sistemas embebidos, como por exemplo os dispositivos para o control de sistemas de videovixilancia, routers, automóbiles, cámaras dixitais, electrodomésticos, e en xeral calquera dispositivo do ámbito IoT. Ao ocupar as instrucións máis utilizadas 16 bits no canto de 32, esta versión reducida do repertorio de instrucións permite incrementar de maneira relevante a densidade de código, co que é posible realizar aplicacións en dispositivos con menores requisitos de memoria, e por tanto cun menor custo de fabricación. Todos os núcleos da arquitectura ARM soportan en maior ou menor medida o repertorio T32, e mesmo hai micros ARM que só soportan instrucións deste repertorio.

O presente manual está baseado na experiencia dos autores de máis de vinte anos impartindo as materias de arquitectura de computadores na Escola de Enxeñaría de Telecomunicación da Universidade de Vigo, e é herdeiro dunha obra anterior en castelá que serviu de base para o presente texto (Fernández-Iglesias y col., 2020). Ademais de diversas correccións e aclaracións realizadas ao texto orixinal, proponse exercicios adicionais e engádese un novo capítulo dedicado á comunicación coa periferia do computador.

Deste xeito, este manual organízase en cinco capítulos. O primeiro deles presenta, a modo de introdución, algunhas das alternativas dispoñibles para pór en práctica os diferentes exercicios de programación en ensamblador propostos neste manual, mentres que o resto desenvolve un conxunto de su-

postos de programación en ensamblador de complexidade crecente, elixidos polo seu valor pedagóxico.

O capítulo 2 propón problemas orientados á iniciación na programación en ensamblador, ilustrando a programación das estruturas básicas de control e a trasfega de información utilizando os rexistros e a memoria principal nunha arquitectura *load/store* como a arquitectura ARM.

O seguinte capítulo está dedicado ao procesado de información en ensamblador. Comezamos con tarefas básicas como contar, sumar e comparar números, para presentar a continuación o procesado de cadeas de caracteres. Máis adiante, propomos un conxunto de problemas centrados na utilización combinada do repertorio de instrucións aritméticas, lóxicas e de desprazamento.

Máis adiante, o capítulo 4 introduce os conceptos de subprograma e de paso de parámetros. En primeiro lugar, trátase o paso de parámetros a subprogramas mediante rexistros. A continuación, presentamos o concepto de pila e o seu manexo, para logo recuperar o concepto de subprograma, esta vez para introducir o paso de parámetros a través da pila, o aniñamento de chamadas a subprograma e a recursividade.

Finalmente, o novo capítulo 5 dedícase á comunicación do ordenador coa súa contorna, é dicir, á resolución de problemas de programación de entrada e saída, ben directamente ou a través de outros módulos software ou do sistema operativo.

O libro complétase con catro apéndices. O primeiro de eles recolle de novo os exercicios introdutorios propostos no capítulo 1, neste caso para a Raspberry Pi. A continuación, o apéndice B describe o convenio para as chamadas ao sistema na arquitectura ARM, convenio que tamén é de aplicación ás funcións ou subprogramas dispoñibles nas bibliotecas do sistema operativo e en xeral ás chamadas a subprograma, e que tamén seguimos neste libro. Inclúese tamén no apéndice C, a modo de referencia rápida, un resumo do modelo do programador Thumb, onde se recolle unha descrición dos rexistros, dos indicadores de estado, da organización da memoria, e finalmente unhas táboas coas instrucións utilizadas no libro. O manual complétase cun apéndice sobre os xogos de caracteres UTF-8 e ISO Latin 9, que inclúe á súa vez información sobre o xogo de caracteres ASCII e a maneira en que este intégrase no modelo UTF (apéndice D).

Finalmente, inclúese unha bibliografía básica sobre arquitectura de ordenadores onde aparecen todos os textos e manuais citados neste libro, xunto con outras referencias relevantes neste campo.

Todos os exercicios do libro están dispoñibles, xunto con outra documentación e referencias interesantes, na páxina Web `libroarm.webs.uvigo.gal`.

O texto foi revisado e corrixido, pero en todo caso os autores asumen toda a responsabilidade polos cazapos que puidesen escaparse deste proceso de revisión. Agradecemos aos nosos lectores e lectoras que nos informen de calquera erro que puidesen localizar para proceder a corrixilo en futuras edicións. Así mesmo, agradecemos todo tipo de comentarios, críticas e suxestións que poidan contribuír a mellorar tanto a presentación como o contido deste libro. Na páxina do libro en Internet é posible atopar os nosos datos de contacto.

Agradecementos

Como indicamos anteriormente, o contido deste manual xorde da experiencia docente dos autores, polo que queremos amosar o noso recoñecemento ao noso alumnado, fonte continua de aprendizaxe para nós. A súa presenza ao longo destes anos fíxonos evolucionar e contribuíu de maneira determinante a manter vivo o noso afán por mellorar. Con eles crecemos ao longo destes anos e grazas a eles mantemos viva nosa curiosidade. Queremos agradecer ademais os comentarios, suxestións e correccións recibidos sobre a obra en castelán que serviu de base para o presente traballo. Gracias a eles esta nova obra en galego pódese humildemente considerar unha obra mellor, máis traballada e máis útil.

Tamén queremos agradecer o seu apoio e os seus comentarios aos compañeiros que ao longo destes anos impartiron as materias de arquitectura de computadores connosco, especialmente a Alberto Gil Solla, Manuel Caeiro Rodríguez e Luís M. Álvarez Sabucedo, así como ao Servizo de Publicacións da Universidade de Vigo polas súas suxestións e consellos, e en definitiva por facer realidade este proxecto editorial en lingua galega.

Finalmente, e non por iso menos importante, queremos deixar patente o noso recoñecemento ás nosas familias polo seu apoio, a súa comprensión e a súa xenerosidade.

Índice xeral

Índice de figuras

Índice de cadros

Capítulo 1
Quentando motores

Calquera texto de exercicios de programación necesita dunha contorna onde pór en práctica os coñecementos adquiridos. Este primeiro capítulo ten como obxectivo presentar dúas alternativas concretas para programar en ensamblador utilizando o repertorio de instrucións Thumb de 16 bits. En ambos os casos eliximos solucións populares na contorna universitaria, que ademais demostraron a súa utilidade pedagóxica. Trátase do simulador QtARMSim e do sistema Raspberry Pi / Raspberry Pi OS / GNU toolchain. Estas dúas alternativas baséanse, respectivamente, en utilizar un simulador da máquina ou familia de máquinas que soportan a linguaxe ensambladora elixida (QtARMSim) e en utilizar unha máquina real cuxa arquitectura é compatible co ensamblador elixido (Raspberry Pi / Raspberry Pi OS / GNU toolchain).

Á hora de escribir programas en ensamblador, ambas as opcións utilizan o ensamblador de GNU. No caso concreto deste manual, utilizaremos o simulador QtARMSim nos capítulos 2 a 4, e a Raspberry Pi no capítulo 5 para ilustrar a comunicación dos nosos programas en ensamblador co exterior, con outros módulos software, e co sistema operativo. Eliximos QtARMSim como ferramenta concreta nos primeiros capítulos por ser a máis adecuada das dúas para o programador novel. Un simulador pedagóxico como QtARMSim abstrae a maior parte dos detalles da máquina subxacente, o que á súa vez permite centrarnos nos retos da programación. Por outra banda, a opción baseada na Raspberry Pi require coñecementos máis profundos sobre arquitectura de computadores e sistemas operativos, pero permitirá ao programador experimentado extraer todo o potencial dunha máquina real. Por esta razón foi a plataforma elixida no caso do capítulo 5.

Aproveitamos para lembrar que o apéndice C inclúe un resumo do modelo do programador Thumb, con todas as instrucións utilizadas nos exercicios e exemplos deste manual.

1.1 O simulador QtARMSim

QtARMSim (Mir y col., 2018) é un simulador gráfico da arquitectura ARM/Thumb sinxelo e fácil de usar que foi deseñado para cursos de introdución á arquitectura de computadores. Distribúese baixo a licenza GPL v3+. Existen versións para Microsoft Windows, macOS de Apple e Linux.

Unha vez instalado de acordo coas instrucións dispoñibles na páxina Web do proxecto QtARMSim, podemos executar o simulador como calquera programa do noso computador. Ao executar a aplicación, aparecerá unha xanela con catro zonas diferenciadas: (i) Rexistros; (ii) Editar/Desensamblar; (iii) Memoria, e (iv) Mensaxes/Envorcado de memoria/Pantalla LCD (cf. figura 1.1).

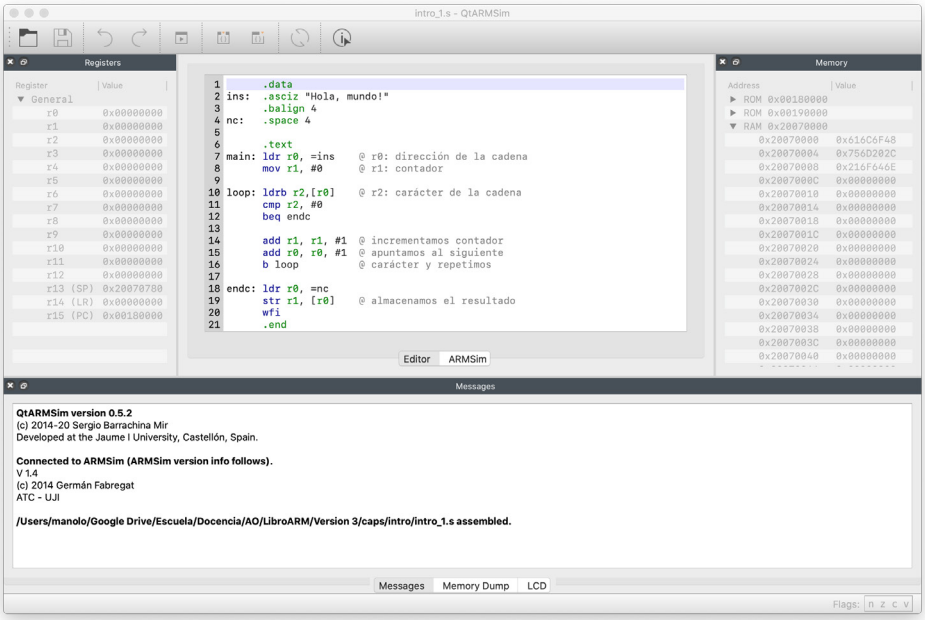


Figura 1.1. Xanela principal de QtARMSim.

Se seleccionamos a pestana Editor da parte inferior da área de edición, entraremos no editor. Nesa zona podemos escribir os nosos programas utilizando a linguaxe ensambladora de GNU para a arquitectura ARM[1].

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

Imos utilizar o programa da figura 1.2 como exemplo para ilustrar o funcionamento do simulador. Unha vez editado, gardariámolo nun arquivo utilizando a opción correspondente da barra de menús (💾) ou o atallo de teclado Ctrl + S (⌘ + S nun computador macOS). O cadro 1.1 presenta un resumo dos atallos de teclado de QtARMSim.

intro_1.s

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

        .text
main:   ldr r0, =ins    @ r0: dirección da cadea
        mov r1, #0      @ r1: contador

loop:   ldrb r2,[r0]    @ r2: carácter da cadea
        cmp r2, #0
        beq endc

        add r1, r1, #1  @ incrementamos contador
        add r0, r0, #1  @ apuntamos ao seguinte
        b loop          @ carácter e repetimos

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

Figura 1.2. Programa de exemplo.

Ao seleccionar a pestana ARMSim na parte inferior da zona de edición, o simulador tentará ensamblar automaticamente o programa. No caso de que haxa erros, notificaranse na área de Mensaxes que está debaixo da zona de edición. Se é así, volveriamos á pestana do Editor, corrixiriamos eses erros e tentariámolo novamente.

Unha vez que o programa estea libre de erros, ao seleccionar a pestana ARMSim ensamblarase automaticamente o código fonte e QtARMSim pasará ao modo de simulación. Neste modo, cada liña correspóndese cunha instrución do nivel de máquina convencional almacenada na memoria. A información que se mostra obtense mediante un proceso chamado desensamblado e consiste no seguinte (de esquerda a dereita):

- A dirección de memoria na que se almacena a instrución.

Cadro 1.1. Atallos de teclado de QtARMSim. No caso dos computadores Apple, hai que substituír a tecla Ctrl pola tecla ⌘.

F1	Axuda
Shift ⇑ + F1	Que é . . . ?
F3	Restablecer a configuración predeterminada
F4	Restablecer a simulación
F5	*Step into* (paso a paso)
F6	*Step over* (saltar subprograma)
Ctrl + F11	Executar
Ctrl + N	Novo arquivo
Ctrl + O	Abrir arquivo
Ctrl + S	Gardar arquivo
Ctrl + Q	Saír de QtARMSim
Alt + P	Preferencias

- A instrución en código máquina expresada en hexadecimal.
- A instrución desensamblada, é dicir, a instrución en linguaxe ensambladora, coa sintaxe ARM completa.
- A liña de texto orixinal do código fonte que produciu a instrución durante o proceso de ensamblado.

Por exemplo, a primeira liña do programa anterior (cf. figura 1.3):

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

indica que:

- A instrución está almacenada a partir da dirección de memoria `0x00180000`.
- A instrución en código máquina expresada en hexadecimal é `0x4805`.
- A instrución desensamblada é **`ldr`** `r0, [pc, #20]`. As instrucións neste campo preséntanse coa sintaxe ARM completa, e as pseudoinstrucións substitúense pola instrución correspondente.
- A instrución xerouse a partir da liña número 7 do código fonte orixinal, cuxo contido era:

```
ldr r0, =ins @r0: dirección da cadea
```

```
[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
[0x0018001A] 0x2007 movs r0, #7        ins = 0x20070000
[0x0018001C] 0x0010 movs r0, r2
```

Figura 1.3. Desensamblado do programa de exemplo. Obsérvese a tradución a unha instrución ARM da pseudoinstrución `ldr r0, =ins` e como se converte `=ins` ao modo de direccionamento relativo a PC, actualizando ao mesmo tempo unha palabra (4 posicións de memoria) despois da instrución `wfi` coa dirección efectiva.

Como acabamos de indicar, no caso de que se trate dunha pseudoinstrución, o código desensamblado mostra a instrución ARM que xera dita pseudoinstrución. No caso da pseudoinstrución **`ldr r0,=ins`**, podemos observar que se substitúe por unha instrución **`ldr`** con direccionamento relativo ao contador de programa e por unha palabra de memoria coa dirección efectiva depositada na proximidade da propia instrución, neste caso 20 posicións máis adiante de onde apunta o contador de programa (**`[pc, #20]`**), na palabra de dirección **`0x00180018`**. Dita palabra contén á súa vez a dirección a partir da cal está almacenada a cadea "Ola Mundo!" ao principio da zona de datos, etiquetada como **`ins`** (dirección **`0x20070000`**, figura 1.3).

Para executar o programa, seleccionamos a icona correspondente da barra de menús (▶), a opción do menú Run, ou o atallo de teclado Ctrl + F11 (⌘ + F11 nun Mac).

Cando o programa finalice, veremos que os rexistros **`r0`**, **`r1`**, **`r2`** e **`r15`** e a posición de memoria **`0x20070010`** teñen fondo azul e están en negriña. Isto débese a que o simulador resalta os rexistros e as posicións de memoria que se modificaron como consecuencia da execución do programa. **`r15`** é o contador de programa (PC), polo que se modifica internamente para apuntar sempre á seguinte instrución a executar.

Ao traballar con QtARMSim séguese o convenio, un tanto artificioso, de que a última instrución que se executa nun programa é sempre **`wfi`**, é dicir, cando o simulador está a executar un programa e atópase unha instrución **`wfi`**, detén a execución e devolve o control ao usuario[2]. Isto é un convenio específico de QtARMSim, que non ten por que ser o mesmo noutras plataformas. Por exemplo, na Raspberry Pi terminaremos os nosos programas cunha instrución de salto **`bx`** `lr` que devolve o control ao sistema operativo (cf. epígrafe 1.2).

Unha vez que se executa a instrución **`wfi`**, non se pode continuar coa execución do programa desde o simulador, ou volver executalo. Se queremos repetir a simulación, podemos volver cargar o programa usando a opción de menú correspondente (🗀), o atallo de teclado Ctrl + O , ou simplemente podemos restablecer a simulación premendo F4 . Tamén podemos modificar o rexistro `r15` (é dicir, o contador de programa) para que apunte a unha instrución executable (p. ex., á dirección `0x00180000` onde está almacenada a primeira instrución do noso programa).

Para executar un programa paso a paso, seleccionariamos a opción do menú de execución paso a paso ({▾}), ou premeriamos F5 . Ao executar paso a paso, podemos monitorizar como cambian os valores dos rexistros e as posicións de memoria durante a execución, e tamén os indicadores do procesador `N`, `Z`, `C` e `V` (signo, cero, carrexo e desbordamento respectivamente).

Ademais de executar paso a paso, é posible executar as instrucións nun programa de maneira que, en caso de executar unha instrución de chamada a subprograma, por exemplo a instrución **`bl`**, execútese o subprograma completo dunha vez antes de devolver o control á execución paso a paso. Trátase da opción *step over* ({}▾ ou F6).

Un punto de ruptura ou *breakpoint* é unha indicación para que o simulador deteña a execución antes de executar a instrución onde se colocou devandito punto de ruptura. Ao facer clic na marxe da xanela de simulación xunto a unha instrución (por exemplo, **`ldr`** `r0, =nc`), aparecerá un círculo vermello que indica que se definiu un punto de ruptura (cf. figura 1.4). Ao executar un programa con Ctrl + F11 , o programa deterase xusto antes de executar a instrución onde colocamos o punto de ruptura.

[2] Nunha contorna real, a instrución **`wfi`** (wait for interrupt) detén a execución á espera de que se produza unha interrupción, cuxa rutina de servizo retomará o control do procesador. Nalgúns microprocesadores, como os da familia Cortex-M, a execución de **`wfi`** implica ademais entrar en modo de baixo consumo.

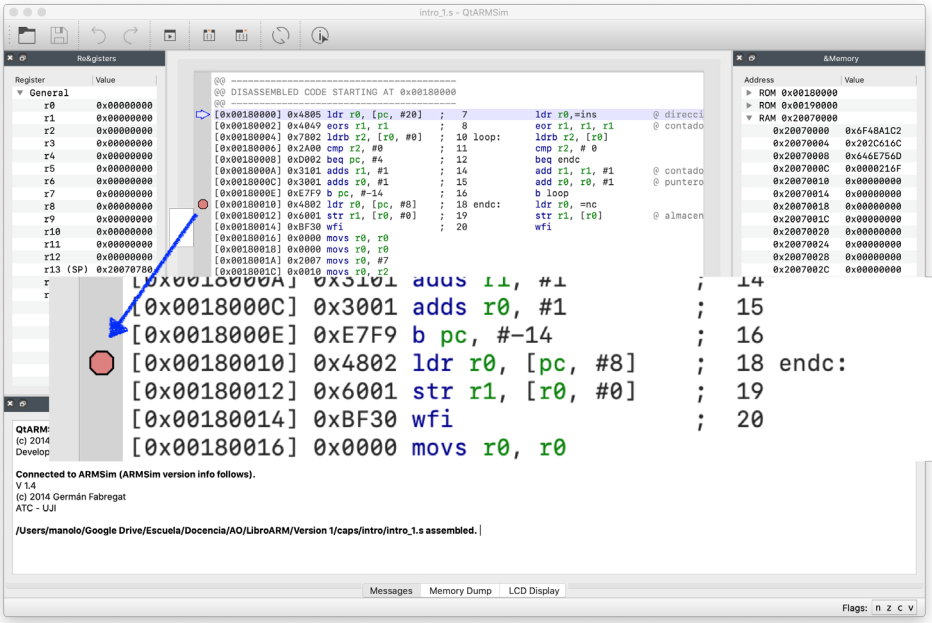


Figura 1.4. Punto de ruptura en QtARMSim.

Para eliminar un punto de ruptura xa definido, simplemente facemos clic no seu círculo vermello.

1.2 Programación en Thumb con Raspberry Pi

Raspberry Pi[3] (Smith, 2013) é un computador de placa única (*single-board computer*, SBC) de baixo custo desenvolvido co obxectivo de promover o ensino da informática. Até o momento de escribir este manual, todos os modelos de Raspberry Pi baséanse nun sistema nun chip (SoC) ARM de Broadcom. Devandito microprocesador inclúe un núcleo ARM con soporte amplo para o repertorio de instrucións T32 (e por tanto para as instrucións Thumb de 16 bits), unha unidade de procesado de gráficos (GPU), a memoria RAM, un procesador de sinal (DSP) e un conxunto amplo de liñas de entrada e saída de propósito xeral, entre outros compoñentes. Os computadores Raspberry Pi dispoñen dun sistema operativo propio (Raspberry Pi OS, derivado de Debian), e varios sistemas operativos baseados en Linux para arquitectura ARM, como Arch Linux ARM (derivado de Arch Linux) e Pidora (derivado de Fedora). Deles, o máis popular é Raspberry Pi OS. A figura 1.5 ilustra un dos modelos máis populares deste dispositivo.

Como case todas as distribucións baseadas en Linux, Raspberry Pi OS inclúe as ferramentas de desenvolvemento de programas do proxecto GNU (*GNU Toolchain*). Entre elas atópase o ensamblador de GNU GAS (`as`), o depurador GDB (`gdb`), o compilador GCC (`gcc`) e o montador de ligazóns LN (`ln`). Como o microprocesador da Raspberry Pi sobre o que se executa Raspberry Pi OS é un procesador ARM que soporta as instrucións Thumb de 16 bits, con estas ferramentas poderemos ensamblar (con `as` e `ld` ou `gcc`), depurar e probar (con `gdb`) programas escritos co repertorio de instrucións Thumb.

Aínda que o procesador da Raspberry Pi soporta Thumb, hai algunhas características que Raspberry Pi OS non soporta, polo que non poderemos programar exclusivamente en Thumb para a Raspberry Pi utilizando as ferramentas de GNU dispoñibles en Raspberry Pi OS. Isto non quere dicir que non sexa posible escribir, ensamblar e probar programas en Thumb. Podemos escribir código en Thumb, pero teremos que facelo de maneira que Raspberry Pi OS permítanos acceder ao noso código e regresar desde el.

Unha maneira de superar este inconveniente baséase en aproveitar a posibilidade que ofrece ARM de mesturar código T32/Thumb con código ARM.

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

Figura 1.5. Raspberry Pi 4 Model B. O chip cadrado brillante cara ao centro da placa, á dereita da frambuesa, é o microprocesador, un SoC Broadcom BCM2711, con catro núcleos ARM Cortex-A72. (De Michael Henzler / Wikimedia Commons, CC BY-SA 4.0).

Os microprocesadores ARM teñen un bit de estado T32 (`T`), de maneira que cando devandito bit toma o valor 1 (`T = 1`), o microprocesador funciona en *estado T32* e é capaz de decodificar as instrucións do repertorio T32, e por tanto as instrucións do repertorio orixinal Thumb de 16 bits. Cando toma o valor 0 (`T = 0`) atópase en *estado A32/ARM* e decodifica instrucións ARM de 32 bits.

Para mesturar código ARM e código Thumb, é dicir, para cambiar de estado A32/ARM a estado T32/Thumb e viceversa, debemos executar unha instrución de *salto e intercambio* (*branch and exchange*). Trátase de instrucións que realizan un salto e ademais cambian o valor do bit de estado `T`. Ditas instrucións son **`bx`** e **`blx`**. A instrución **`blx`** realiza unha chamada a subprograma (cf. páxina 115 do capítulo 4) e ademais intercambia o repertorio de instrucións.

Utilizando as instrucións anteriores, é posible *envolver* o noso código escrito co repertorio Thumb cun envoltorio ARM para pasarllo ao sistema operativo e para que o sistema operativo devólvanos o control unha vez executado o noso código. Os nosos programas escritos para ser analizados na Raspberry Pi terán a estrutura seguinte:

1. Un sinxelo programa principal de entrada, escrito en ARM, que simplemente chama a un subprograma que contén o noso código Thumb utilizando a instrución de chamada a subprograma **`blx`** (*branch, link and exchange*).

```
            .arm           @ Indicamos que este código é A32.
            .align  4      @ Instrucións aliñadas a 32 bits.
            .global _start @ ou .global main

_start:     push {r4, lr}  @ Apilamos o rexistro de ligazón lr.

            blx meu_progr  @ Utilizamos blx para conmutar a T32
                           @ ao pasar o control ao noso programa.
            pop {r4, lr}   @ Recuperamos o rexistro de ligazón
            bx lr          @ e devolvemos o control ao SO.
           .end
```

   Podemos observar tamén que o programa principal salvagarda na pila o rexistro de ligazón `lr` antes de executar a instrución **`blx`**, para así permitir o aniñamento do noso subprograma (cf. apartado 4.3).

2. O noso código Thumb/T32, escrito como un subprograma. Por iso, no canto de terminar os nosos programas con **`wfi`** como faciamos nos programas para QtARMSim, terminarémolos coa instrución de salto e intercambio **`bx lr`** do repertorio Thumb de 16 bits. Esta instrución recupera a dirección de retorno ao programa principal almacenada no rexistro de ligazón `lr`, onde a almacenou a instrución **`blx`** do punto anterior, e cambia ao microprocesador de volta ao estado ARM (cf. apartado C.3.4 e táboa C.8).

O programa-envoltorio terá que estar declarado de maneira que o montador de ligazóns-cargador poida atopar o punto de entrada para pasarlle o control cando o sistema operativo o cargue para ser executado. Para iso, o convenio é etiquetar o punto de entrada desexado (no noso caso, a primeira instrución do programa-envoltorio) coa etiqueta `_start` declarada globalmente:

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

Outra opción consiste en declarar noso programa-envoltorio como o programa principal dun programa en C e logo enlazar o noso código coa biblioteca estándar de C `libc` utilizando o compilador de C `gcc`. Con iso, teremos á

nosa disposición todas as funcionalidades da devandita biblioteca estándar, o que nos permitirá ler e mostrar caracteres por pantalla entre outras moitas cousas. O único cambio será que agora o punto de entrada estará etiquetado coa etiqueta `main` declarada globalmente:

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

A figura 1.6 presenta un exemplo desta estratexia baseada no programa da figura 1.2.

Podemos observar que o programa-envoltorio salvagarda na pila o rexistro `r4` ademais do rexistro de ligazón. Isto é así porque o convenio da arquitectura ARM para a chamada a procedementos (AAPCS, *ARM Architecture Procedure Call Standard*, Arm Ltd., 2020) especifica que nos interfaces públicos (como os *procedementos* principais `_start` ou `main`) a pila ten que estar sempre aliñada a unha dirección múltiplo de 8 (cf. apéndice B). Simplemente temos que garantir que ao introducir valores na pila, o punteiro de pila `sp` segue sendo un múltiplo de 8. En realidade, `r4` podería ser calquera rexistro, aínda que seguindo o convenio AAPCS `r4` sería o primeiro rexistro a conservar dentro dun subprograma (seguindo este convenio, ningún subprograma ten por que garantir que se conservan os rexistros `r0` a `r3`).

Se o noso código Thumb chama á súa vez a outros subprogramas, será necesario conservar o rexistro `lr` para recuperalo ao finalizar, seguindo por suposto o convenio AAPCS. O código resultante tería a estrutura da figura 1.7. Hai que ter en conta que a instrución **`pop {pc}`** do repertorio Thumb é equivalente á secuencia de instrucións **`pop {lr}`** seguido de **`bx lr`** do repertorio A32.

Finalmente, se queremos chamar a un subprograma en código A32 desde o noso código Thumb (p. ex., se queremos chamar a unha función desde a biblioteca estándar), seguiremos o mesmo esquema, só que utilizaremos a instrución **`blx`** para cambiar de novo o procesador ao estado A32. Cando devandito subprograma termine, volverase a producir un cambio de modo, continuándose a execución do noso código Thumb (cf. Figura 1.8).

Para escribir os nosos programas podemos utilizar calquera editor de texto dispoñible para Raspberry Pi OS, como `vim`, `nano`, `emacs`, o editor gráfico `gedit`, ou calquera outro. Asignaremos a extensión `.s` aos ficheiros que cremos, igual que fai QtARMSim.

intro_2a.s

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

     .text
/*
 * Os nosos programas serán funcións invocadas desde o
 * punto de entrada dun programa escrito para Raspberry Pi OS.
 */
     .code16        @ Esta directiva indica que usaremos Thumb.
     .align 2       @ As instrucións de Thumb son de 16 bits
                    @ (aliñadas en fronteiras de 2 bytes).
meu_prog:
    ldr r0,=ins     @ Dirección cadea 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  @ Punteiro de incremento.
    b loop

endc:
    ldr r0, =nc
    str r1, [r0]    @ Almacenamos o resultado
                    @ e volvemos a main.
    bx lr           @ Esta instrución substitúe a wfi en QtARMSim
/*
 * Aquí estaría o punto de entrada desde o SO: unha función chamada
 * "main", declarada obrigatoriamente como main.
 */
    .arm            @ Indicamos que o noso código é ARM completo.
    .align 4        @ Instrucións aliñadas a 32 bits.
    .global main
main:
    push {r4, lr}   @ Apilamos o rexistro de ligazón lr.

    blx meu_prog    @ Utilizamos blx para conmutar de ARM a Thumb
                    @ ao pasar o control ao noso programa.
    pop {r4, lr}    @ Recuperamos o rexistro de ligazón
    bx lr           @ e devolvemos o control ao SO.
   .end
```

Figura 1.6. Programa da figura 1.2 adaptado para a súa execución baixo Raspberry Pi OS tras ensamblalo e montalo con `as` e `gcc`.

intro_2b.s

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

     .text
     .code16         @ Esta directiva indica que usaremos Thumb.
     .align 2        @ As instrucións de Thumb son de 16 bits
                     @ (aliñadas en fronteiras de 2 bytes).

/* Subprograma do noso programa en Thumb */

subp_conta:
    ldrb r2, [r0]
    cmp  r2, # 0
    beq  ends

    add  r1, r1, #1  @ Contador de incrementos.
    add  r0, r0, #1  @ Punteiro de incremento.
    b    subp_conta

ends:
    mov  lr, pc      @ Final do subprograma en Thumb.

/* O noso programa principal en Thumb. */

meu_prog:
    push {r4, lr}    @ Apilamos lr seguindo o AAPCS.

    ldr  r0,=ins     @ Dirección cadea de entrada.
    eor  r1, r1, r1  @ Contador a 0.

    bl   subp_conta  @ Chamada a un subprograma en Thumb.

    ldr  r0, =nc
    str  r1, [r0]    @ Almacenamos o resultado.

    pop {r4, pc}     @ Regresamos ao envoltorio en ARM completo.
```

Figura 1.7. Chamada a un subprograma escrito en Thumb desde o noso código Thumb en Raspberry Pi OS.

intro_2c.s

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

     .text
     .code16          @ Esta directiva indica que usaremos Thumb.
     .align 2         @ As instrucións de Thumb son de 16 bits
                      @ (aliñadas en fronteiras de 2 bytes).

/* O noso programa principal en Thumb. */

meu_prog:
    push {r4, lr}     @ Apilamos lr seguindo o AAPCS.

    ldr  r0,=ins      @ Dirección da cadea de entrada (para printf).
    blx  printf       @ Chamada á función estándar printf (A32).

    pop {r4, pc}      @ Regresamos ao envoltorio en ARM completo.
```

Figura 1.8. Chamada a un subprograma escrito en ARM completo desde o noso código Thumb en Raspberry Pi OS. Este segmento de código imprimiría "Ola mundo!" na saída estándar.

Para ensamblar un programa utilizaremos o comando `as` para invocar ao ensamblador. No caso de que o noso código estea nun ficheiro chamado `o meu_prog.s`, executariamos o comando:

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

Se non houbo erros, o comando anterior xerará un ficheiro obxecto coa extensión `.o`, no noso exemplo `o meu_prog.o`, con información para o depurador. No caso de que o código fonte tivese algún erro, o ensamblador dará conta dos devanditos erros e non se xerará o ficheiro anterior. Volveremos ao editor para corrixir os erros.

Agora necesitamos enlazar o ficheiro obxecto coas bibliotecas do sistema para xerar un programa executable utilizando o compilador `gcc`, incluíndo información para o depurador:

```
$ gcc -g -o meu_prog meu_prog.o
```

No caso de que decidísemos prescindir da biblioteca estándar de C, declararíamos `_start` como punto de entrada e simplemente chamaríamos ao enlazador-cargador con:

```
$ ld -o meu_prog meu_prog.o
```

En calquera dos dous casos, agora teríamos un programa que podemos executar directamente:

```
$ ./meu_prog
```

Ademais, podemos realizar operacións similares ás que poderíamos realizar cun simulador como QtARMSim utilizando un depurador como GDB (GNU Debugger):

```
$ gdb meu_prog
```

O cadro 1.2 recolle algúns comandos útiles de GDB, aínda que as opcións dispoñibles son moito máis amplas e versátiles, sobre todo se estendemos o depurador con algún paquete como GEF. A figura 1.9 ilustra o interfaz de usuario de GFB con GEF.

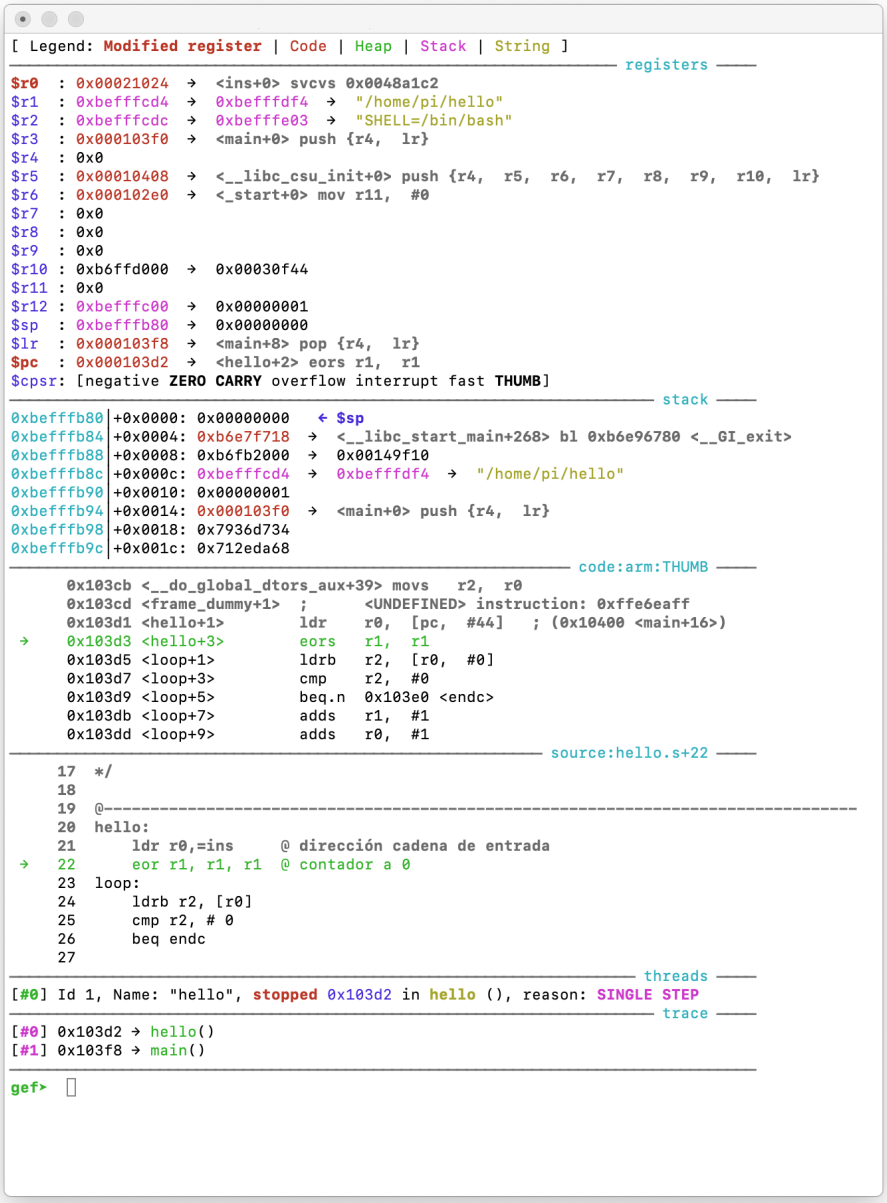


Figura 1.9. Analizando o programa de exemplo utilizando GDB coa extensión GEF.

Cadro 1.2. Algúns comandos útiles de `gdb`.

`b <n>`	Coloca un breakpoint na liña `<n>`.
`c`	Continúa coa execución do programa a partir da posición actual.
`d <n>`	Borra o breakpoint da liña `<n>`.
`h <c>`	Mostra axuda sobre o comando `<c>`.
`i <s>`	Mostra diversa información sobre o programa, en función de `<s>` (`registers`, `args`, `breakpoints`...) `h i` ofrece detalles sobre todas as opcións posibles.
`l <n>`	Lista 10 liñas do código fonte, centradas na liña especificada por `<n>`.
`p <e>`	Avalía a expresión `<e>` e imprime o seu valor. Se `<e>` é unha etiqueta, mostra o contido da posición de memoria etiquetada.
`r`	Comeza a execución dun programa.
`n`	Executa a seguinte liña de código (*step over*).
`s`	Executa a seguinte instrución (*step into*).
`f`	Executa até o final do subprograma actual.
`x/<n><f><t> <d>`	Mostra `<n>` valores de memoria en formato `<f>`, de tamaño `<t>`, comezando pola dirección `<d>`. `h x` ofrece detalles sobre todas as opcións posibles para `<f>` e `<t>`.
`q`	Sae do programa `gdb`.

1.3 Exercicios introdutorios

Neste apartado propomos unha serie de exercicios sinxelos orientados a familiarizarse con QtARMSim e a repasar algúns conceptos básicos relacionados coa representación e almacenamento de información na arquitectura ARM T32/Thumb. Como indicamos ao principio deste capótulo, o apéndice C inclúe, a modo de referencia rápida, información sobre as instrucións do repertorio Thumb de 16 bits (cf. cadros C.4 a C.8) e sobre as directivas do ensamblador de GNU utilizadas neste libro (cf. cadro C.9). O apéndice A presenta estes mesmos exercicios para o caso da Raspberry Pi.

Exercicio 1.1

O seguinte programa inicializa os rexistros `r0` a `r3` con 4 valores numéricos en decimal, hexadecimal, octal e binario, respectivamente. Edita o programa, ensámblalo e responde as preguntas.

e_intro_1.s

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

1. Como se mostran os números anteriores ao desensamblar o programa na pestana ARMSim de QtARMSim? Están na mesma base que no código orixinal? En que base están representados?

2. Executa o programa paso a paso. Que números almacénanse nos rexistros `r0` a `r3`?

   Solución

   Os datos inmediatos no código desensamblado represéntanse en decimal, co que ao observar o código na xanela de simulación obteriamos 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 a representación en hexadecimal dos datos inmediatos a partir do formato de instrución, xa que os datos inmediatos se codifican na propia instrución. No exemplo anterior, correspóndense co segundo byte da instrución **movs**: `0x1E`, `0x42`, `0x42` e `0x42`.

En canto á representación do contido dos rexistros, represéntanse en hexadecimal. Así, ao executar o programa o contido dos rexistros aparece como segue no panel de rexistros de QtARMSim:

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

Exercicio 1.2

Edita o código presentado a continuación, ensámblalo, analízao e responde as preguntas.

e_intro_2.s

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

       .text
stop:  wfi
       .end
```

1. Identifica as posicións de memoria onde se almacenaron os datos definidos no programa. Identifica o catro valores en hexadecimal.

2. En que direccións almacénanse o catro valores? Por que estas direccións de memoria son múltiplos de catro en lugar de ser direccións consecutivas?

3. Cales son os valores das etiquetas `word1`, `word2`, `word3` e `word4`?

Solución

As posicións de memoria onde se almacenan as 4 palabras definidas no código anterior son as seguintes:

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

Cada palabra ocupa 4 bytes e o convenio de almacenamento ou *endianness* é o extremista menor (*little endian*), é dicir, o byte menos significativo da palabra almacénase na posición de memoria de dirección máis baixa.

Por tanto, os valores das etiquetas `word1`, `word2`, `word3` e `word4` serán os indicados a continuación:

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

Exercicio 1.3

Ensambla o seguinte código:

e_intro_3.s

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

Hai algún cambio nos valores almacenados na memoria con respecto aos almacenados no caso do programa anterior? Están no mesmo lugar?

Solución

Non hai ningún cambio no que respecta ao almacenamento dos datos anteriores. O único cambio sería que agora definimos unha única etiqueta (`wrds`) con valor `0x20070000`.

Exercicio 1.4

Ensambla o seguinte código:

e_intro_4.s

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

```
      .text
stop: wfi
      .end
```

1. Que valores almacenáronse na memoria? En que posicións?
2. Cal é o valor da etiqueta `bys`?

   Solución

   Almacénase un byte en cada posición de memoria, comezando na dirección `0x20070000`. A etiqueta `bys` tomará devandito valor.

Exercicio 1.5

Agora ensambla o seguinte código:

e_intro_5.s

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

1. Que rango de posicións de memoria reservouse para a variable etiquetada como `strg`?
2. Como se representa en memoria a cadea `abác`? Como se representa o carácter `á`?
3. Pescuda cal é o sistema de codificación de caracteres da túa contorna de traballo (UTF-8, ASCII, ISO Latin 9 ...).
4. Cal é o valor da etiqueta `byte`?

   Solución

   Trátase dunha cadea de caracteres. QtARMSim baséase no ensamblador de GNU, que utiliza UTF-8 como sistema de representación de caracteres na maioría das plataformas, sendo a excepción máis relevante Microsoft Windows, onde o sistema de representación utilizado é ISO Latin 9 (ISO/IEC 8859-15).

   Por tanto, para a cadea etiquetada como `streg` reservaranse 5 posicións no caso da maioría das plataformas excepto Windows e a representación en memoria byte a byte será:

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

No caso de QtARMSim baixo Microsoft Windows, a representación será:

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

e reservaranse 4 posicións de memoria.

Vemos que o carácter `á` represéntase co código de dous bytes (`0xc3`, `0xa1`) en UTF-8, de acordo co descrito no apéndice D, e como `0xe1` no caso diso Latin 9.

Finalmente, o valor da etiqueta `byte` será `0x2007005` ou `0x2007004` en QtARMSim, dependendo de que o sistema de representación de caracteres sexa UTF-8 ou ISO Latin 9.

Exercicio 1.6

Ensambla e analiza o seguinte código:

e_intro_6.s

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

1. Cantas posicións de memoria resérvanse para a variable `gap`?
2. Poderían lerse ou escribirse o catro bytes utilizados pola variable `gap` coma se fosen unha palabra? Por que?
3. Cal é o valor da etiqueta `b1`? E da etiqueta `b2`?
4. Cal é o valor da etiqueta `bign`? Poderían lerse ou escribirse o catro bytes que comezan na dirección `bign` coma se fosen unha palabra? Por que?
5. Agrega unha directiva **`.balign`** ao código anterior para que a variable etiquetada como `bign` alíñese cun límite de palabra (é dicir, cunha dirección múltiplo de catro).

Solución

Para a variable `gap` resérvanse 4 posicións de memoria. Non poderemos acceder en lectura ou escritura aos 4 bytes de `gap` coma se fosen unha palabra porque os accesos a palabra en ARM teñen que estar alienados a unha dirección múltiplo de 4.

Para poder acceder a `bign` como unha palabra teriamos que engadir unha directiva de aliñación `.balign 4` ou ben `.align 2` antes da definición da devandita etiqueta. En ambos os casos aliñaríase o espazo etiquetado por `bign` a unha dirección múltiplo de 4.

O código é o seguinte:

e_intro_7.s

```
1       .data
2  b1:  .hword  0x11       @ isto ocupa dous bytes
3  gap: .space  4          @ catro bytes adicionais
4  b2:  .byte   0x22       @ un byte adicional (total 7 bytes)
5       .balign 4          @ aliñamos a un múltiplo de 4
6  bign: .word  0x33445566 @ agora podemos definir unha palabra
7
8       .text
9  stop: wfi
10      .end
```

Capítulo 2
Soltando amarras

Neste capítulo propomos unha serie de exercicios orientados á iniciación na programación en ensamblador. Estes exercicios están pensados para axudar a comprender o funcionamento das instrucións máis básicas, así como a organización dos datos e o seu procesado no nivel de máquina convencional, utilizando no noso caso como referencia a arquitectura ARM e o repertorio Thumb do núcleo ARM7TDMI.

Os primeiros exercicios do capítulo dedícanse a presentar as estruturas básicas de programación `if-then`, `if-then-else`, `do-while` e `for`, e a súa programación en ensamblador. A continuación, proponse algúns exercicios onde se pide a codificación de programas que teñen como obxectivo transferir información entre a memoria e os rexistros, e entre zonas de memoria, por exemplo intercambiar o contido dunha zona de memoria segundo diversos criterios, transferir de maneira selectiva información dunha zona de memoria a outra, mesturar os contidos de varias zonas de memoria, ou identificar contidos en zonas de memoria.

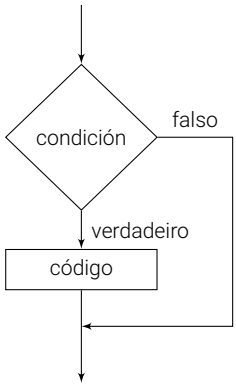
2.1 Estruturas básicas de control

Os exercicios seguintes teñen como obxectivo presentar a programación en ensamblador das catro estruturas de control máis comúns en programación imperativa: `if-then`, `if-then-else`, `do-while` e `for`. Polo menos unha destas estruturas aparecerá en practicamente todos os exercicios do libro.

Os catro exercicios teñen unha estrutura similar. Primeiro preséntase a estrutura de control, mediante pseudocódigo e mediante unha descrición textual. A continuación, proponse un exercicio concreto de programación en en-

```
if (condición) {
      código
}
```

(a) Código.



(b) Diagrama de fluxo.

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

samblador, que tamén se ilustra mediante pseudocódigo. Finalmente, preséntase a solución ao exercicio mediante código Thumb/T32 de 16 bits.

Exercicio 2.1

Neste exercicio ilustramos a programación en ensamblador da estrutura de control `if - then` (cf. figura 2.1).

Esta estrutura de control permite executar certo código segundo a avaliación dunha condición simple, sexa falsa ou verdadeira. Se a condición é verdadeira, execútase o bloque de código **`código`**; pola contra, non se executa devandito código.

Para avaliar a condición en ensamblador, utilizamos instrucións que actúan sobre os indicadores `Z, N, V, C`, e a continuación executamos unha instrución de salto condicional que nos permita ignorar as instrucións que forman parte de **`código`** no caso de que non se cumpra a condición.

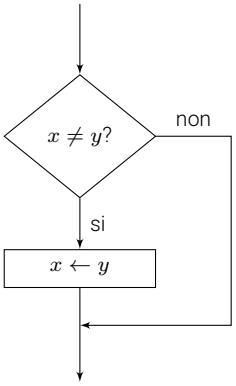
Para ilustrar esta estrutura de control, imos programar o exemplo da figura 2.2).

Solución

Basicamente, traducimos a linguaxe ensamblador o código da figura 2.2). A comparación entre os dous valores de `x` e `y` realizámola utilizando a instrución **`cmp`**. Esta instrución é equivalente a unha instrución de resta, só que se modifican unicamente os indicadores de condición `N`, `Z`, `C` e `V` dependendo do resultado da operación (non se modifica ningún rexistro de datos). No caso de que `x` e `y` sexan diferentes, é dicir, no caso

```
x = 1;
y = 100;
if (x != y) {
      x = y;
}
```

(a) Código.



(b) Diagrama de fluxo.

Figura 2.2. Exemplo de estrutura de control `if-then`.

de que o seu resta sexa distinta de cero, tras executar a instrución **`cmp`** o indicador `Z` tomará o valor 0 (é dicir, o resultado da operación non foi cero).

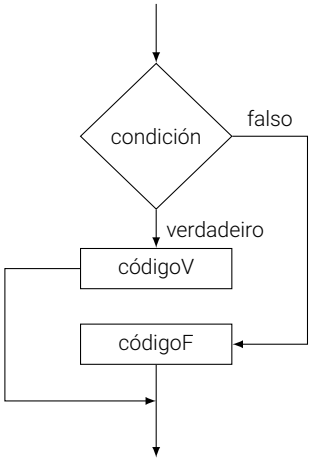
Despois, utilizamos a instrución de salto condicional **`beq`** para elixir que parte do código executar en función de se se cumpre a condición do `if` ou non (cf. cadro C.1 do apéndice C).

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 se se cumple a condición
16  @ {
17        ldr r3,=x
18        str r1,[r3]  @ (x) <- (y)
19  @ }
20  @     fin do código se se cumple a condición
21  @
22  endif:
```

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

(a) Código.



(b) Diagrama de fluxo.

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

```
23      wfi
24      .end
```

Exercicio 2.2

Neste exercicio ilustramos a programación en ensamblador da estrutura de control `if - then - else` (cf. figura 2.3).

Esta estrutura de control é unha extensión da estrutura do exercicio anterior. Tamén permite executar certo código segundo a avaliación dunha condición simple, sexa falsa ou verdadeira. Se a condición é verdadeira, execútase o bloque de código **`códigoV`**; pero pola contra, execútase o bloque de código **`códigoF`**.

Do mesmo xeito que no caso anterior, para avaliar a condición en ensamblador utilizamos instrucións que actúan sobre os indicadores e a continuación executamos unha instrución de salto condicional.

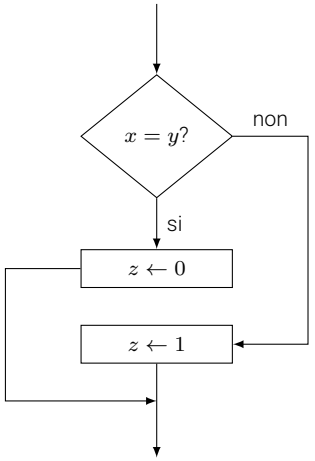
Para ilustrar esta estrutura de control, imos programar o exemplo da figura 2.4).

Solución

Este exercicio é moi similar ao exercicio anterior, só que teremos que executar un bloque de código diferente no caso de que se cumpra a condición e no caso de que non se cumpra. Do mesmo xeito que antes,

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

(a) Código.



(b) Diagrama de fluxo.

Figura 2.4. Exemplo de estrutura de control `if-then-else`.

a comparación entre os dous valores de `x` e `y` realizámola utilizando a instrución **`cmp`**. Despois utilizamos a instrución de salto condicional **`bne`** para elixir que parte do código executar en función de se se cumpre a condición do `if` ou non.

É dicir, se os valores de `x` e `y` eran iguais, `Z` tomará o valor 1 tras a execución de **`cmp`**, non se cumprirá a condición de salto **`bne`** e executaranse as liñas de código que están a continuación da devandita instrución de salto. Se non son iguais, `Z` tomará o valor 0, cumprirase a condición de salto e executarase o código etiquetado como `else` (cf. cadro C.1 do apéndice C).

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 se se cumpre a condición
17 @ {
18       mov r2,#0    @ r2 <- 0
19       b   endif
20 @ }
21 @     código se non se cumpre a condición
22 @ {
23 else:
24       mov r2,#1 @ r1 <- 1
25 @ }
26 @     código común a ambas as opcións
27 @
28 endif:
29       ldr r3,=z
30       str r2,[r3] @ (z) <- r2 (un 0 ou un 1)
31       wfi
32       .end
```

Exercicio 2.3

Neste exercicio ilustramos a programación en ensamblador da estrutura de control `while` (cf. figura 2.5).

Esta estrutura de control permite executar certo código mentres sexa verdadeira unha condición simple. Se **`condición`** é verdadeira, execútase o bloque de código **`código`** e a continuación vólvese a comprobar a dicir condición. O bucle repítese ata que **`condición`** sexa falsa.

Como nos casos anteriores, para avaliar a condición en ensamblador utilizamos instrucións que actúan sobre os indicadores `C`, `N`, `V` e `Z`, e a continuación executamos unha instrución de salto condicional.

Para ilustrar esta estrutura de control, imos programar o exemplo da figura 2.6.

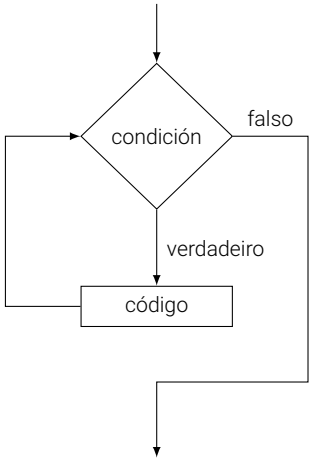
Solución

Do mesmo xeito que nos exercicios 2.1 e 2.2, utilizamos unha instrución **`cmp`** e unha instrución de salto condicional, neste caso **`ble`**, para tomar decisións sobre o fluxo de programa.

No caso desta estrutura `while`, sairemos do bucle cando `num` deixe de ser maior que 0. Para iso, comparamos `num` (almacenado no rexistro `r1`) con cero. No caso de que o resultado da operación de resta asociada sexa menor ou igual que cero, ou o que é o mesmo, que o resultado non sexa maior que cero, a condición de salto de **`ble`** farase efectiva e

```
while (condición) {
      código
}
```

(a) Código.

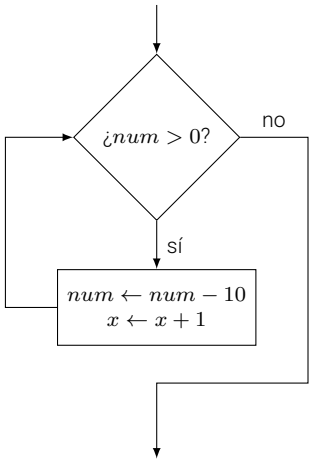


(b) Diagrama de fluxo.

Figura 2.5. Estrutura de control `while`.

```
x = 0;
num = 100;
while (num > 0) {
      num = num - 10;
      x = x + 1;
}
```

(a) Código.



(b) Diagrama de fluxo.

Figura 2.6. Exemplo de estrutura de control `while`.

sairemos do bucle saltando á posición etiquetada como `finb` (cf. cadro C.1 do apéndice C).

while.s

```
        .data
x:      .word 0
num:    .word 100

        .text
main:
        ldr r0,=x
        ldr r0,[r0]  @ r0 <- (x)
        ldr r1,=num
        ldr r1,[r1]  @ r1 <- (num)
        mov r2,#10

bucle:
        cmp r1,#0    @ condición de fin
        ble finb     @ de bucle: r1 <= 0?
@
@       contido do bucle
@ {
        sub r1,r1,r2 @ r1 <- r1 - 10
        add r0,r0,#1 @ r0 <- r0 + 1
@ }
@       fin do contido do bucle
@
        b bucle

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

Exercicio 2.4

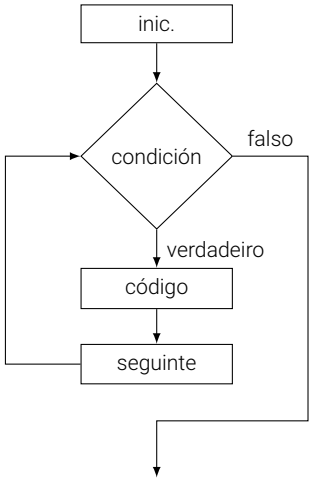
Finalmente, neste exercicio ilustramos a programación en ensamblador da estrutura de control `for` (cf. figura 2.7).

Do mesmo xeito que no caso da estrutura `while`, a estrutura `for` dá lugar a un lazo ou bucle, pero neste caso permite executar un bloque de código un número determinado de veces.

1. Avalíase a expresión `inic` dando como resultado un valor.
2. Avalíase a expresión `condición`.

```
for (inic; condición; seguinte) {
      código
}
```

(a) Código.



(b) Diagrama de fluxo.

Figura 2.7. Estrutura de control `for`.

3. Se se cumpre a condición avaliada, execútase o bloque `código`. Se non, termínase o bucle.

4. Avalíase a expresión `seguinte` e vólvese ao momento 2.

Como caso particular, esta estrutura permite definir un bucle controlado por unha variable desde un valor inicial até un valor final. Esta versión da estrutura é útil para procesar un conxunto indexado de valores, por exemplo un vector.

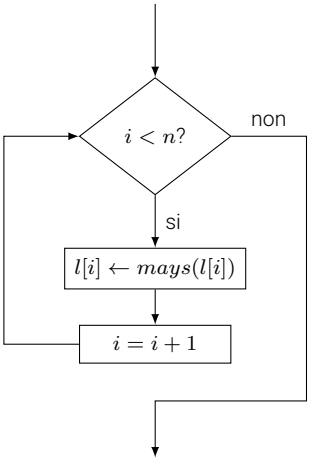
```
for (i = 0; i < límite; i = i + 1) {
      procesar elemento i-ésimo
}
```

Neste caso, a expresión `i = 0` correspóndese coa expresión `inic` do caso xeral, a expresión `i < límite` coa expresión `condición` e `i = i + 1` coa expresión `seguinte`. O valor da expresión `i < límite` vainos a servir de condición para seguir dentro do bucle ou non. Dáselle un valor inicial con `i = 0` (punto 1 anterior) e logo vaise actualizando con `i = i + 1` (punto 4 anterior). Cando se cumpra `i < límite` (punto 3 anterior), o bucle termina.

Para ilustrar esta estrutura de control, imos realizar un programa que pasa a maiúsculas unha lista de cinco caracteres ASCII (cf. figura 2.8).

```
l = {'a','b','c','d','e'};
n = 5;
for (i = 0; i < n; i++) {
      l[i] = mays(l[i]);
}
```

(a) Código.



(b) Diagrama de fluxo.

Figura 2.8. Exemplo de estrutura de control `for`.

Cadro 2.1. Instrucións lóxicas e máscaras. `n` representa un bit (0 ou 1)

and	**orr**	**eor**
`n and 0 = 0`	`n orr 0 = n`	`n eor 0 = n`
`n and 1 = n`	`n orr 1 = 1`	`n eor 1 = not(n)`

Solución

Traducimos o código anterior a linguaxe ensambladora, utilizando unha instrución **`cmp`** e unha instrución de salto condicional **`beq`** para tomar decisións sobre a terminación do bucle.

As letras maiúsculas e minúsculas diferéncianse unicamente nun bit. Para pasar a maiúsculas, pomos a 0 ese bit mediante unha máscara adecuada (`0xdf`) e unha instrución **`and`** (cf. exercicio 3.14).

En xeral, podemos utilizar as instrucións lóxicas **`and`**, **`orr`** e **`eor`** (AND, OR e XOR (eXclusive OR)) xunto con combinacións binarias denominadas *máscaras* para activar ou desactivar bits concretos dun operando (cf. cadro 2.1). Así, para poñer a cero un bit efectuaremos unha operación AND (instrución **`and`**) cunha máscara cuxo bit correspondente está a cero; para pór a un un bit utilizaremos unha instrución **`orr`** cunha máscara co bit obxectivo co valor un, e para investir un bit (cambiar o seu

estado de cero a un ou de un a cero) fariámolo mediante unha instrución **eor** e unha máscara co bit correspondente a un.

for.s

```
1       .data
2  l:   .byte 'a', 'b', 'c', 'd', 'e'
3       .balign 4
4  n:   .word 5
5
6       .text
7  main:
8       ldr r0,=l    @ r0 <- dir. da lista
9       ldr r1,=n
10      ldr r1,[r1]  @ r1 <- (n) (tamaño da lista)
11      mov r2,#0    @ r2 <- 0 (índice do bucle for)
12      mov r4,#0xdf @ r4: máscara para pasar a maiúsculas
13
14 bucle:
15      cmp r2,r1    @ condición de fin de beq
16      finb     @ bucle: índice = tamaño? r2 = r1?
17 @
18 @    contido do bucle
19 @ {
20      ldrb r3,[r0,r2]  @ r3: cargamos un carácter
21      and  r3,r3,r4    @ pasámolo a maiúscula
22      strb r3,[r0,r2]  @ almacenámolo modificado.
23 @ }
24 @    fin do contido do bucle
25 @
26 @    incrementamos o contador (índice = índice + 1)
27 @
28      add r2,r2,#1
29      b bucle
30
31 finb: wfi
32      .end
```

2.2 Trasfega do contido da memoria

Unha das características esenciais de todos os procesadores RISC como os da familia ARM é a súa arquitectura *load/store*. O repertorio de instrucións divídese en dous grandes grupos:

- Acceso á memoria, con instrucións de carga (*load*) e almacenamento (*store*) que levan a cabo entre a memoria e os rexistros.

- Operacións aritméticas e lóxicas, que unicamente son posibles entre valores almacenados nos rexistros.

Os exercicios que se propoñen a continuación teñen como obxectivo practicar a transferencia de información desde a memoria principal do computador aos rexistros e viceversa, utilizando os modos de direccionamento dispoñibles no repertorio Thumb/T32 de 16 bits. As instrucións correspondentes ás operacións aritméticas e lóxicas trataranse no capítulo 3.

Exercicio 2.5

Realizar un programa que intercambie unha zona de memoria, utilizando o seguinte método:

> O método baséase na comparación por cambio de elementos, xa que se van comparando de dous en dous os elementos (números enteiros de 1 byte) da zona de memoria. Devanditos elementos intercambiaranse, se e só se, tanto o primeiro como o segundo son menores ou iguais que o contido do rexistro `r7`.

Tras a execución do programa, a zona de memoria cos datos deberá continuar tendo os mesmos números enteiros que antes da execución do mesmo. Obviamente, devanditos números poderán estar nunha orde diferente, pero non poderá faltar nin repetirse ningún dentro da citada zona.

Como exemplo supor que a zona de memoria contén:

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

Solución

Utilizamos o rexistro `r0` como rexistro índice ao longo da zona de memoria. En cada iteración, utilizamos o modo de direccionamiento indexado para obter os dous números consecutivos a comparar, cuxas direccións efectivas serán `r0` e (`r0 + 1`).

b_mem1.s

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

```
7  main: ldr r0, =zona      @ r0: punteiro ao principio
8        mov r1, r0         @ da zona de memoria
9        add r1, r1, #ult   @ r1: punteiro ao final
10       mov r7, #comp      @ inicializamos r7
11
12 buc:  ldrb r2, [r0]      @ vemos se este e o
13       ldrb r3, [r0, #1]  @ seguinte son ambos os
14       cmp r7, r2         @ menores ou iguais
15       bmi sigo           @ que r7
16       cmp r7, r3
17       bmi sigo
18
19       strb r2, [r0, #1]  @ se son, intercambiamos
20       strb r3, [r0]
21
22 sigo: add r0, r0, #1     @ apuntamos ao seguinte
23       cmp r0, r1         @ até terminar
24       bne buc
25
26       wfi
27       .end
```

Exercicio 2.6

Realizar un programa que intercambie unha zona de memoria, utilizando o seguinte método:

> O método baséase na comparación por cambio de elementos, xa que se van comparando de dous en dous os elementos (números enteiros de 1 byte) da zona de memoria. Devanditos elementos intercambiaranse, se e só se, o segundo deles é menor ou igual que o elemento seguinte a si mesmo e na anterior iteración non se realizou un intercambio.

Tras a execución do programa, a zona de memoria cos datos deberá continuar tendo os mesmos números enteiros que antes da execución do mesmo. Obviamente, devanditos números poderán estar nunha orde diferente, pero non poderá faltar nin repetirse ningún dentro da citada zona.

Como exemplo supor que a zona de memoria contén:

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

Solución

Utilizamos o rexistro `r1` como rexistro índice para navegar pola zona de memoria. En cada iteración, `r1` apuntará ao primeiro número dos dous números a intercambiar, (`r1` + 1) ao segundo deles e (`r1` + 2) ao elemento seguinte ao segundo, cuxo valor nos dirá se se ha de realizar o intercambio ou non.

Ademais, utilizaremos o rexistro `r0` para indicar se na anterior iteración realizamos un intercambio ou non.

- `r0 = 0` indica que non se realizou un intercambio na iteración anterior. Se nesta iteración realizamos un intercambio, poremos ademais `r0 = 1`.
- Se `r0 = 1` significa que na iteración anterior fixemos un intercambio, e por tanto nesta iteración simplemente faremos `r0 = 0` e non faremos nada máis á parte de actualizar `r1` para apuntar ao seguinte 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: sinala se houbo cambio
7        ldr    r1,=zona        @ r1: punteiro á zona de memoria
8        mov    r2, r1
9        add    r2, r2, #penult @ r2: punteiro ao penúltimo número
10                              @ (último a comprobar)
11 bucle:
12       cmp    r0, #0
13       bne    outro            @ se r0 é cero, non houbo cambio
14       ldrb   r7, [r1, #1]
15       ldrb   r6, [r1, #2]
16       cmp    r7, r6          @ comprobamos a condición
17       bgt    nocam           @ do enunciado
18
19       ldrb   r5, [r1, #0]    @ realizamos o intercambio
20       strb   r5, [r1, #1]
21       strb   r7, [r1, #0]
22       mov    r0, #1          @ e marcamos o cambio
23
24 nocam:
25       add    r1, #1          @ apuntamos ao seguinte
26       cmp    r1, r2          @ e vemos se terminamos
27       bne    bucle
```

```
28      wfi
29
30  outro: mov   r0, #0          @ reseteamos o cambio
31      b     nocam
32      .end
```

Exercicio 2.7

Realizar un programa que intercambie unha zona de memoria, utilizando o seguinte método:

> O método baséase na comparación por cambio de elementos (números enteiros de 1 byte) da zona de memoria. Supostos os elementos ordenados $X1|X2|X3$. Intercambiaranse os elementos $X1$ e $X3$ se e só se, $X1 < X2 < X3$.

En sucesivas iteracións, os números que ocupan a posición X2, pasarán a ocupar a posición $X1$, os que ocupan a $X3$ pasarán a ocupar a $X2$, e por último os que ocupan a posición seguinte á $X3$ pasarán a ocupar a $X3$.

Tras a execución do programa, a zona de memoria considerada deberá continuar tendo os mesmos números enteiros que antes da execución do mesmo. Obviamente, devanditos números poderán estar nunha orde diferente, pero non poderá faltar nin repetirse ningún dentro da citada zona.

Como exemplo supor que a zona de memoria contén:

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

Solución

Do mesmo xeito que nos exercicios anteriores, utilizaremos o rexistro `r1` como punteiro aos elementos na zona de memoria. En cada iteración `r1` apuntará a $X1$, (`r1` + 1) apuntará a $X2$ e (`r1` + 2) apuntará a $X3$. Inicialmente, `r1` apuntará ao primeiro elemento da zona de memoria e irase incrementando en cada iteración até chegar ao 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: punteiro ao primeiro dato
```

```
7         mov   r2, r1
8         add   r2,r2,#penult   @ r2: punteiro ao penúltimo elemento
9
10    bucle:
11        cmp   r1, r2          @ comprobamos se terminamos
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 ao seguinte
28        b     bucle
29
30    final:
31        wfi
32        .end
```

Exercicio 2.8

Considere unha zona de memoria que contén 10 números enteiros de 1 byte. Realice un programa que, tomando devanditos números agrupados de 3 en 3 ($X1$, $X2$, $X3$), intercambie os números $X1$ e $X3$ se e só se $X1$ é maior ou igual que $X3$.

Utilice o seguinte código como exemplo da zona de memoria:

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

Solución

Este exercicio ten unha estrutura similar ao exercicio 2.7. Cambian as condicións para realizar o intercambio dos números en cada iteración.

b_mem4.s

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

```
3       .equ  penult, 8
4
5       .text
6  main: ldr   r1,=zona      @ r1: punteiro ao primeiro dato
7       mov   r2, r1
8       add   r2,r2,#penult @ r2: punteiro ao penúltimo dato
9
10 bucle:
11      ldrb  r7, [r1, #0]  @ r7 <- X(i)
12      ldrb  r5, [r1, #2]  @ r5 <- X(i+2)
13      cmp   r7, r5        @ se X(i) >= X(i+2) cambiar
14      blt   esteno
15
16      strb  r7, [r1, #2]  @ intercambio
17      strb  r5, [r1, #0]
18
19 esteno:
20      add   r1, #1        @ apuntamos ao seguinte
21      cmp   r1, r2        @ e vemos se acabamos
22      bne   bucle
23
24      wfi
25      .end
```

Exercicio 2.9

Considere unha zona de memoria que contén 10 números enteiros positivos de 1 byte.

Realizar un programa que faga as seguintes operacións:

- Tomando os números agrupados de 3 en 3 ($X1$, $X2$, $X3$), intercambiar os números $X1$ e $X3$ se e só se $X1$ é menor que $X3$.
- Tras isto, gardar o último número, é dicir, o que ocupa a posición 10 da zona de memoria, na dirección de memoria etiquetada como `ultimo`.

Utilizar o seguinte código como exemplo da zona de memoria:

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

Solución

Exercicio estruturalmente similar ao exercicio 2.8. Cambia a condición de intercambio (neste caso se $X1 < X3$ no canto de $X1 \geq X3$) e antes de terminar almacénase o último número na posición indicada.

b_mem5.s

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

      .text
main: ldr   r1,=zona      @ r1: punteiro ao primeiro dato
      mov   r2, r1
      add   r2,r2,#penult @ r2: punteiro ao penúltimo dato
      ldr   r3,=ultimo

bucle:
      ldrb  r7, [r1, #0]  @ r7 <- X(i)
      ldrb  r5, [r1, #2]  @ r5 <- X(i+2)
      cmp   r7, r5        @ se X(i) < X(i+2) cambiar
      bge   esteno

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

esteno:
      add   r1, #1        @ apuntamos ao seguinte
      cmp   r1, r2        @ e vemos se acabamos
      bne   bucle

      ldrb  r6, [r1, #1]  @ antes de terminar, almacenamos
      strb  r6, [r3]      @ o último elemento na
                          @ posición indicada.
      wfi
      .end
```

Capítulo 3
Velocidade de cruceiro

Unha vez que coñecemos a programación en ensamblador das estruturas de control máis habituais e as diferentes maneiras de transferir información utilizando a memoria e os rexistros, dedicamos este capítulo ao procesado da devandita información. Comezamos coas operacións máis básicas, como contar e comparar números, para introducir a continuación o procesado de cadeas de caracteres. Finalmente, propomos un conxunto de problemas centrados na utilización combinada das diferentes instrucións aritméticas, lóxicas e de desprazamento.

3.1 Contar e sumar

Esta sección propón un conxunto de exercicios para practicar a realización de operacións básicas entre os valores almacenados nos rexistros. Os exercicios concíbense de maneira incremental, é dicir, seguimos reforzando a experiencia adquirida no capítulo anterior relativa á carga e almacenamento de valores nos rexistros e complementámola coa realización de operacións básicas, como sumar e comparar.

Exercicio 3.1

Realizar un programa que some os números enteiros situados nunha zona de memoria determinada, excepto aquel que se atope na posición de memoria etiquetada como `este_non`. O resultado da suma almacenarase no rexistro `r4`.

Tras a execución do programa, a zona de memoria cos datos deberá continuar tendo os mesmos números enteiros que antes da execución do mesmo e no mesmo orde.

A zona de memoria conterá 9 números enteiros, e por tanto a posición de memoria etiquetada como `este_non` só poderá conter valores entre 1 e 9, ambos inclusive.

Como exemplo, supor que a zona de memoria contén os valores seguintes:

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

Solución

O labor do exercicio é ler os bytes da zona de memoria e sumalos un tras outro, omitindo o indicado pola posición de memoria etiquetada como `este_non`.

Para iso, utilizamos o direccionamento relativo a rexistro con desprazamento, onde o rexistro `r0` apunta ao principio da zona e o rexistro `r1` contén o desprazamento ou índice desde o principio da zona. A dirección efectiva de cada dato será entón `r0` + `r1`.

Ademais, utilizamos o rexistro `r2` para comprobar se terminamos de sumar todos os números e o rexistro `r7` para gardar o índice do elemento que temos que omitir.

b_cs1.s

```
1         .data
2   zona:   .byte 2,1,7,8,5,6,4,9,3
3   este_non: .byte 3
4         .equ tam, 9 @ tamaño da zona
5
6         .text
7   main: ldr  r0, =zona     @ r0: punteiro á zona
8         mov  r1, #0        @ r1: índice na zona
9         mov  r2, #tam      @ r2: tamaño da zona
10        ldr  r7, =este_non
11        ldrb r7, [r7]      @ r7: posición a omitir
12        mov  r4, #0        @ r4: acumulador para as sumas
13
14  bucle:
15        cmp  r1, r7        @ vemos se este sumar ou non
16        beq  estenon       @ se o índice en r1 é distinto
17        ldrb r3, [r0, r1]  @ do valor en r7, sumamos
18        add  r4, r4, r3    @ este número ao resto
19
20  estenon:
21        add  r1, r1, #1    @ apuntamos ao seguinte
```

```
22        cmp  r1, r2        @ vemos se acabamos
23        bne  bucle
24        wfi
25        .end
```

Exercicio 3.2

Realizar un programa que some os números enteiros de 1 byte que se atopan nunhas posicións de memoria determinadas, excepto aqueles que sexan menores que o contido da posición de memoria etiquetada como `lim`.

Os números enteiros atópanse nas posicións de memoria indicadas por unha táboa de direccións.

O resultado final deberá gardarse no rexistro `r4`.

Tras a execución do programa, a zona de memoria considerada deberá continuar tendo os mesmos números enteiros que antes da execución do mesmo e no mesmo orde.

Como exemplo supor que a táboa de direccións (etiquetada como `taboa`) onde están almacenados os números defínese da seguinte maneira:

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

Solución

Neste caso, a táboa de datos non contén os números a sumar, senón as direccións dos devanditos números. Para ler cada número a sumar, teremos que acceder dúas veces á memoria.

- A primeira vez, leremos unha dirección da táboa de direccións `taboa`. Para iso utilizaremos o rexistro `r0` como punteiro. O inicializaremos coa dirección de comezo da táboa e irémolo incrementando de 4 en 4 unidades en cada iteración (as direccións en

ARM ocupan 4 posicións de memoria). Almacenaremos en `r1` a dirección do dato lida da táboa.

- A segunda vez, utilizaremos a dirección obtida no paso anterior que temos no rexistro `r1` para ler o dato de 1 byte en si, que almacenaremos tamén en `r1`.

Logo, simplemente quédanos ir sumando os bytes lidos, sempre que o seu valor sexa maior que o indicado pola posición de memoria etiquetada como `lim`. Para iso, cargaremos devandito valor no rexistro `r3` e compararémolo co dato en `r1`.

Utilizaremos tamén o rexistro `r2` para comprobar se terminamos de sumar todos os números. Para iso o inicializaremos co tamaño da táboa, almacenado na dirección de memoria etiquetada como `size`.

b_cs2.s

```
      .data
b1:   .byte   4
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 @ límite para non sumar
      .balign 4
size: .word   9 @ tamaño da táboa
taboa:
      .word   b1, b2, b3, b4, b5, b6, b7, b8, b9


   .text
main:
   ldr  r3, =lim
   ldrb r3, [r3]   @ r3: limite das sumas

   ldr  r2, =size
   ldr  r2, [r2]   @ r2: tamaño da zona de contador

   ldr  r0, =taboa @ r0: punteiro á táboa de punteiros
   mov  r4, #0     @ r4: acumulador de sumas
bucle:
   cmp  r2, #0     @ vemos se final de táboa
   beq  final
```

```
32      sub  r2, r2, #1 @ actualizamos o contador
33
34      ldr  r1, [r0]   @ r1: punteiro na táboa de direccións
35      ldrb r1, [r1]   @ lemos o byte ao que apunta o punteiro
36      cmp  r1, r3
37      blt  outro      @ dato < r3: saltamos a suma
38      add  r4, r4, r1 @ actualizamos a suma
39  outro:
40      add  r0, r0, #4 @ actualizamos o punteiro á táboa
41      b    bucle
42
43  final:
44      wfi
45      .end
```

Exercicio 3.3

Realizar un programa que some os números enteiros que se atopan nunha zona de memoria determinada, excepto aqueles que:

- Sexan maiores que o contido da posición de memoria etiquetada como `lim`.
- Atópense en posicións de memoria maiores ou iguais que a dirección almacenada na posición de memoria etiquetada como `desde`.

Os números enteiros de 1 byte atópanse nas direccións de memoria indicadas por unha táboa de direccións de memoria, etiquetada como `taboa`, e o resultado final deberá gardarse na pila.

Tras a execución do programa, a zona de memoria considerada deberá continuar tendo os mesmos números enteiros que antes da execución do mesmo e no mesmo orde.

Para realizar o exercicio, utilice a información seguinte:

```
        .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
taboa: .word   b1, b2, b3, b4, b5, b6, b7, b8, b9
desde: .word   b3
```

Solución

Este exercicio é unha combinación dos dous exercicios anteriores.

b_cs3.s

```
       .data
b1:    .byte   2   @ táboa 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 do dato a omitir
       .balign 4

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

desde: .word   b3  @ posición desde a cal simar
size:  .word   9   @ tamaño da táboa

   .text
main:
   ldr  r3, =lim
   ldrb r3, [r3]   @ r3: limite para sumar o número
                   @ debe ser maior estrito que r3
   ldr  r2, =size
   ldr  r2, [r2]   @ r2: tamaño da zona contador

   ldr  r0, =taboa @ r0: punteiro á táboa de punteiros
   mov  r5, #0     @ r5: suma


   ldr  r4, =desde @ r4: metemos a dirección a partir
   ldr  r4, [r4]   @ da cal sumamos

bucle:
   cmp  r2, #0      @ vemos se percorremos toda a táboa
   beq  final
   sub  r2, r2, #1  @ actualizamos o contador

   ldr  r1, [r0]    @ r1: punteiro da táboa
```

```
41      cmp  r1, r4      @ comprobamos se dir. do dato >= r4
42      blt  outro       @ (dirección a partir da cal sumamos)
43
44      ldrb r1, [r1]    @ byte ao que apunta o punteiro
45
46      cmp  r1, r3
47      bls  outro       @ se dato <= r3 saltamos a suma
48      add  r5, r5, r1  @ actualizamos a suma
49  outro:
50      add  r0, r0, #4  @ actualizamos o punteiro á táboa
51      b    bucle
52
53  final:
54      push {r5}
55      wfi
56      .end
```

Exercicio 3.4

Considere unha zona de memoria etiquetada como `taboa`, que denominaremos *zona de punteiros* e que contén 3 *punteiros*, é dicir, as direccións de memoria nas que se atopan almacenados 3 números que consideraremos *datos*.

Realizar un programa que some aqueles datos que se atopen en direccións de memoria maiores ou iguais que o dato que conteñen, gardando o resultado da suma no rexistro `r4`.

Para realizar o exercicio, utilice a información seguinte:

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

táboa:
        .word d1, d2, d3
```

Solución

Neste exercicio tamén partimos dunha táboa de direccións (punteiros) que nos indican as posicións de memoria onde están almacenados os datos. Utilizaremos o rexistro `r1` para navegar pola táboa, o rexistro `r2` para almacenar as direccións dos datos lidas da táboa e o rexistro `r3` para almacenar os datos en si lidos a partir da dirección en `r2`.

A condición para sumar números que aparece no enunciado, tendo en conta os rexistros utilizados, será entón que `r2 >= r3`, é dicir, que a dirección de memoria sexa maior ou igual que o dato contido nela.

b_cs4.s

```
.data

@ Almacenamos os datos en memoria

d1:   .word   0x000023f0 @ sumamos este d2:
   .word   0x34234323
d3:   .word   0x0000dc0f @ e leste. Total 0x0000ffff

@ A continuación creamos a táboa de direccións

taboa:
   .word   d1, d2, d3
   .equ    tam, 12      @ táboa de 3 x 4 = 12 posicións

   .text
main:
   ldr   r1, =taboa  @ r1: punteiro á táboa de punteiros
   mov   r0, #3      @ r0: contador
   mov   r4, #0      @ r4: acumulador de sumas

bucle:
   cmp   r0, #0      @ comprobamos se terminamos
   beq   final
   ldr   r2, [r1]    @ r2: punteiro da táboa de punteiros
   ldr   r3, [r2]    @ r3: dato ao que se apunta desde a táboa
   cmp   r2, r3      @ comparamos o dato coa dirección
   bls   outro       @ se dir < dato non sumamos
   add   r4, r4, r3  @ se dir >= dato sumámolo
outro:
   add   r1, r1, #4  @ actualizamos punteiro á táboa
   sub   r0, r0, #1  @ actualizamos o contador
   b   bucle

final:
   wfi
   .end
```

3.2 Comparar números

De acordo co enfoque incremental deste manual, engadimos ao noso repertorio de actividades un conxunto de exercicios centrados na comparación

de valores en rexistros e na toma de decisións en función dos resultados da devandita comparación.

Exercicio 3.5

Considere unha zona de memoria que contén números enteiros positivos dun byte. Realizar un programa que faga as seguintes operacións, tendo en conta que consideraremos os números da zona de memoria en grupos de 3, que denominaremos X1, X2 e X3:

- Se X1 é igual a X2 e distinto de X3 entón sumarase o valor de X1 ao rexistro `r0`.
- Se X1 é distinto de X2 e igual a X3 entón o valor de X1 sumarase ao rexistro `r2`.
- Se X1 = X2 = X3 terminarase o programa.
- En calquera outro caso continúase coa iteración seguinte do programa.

En sucesivas iteracións, os números que ocupan a posición X2, pasarán a ocupar a posición X1, os que ocupan a X3 pasarán a ocupar a X2, e por último os que ocupan a posición seguinte á X3 pasarán a ocupar a X3. Os rexistros `r0` e `r2` deben inicializarse co valor 0.

Utilizar o seguinte código como exemplo da zona de memoria (hai que ter en conta que a cantidade de números enteiros na zona de memoria non se coñece a priori, aínda que obviamente dita cantidade deberá ser maior que 2):

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

Solución

Utilizaremos o rexistro `r1` como punteiro para navegar pola zona de memoria. Para acceder ao tres valores identificados como X1, X2 e X3 no enunciado, utilizaremos o direccionamento indirecto a rexistro con desprazamento utilizando como desprazamentos os valores 0 para X1, 1 para X2 e 2 para X3. En cada iteración, iremos incrementando o valor de `r1` nunha unidade, xa que os datos almacenados na zona de memoria son bytes.

Utilizaremos tamén os rexistros `r5`, `r6` e `r7` para almacenar os datos e facer as comparacións oportunas indicadas polo enunciado.

A zona de memoria non ten un tamaño predeterminado, senón que o final da zona identifícase porque hai tres valores iguais consecutivos.

b_cn1.s

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

      .text
main:
   ldr   r1,=zona  @ punteiro á zona de memoria
   mov   r0, #0    @ acumulador da suma 1
   mov   r2, #0    @ acumulador da 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) e 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) ou X(i) = X(i+2)
   b     itera

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

Exercicio 3.6

Considere unha zona de memoria que contén unha serie de números enteiros positivos dun byte. Realizar un programa que faga as seguintes operacións, tendo en conta que consideraremos os números da zona de memoria en grupos de 3, que denominaremos X1, X2 e X3:

- Se X1 = X2 = X3 sumarase o valor de X1 ao rexistro `r0` e continúase coa iteración seguinte do programa.

- Se X1 é distinto de X2 e X3 terminarase o programa.
- En calquera outro caso sumarase o valor de X2 e o de X3 ao rexistro `r2` e continúase coa iteración seguinte do programa.

En iteracións sucesivas X1 será o que na iteración anterior era X2, X2 o que na iteración anterior era X3, e X3 o que na iteración anterior era o seguinte a X3.

Utilizar o seguinte código como exemplo da zona de memoria (hai que ter en conta que a cantidade de números enteiros na zona de memoria non se coñece a priori, aínda que obviamente dita cantidade deberá ser maior que 2):

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

Solución

En canto á súa estrutura, este exercicio é similar ao exercicio anterior. Aumenta lixeiramente a complexidade en canto a comprobar a condición de terminación e as tarefas a realizar co tres valores consecutivos X1, X2 e X3.

b_cn2.s

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

   .text
main:
   ldr   r1,=zona   @ punteiro á zona de memoria
   mov   r0, #0     @ acumulador para a suma 1
   mov   r2, #0     @ acumulador para a 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
   beq   van2

   cmp   r7, r5       @ X(i) != X(i+1)
   bne   fin

outro:
   add   r2, r2, r6   @ calquera outro caso
   add   r2, r2, r5
   b     itera

```

```
25 van2:    cmp   r7, r5
26    bne   outro
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) e  X(i) != X(i+2)
35    .end
```

Exercicio 3.7

Considere unha zona de memoria que contén unha serie de números enteiros positivos dun byte. Realizar un programa que faga as seguintes operacións:

- Coller un número da zona de memoria, e nas sucesivas iteracións, en caso de habelas, os seguintes.
- Se ese número (ao cal chamaremos X) é menor que o seguinte (ao cal chamaremos Y) o programa debe intercambiar ambos os números e continuar cunha nova iteración, quedando gardado no rexistro `r5` o número de intercambios realizados até o momento.
- Se X é igual a Y e ao número de intercambios realizados até o momento, o programa debe terminar.
- Se non se cumpren ningunha das condicións anteriores o programa debe sumar X e Y ao contido do rexistro `r2` e continuar cunha nova iteración.

En cada nova iteración, Y pasará a ser X, e o número seguinte a Y pasará a ser Y.

Utilizar o seguinte código como exemplo da zona de memoria (hai que ter en conta que a cantidade de números enteiros na zona de memoria non se coñece a priori, aínda que obviamente dita cantidade deberá ser maior que 1):

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

Solución

Neste caso, ademais de realizar operacións con valores os consecutivos dunha táboa de números, teremos que intercambiar algúns dos devanditos valores.

Do mesmo xeito que nos exercicios anteriores, utilizaremos o direccionamento indirecto a rexistro con desprazamento para navegar pola táboa, co rexistro `r0` como rexistro índice.

b_cn3.s

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

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

bucle:
   ldrb  r3, [r0]     @ cargamos o numero X en r3
   ldrb  r4, [r0, #1] @ e o seguinte (Y) en r4
   cmp   r3, r4       @ comprobamos se X = Y
   blt   inter        @ X < E: intercambiar
   bne   seguir

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

   wfi                @ números iguais, e iguais
                      @ ao número de intercambios
inter:
   add   r5, r5, #1   @ un intercambio máis, xa que
   strb  r3, [r0, #1] @ é menor: intercambiamos
   strb  r4, [r0]

outro:
   add r0,r0,#1       @ incrementamos o punteiro
   b     bucle

seguir:
   add   r2, r2, r3   @ é maior: sumamos X e Y
   add   r2, r2, r4
   b     outro
   .end
```

Exercicio 3.8

Considere unha zona de memoria que contén unha serie de números enteiros de 16 bits maiores ou iguais que cero. Realizar un programa que faga as seguintes operacións:

- Tomar un número da zona de memoria. A ese número denominarémoslle X.
- Se X = 0 o programa termina.
- Se X é maior que o contido do rexistro `r7` o programa debe gardar o contido do rexistro `r7` en X (é dicir, na posición de memoria onde está X), contabilizar que se realizou un cambio e continuar coa seguinte iteración.
- Se X é menor ou igual que o contido do rexistro `r7` o programa debe gardar o contido do rexistro `r2` no rexistro `r7` e terminar.

No rexistro `r2` contabilizarase o número de cambios realizados, é dicir, o número de veces que se gardou o contido do rexistro `r7` na dirección de memoria onde se atopaba X.

En cada nova iteración, o número seguinte a X pasará a ser X. O rexistro `r2` debe inicializarse co valor 0 e o rexistro `r7` co valor 3.

Utilizar o seguinte código como exemplo da zona de memoria. Hai que ter en conta que a cantidade de números enteiros na zona de memoria non se coñece a priori.

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

Solución

Este exercicio é semellante ao anterior, só que os datos ocupan 16 bits (medias palabras) e as operacións a realizar son un pouco máis complexas.

Neste caso, as operacións de carga e almacenamento referiranse con medias palabras (**`ldrh`**, **`strh`**) no canto da bytes e teremos que incrementar o punteiro para navegar a táboa (o rexistro `r0` neste caso) en dúas 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: punteiro ao comezo da 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 ou igual que r7
21
22 @ X é maior que r7
23
24    strh  r7, [r0]
25    add   r2, r2, #1  @ contabilizamos o cambio
26    add   r0, r0, #2  @ incrementamos o punteiro
27    b     bucle       @ saltamos ao bucle
28
29 termina:
30    mov   r7, r5      @ movemos r2 a r7
31
32 final:
33    wfi
34    .end
```

Exercicio 3.9

Considere unha zona de memoria que contén unha serie de números de 32 bits enteiros maiores ou iguais que cero. Realizar un programa que faga as seguintes operacións:

- Coller un número da zona de memoria. A ese número denominarémoslle X e ao seguinte Y.
- Se X é maior que o contido do rexistro `r7`:
  - Se X é maior ou igual que Y, cópiase X a partir da dirección etiquetada como **`copia`** (a primeira vez, e en direccións seguintes, tantas veces como sexa necesario).
  - En caso contrario, se garda Y no rexistro `r6` .
- En caso contrario: Acábase o programa.

No rexistro `r2` contabilizarase o número de iteracións do programa (sen incluír cando se cumpre a condición de saída). No caso de que nunha iteración determinada o programa non acabe, procederase a realizar unha nova iteración, na cal Y pasará a ser X e o número seguinte a Y pasará a ser Y.

O rexistro `r7` debe inicializarse co valor 5.

Utilizar o seguinte código como exemplo da zona de memoria. Hai que ter en conta que a cantidade de números enteiros na zona de memoria non se coñece a priori.

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

Solución

Ademais do lixeiro aumento en complexidade das operacións a realizar, neste exercicio traballamos cunha táboa de palabras de 4 bytes. As operacións de carga e almacenamento dos datos referiranse a palabras (**`ldr`**, **`str`**) no canto de con medias palabras e teremos que incrementar o punteiro para navegar pola táboa (o rexistro `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: punteiro ao comezo da zona
11    ldr   r3, =copia  @ r3: punteiro á 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 se terminamos
18    ble   final       @ se X <= r7
19
20    ldr   r4, [r0,#4] @ cargamos Y en r4
21    cmp   r1, r4      @ X >= Y? r1 >= r4?
22    bge   copiar
23
```

```
@ X < E : gardamos Y en r6

   mov   r6, r4
outro:
   add   r2, r2, #1  @ incrementamos o contador,
   add   r0, r0, #4  @ incrementamos o punteiro
   b     bucle       @ e iteramos unha vez máis

@ X >= E

copiar:
   str   r1, [r3]    @ almacenamos o dato en add
   r3, r3, #4        @ a nova táboa, e incrementamos
   b     outro       @ o punteiro á nova táboa

final:
   wfi
   .end
```

3.3 Operacións con cadeas de caracteres

O conxunto de exercicios proposto a continuación céntrase na realización de operacións con cadeas de caracteres: contar caracteres, pasar de minúsculas a maiúsculas, comprobar características simples de grupos de caracteres, etc.

Exercicio 3.10

Realizar un programa que conte o número de caracteres nunha cadea terminada en 0. A cadea componse de caracteres ASCII de 7 bits (puntos Unicode U+0000 a U+007F).

Para probar o programa, utilice a cadea:

```
        .data
cadea: .asciz "Ola Mundo!"
```

Solución

A solución consiste en ir percorrendo a cadea de caracteres á vez que incrementamos un contador, ata que detectemos un carácter `\0`. Utilizamos o rexistro `r0` para percorre a cadea e o rexistro `r2` como contador.

cuentacad_ascii.s

```
   .data
```

```
3  cadea:
4     .asciz   "Ola Mundo!"
5     .balign  4
6  ncars:
7     .space   4
8
9     .text
10 main:
11    ldr  r0, =cadea   @ r0: punteiro á cadea
12    mov  r2, #0       @ r2: contador a 0
13
14 bucle:
15    ldrb r1, [r0]     @ lemos un carácter
16    cmp  r1, #0       @ comprobamos se chegamos ao final
17    beq  final
18
19    add  r2, r2, #1   @ incrementamos contador,
20    add  r0, r0, #1   @ incrementamos punteiro
21    b    bucle        @ e iteramos outra vez
22
23 final:
24    ldr  r0,=ncars    @ r0: punteiro a ncars
25    str  r1, [r0]     @ gardamos resultado
26    wfi
27    .end
```

Exercicio 3.11

Realizar un programa que conte o número de díxitos `5` que hai nun número representado por unha cadea de caracteres cos seus caracteres ASCII, almacenada a partir da dirección de memoria etiquetada como `cad`. O final da cadea vén determinado por un byte de valor 0. O resultado (un enteiro) deixarase na palabra de memoria etiquetada como `num`.

Para probar o programa, utilice a cadea:

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

Solución

Similar ao exercicio 3.10, só que unicamente contamos os caracteres `'5'`.

nde5.s

```
1
2       .data
3  cad:  .asciz "12546579"
```

```
4       .balign 4
5   num:  .space 4          @ dirección do resultado
6
7       .text
8   main:
9       mov r1,#0           @ r1: desprazamento pola cadea
10      ldr r0,=cad         @ r0: punteiro ao principio da cadea
11      mov r3,#0           @ r3: contador de coincidencias con '5'
12  bucle:
13      ldrb r2,[r0, r1]    @ lemos un carácter
14      cmp r2,#0           @ vemos se terminamos
15      beq termina         @ (a cadea termina con 0)
16
17      cmp r2,#'5'         @ vemos se é un '5'
18      bne next            @ se é un '5',
19      add r3,r3,#1        @ incrementamos o contador de '5's
20  next:
21      add r1,r1,#1        @ apuntamos ao seguinte
22      b bucle             @ e iteramos outra vez
23
24  termina:
25      ldr r0,=num         @ ro:  dirección do resultado
26      str r3,[r0]         @ almacenamos o resultado
27      wfi
28      .end
```

Exercicio 3.12

Realizar un programa que, dado un número natural representado por unha cadea de caracteres ASCII que empeza na dirección `cad` e termina cun byte de valor 0, indique se o número é capicúa. Se o é, deixarase un 1 na posición `capi`, e se non o é, deixarase un 0 na devandita posición de memoria.

Para probar o programa, utilice a cadea:

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

Solución

Imos percorrendo a cadea de caracteres numéricos simultaneamente desde o principio e desde o final, utilizando como punteiros os rexistros `r1` e `r2`. `r1` se inicializa coa dirección de memoria etiquetada como `cad` (apunta ao principio da cadea). No caso de `r2`, buscamos o final da cadea desde o principio desta, marcado cun byte de valor 0, e deixamos `r2` apuntando ao carácter anterior a devandito 0.

Se ao percorrer a cadea desde os dous extremos detectamos dous caracteres numéricos diferentes antes de que os punteiros atópense no medio da mesma, o número non será capicúa e terminamos con "`0`". No caso de que `r1` e `r2` atópense, terminariamos cun número capicúa, polo que o resultado será "`1`". Neste último caso temos dúas posibilidades, dependendo de que o número de díxitos sexa par ou impar.

- Que `r1` e `r2` acaben apuntando ao mesmo díxito (p. ex. en "`313`"). Por tanto, a condición de terminación será que `r1` e `r2` tomen o mesmo valor.
- Que `r1` e `r2` crúcense (p. ex. en "`3113`"). Neste caso, a condición de terminación será que `r1` pase a ser maior que `r2`.

En conxunto, a condición de terminación será que `r1 >= r2` ou de maneira equivalente que `r2 <= r1`.

ncap.s

```
1
2
3       .data
4  cad: .asciz   "123454321"
5
6       .balign  4
7  capi: .space   4
8
9
10    .text
11 main:
12    ldr   r1, =cad    @ r1: punteiro ao principio
13    mov   r2, r1      @ r2: punteiro ao final
14    ldr   r5, =capi   @ r5: punteiro a capi
15
16 buc1:
17    ldrb  r3, [r2]
18    cmp   r3, #0
19    beq   chegue
20    add   r2, r2,#1
21    b    buc1
22
23 chegue:              @ restamos 1 a r2 para que quede
24    sub   r2, r2, #1  @ apuntando ao último
25
26
27 buc2:
28    cmp   r2, r1      @ Comparamos os dous punteiros
29    bls   ecapi       @ Se r2 é menor igual a r1 é que
```

```
30                      @ coincidiron todos
31                      @ e crúzanse os punteiros,
32                      @ por tanto é capicua
33
34      ldrb  r3, [r1]    @ r3: número desde o inicio
35      ldrb  r4, [r2]    @ r4: número desde o final
36      cmp   r3, r4
37      bne   nonecapi    @ se son distintos, non é capicúa
38
39  @ se son iguais actualizamos os punteiros
40
41      add   r1, r1, #1
42      sub   r2, r2, #1
43      b     buc2
44
45  ecapi:
46      mov   r0, #1
47      str   r0, [r5]
48      wfi
49
50  nonecapi:
51      mov   r0, #0
52      str   r0, [r5]
53
54      wfi
55      .end
```

Exercicio 3.13

Realizar un programa que conte o número de caracteres dunha cadea terminada en 0. A cadea componse de caracteres UTF-8 de 1 ou 2 bytes (puntos Unicode U+0000 a U+00FF, cf. apéndice D). Téñase en conta que, por tanto, caracteres como "ñ" (`0xc2b1`) ou "í" (`0xc3ad`) contan como 1 carácter.

Para probar o programa, utilice a cadea:

```
        .data
cadea: .asciz "Bos días pequeno mundo!"
```

Solución

O exercicio é similar ao exercicio 3.10, salvo que temos caracteres de 1 e 2 bytes. Os caracteres UTF-8 de 2 bytes teñen como primeiro byte `0xc2` ou `0xc3` e o seu segundo byte ten o bit máis significativo a 1 (cf. apéndice D). Os caracteres de 1 byte teñen sempre o bit máis significativo a 0.

Por tanto, á hora de contar os caracteres da cadea, no caso de que leamos un byte co primeiro bit a 1 estaremos ante o primeiro ou o se-

gundo byte dun carácter de dous bytes, que teremos que contar só unha vez. Utilizamos o rexistro `r3` para indicar se na iteración anterior lemos un carácter co primeiro bit a 1. Ao ler un carácter co primeiro bit a 1 pódense dar dous casos:

- Se `r3 = 0`, trátase do primeiro carácter co primeiro bit a 1. Facemos `r3 = 1` e lemos o seguinte byte.
- Se `r3 = 1`, trátase do segundo carácter co primeiro bit a 1. Contamos un carácter máis, facemos `r3 = 0` e lemos o seguinte byte.

cuentacad_utf8.s

```
   .data
cadea:
   .asciz  "¡Bos días rapaciños!"
   .balign 4
ncars:
   .space  4

   .text
main:
   ldr  r0,=cadea  @ r0: punteiro á cadea
   mov r2, #0      @ r2: contador a 0
/*
  Os caracteres de dous bytes UTF-8 teñen o bit máis
significativo a un e están compostos por dous bytes,
ambos co bit máis significativo a 1.
*/
   mov r3, #0      @ r3: para indicar se o carácter
                   @ anterior era auxiliar
   ldr r4,=0x80    @ r4: máscara para detectar especiais
                   @ (teñen o primeiro bit a 1)
bucle:
   ldrb r1, [r0]
   cmp  r1, #0     @ comprobamos se chegamos ao final
   beq  final      @ da cadea

   tst  r1, r4     @ Comprobamos o bit máis significativo
   beq  ecar       @ se é cero, é un carácter de 1 byte

@ o bit máis significativo está a 1: carácter de 2 bytes

   cmp  r3, #0     @ comprobamos se no anterior tamén
   bne  ecar       @ se estaba, estamos ante un segundo byte UTF-8
   mov  r3, #1     @ en caso contrario é o primeiro byte dos dous
   b    outro      @ cambiamos o valor de r3 e iteramos de novo

```

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

outro:
   add  r0, r0, #1 @ incrementamos o punteiro
   b    bucle

final:
   ldr  r0, =ncars @ almacenamos o resultado.
   str  r2, [r0]
   wfi
   .end
```

Exercicio 3.14

Realizar un programa que pase a maiúsculas unha cadea terminada en 0. A cadea componse de caracteres ASCII de 7 bits (puntos Unicode U+0000 a U+007F).

Para probar o programa, utilice a cadea:

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

Solución

Imos percorrendo a carreira utilizando o rexistro `r0` como punteiro, ata que atopemos o byte de terminación de valor 0, saltándonos os caracteres que non sexan letras minúsculas, é dicir, todos aqueles cuxo código ASCII sexa menor que o código da `'a'` ou maior que o da `'z'`.

Os códigos ASCII das letras maiúsculas e minúsculas diferéncianse unicamente no seu bit de peso 5. No caso de que o carácter sexa unha letra minúscula, terá o seu bit de peso 5 a 1 (cf. cadro D.3 do apéndice D). Para pasala a maiúscula, simplemente pomos ese bit a 0.

amayusc.s

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

     .text
main:
     ldr   r0, =cad   @ r0: punteiro á cadea

@ A diferenza entre maiúsculas e minúsculas é que
@ as primeiras teñen o bit de peso 5 a 1
```

```
@ e as segundas a 0

     mov   r4, #0x20  @ usamos r4 como mascara

buc: ldrb  r1, [r0]
     cmp   r1, #0
     beq   final

     cmp   r1, #'a'   @ é menor que 'a'?
     blt   salta      @ se é así nolo saltamos

     cmp   r1, #'z'   @ é maior que 'z'?
     bgt   salta      @ se é así nolo saltamos
                      @ se non, pasamos a maiúsculas
     bic   r1, r4     @ pomos a 0 o bit de peso 5
     strb  r1,[r0]    @ e actualizamos en memoria

salta:
     add   r0, r0, #1
     b     buc

final:
     wfi
     .end
```

Exercicio 3.15

Realizar un programa que pase a maiúsculas unha cadea terminada en 0. A cadea componse de caracteres UTF-8 de 1 ou 2 bytes (puntos Unicode U+0000 a U+00FF, cf. apéndice D). Ten en conta que, por tanto, caracteres como "ñ" (`0xc2b1`) ou "í" (`0xc3ad`) contan como un carácter.

Para probar o programa, utilice a cadea:

```
      .data
cad:  .asciz "¡Bos días rapaciños!"
```

Solución

Este exercicio é similar ao 3.14, pero ademais temos que ter en conta as consideracións sobre os caracteres UTF-8 de dous bytes do exercicio 3.13. En calquera caso, do mesmo xeito que ocorre cos caracteres ASCII de 1 byte, a diferenza entre maiúsculas e minúsculas no caso dos caracteres alfanuméricos de dous bytes tamén está unicamente no seu bit de peso 5 do segundo byte (cf. cadro D.5 do apéndice D).

amayusc_utf8.s

```
1       .data
2  cad:  .asciz "Bos días pequeno mundo!"
3
4
5       .text
6  main: ldr  r0, =cad  @ punteiro ao comezo da cadea
7
8  @ A diferenza entre maiúsculas e minúsculas é que as
9  @ primeiras teñen o bit de peso 5 a 1 e as 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 o primeiro byte con 128.
17      cmp  r1, #128  @ Os cars. dun byte serán menores.
18                     @ Se é maior de 128, é o primeiro de
19      bcs  tendous   @ dous bytes.
20      cmp  r1, #'a'  @ En caso de ter un byte
21      bcc  outro     @ vemos se é minúscula
22      cmp  r2, #'z'
23      bhi  outro
24
25      bic  r1, r4
26      strb r1, [r0]
27
28 outro: add  r0, r0, #1
29      b    bucle
30 tendous:              @ Se o primeiro carácter é 0xC2
31      cmp  r1, #0xC2   @ é un carácter especial
32      beq  outro       @ de dous bytes.
33
34      add  r0, r0, #1 @ Lemos o segundo byte.
35      ldrb r1, [r0]
36
37      cmp  r1, #0xB7  @ Excepto o signo de dividir
38      beq  outro
39
40      cmp  r1, #0xBF  @ e o glifo e con diéresis
41      beq  outro
42                      @ no resto dos casos son
43      bic  r1, r4     @ letras 'especiais' susceptibles de
44                      @ pasar a maiúsculas
45      strb r1, [r0]
46      b    outro
47
48 final:
49      wfi
```

```
50          .end
```

Exercicio 3.16

Un palíndromo é unha palabra ou frase que se le igual nun sentido que noutro, por exemplo *A torre da derrota*. Realizar un programa que, dada unha cadea de caracteres ASCII que empeza na posición de memoria etiquetada como `frase` e terminada por un punto "`.`", indique se dita frase é un palíndromo ou non. Se é un palíndromo deixará unha "`S`" na posición de memoria etiquetada como `pal` e se non o é deixará unha "`N`" na devandita posición de memoria. No caso de que a frase non conteña ningunha letra, o programa deixará unha "`V`" en `pal`. A frase soamente conterá, ademais do punto que indica o final da mesma, caracteres ASCII de letras maiúsculas ou minúsculas e espazos en branco.

Para probar o programa, utilice a cadea:

```
      .data
frase: .ascii "A torre da derrota."
```

Solución

Este exercicio é moi similar ao exercicio 3.12, salvo que a cadea termina cun "." e pode ter espazos en branco. Neste caso, se a cadea non ten caracteres alfanuméricos, o programa terminará con "`V`". Se ao percorrer a frase desde os dous extremos detectamos dous caracteres diferentes antes de que os punteiros atópense no medio da frase, non se tratará dun palíndromo e terminamos con "`N`". No caso de que os punteiros (`r0` e `r1` neste caso) atópense, terminariamos cun palíndromo, polo que o resultado será "`S`". A condición de terminación será que `r0` >= `r1`.

capicua.s

```
1       .data
2
3   pal:   .byte      1
4   frase: .ascii "A torre dá derrota."
5   ese:   .byte 'S'     @ resultado se é palíndromo
6   ene:   .byte 'N'     @ resultado se non é palíndromo
7   uve:   .byte 'V'     @ resultado no caso de que non haxa letras
8          .equ punto,46 @ código ASCII do "."
9
10         .text
11
12  main: ldr  r0,=frase @ r0: punteiro desde o principio da frase
13        mov  r1,r0     @ r1: punteiro desde o final da frase
14        mov  r4,#0     @ r4: indicador de se hai letras
```

```
bucle1:
      ldrb r2,[r1]   @ lemos caracteres até chegar ao momento
      cmp  r2,#punto @ comprobamos se final da cadea
      beq  lerc1     @ xa chegamos ao final
      add  r1,r1,#1  @ actualizamos r2 se non chegamos ao final
      b    bucle1
lerc1:
      cmp  r0,r1
      bge  sioe      @ se os punteiros crúzanse acabamos
      ldrb r2,[r0]   @ lemos en r2 o carácter desde esquerda

      cmp  r2,#punto @ se chega ao final podería selo
      beq  sioe

      cmp  r2, #65   @ comparamos coa
      bmi  leoutroc1 @ non é unha letra
      cmp  r2, #91   @ Z é 90
      bmi  c1emai

      cmp  r2, #97   @ 'a' é 97
      bmi  leoutroc1
      cmp  r2, #123  @'z' é 122
      bmi  lec2      @ c1 é minúscula
c1emai:
      add  r2,r2,#32 @ pasámola a minúscula
      b    lec2
leoutroc1:
      add  r0,r0,#1
      b    lerc1
lec2:
      mov  r4,#1     @ hai polo menos unha letra
      sub  r1,r1,#1  @ r1 apuntando ao carácter anterior
      ldrb r3,[r1]

      cmp  r3,#65    @ comparamos coa
      bmi  lec2      @ non é unha letra
      cmp  r3,#91    @ 'Z' é 90
      bmi  c2emai    @ menor de 91 é maiúscula

      cmp  r3, #97   @ 'a' é 97
      bmi  lec2
      cmp  r3, #123  @'z' é 122
      bmi  compara   @ c2 é minúscula

c2emai:
      add  r3,r3,#32
compara:
      cmp  r2,r3
      bne  nonoe     @ non é palíndromo
```

```
64        add  r0,r0,#1
65        b    lerc1     @ se son iguais, a por o seguinte
66
67  nonoe:
68        ldr  r0,=ene   @ almacenamos unha 'N'
69        ldrb r1,[r0]   @ na dirección do resultado
70        ldr  r0,=pal
71        strb r1,[r0]
72
73        wfi
74
75  sioe: cmp  r4, #0
76        beq  baleira
77        ldr  r0,=ese   @   almacenamos unha 'S'
78        ldrb r1,[r0]   @ na dirección do resultado
79        ldrb r0,=pal
80        strb r1,[r0]
81
82        wfi
83
84  baleira:
85        ldr  r0, =uve  @ almacenamos unha 'V'
86        ldrb r1, [r0]  @ na dirección do resultado
87        ldrb r0, =pal
88        strb r1, [r0]
89
90        wfi
91        .end
```

3.4 Operacións aritmético-lóxicas

A continuación, propomos un conxunto de problemas centrados na utilización de maneira combinada das instrucións aritméticas, lóxicas e de desprazamento. Inclúense problemas para manexar táboas de bits, converter entre diferentes formatos de representación de información que requiren a realización de operacións de desprazamento e o traballo con máscaras, a realización de operacións aritméticas complexas a partir de operacións simples, ou o procesado de cadeas de caracteres.

Exercicio 3.17

Realizar un programa que mostre o factorial dun número N almacenado na palabra de memoria etiquetada como `input`, con $0 \leq N \leq 7$, buscándoo nunha táboa de factoriales. O programa almacenará o resultado na palabra de memoria etiquetada como `output`.

Para probar o programa, utilice as definicións seguintes:

```
      .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

O exercicio consiste basicamente en acceder a unha táboa de factoriais etiquetada como `factab` utilizando como índice o número do cal queremos calcular o seu factorial. Como cada palabra de memoria ocupa 4 bytes e en ARM a memoria organízase en bytes (cada byte ten a súa propia dirección), o factorial de `N` estará almacenado na palabra cuxa dirección é `factab + (N x 4)`. Para multiplicar por 4, desprazamos o número `N` dúas veces cara á esquerda coa instrución **`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: comezo da táboa
```

```
21    ldr   r1, =input   @  r1: dirección do dato de entrada
22    ldr   r1, [r1]     @  r1: dato de entrada N
23    lsl   r1, #2       @  convertemos N nun desprazamento (N x 4)
24    ldr   r2, [r0, r1] @  r2: factorial sacado da táboa
25    ldr   r0, =output  @  r0: dirección do resultado
26    str   r2, [r0]     @  almacenamos o resultado
27
28    wfi
29    .end
```

Exercicio 3.18

Realizar un programa que some dous números enteiros de 64 bits (2 palabras) almacenados en memoria nas posicións etiquetadas como `num1` e `num2`, deixando o resultado nunha zona de memoria etiquetada como `res`.

Para probar o programa, utilice as seguintes definicións:

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

Solución

Este exercicio ilustra a suma con carrexo. Primeiramente, sumamos as palabras menos significativas do número utilizando a instrución **add**, `0x9abcdef0` e `0x6543210f` segundo as definicións que propón o enunciado. A continuación, sumamos as palabras máis significativas xunto co carrexo da primeira suma (`0x12345678` e `0xedcba987`), utilizando a instrución **adc**.

Como en todas as arquitecturas *load/store*, as operacións aritméticas realízanse con valores almacenados en rexistros e o resultado depósitase nun rexistro. No noso caso, cargamos o primeiro número nos rexistros `r1` (parte menos significativa) e `r2` (parte máis significativa), e o segundo número nos rexistros `r3` e `r4`. O resultado da suma queda nos rexistros `r1` (parte menos significativa) e `r2` (parte máis significativa).

suma64.s

```
1
2       .data
3   num1: .quad   0x123456789abcdef0   @ valores a sumar
4   num2: .quad   0xedcba9876543210f   @
5   res:  .space  8                    @ espazo para o resultado
6
```

```
7     .text
8  main:
9     ldr   r0, =num1    @ dir do primeiro valor
10    ldr   r1, [r0]     @ primeira parte de n1 (menos sig.)
11    ldr   r2, [r0, #4] @ segunda parte de n1 (mais sig.)
12    ldr   r0, =num2    @ dir do segundo valor
13    ldr   r3, [r0]     @ primeira parte de n2
14    ldr   r4, [r0, #4] @ segunda parte de n2
15
16    add   r1, r1, r3   @ suma parte menos significativa (sen carrexo)
17    adc   r2, r2, r4   @ suma parte máis significativa (con carrexo)
18
19    ldr   r0, =res     @ almacenamos o resultado
20    str   r1, [r0]
21    str   r2, [r0, #4]
22
23    wfi
24    .end
```

Exercicio 3.19

Realizar un programa que divida un número enteiro positivo de 32 bits entre un número enteiro positivo de 16 bits, deixando o resultado nas palabras de memoria etiquetadas cono `coci` (valor do cociente da división) e `resto` (valor do resto). Os datos atópanse almacenados nas posicións de memoria etiquetadas como `dendo` (dividendo) e `dsor` (divisor). Para probar o programa, utilice as seguintes definicións:

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

Neste caso, o resultado depositado en `coci` será `0x25` e resto obtido en `resto` será `0x1000`.

Solución

Para realizar a división, utilizaremos o método das diferenzas sucesivas ou división euclídea:

```
int cociente;
int resto;
```

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

obténdose finalmente en `cociente` e `resto` os valores do cociente e do resto da división. No noso caso, implementaremos o algoritmo anterior utilizando como variables os rexistros `r0` (dividendo / resto), `r1` (divisor) e `r3` (cociente).

Teremos que ter en conta ademais que o dividendo é unha palabra de 4 bytes e o divisor unha 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 se dividimos por cero
19    beq   error
20 bucle:
21    cmp   r0, r1       @ ¿divisor menor que dividendo?
22    blt   fin          @ se o é, rematamos
23
24    add   r3, r3, #1   @ sumamos 1 ao cociente
25    sub   r0, r0, r1   @ restamos o divisor do 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 o resto
35    str   r3, [r7]     @ e o cociente
36
37    wfi
38    .end
```

Exercicio 3.20

Realizar un programa que, dado un número positivo de 32 bits en octal representado por unha cadea de caracteres ASCII terminada en cero e almacenada a partir da dirección `numoc`, transfórmeo a binario e almacéneo na posición etiquetada como `resul`. Compróbeo co seguinte exemplo.

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

O programa deberá almacenar na palabra etiquetada como `resul` o valor `03502570`.

Solución

O algoritmo para converter o número en octal $0d_1d_2\ldots d_n$ de octal a decimal é:

```
int num = 0;

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

onde d_i/`di` son os díxitos da representación en octal do número.

Por tanto, para resolver o problema convertemos os caracteres ASCII que representan os díxitos a números enteiros para obter os d_i/`di` e aplicamos o 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 da cadea
8     mov  r2,#0      @ r2: resultado (num no algoritmo)
9
10 outro:
11    ldrb r1,[r0]    @ r1: cada díxito dei
```

```
12      cmp  r1,#0      @ terminamos?
13      beq  listo
14
15      sub  r1,r1,#'0' @ pasamos di a binario
16      lsl  r2,#3      @ multiplicamos por oito
17      add  r2,r2,r1   @ e sumamos ao resultado (num)
18      add  r0,r0,#1   @ incrementamos o punteiro
19      b    outro      @ e pasamos ao seguinte.
20
21  listo:
22      ldr  r0,=resul  @ almacenamos o resultado
23      str  r2,[r0]
24
25      wfi
26      .end
```

Exercicio 3.21

Realizar un programa que separe as dúas metades dun byte, almacenado na posición de memoria etiquetada como `val` e deixe cada metade en cada un dos bytes dunha media palabra de 16 bits, etiquetada como `res`.

Para probar o programa, utilice as seguintes definicións:

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

Solución

Para resolver o problema, cargamos o byte almacenado en `val` en `r1` e pasamos os seus 4 bits máis significativos a `r3` con **`lsr`**. A continuación, deixamos eses 4 bits no seu lugar definitivo nunha media palabra con **`lsl`**. Finalmente, deixamos en `r1` os 4 bits menos significativos e engadímoslle o que temos en `r3`. Así, os 16 bits menos significativos de `r1` terán o resultado esperado, que almacenamos en `res`.

nibbles.s

```
1
2       .data
3   val: .byte   0x6c
4        .balign 2
5   res: .space  2
6
7       .equ   mask,  0x000f  @ máscara para aillar un nibble
8       .equ   nlsb,  0x04    @ contador para mover o primeiro nibble
9       .equ   bdesp, 0x08    @ contador para mover o segundo nibble
```

```
10
11      .text
12  main:
13      ldr   r1, =val
14      ldr   r1, [r1]        @ dato a separar
15      lsr   r3, r1, #nlsb   @ pasamos o primeiro nibble a r3
16      lsl   r3, r3, #bdesp  @ movémolo un byte
17
18      mov   r4, #mask       @ cargamos a máscara
19      and   r1, r1, r4      @ deixamos o segundo nibble
20      orr   r1, r1, r3      @ engadímolos a r4
21
22      ldr   r2, =res
23      str   r1, [r2]        @ almacenamos o resultado
24
25      wfi
26      .end
```

Exercicio 3.22

Realizar un programa que, dado un número almacenado na posición de memoria etiquetada como `num`, conte a cantidade de ceros que ten a representación binaria dese número e deixe o resultado na palabra etiquetada como `ncero`. Compróbeo co seguinte exemplo.

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

Solución

Cargamos o número almacenado en `num` no rexistro `r1` e desprazámolo bit a bit á dereita con **`asr`**, de maneira que o bit menos significativo pasa ao bit de carrexo `C`. Logo, utilizamos a instrución de salto condicional **`bcs`** (saltar se o indicador `C` está activado) para incrementar ou non un contador co número de ceros (rexistro `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 desprazamentos
9       mov r3,#0     @ contador de ceros
10      ldr r1,[r0]
```

```
11 outro:
12    asr r1,#1     @ desp. a dereitas (o bit que sae vai a C)
13    bcs noncero   @ se saíu un un, pasamos
14    add r3,r3,#1  @ incrementamos o contador de ceros
15 noncero:
16    sub r2,r2,#1  @ decrementamos o contador de desp.
17    bne outro
18
19    ldr r0,=ncero @ almacenamos o resultado
20    str r3,[r0]
21
22    wfi
23    .end
```

Exercicio 3.23

Realizar un programa que active un bit dunha matriz de 8×8 bits serializada por filas. A matriz estará almacenada nunha zona de memoria composta por 8 bytes a partir da dirección de memoria etiquetada como `matriz`. O programa recibirá en dous bytes etiquetados como `col` e `fila` dous números comprendidos entre 0 e 7 que indican respectivamente as coordenadas (columna, fila) do bit a activar. As filas e columnas numéranse do bit máis significativo do byte ó menos significativo (columnas) e da dirección máis baixa á máis alta (filas).

Para probar o programa, utilice as seguintes definicións:

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

Solución

Unha matriz é unha estrutura bidimensional formada por filas e columnas, mentres que a memoria é lineal e por tanto unidimensional. Por tanto, á hora de almacenar unha matriz en memoria teremos que adoptar un convenio para facelo de maneira consistente. As dúas opcións dispoñibles serían almacenar as filas de maneira consecutiva en memoria, unha detrás da outra, ou almacenar as columnas de maneira consecutiva. No primeiro caso dicimos que a matriz está almacenada en memoria serializada por filas e no segundo que está serializada por columnas.

Serializar unha matriz bidimensional consiste en último termo en converter dita matriz nun vector unidimensional. No caso dunha matriz serializada por filas, supondo que o primeiro elemento da matriz é o de coordenadas $(0, 0)$ e que o primeiro elemento do vector é o de índice 0, o elemento de coordenadas (i, j) corresponderase co elemento do vector de índice $i \times d + j$, onde d é o número de columnas da matriz. Por exemplo, nunha matriz 8 x 8 ($d = 8$), o índice do elemento de coordenadas $(0, 7)$ tomará o valor 7, o índice do elemento de coordenadas $(1, 7)$ tomará o valor 15 e o índice do elemento de coordenadas $(7, 7)$ tomará o valor 63.

Neste caso, as dimensións da matriz (8 x 8) coinciden co tamaño das posicións de memoria do computador (8 bits), co que o elemento de coordenadas (i, j) corresponderase co bit j-ésimo do byte i-esimo da matriz serializada a partir da posición etiquetada como `matriz`.

tablabit.s

```
        .data
col:    .byte   0
fila:   .byte   0
        .balign 4
matriz: .space  8

   .text
main:
   ldr  r0,=matriz @ ro: dirección da táboa
   ldr  r1,=fila   @ r1: dir. do número de fila
   ldr  r2,=col    @ r2: dir. do número de columna
   ldrb r1,[r1]    @ r1: número de fila
   ldrb r2,[r2]    @ r2: número de columna

   mov  r3,#0x80   @ r3: máscara para seleccionar
   lsr  r3,r3,r2   @ a columna (0 = máis significativo)
   ldrb r4,[r0,r1] @ cargamos a fila
   orr  r4,r4,r3   @ activamos o bit
   strb r4,[r0,r1] @ e almacenamos o resultado

   wfi
   .end
```

Exercicio 3.24

Existe unha matriz de 4×16 bits almacenada a partir da posición etiquetada como `array`. A matriz almacénase serializada por filas. Escriba un programa que, dada unha fila (entre 0 e 3), unha columna (entre 0 e 15) e un valor (0 ou 1), actualice a matriz de maneira acorde a eses datos.

Por exemplo, dada a 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 e 3
col:    .word 12      @ columna entre 0 e 15
val:    .byte 1       @ novo valor, 0 ou 1
```

O programa actualizará o sexto byte da matriz (o que contén o bit da columna 12) co novo valor `0b0000 1000` = `0x08`.

Solución

No exercicio 3.23 describimos o proceso de serialización da matriz e a maneira de calcular o índice sobre a matriz serializada, neste caso almacenada a partir da dirección `array`. Para resolver o exercicio, primeiro calculamos o índice do bit a modificar, de acordo coa fórmula do exercicio 3.23:

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

Unha vez que calculamos o índice, buscamos o byte onde se atopa o bit a modificar. Para iso dividimos o índice por 8. O cociente da división indicaranos o índice do byte en `array` e o resto o desprazamento dentro de devandito byte, contando de esquerda a dereita.

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 e 3
11  | col:  .word 12      @ columna entre 0 e 15
12  | val:  .byte 1       @ novo valor, 0 ou 1
13  |
14  |       .text
15  |
```

```
16        @ Calculamos o índice do 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 do bit na matriz serializada
26
27        @ Calcula o byte que ten o bit a modificar (b_mod)
28        @ índice  = b_mod * 8 + offset
29        @ (contando de esquerda a dereita)
30
31        mov r1,r0        @ r1: índice
32        lsr r0,#3        @ r0: cociente de dividir por oito
33        mov r2,#0b111
34        and r1,r2        @ r1: resto da división
35
36        ldr r2,=#0x80    @ máscara inicial para activar ou desactivar
37        lsr r2,r1        @ despraza a máscara ao bit obxectivo
38
39        ldr r5,=array
40        ldrb r6,[r5,r0]  @ carga o byte obxectivo (co offset)
41        ldr r7,=val
42        ldr r7,[r7]
43        cmp r7,#0        @ activar ou desactivar?
44        beq set0
45
46  set1: orr r6,r2       @ activamos usando a máscara
47        b next
48
49  set0: mvn r2,r2
50        and r6,r2       @ desactivamos usando a máscara inversa
51
52  next: strb r6,[r5,r0] @ substituímos o byte orixinal
53
54        wfi
55       .end
```

Exercicio 3.25

Realizar un programa que compare dúas cadeas de caracteres terminadas con 0, almacenadas en memoria a partir das posicións etiquetadas como `cad1` e `cad2`, deixando o resultado da comparación na posición de memoria etiquetada como `res`. O programa ofrecerá como resultado o valor `0xff`

no caso de que as cadeas sexan diferentes, e o valor `0x00` no caso de que sexan iguais.

Para probar o programa, utilice as cadeas seguintes:

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

Solución

Imos percorrendo as cadeas utilizando o rexistro `r0` como punteiro para percorrer a primeira cadea e o rexistro `r1` para percorrer a segunda. Comparamos dous a dous os caracteres de ambas as cadeas, almacenados respectivamente nos rexistros `r2` e `r3`.

As condicións de terminación son:

- Que os caracteres lidos de ambas as cadeas sexan diferentes. Neste caso, as cadeas serán diferentes.
- Que unha das cadeas termine, é dicir, que o byte lido tome o valor 0. Neste caso, se a outra tamén termina, as cadeas serán iguais, e se non serán diferentes.

comp_cad.s

```
1
2       .data
3  cad1: .asciz "Ola Mundo!"
4  cad2: .asciz "Ola Mundo!"
5  res:  .space 1
6
7     .text
8  main:
9     ldr r0, =cad1    @ r0: punteiro á primeira cadea
10    ldr r1, =cad2    @ r1: punteiro á segunda cadea
11
12 buc:
13    ldrb r2, [r0]    @ lemos un carácter da primeira cadea
14    cmp r2, #0       @ se é 0, terminamos
15    beq finc1
16
17 @ o carácter da primeira cadea non é a cero
18
19    ldrb r3, [r1]    @ lemos un carácter da segunda cadea
20    cmp r3,#0        @ se é cero, terminamos
21    beq dif          @ e as cadeas son diferentes
```

```
22
23  @ quedan caracteres en ambas as cadeas
24
25      cmp r2, r3        @ se son diferentes, terminamos
26      bne dif           @ e as cadeas son diferentes
27
28  @ os caracteres seguen sendo iguais,
29  @ seguimos cos seguintes caracteres de ambas as cadeas
30
31      add r0, r0, #1
32      add r1, r1, #1
33      b buc
34
35  @ primeira cadea finalizada
36
37  finc1:
38      ldrb r3, [r1]     @ lemos un carácter da segunda cadea
39      cmp r3, #0        @ se é 0, terminamos
40      bne dif           @ e as dúas cadeas son iguais
41
42  gres:
43      ldr r1, =res      @ almacenamos o resultado da
44      strb r2, [r1]     @ comparación
45      wfi
46
47  @ as cadeas son diferentes
48
49  dif:
50      mov r2, #0xff     @ resultado para cadeas diferentes
51      b gres
52
53  @ as cadeas son iguais
54
55  igual:
56      mov r2, #0        @ resultado para cadeas iguais
57      b gres
58      .end
```

Exercicio 3.26

Realizar un programa para calcular a representación en código ASCII do valor en hexadecimal dun byte almacenado na posición de memoria etiquetada como `val`. Teña en conta de que o resultado se representará con dous díxitos hexadecimais (0 a 9 e letras da A a F). O resultado almacenarase en 2 bytes a partir da posición etiquetada como `res`.

Para probar o programa, utilice as seguintes definicións:

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

Solución

Cada un dos dous díxitos hexadecimais dun byte se codifica con 4 bits, é dicir, cun *nibble*. Convertemos aos caracteres ASCII correspondentes cada un dos nibbles, primeiro o menos significativo e logo o máis significativo.

Para converter un valor de 4 bits ao carácter ASCII que representa o seu valor en hexadecimal, primeiro comprobamos se o número está entre `0x0` e `0x9` ou entre `0xA` e `0xF` (cf. cadro D.3 do apéndice D):

- No primeiro caso, para converter o número ao carácter que o representa teremos que sumarlle o código ASCII do carácter `'0'`, é dicir, teremos que sumar o valor `0x30`.
- No segundo caso, para realizar a conversión teremos que sumar `0x37`. Por exemplo, `0xA + 0x37 = 0x41`, que é o código ASCII do carácter `'A'`.

hextoascii.s

```
1
2       .data
3   val: .byte  0xa7 @ dato de exemplo
4   res: .space 2     @ resultado
5
6       .text
7   main:
8       mov   r2, #0x0f     @ enmascaramos todo menos o nibble
9       ldr   r0, =val      @ menos significativo
10      ldrb  r1, [r0]      @ cargamos o byte en r0
11      and   r1, r1, r2
12      cmp   r1, #0x09     @ vemos se '0' - '9' ou 'A' - 'Z'
13      bhi   maior1
14
15      add   r1, r1, #'0'  @ menor que 9
16      b     dig2
17
18  maior1:
19      add   r1, r1, #0x37 @ maior que 9 : 0x0A + 0x37 = 0x41 = 'A'
20
21  dig2:
22      ldr   r3, =res
23      strb  r1, [r3, #1]  @ almacenámolo no seu sitio
```

```
@
@ facemos o mesmo co nibble (grupo de 4 bits)
@ máis significativo.
@
   ldrb  r1, [r0]
   lsr   r1, r1, #4    @ eliminamos a parte xa convertida
   cmp   r1, #0x09
   bhi   maior2

   add   r1, r1, #0x30
   b     fin

maior2:
   add   r1, r1, #0x37

fin:
   strb  r1, [r3]    @ almacenámolo no seu sitio

   wfi
   .end
```

Exercicio 3.27

Temos unha lista de caracteres ASCII almacenados nun bloque de memoria etiquetado como `cad`. O tamaño da lista de caracteres está indicada polo número almacenado na posición de memoria etiquetada como `cant`. Realice un programa que:

1. Conte a cantidade de caracteres numéricos que se atopan na devandita lista, deixando o resultado nunha posición etiquetada como `nums`.
2. Realice a suma dos valores dos devanditos caracteres numéricos, previamente convertidos a números enteiros, deixando o resultado na posición etiquetada como `result`. No caso de que a lista non teña ningún carácter numérico o resultado da suma debe ser 0.

Tanto `nums` como `result` son palabras de memoria de 32 bits.

Por exemplo, dadas as definicións seguintes, o resultado almacenado na posición `nums` será 5 e o resultado da suma transferido á posición `result` será 10.

```
        .data
cant:   .word   17
nums:   .space   4
result: .space   4
cad:    .asci    "H014 Mu'{n!!dO23"
```

Solución

Imos percorrendo a cadea de caracteres utilizando o rexistro `r1` como punteiro ao longo da mesma. En cada iteración, lemos un carácter en `r0`. Utilizamos `r5` como contador de caracteres restantes, `r3` como contador de caracteres numéricos e `r2` como acumulador para as sumas parciais. Daremos os pasos seguintes ata que o contador de caracteres restantes chegue a 0.

- No caso de que teñamos un carácter non numérico, o saltamos: incrementamos o punteiro, decrementamos o contador de caracteres restantes e pasamos á seguinte iteración.
- No caso de que teñamos un carácter numérico, actualizamos o contador de números en `r2`, o convertemos a número e actualizamos a suma en `r2`. Finalmente facemos como no caso anterior (actualizamos punteiro e contador e pasamos á seguinte iteración).

Ao chegar ao final da lista, almacenamos os resultados en cadansúa palabra de memoria.

contasuma.s

```
        .data
cant:   .word 17
nums:   .space 4
result: .space 4
cad:    .ascii "H014 Mu‘ {n!!dO23"

      .text
contasuma:
      ldr   r4, =cant   @ r4 puntero a cant
      ldr   r5, [r4]    @ inicializamos r5 coa cantidade de caracteres
      ldr   r1, =cad    @ r1 puntero á cadea
      mov   r2, #0      @ inicializamos o resultado da suma
      mov   r3, #0      @ inicializamos contador de caracteres numéricos
bucle:
      cmp   r5, #0
      beq   final       @ se é cero chegamos ao final
      ldrb  r0, [r1]    @ usamos r0 para ler caracteres
      cmp   r0, #’0’
      bmi   segue       @ se é menor non é carácter numérico
      cmp   r0, #’9’
      bgt   segue       @ se é maior non é carácter numérico
      add   r3, r3, #1  @ actualizamos contador
      sub   r0, r0, #48 @ pasámolo a número
      add   r2, r2, r0  @ actualizamos a suma
```

```
25 segue:
26       add   r1, r1, #1  @ actualizamos punteiro á cadea
27       sub   r5, r5, #1  @ actualizamos cantidade de caracteres que faltan
28       b     bucle
29 final:
30       ldr   r1, =nums   @ actualizamos punteiro a nums
31       str   r3, [r1]    @ gardamos o contador
32       ldr   r1, =result @ actualizamos punteiro a result
33       str   r2, [r1]    @ gardamos o resultado da suma
34       wfi
```

Exercicio 3.28

Realizar un programa que multiplique dous números de 32 bits almacenados nas palabras de memoria etiquetadas como `num1` e `num2`, deixando o resultado (64 bits) na dobre palabra etiquetada como `res`.

Para probar o programa, utilice as seguintes definicións:

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

Solución

Cargamos os números a multiplicar en rexistros, separando as súas metades altas e baixas, para poder utilizar a instrución **`mul`** de Thumb que multiplica medias palabras de 16 bits:

- `num1` cárgase en (os 16 bits menos significativos) de `r5` e `r0`.
- `num2` cárgase en (os 16 bits menos significativos) de `r3` e `r1`.

Facemos ademais unha copia dos 16 bits menos significativos de `num2` en `r7` para facilitar a multiplicación.

A continuación, aplicamos directamente o método de multiplicación seguinte (`num1` = `{r5r0}` e `num2` = `{r3r1}` ou `{r3r7}`):

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

Quedando o resultado nos rexistros `{r3r0r7}`.

Ao realizar a suma `(r5xr1+r3xr0)` temos que ter en conta os posibles acarreos que se produzan.

Para os datos do enunciado (`0x1122 3344` x `0x5566 7788`) o resultado é `0x05b7 36a6 a0117 d820`.

mul32.s

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

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

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

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

   mov   r7, r1        @ r7: copia da metade baixa de num1

/*

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

O 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 se temos que trasladar
```

```
42                          @ un carrexo
43      ldr   r6, =carry    @ e se é así,
44      add   r3, r3, r6    @ engadímolo no lugar correcto
45
46  saltar:
47      lsl   r4, r0, #16   @ colocamos no seu sitio as
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 máis significativos
51
52      ldr   r6, =res      @ dirección do resultado
53      str   r7, [r6]      @ almacenamos os 32 bits menos sig.
54      str   r3, [r6, #4] @ e os 32 bits máis significativos
55
56      wfi
57      .end
```

Exercicio 3.29

Realizar un programa que transforme un número enteiro de 32 bits almacenado a partir da posición de memoria etiquetada como `num` na súa representación en punto flotante normalizada, con base 2, normalización enteira, cunha mantisa de 31 bits, 1 bit para o signo (bit máis significativo), representación con signo e magnitude para a mantisa para os números negativos e unha característica de 8 bits en exceso de 128, é dicir $C = E + 128$ onde C é a característica e E o expoñente. O número 0 represéntase cunha mantisa e unha característica de valor 0.

Por exemplo, o número `0x1c00abba` representarase como `0x7002aee8 x 2exp-2` (signo e mantisa SM = `0x7002aee8` e característica C = -2 + 128 = 126 = `0x7e`) e o número enteiro `0x0000abba` como `0x55dd0000 x 2exp-15` (signo e mantisa SM = `0x55dd0000` e característica C = -15 + 128 = 113 = `0x71`), onde `exp` representa o operador exponenciación.

No caso do número negativo `-3000`, cuxa representación en complemento a 2 é `0xfffff448` e cuxo valor absoluto en hexadecimal é `0xbb8`, representarase como `0x-5dc00000 x 2exp-19` (signo e mantisa SM = `0xddc00000` e característica C = -19 + 128 = 109 = `0x6d`)

O programa almacenará o signo e a mantisa a partir da posición de memoria etiquetada como `mant` e a característica na posición etiquetada como `car`.

Para probar o programa, utilice as seguintes definicións:

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

Solución

Se o número orixinal é positivo, cargámolo no rexistro `r0` e ímolo desprazando á esquerda ata que o bit de signo cambia a `1`, á vez que incrementamos un contador en `r1`. Cada vez que desprazamos unha unidade á esquerda multiplicariamos devandito número por dous, co que o rexistro `r1` representa o número de multiplicacións realizadas ata que o bit máis significativo da representación binaria do número pasa a ser `1`.

No momento en que o bit de signo pase de `0` a `1`, teremos:

- En `r1`, o valor do expoñente da representación en punto flotante normalizada cambiado de signo (p. ex. 3 desprazamentos significan que temos que multiplicar o resultado final por 2^{-3} para recuperar o enteiro positivo orixinal).
- En `r0`, o valor da mantisa, sen o bit máis significativo, que pasou ao bit de signo.

Neste punto só nos queda:

- Calcular a característica C como `-r1 + 128` (é dicir, $C = E + 128$. Para iso restamos de 128 do valor do expoñente cambiado de signo que temos en `r1` utilizando a instrución **`sub`**, deixando o resultado en `r1`.
- Recuperar o bit perdido que pasou ao bit de signo e o bit de signo orixinal (que debe de ser `0`, xa que supomos que o número era positivo). Para iso utilizamos a instrución **`lsr`**.

Se o número orixinal é negativo, tomamos nota do signo no rexistro `r2`, calculamos o seu valor absoluto ou magnitude utilizando a instrución **`neg`** e normalizamos segundo os pasos anteriores. Ao completar a normalización, engadimos o bit de signo correcto á mantisa para obter a representación requirida de signo e magnitude.

Cos datos das definicións do exercicio, o signo S é `0`, a mantisa M é `0x7bbaacc0` e a característica C é `0x7c`. Podemos probar tamén co resto dos exemplos do enunciado.

normaliza.s

```
        .data
num:    .word   0x07bbaacc      @ número de exemplo
mant:   .space  4               @ mantisa e signo
car:    .space  1               @ característica
        .equ signo, 0x80000000  @ signo negativo

    .text
main:
    ldr   r0,=num
    ldr   r0, [r0]      @ r0: número a normalizar
    mov   r1, #0        @ r1: para acumular o expoñente
    mov   r2, #0        @ r2: signo (0 = positivo)
    cmp   r0, #0        @ comprobamos se o número é cero
    beq   nada          @ se o é, terminamos
    bgt   bucle         @ comprobamos se é positivo

    ldr   r2, =signo    @ se é negativo, gardamos o signo
    neg   r0, r0        @ e normalizamos o valor absoluto

bucle:
    lsl   r0, r0, #1    @ movemos os bits á esquerda
    blt   final         @ saíu un 1 por Z -> acabamos
    add   r1, r1, #1    @ incrementamos en 1 o exp.
    b     bucle         @ e seguimos buscando

final:
    lsr   r0, r0, #1    @ recuperamos o 1 perdido por Z
    orr   r0, r0, r2    @ pomos o signo correcto
    mov   r2, #128
    sub   r1, r2, r1    @ característica: -r1 + 128

nada:
    ldr   r3, =mant     @ almacenamos signo e mantisa
    str   r0, [r3]      @ e característica
    strb  r1, [r3, #4] @ (0's se orixinalmente num = 0)

    wfi
    .end
```

Exercicio 3.30

Dada unha lista de caracteres ASCII almacenados nun bloque de memoria etiquetado como `cad`, realizar un programa que converta todas as palabras (entendidas como secuencias de letras separadas por caracteres ASCII non alfabéticos) de tal modo que a primeira letra sexa maiúscula e o resto de letras

consecutivas sexan minúsculas, deixando o resultado no bloque de memoria etiquetado como `conv`. O final da cadea delimítase cun cero.

Por exemplo, dadas as definicións seguintes, o resultado almacenado a partir da posición `conv` será a cadea:

```
Ola Mundo Cruel: []Vaia Par De2Xemelgos{}
```

```
      .data
cad:  .asciz "ola mundo CRUEL: []vaia PAR de2xemelgos{}"
      .balign 4
conv: .space  64
```

Solución

Imos percorrendo a cadea de caracteres utilizando o rexistro `r0` como punteiro ao longo da mesma. En cada iteración, lemos un carácter en `r1`.

- No caso de que sexa un carácter alfabético ao principio dunha secuencia de caracteres alfabéticos (é dicir, que o carácter anterior non era alfabético), pasámolo a maiúscula.
- No caso de que sexa un carácter alfabético dentro dunha secuencia de caracteres alfabéticos (é dicir, que o carácter anterior era alfabético), pasámolo a minúscula.
- No resto dos casos, tratarase dun carácter non alfabético e por tanto deixámolo como está.

Utilizamos o rexistro `r2` para indicar se nesta iteración estamos ao principio dunha secuencia de caracteres alfabéticos. Se (`r2` = 0) indicará que estamos ao principio dunha secuencia de caracteres alfabéticos ou palabra.

- Se (`r2` = 0) e este carácter é alfabético, facemos (`r2` = 1). Xa non estamos a principio de palabra.
- Se (`r2` = 0) e este carácter non é alfabético, non facemos nada e deixamos `r2` como está.
- Se (`r2` = 1) e este carácter é alfabético, seguimos dentro dunha palabra, polo que deixamos `r2` como está.

- Se (r2 = 1) e este carácter non é alfabético, facemos (r2 = 0). Terminouse a palabra e o seguinte carácter alfabético que apareza será principio de palabra.

capita.s

```
      .data
cad:  .asciz "ola mundo CRUEL: []vaia PAR de2xemelgos{}"
      .balign 4
conv: .space  64 @ espazo para almacenar a cadea convertida

   .text
main:
   mov  r2,#0    @ r2: indica se principio de palabra (r2 = 0)
   mov  r4,#0xdf @ r4: máscara para pasar a maiúsculas
   mov  r5,#0x20 @ r5: máscara para pasar a minúsculas
   ldr  r3,=conv @ r3: punteiro ao destino
   ldr  r0,=cad  @ r0: punteiro ao principio da cadea
bucle:
   ldrb r1,[r0]  @ cargamos un carácter en r1
   cmp  r1,#0    @ vemos se terminamos
   beq  listo    @ (a cadea termina cun 0)

   cmp  r1,#'A'  @ filtramos os caracteres non alfanuméricos
   bmi  initp    @ os menores que 'A' non son alfanuméricos
   cmp  r1,#'z'  @ os maiores que 'z' tampouco
   bpl  initp    @ agora só quedan entre 'A' e 'z'
   cmp  r1,#'Z'  @ os menores que 'Z' son letras maiúsculas
   ble  elet     @ os do medio non son alfanuméricos
   cmp  r1,#'a'  @ os maiores que 'a' son letras minúsculas
   bmi  initp

elet:
   cmp  r2,#0    @ trátase dunha letra
   bne  aminus   @ vemos se principio de palabra (r2 = 0)

   and  r1,r1,r4 @ convertemos a maiúscula
   mov  r2,#1    @ xa non é principio de palabra (r2 = 1)
   b    outro

aminus:
   orr  r1,r1,r5 @ non estamos a principio de palabra
   b    outro    @ convertemos a minúscula

initp:
   mov  r2,#0    @ vimos dun carácter non alfabético
                 @ segu. letra: principio de palabra (r2 = 0)
outro:
   strb r1,[r3]  @ almacenamos o carácter
   add  r0,r0,#1 @ e apuntamos ao seguinte
```

```
45      add  r3,r3,#1 @ en orixe e destino
46      b    bucle    @ seguinte carácter
47
48  listo:
49      strb r1,[r3]  @ almacenamos o cero final
50      wfi
51      .end
```

Exercicio 3.31

O obxectivo deste exercicio é realizar un programa que divida un número enteiro de 32 bits entre 10 . Xa coñecemos o método da división euclídea (cf. exercicio 3.19). Neste exercicio imos aproveitar o feito de que

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

e por tanto

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

Se facemos $q = N/2^1 + N/2^2$, temos 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 nun computador a precisión de N é limitada (p. ex., un número de 32 bits), se tomamos suficientes termos da aproximación conseguiremos un valor o suficientemente preciso como para non cometer erros no resultado. No caso dun número de 32 bits, cos termos indicados na última expresión é suficiente para cometer un erro de como máximo unha unidade.

Por tanto, podemos utilizar o algoritmo seguinte para dividir un enteiro por 10, considerando que cada desprazamento á dereita dun enteiro divide devandito enteiro por 2 (o operador **>>** é o desprazamento á dereita):

```
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 o resto como num - coci x 10 = num - coci x (4 + 1) x 2

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

// axustamos cociente e resto se o resto é maior que 9

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

}
```

Realice un programa que divida un enteiro de 32 bits entre 10, utilizando o algoritmo descrito e as definicións seguintes:

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

Solución

A solución consiste en programar directamente o algoritmo anterior. Supondo que imos acumulando en `r1` o valor de `coci`, para calcular a suma con desprazamento

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

utilizamos o código

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

utilizando `r3` como almacenamento temporal.

div10.s

```
1       .data
2  num: .word  235452
3  coc: .space 1
4  res: .space 1
5
6       .text
```

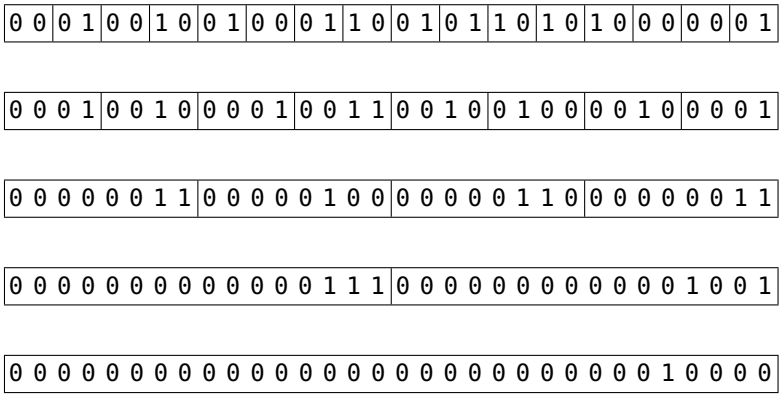
```
7  main: ldr r1,=num
8        ldr r1,[r1]    @ r1: coci, inicialmente num
9        mov r2,r1      @ r2: resto, inicialmente num
10       lsr r3,r1,#2   @ r3: coci >> 2
11       sub r1,r3      @ r1: coci - (coci>>2) = (coci>>1 + coci>>2)
12       lsr r3,r1,#4
13       add r1,r3      @ r1: coci + (coci >> 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 o resultado
31       str r1,[r3]    @ cociente
32       str r2,[r3,#4] @ resto
33
34       wfi
35      .end
```

Exercicio 3.32

O obxectivo deste exercicio é realizar un programa que conte o número de bits a 1 que ten unha palabra de 32 bits utilizando o método *divide e vencerás*. O método consiste en substituír primeiro cada campo de 2 bits coa suma dos dous bits individuais que estaban orixinalmente no campo e logo sumar os campos adxacentes de 2 bits, colocando os resultados en cada campo de 4 bits, e así sucesivamente.

O método ilústrase a continuación. Na primeira fila vemos unha palabra de 32 bits cuxos bits a 1 débense sumar (`0x23475fc1`), as filas sucesivas mostran as agrupacións e sumas de bits descritas e a última fila amosa o resultado (16 bits a un).

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



Este método pódese aplicar de maneira sinxela mediante máscaras, desprazamentos e sumas. Cunha máscara adecuada e desprazamentos, podemos filtrar e aliñar os bits a sumar en cada unha do cinco etapas. Un posible algoritmo baseado neste método sería o seguinte

```
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;
}
```

No canto de realizar un bucle con 32 iteraciones, podemos resolver o problema en 5 pasos.

Realice o programa solicitado de acordo co algoritmo anterior. Utilice as definicións seguintes para probar o programa:

```
      .data
num:  .word   0x23475fc1 @ ten 16 bits a un
res:  .byte   1
```

Solución

A solución é unha implementación directa do algoritmo anterior. Utilizamos os rexistros `r0`, `r1` e `r2` para programar a etapa

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

da seguinte maneira:

```
    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

```
      .data
num:  .word 0x23652fe1
res:  .space 1

      .text

main: ldr r0,=num
      ldr r0,[r0]

@ r0 = (r0 & 0x55555555) + ((r0 >> 1) & 0x55555555)

      mov r1,r0
      ldr r2,=0x55555555
      and r1,r1,r2
      lsr r0,r0,#1
      and r0,r0,r2
      add r0,r0,r1

@ r0 = (r0 & 0x33333333) + ((r0 >> 2) & 0x33333333)

      mov r1,r0
      ldr r2,=0x33333333
      and r1,r1,r2
      lsr r0,r0,#2
      and r0,r0,r2
      add r0,r0,r1

@ r0 = (r0 & 0x0f0f0f0f) + ((r0 >> 4) & 0x0f0f0f0f)

      mov r1,r0
      ldr r2,=0x0f0f0f0f
      and r1,r1,r2
      lsr r0,r0,#4
      and r0,r0,r2
      add r0,r0,r1

@ r0 = (r0 & 0x00ff00ff) + ((r0 >> 8) & 0x00ff00ff)

      mov r1,r0
      ldr r2,=0x00ff00ff
      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
```

Exercicio 3.33

Realizar un programa que some dous números BCD empaquetados de 8 díxitos almacenados a partir das posicións de memoria etiquetadas como `num1` e `num2` e deixe o resultado, tamén en formato BCD empaquetado, a partir da posición de memoria etiquetada como `res`.

Para probar o programa, utilice as seguintes definicións:

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

Solución

Cada sumando ocupa catro bytes consecutivos e cada byte almacena dous díxitos BCD, un en cada un dos dous *nibbles* (grupos de 4 bits) de cada byte.

Para realizar a suma, imos lendo os bytes de cada sumando, comezando polo menos significativo, e sumámolos, tendo en conta se hai algún carrexo da suma dos díxitos no byte anterior, que tamén haberá que sumar.

Para sumar os dous díxitos dun byte, filtramos os *nibbles* correspondentes, deixando en `r3` e `r4` os *nibbles* / díxitos menos significativos e

en `r1` e `r2` os *nibbles* / díxitos máis significativos. Sumamos os díxitos por separado.

Os díxitos en BCD toman os valores de 0 a 9. Se ao sumar dous díxitos obtemos un valor maior que 9, haberá que realizar un axuste. Nese caso, restamos 10 ao resultado para obter o díxito correcto e indicamos que se produciu un carrexo. Por exemplo: `0x8 + 0x7 = 0xf`. O resultado correcto será `0xf` - `0xa` = `0x5` e levámonos unha (indicamos un carrexo de 1, e o resultado en BCD será `0x15`).

O resultado de sumar os números 35.642.801 e 46.589.001 do enunciado, cuxas representacións en BCD empaquetado son respectivamente `0x35 64 28 01` e `0x35 64 28 01`, sería:

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

É dicir, o número 82.231.802, cuxa representación en BCD empaquetado é `0x0x82 23 18 02`.

sumabcd.s

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

   .equ   lon, 4        @ lonxitude do número en bytes
   .equ   mask, 0x0f    @ mascara
   .equ   offset, 0x03 @ desp. ao díxito menos signif.

   .text
main:
   ldr   r1, =num1      @ cargamos en r1 o byte menos
   add   r1, r1, #offset
   ldrb  r1, [r1]       @ significativo do primeiro numero

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

   mov   r3, #lon       @ inicializamos o contador
   ldr   r4, =cont      @ co tamaño en bytes dos sumandos
   str   r3, [r4]
```

```
26
27      ldr   r3, =res      @ inicializamos un punteiro ao resultado
28      add   r3, r3, #offset
29      ldr   r4, = dres    @ empezando polo byte menos significativo
30      str   r3, [r4]
31
32      mov   r5, #0        @ r5: carrexo entre díxitos, inicialmente a 0
33
34  bucle:
35      mov   r3, r1        @ copiamos o byte en curso
36      mov   r4, r2        @ de ambos os números a r3 e r4
37      mov   r0, #mask     @ r0: máscara para filtrar 4 bits
38
39      and   r3, r3, r0    @ quedamos co nibble menos
40      and   r4, r4, r0    @ significativo de ambos os números (r3 e r4)
41      lsr   r1, r1, #4    @ e deixamos preparado o nibble
42      lsr   r2, r2, #4    @ mais significativo (r1 e r2)
43
44      add   r6, r3, r4    @ sumamos os díxitos menos significativos
45      add   r6, r6, r5    @ e o carrexo (de habelo) en r6
46      cmp   r6, #0x0a @     dá mais de 10?
47      blt   rc1           @ se non, o carrexo é 0
48      mov   r5, #1        @ se dá mais de 10, o carrexo é 1
49      sub   r6, r6, #0x0a @ e restamos 10 do resultado
50      b     segu
51
52  rc1:
53      mov   r5, #0        @ carrexo a 0
54
55  @ temos sumados os dous díxitos menos significativos de
56  @ un byte en r6, e en r5 o carrexo correspondente
57
58  segu:
59      mov   r3, r1        @ seguimos co segundo díxito do byte
60      mov   r4, r2        @ facemos o mesmo que co primeiro díxito
61      and   r3, r3, r0    @ quedamos co díxito a sumar
62      and   r4, r4, r0    @ de ambos os bytes
63      lsr   r1, r1, #4    @ desprazamos á dereita
64      lsr   r2, r2, #4    @ ambos os nibbles e
65      add   r7, r3, r4    @ sumamos ambos os nibbles co carrexo
66      add   r7, r7, r5    @ do menos significativo ao mais significativo
67      cmp   r7, #0x0a     @ vemos se hai carrexo, e axustamos
68      blt   rc2           @ o resultado restando 10 se é necesario
69      mov   r5, #1
70      sub   r7, r7, #0x0a
71      b fin
72
73  rc2:
74      mov   r5, #0        @ carrexo a 0
```

```
75
76  @ temos sumados os dous díxitos máis significativos de
77  @ un byte en r7, e en r5 o carrexo correspondente
78
79  fin:
80     lsl   r7, r7, #4     @ montamos os dous nibbles
81     orr   r6, r6, r7     @ no byte menos significativo de r6
82     ldr   r0, =dres      @ recuperamos a dir. de destino
83     ldr   r0, [r0]
84     strb  r6, [r0]       @ almacenamos o resultado
85     sub   r0, r0, #1     @ e apuntamos ao seguinte díxito
86
87     ldr   r3, =cont      @ recuperamos o contador
88     ldr   r3, [r3]
89     sub   r3, r3, #1     @ o decrementamos
90     beq   acabo          @ e vemos se acabamos
91
92     ldr   r1, =cont      @ non acabamos: actualizamos
93     str   r3, [r1]       @ a dirección de destino
94     ldr   r1, =dres      @ e o contador
95     str   r0, [r1]
96
97     ldr   r1, =num1      @ cargamos os seguintes díxitos
98     add   r1, r1, r3     @ en r1 e r2
99     sub   r1, r1, #1     @ utilizamos o contador como
100    ldrb  r1, [r1]       @ punteiro á zona de destino
101                         @ (o desprazamento é un menos
102    ldr   r2, =num2      @ do que marca o contador)
103    add   r2, r2, r3     @ diri = numi + cont - 1
104    sub   r2, r2, #1
105    ldrb  r2, [r2]
106
107    b     bucle          @ nova iteración
108
109 acabo:
110    wfi
111    .end
```

Capítulo 4
A toda máquina

Neste capítulo comezamos introducindo o concepto de subrutina ou subprograma e a programación de subprogramas tomando como referencia a arquitectura ARM T32/Thumb. En definitiva, ilustramos o uso da instrución **`bl`** para invocar un subprograma e o uso de **`mov`** `pc,lr` para retornar ao programa principal.

Máis adiante traballamos o concepto de pila implementada na memoria de acceso aleatorio do computador.

Unha vez que nos familiarizamos co uso da pila, retomamos o concepto de subprograma co obxectivo de introducir o paso de parámetros a través da pila e a salvagarda de rexistros na pila. Tamén estudaremos as chamadas aniñadas a subprogramas e a recursividade.

4.1 Subprogramas

Neste primeiro contacto co concepto de subprograma centrámonos no paso de parámetros a través de rexistros, e ilustramos este concepto a través da súa utilización nunha serie de problemas elixidos pola súa idoneidade pedagóxica. Seguimos o convenio de chamada a subprogramas do AAPCS (Arm Ltd., 2020, cf. apéndice B), que propón o uso dos rexistros `r0` a `r3` para o paso de argumentos a subprogramas e para o paso dos resultados da execución dun subprograma ao programa chamante. Ademais, as variables locais débense almacenar tamén nos rexistros `r0` a `r3` de ser posible. O programa chamante non pode supor que o subprograma vai preservar os valores de ningún dos rexistros `r0` – `r3`. Máis adiante, no apartado 4.3, veremos como a pila proporciónanos un mecanismo adecuado para permitir que os subprogramas

usen todos os rexistros sen risco de destruír datos doutros subprogramas ou programas chamantes.

Aproveitamos ademais para introducir a codificación de caracteres UTF-8 (cf. apéndice D) mediante un exercicio de conversión de ASCII estendido ao devandito sistema de codificación, e un concepto importante na programación de sistemas como é o concepto de paridade.

Exercicio 4.1

Realizar un programa que reorganice unha zona de memoria que contén números enteiros positivos de dous bytes (medias palabras). O programa deberá rotar os números da zona de memoria cara a dúas posicións máis baixas (4 direccións de memoria) que a que ocupan actualmente, de maneira circular, é dicir, o que ocupa a posición 3 deberá ocupar a posición 1, o que ocupa a 4 deberá almacenarse na 2 e así sucesivamente. Finalmente, o elemento que ocupa a posición de memoria 1 deberá pasarse á penúltima posición e o que ocupa a posición 2 a última (cf. figura 4.1).

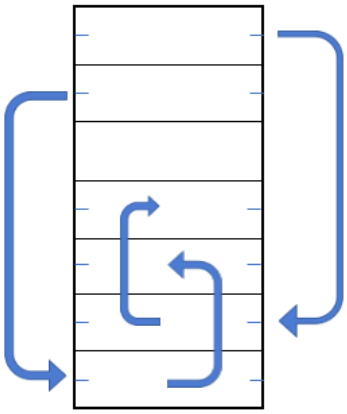


Figura 4.1. Reorganización dunha zona de memoria proposta no enunciado. A primeira media palabra pasará a ocupar a penúltima posición da zona; a última palabra almacenarase dúas posicións (4 direccións de memoria) máis arriba.

Para iso, realice un subprograma de apoio que leve a cabo a rotación dos números (medias palabras) almacenados nunha zona de memoria. Devandito subprograma recibirá como argumentos:

- `r0`: dirección de comezo da zona de memoria.
- `r1`: número de elementos na zona de memoria (número de medias palabras).

Tras a execución do programa principal, a zona de memoria considerada deberá continuar tendo os mesmos números enteiros que antes da execución do mesmo. Obviamente, devanditos números poderán estar nunha orde diferente, pero non poderá faltar nin repetirse ningún dentro da citada zona).

Para realizar o exercicio, utilice a zona de memoria seguinte:

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

Solución

O subprograma `rota` será o subprograma de apoio encargado de realizar unha rotación da zona de memoria. Para iso, primeiro toma o primeiro elemento da mesma e deposítao no último lugar, gardando o valor orixinal da última posición en `r2`. A continuación, imos desprazando os elementos da zona cara a direccións decrecentes desde o segundo elemento:

```
      ldrh r3,[r0,#2]  @ lemos un dato
      strh r3,[r0]     @ almacenámolo na posición anterior
```

Finalmente, almacenamos o valor que vai en penúltimo lugar, que estaba orixinalmente no último e que reservaramos en `r2`.

O programa principal simplemente chamará ao subprograma de apoio dúas 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  /* Temos que rotar a zona dúas veces */
8
9  main: ldr  r0,=zona    @ r0: dirección da zona
10      mov  r1,#tam     @ r1: tamaño
11      bl   rota        @ rotamos
12
13      ldr  r0,=zona
14      mov  r1,#tam
```

```
15        bl    rota        @ rotamos outra vez
16        wfi
17
18  /*
19   rota: Subprograma para rotar as medias palabras
20   almacenadas nunha zona de memoria.
21   Argumentos:
22        r0: dirección de comezo da zona.
23        r1: tamaño (en medias palabras) da zona.
24  */
25  rota:
26  /*
27  Primeiro comprobamos se hai medias palabras que rotar
28  */
29
30        cmp   r1,#1       @ se a zona non ten polo menos dúas
31        ble   feito        @ medias palabras, terminamos.
32
33        sub   r1,r1,#1    @ calculamos o fin da zona
34        lsl   r1,#1
35        add   r1,r1,r0    @ r1: apunta agora ao último elemento
36
37        ldrh r2,[r1]      @ movemos o primeiro ao
38        ldrh r3,[r0]      @ último a través de r3
39        strh r3,[r1]
40
41  outro: ldrh r3,[r0,#2] @ movemos cara abaixo
42        strh r3,[r0]      @ desde o segundo
43        add   r0,#2
44        cmp   r0, r1
45        bne   outro
46
47        sub   r0,#2
48        strh r2,[r0]      @ movemos o último ao penúltimo
49
50  feito: mov   pc,lr      @ e terminamos
51        .end
```

Exercicio 4.2

Realizar un subprograma que divida un número enteiro positivo de 32 bits entre un número enteiro positivo de 32 bits. O subprograma recibirá no rexistro `r0` o valor do dividendo e en `r1` o valor do divisor. O resultado da división devolverase nos rexistros `r0` (cociente) e `r1` (resto). Para probar o subprograma, utilice o seguinte programa principal:

```
        .data
dendo:  .word   1234
```

```
dsor:  .word    56

       .text
main:
   ldr  r0, =dendo   @ dividendo a r0
   ldr  r0, [r0]
   ldr  r1, =dsor    @ divisor a r1
   ldr  r1, [r1]
   bl   dividir
   wfi
```

Neste caso, o subprograma devolverá en `r0` o valor `0x16` (cociente da división) e en `r1` o valor `0x2` (resto da división).

Solución

Para realizar a división, utilizaremos o método das diferenzas sucesivas ou división euclídea igual que fixemos no exercicio 3.19 da páxina 85:

```
int cociente;
int resto;

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

Neste caso, os valores de `dividendo` e `divisor` estarán respectivamente nos rexistros `r0` e `r1`, e devolveranse `cociente` e `resto` nos mesmos rexistros.

dividir_sub.s

```
1
2  dividir:
3     mov   r2, #0       @ iniciamos cociente a 0 (r2)
4
5     cmp   r1, #0       @ vemos se dividimos por cero
6     beq   error
7  bucle:
8     cmp   r0, r1       @ ¿divisor menor que dividendo?
9     blt   fin          @ se o é, rematamos
10
11    add   r2, r2, #1   @ sumamos 1 ao cociente
12    sub   r0, r0, r1   @ restamos o divisor do dividendo
13    b     bucle
14
```

```
15 error:
16    eor   r2, r2, r2   @ indicamos error
17    mvn   r2, r2
18
19 fin:
20    mov   r1, r0       @ deixamos o reto en r1
21    mov   r0, r2       @ e o cociente en r0
22    mov   pc, lr       @ e volvemos
23
24    .end
```

Exercicio 4.3

Realizar un programa que compare dúas cadeas de caracteres terminadas con 0, almacenadas en memoria a partir das posicións etiquetadas como `cad1` e `cad2`, deixando o resultado da comparación na posición de memoria etiquetada como `res`. O programa ofrecerá como resultado o valor `0xff` no caso de que as cadeas sexan diferentes e o valor `0x00` no caso de que sexan iguais.

Utilice dous subprogramas de soporte, un para calcular a lonxitude dunha cadea e outro para comparar dúas cadeas de igual lonxitude.

Para probar o programa, utilice as cadeas seguintes:

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

Solución

O primeiro subprograma, etiquetado como `tamc`, vai lendo caracteres da cadea ata que se atopa o carácter 0, á vez que vai contando o número de iteracións. O número de iteracións será o tamaño da cadea.

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

O segundo subprograma, etiquetado como `eqc`, compara os caracteres de dúas cadeas dúas a dúas, supondo que ambas son de igual lonxitude, ata que atopamos caracteres diferentes ou chegamos ao final das cadeas:

```
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 */
```

O programa principal chama aos dous subprogramas en secuencia.

1. Primeiro compróbase o tamaño de ambas cadeas con **tamc**. Se as cadeas son de distinto tamaño, son diferentes.
2. Se son do mesmo tamaño, compróbase se son iguais con **eqc**.

comp_cad_srt.s

```
/*
 Comparamos dúas cadeas terminadas en 0

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

      .text
main: ldr     r5, =cad1      @ direccións das cadeas
      ldr     r6, =cad2
      mov     r7, #0xff      @ supomos que son distintas

      mov     r0, r5
      bl      tamc           @ calculamos o tamaño da primeira
      mov     r4, r0

      mov     r0,r6
      bl      tamc           @ e o da segunda
      mov     r3, r0

      cmp     r3, r4
      bne     fin            @ se os tamaños son diferentes
                             @ as cadeas non poden ser iguais
      mov     r0, r5
      mov     r1, r6
      bl      eqc            @ vemos se son iguais
      mov     r7, r0         @ recuperamos o resultado
```

```
30
31  fin:  ldr     r1, =res
32        strb    r7, [r1]        @ almacenamos o resultado
33        wfi
34
35  @
36  @ subrutina de cálculo do tamaño dunha cadea de caracteres
37  @ terminada en 0
38  @
39  @       entrada: en r0: dir. de comezo da cadea (1 word)
40  @       saída:  en r0: tamaño (1 word)
41  @
42  tamc:
43        mov     r2, #0      @ contador a cero
44  buctc:
45        ldrb    r1, [r0]    @ cargamos byte de primeira cadea
46        cmp     r1, #0      @ vemos se terminamos
47        beq     tcd
48        add     r2, r2, #1 @ incrementamos contador
49        add     r0, r0, #1 @ incrementamos o punteiro
50        b       buctc
51
52  tcd:  mov     r0, r2      @ almacenamos o resultado
53        mov     pc,lr       @ volvemos
54  @
55  @ subrutina que determina se dúas cadeas terminadas en 0 da mesma
56  @ lonxitude son iguais
57  @
58  @       entrada: r0, r1: direccións de comezo das cadeas
59  @       saída:  r0: 0x00000000 = son iguais, 0x000000ff son
60  @                diferentes.
61  @
62  eqc:  ldrb    r2, [r0]        @ cargamos un carácter
63        cmp     r2, #0          @ se chegamos ao final
64        beq     fieqc           @ é que son iguais
65        ldrb    r3, [r1]        @ cargamos un da outra cadea
66        cmp     r2, r3          @ comparámolos
67        bne     disteqc         @ se non, cadeas distintas
68        add     r0, r0, #1      @ actualizamos punteiros
69        add     r1, r1, #1
70        b       eqc             @ outro carácter
71
72  fieqc:
73        mov     r0, #0x00       @ son iguais
74        mov     pc,lr
75
76  disteqc:                      @ son diferentes
77        mov     r0, #0xff
78        mov     pc, lr
```

```
79          .end
```

Exercicio 4.4

Realizar un programa que converta unha secuencia de bytes ISO Latin 1 (ASCII estendido) no rango `0x00` a `0xff`, almacenados a partir da posición de memoria etiquetada como `cad`, en caracteres UTF-8 dun ou dous bytes, segundo a codificación descrita no Apéndice D. O resultado almacenarase a partir da dirección etiquetada como `res`.

Para probar o programa, utilice a cadea de exemplo seguinte:

```
        .data
cad:    .byte       0xa1, 'O', 'l', 'a', ' ', 'r', 'a', 'p', 'a', 'c', 'i'
        .byte       0xf1, 'o', 's ', '!', 0x00
res:    .space      20
```

Solución

Facemos uso dun subprograma de soporte (`btoutf8`) que recibe en `r0` un carácter ISO Latin 1 e devolve en `r0` a súa representación en UTF-8 (un ou dous bytes). Percorremos a secuencia de bytes e imos convertendo cada byte na devandita secuencia utilizando o programa de soporte.

btoutf8.s

```
1
2           .data
3   cad:    .byte       0xa1, 'O', 'l', 'a', ' ', 'r', 'a', 'p', 'a', 'c', 'i'
4           .byte       0xf1, 'o', 's ', '!', 0x00
5   res:    .space      20
6
7       .text
8   main:
9       ldr     r4, =cad        @ r4: dir. da cadea de partida
10      ldr     r5, =res        @ r5: dir. do resultado
11  buc:
12      ldrb    r0, [r4]        @ r0: cargamos un carácter
13      cmp     r0, #0          @ vemos se fin de cadea
14      beq     acab
15
16      bl      btoutf8         @ convertemos o carácter utilizando
17      cmp     r0, #0x80       @ o subprograma; vemos se é de 2 bytes
18      blt     so1
19
20      strb    r0, [r5, #1] @ almacenamos o byte menos signific. (de segundo)
21      lsr     r0, r0, #8   @ preparamos o byte máis significativo (0xCX)
22      strb    r0, [r5, #0] @ para almacenar o 0xCX no seu sitio
```

```
23      add   r5, r5, #2   @ incrementamos o punteiro
24      b     outro
25
26  so1:
27      strb  r0, [r5]     @ almacenamos tal cal
28      add   r5, r5, #1
29  outro:
30      add   r4, r4, #1
31      b     buc
32
33  acab:   wfi
34
35
36  /*
37
38   btoutf8: subprograma para converter caracteres no rango
39   0x00 a 0xff en caracteres UTF8 de 1 ou 2 bytes.
40
41      entrada: r0, byte menos significativo, carácter de entrada
42      saída:  r0, conversión de 1 ou 2 bytes. resto a cero.
43  */
44  btoutf8:
45      mov   r1,#ff
46      and   r0,  r0, r1 @ quedamos co byte menos significativo
47      cmp   r0, #0x80
48      blt   xaconv      @ carácter de 1 byte. Non nay que convertelo.
49
50      ldr   r2, =0xC000 @ máscara para o byte máis significativo
51      ldr   r3, =0xff00 @ máscara para borrar o byte máis sign.
52
53      lsl   r1, r0, #2  @ byte de r1: o byte máis significativo
54      orr   r1, r1, r2  @ do resultado
55      and   r1, r1, r3
56
57      mov   r2,#0x3f    @ cambiamos os dous bits máis signif.
58      and   r0, r0, r2  @ de r0 por 0b10
59      mov   r2,#0x80
60      orr   r0, r0, r2
61
62      orr   r0, r0, r1  @ e montamos todo en r0
63
64  xaconv:
65      mov   pc,lr
66      .end
```

Exercicio 4.5

Realizar un programa para calcular a representación en código ASCII do valor en hexadecimal dun número enteiro non negativo de 4 bytes etiquetado como `val`. O resultado almacenarase na partir da posición de memoria etiquetada como `res`. Para iso utilizarase un subprograma, etiquetado como `hextoa`, que reciba nos catro bits menos significativos de `r0` un número entre 0 e 15 e devolva, tamén en `r0`, o código ASCII da súa representación en hexadecimal.

Por exemplo, se o número recibido é `0xa7b5ff90`, o programa deberá almacenar a partir da posición de memoria etiquetada como `res` a cadea de caracteres "`0xA7B5FF90`".

Para probar o programa, utilice as seguintes definicións:

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

Solución

Primeiro xeramos os caracteres "`0x`". A continuación, imos tomando os bits do número de entrada de catro en catro, depositándoos en rexistro `r0`, converténdoos a ASCII utilizando o subprograma `hextoa` e almacenando os códigos ASCII obtidos no seu lugar da cadea resultado `res`.

hextoascii_srt.s

```
1
2       .data
3   val: .word  0xa7b5ff90  @ dato de exemplo
4   res: .space 10          @ resultado
5
6      .text
7   main:
8      ldr   r4, =val     @ dirección do dato
9      ldr   r5, =res     @ dirección do resultado
10  @
11  @ pomos o 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  @ convertemos e almacenamos
20  @
21     mov   r6, #7       @ contador de díxitos
```

```
22    ldr   r4, [r4]     @ cargamos o número en r4
23
24 outro:
25    mov   r0, r4
26    bl    hextoa       @ convertemos o nibble menos significativo
27    strb  r0, [r5, r6] @ almacenámolo no seu sitio
28    cmp   r6, #0       @ terminamos?
29    beq   final
30
31    lsr   r4, r4, #4   @ desprazámolo un nibble
32    sub   r6, r6, #1   @ restamos 1 do contador / punteiro
33    b     outro        @ e outra vez
34
35 final:
36    wfi
37
38 /* subrutina para pasar un díxito en binario a ASCII en hexadecimal
39    entrada: r0 díxito en binario (4 bits menos significativos).
40       Ignórase o resto.
41    saída:  r0: carácter ASCII do número (8 bits menos
42       significativos). Resto a cero.
43 */
44
45 hextoa:
46    mov   r1, #0x0f     @ enmascaramos todo menos o nibble menos
47    and   r0, r0, r1    @ significativo
48    cmp   r0, #0x09     @ vemos se '0' - '9' ou '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 @ maior que 9 : 0x0A + 0x37 = 0x41 = 'A'
56 dsr1:
57    mov   pc,lr
58    .end
```

Exercicio 4.6

Realizar un programa que tome unha secuencia de caracteres ASCII (rango `0x00` a `0x7f`) e converta o bit máis significativo de cada carácter no bit de paridade correspondente, de maneira que o numero total de bits a 1 no carácter sexa sempre par.

Por exemplo, este programa convertería `0x34` en `0xb4` (para que teña un número par de bits) e deixaría `0x33` como está (xa ten un número par de bits).

A secuencia a converter comeza na dirección de memoria etiquetada como `strm` e termina co carácter `0x04` (carácter de control de fin de texto `eot`).

Para probar o programa, utilice a cadea de exemplo seguinte:

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

Solución

Utilizamos un programa de apoio, etiquetado como `setpar`, que conta o numero de bits a 1 no byte menos significativo de `r0`, e que devolve no mesmo rexistro `r0` o byte de entrada, pero coa paridade correcta. Para contar os bits a 1, utilizamos a instrución **`lsl`** e logo incrementamos un contador en función do valor desprazado ao bit de signo, tomando a decisión coa instrución **`bpl`**.

O programa principal analizará a cadea de caracteres até atopar o carácter `eot`, modificando cada un deles utilizando o 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 a lista de números
8  bucle:
9     ldrb  r0, [r4]
10    cmp   r0, #0x04
11    beq   fin        @ se é EOT, terminamos (sen convertelo, é par)
12
13    bl    setppar    @ chamamos ao subprograma para
14    strb  r0, [r4]   @ converter este carácter.
15    add   r4, r4, #1
16    b     bucle
17
18 fin:   wfi
19
20 /*
21  setppar: subprograma que conta o numero de bits a 1
22  no byte menos significativo de r0,
23  e devolve en r0 o byte coa paridade correcta.
24
25 Entrada: r0: byte a comprobar.
26 Saída:  r0: con paridade par.
```

```
27
28 */
29
30 setppar:
31    mov   r3, r0      @ gardamos unha copia de r0
32    lsl   r3, r3, #24 @ desprazamos o byte ata que quede ao bordo
33    mov   r2, #0      @ contador de bits a 1
34    mov   r1, #7      @ contador de desprazamentos
35
36 pbuc:
37    lsl   r3, r3, #1  @ o bit menos significativo vai ao signo
38    bpl   nonsuma      @ se non é un, pasamos ao seguinte
39
40    add   r2, r2, #1  @ incrementamos o contador duns
41 nonsuma:
42    sub   r1, r1, #1
43    bne   pbuc        @ ao seguinte se non acabamos
44
45    lsr   r2, #1      @ vemos se a conta é par ou impar
46    bcc   epar       @ se é par, acabamos
47
48    mov   r1, #0x80   @ máscara para activar o bit de paridade
49    orr   r0, r0, r1  @ activámolo
50
51 epar:
52    mov   pc,lr
53    .end
```

Exercicio 4.7

Realizar un programa que calcule a representación en decimal dun número enteiro non negativo de 16 bits utilizando caracteres ASCII. O programa almacenará a representación como unha cadea, pondo o carácter "0" nas posicións máis significativas se é necesario. O número orixinal estará almacenado na media palabra etiquetada como `num` e o resultado almacenarase a partir da dirección etiquetada como `res`.

Por exemplo, o número 22 representaríase mediante a cadea de caracteres "00022" e o número 65321 mediante a cadea "65321". O número 0 representaríase coa cadea "00000". Teña en conta que o número máximo de caracteres necesarios para representar un número de 16 bits é de 5.

Utilice un subprograma para dividir por 10. Devandito subprograma deberá recibir en `r0` o número a dividir por 10, e devolverá o cociente da división en `r0` e o resto en `r1`.

Para probar o programa, utilice as definicións seguintes:

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

Solución

O noso programa irá dividindo o número orixinal por 10, ata que o número sexa menor que 10. En cada iteración, o resto da división será un dos díxitos do número orixinal, comezando polo díxito menos significativo. Sumamos ao devandito resto o valor do código ASCII que representa o '0' para convertelo no código ASCII dun número.

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

Por exemplo, os díxitos do número 567 obtéñense como 567 / 10 = 56 con resto/díxito 7; 56 / 10 = 5, con resto/díxito 6 e 5 / 10 = 0, con resto/díxito final 5.

Como coñecemos o tamaño máximo da representación ASCII do número (5 díxitos), non é necesario considerar como caso especial a posibilidade de engadir caracteres '0' nas posicións máis significativas. Por exemplo, se o número orixinal fose 0, obteriamos "00000" executando o algoritmo 5 veces consecutivas, xa que 0 / 10 = 0, con resto 0.

O subprograma de dividir por 10, etiquetado como `div10`, realiza a división mediante o algoritmo de restas sucesivas ou algoritmo de Euclides (cf. exercicio 3.19).

bin2dec.s

```
/*
 Representación dun número en decimal con caracteres ASCII.

      Almacénase a representación como cadea, pondo 0's
      nas posicións máis significativas se é necesario.

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

```

```
12        .text
13  main: ldr     r4, =res      @ dirección do resultado
14        mov     r3, #5        @ contador de caracteres
15        ldr     r0, =num
16        ldr     r0, [r0]      @ numero a converter
17
18  outro:                      @ pasamos (o que queda d)o número
19        bl      div10         @ dividimos por 10
20                              @ o resto en r1 será o novo díxito.
21        add     r1, #'0'      @ pasamos a ascii
22        sub     r3, r3, #1
23        strb    r1, [r4, r3]  @ almacenamos na súa posición
24
25        cmp     r0, #0        @ terminamos?
26        beq     ecero
27
28        cmp     r3, #0        @ o último díxito era o de orde 5?
29        beq     fin
30
31  ecero:
32        mov     r0, #'0'      @ enchemos con ceros, é dicir,
33
34  outrop:
35        sub     r3, r3, #1    @ pomos 0's nos caracteres que
36        strb    r0, [r4, r3]  @ faltan
37        cmp     r3, #0
38        bne     outrop
39
40  fin:  wfi
41
42  /*
43   Subprograma para dividir por 10.
44
45        Entrada: en r0 unha palabra co dividendo
46        Saída:  en r0 o cociente, e en r1 o resto
47
48        Neste exemplo, realizamos a división por restas sucesivas
49        (algoritmo de Euclides da 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           @ se o é, acabamos
57
58        add     r2, r2, #1     @ sumamos 1 ao cociente
59        sub     r0, r0, #10    @ restamos 10 do dividendo
60        b       bucle
```

```
61
62  fins:
63        mov       r1, r0          @ almacenamos o resto
64        mov       r0, r2          @ e o cociente
65        mov       pc, lr
66        .end
```

4.2 A pila

Os exercicios incluídos neste apartado traballan o concepto de pila. Introdúcese en primeiro lugar a súa estrutura, funcionamento e como se opera nunha *máquina de pila*. A continuación, proponse varios exercicios nos que se ilustra o procesado e modificación de datos almacenados na pila, incluíndo casos nos que o número de elementos resultado das operacións realizadas é diferente do número de elementos almacenados inicialmente na pila.

Exercicio 4.8

Dada unha cadea de caracteres terminada en 0 almacenada a partir da dirección `cad`, realice un programa que, usando a pila, almacénea en orde inversa. Por exemplo, dadas as definicións seguintes, o resultado sería a cadea `retrevni a seretcarac ed aedaC`.

```
     .data
cad: .asciz "Cadea de caracteres a inverter"
```

Solución

Utilizamos o feito de que unha memoria de pila é unha memoria LIFO (*last in, first out* ou *último en entrar, primeiro en saír*), é dicir, que o elemento que extraemos da pila cunha instrución **`pop`** é o último que introducimos cunha instrución **`push`**. Noutras palabras, se apilamos os caracteres da cadea na orde en que están, ao desapilarlos obterémolos en orde inversa.

invcad.s

```
1
2       .data
3  cad: .asciz "Cadea de caracteres a investir"
4
5     .text
6  main:
7     ldr  r0,=cad     @ dir. comezo da cadea
8     mov  r1,#0       @ índice sobre a cadea
```

```
outro:
   ldrb r2,[r0,r1] @ lemos un carácter
   cmp  r2,#0      @ vemos se terminamos de ler
   beq  lida

   push {r2}       @ apilamos este carácter
   add  r1,r1,#1   @ incrementamos o índice
   b    outro
lida:
   cmp  r1,#0      @ vemos se terminamos
   beq  feito      @ r1 ten o tamaño da cadea

   pop  {r2}       @ sacamos un carácter
   strb r2,[r0]    @ almacenámolo
   sub  r1,r1,#1   @ actualizamos contador
   add  r0,r0,#1   @ e punteiro
   b    lida
feito:
   wfi
   .end
```

Exercicio 4.9

Unha máquina de pila é un computador no cal a memoria se comporta como unha pila. Nestas máquinas, as operacións realízanse mediante un conxunto de instrucións que operan implicitamente cos valores na pila e substitúen devanditos valores polo resultado das devanditas operacións.

O obxectivo deste exercicio é simular unha máquina de pila nun computador ARM/Thumb. A pila simulada almacenará palabras de 32 bits, terá un tamaño de 64 palabras de memoria e simúlase en memoria principal de acordo coas seguintes definicións:

```
        .data
pila:   .space 63*4  @ pila é a dirección do comezo da pila
bpila:  .space 4     @ bpila é a dirección da base (fondo) da pila
ppila:  .word 1      @ ppila almacena en cada momento o punteiro de pila
```

O punteiro de pila (`pp`) estará sempre dispoñible na dirección de memoria etiquetada como `ppila` e apuntará sempre ao último dato introducido na pila. A pila crece cara a direccións decrecentes de memoria. Cando `pp` tome o valor `bpila` quererá dicir que a pila está baleira e cando tome o valor `tpila` quererá dicir que a pila está chea.

Para simular a pila, realice os seguintes subprogramas:

1. `initp`: Inicializa a pila simulada. Inicializa o punteiro de pila a `bpila+4`, de maneira que cando se apile o primeiro elemento este quede apilado no fondo da pila (na dirección de memoria `bpila`).

2. `apila`: Apila a palabra de memoria almacenada en `r0` e actualiza de maneira adecuada o punteiro de pila almacenado en `ppila`. No caso de que non sexa posible porque a pila está chea, indicarase mediante algún dos indicadores do rexistro de estado.

3. `desap`: Saca a palabra da cima da pila e déixaa en `r0` e actualiza de maneira adecuada o punteiro de pila almacenado en `ppila`. No caso de que non sexa posible porque a pila está baleira, indicarase mediante o indicador Z do rexistro de estado.

4. `sumap`: Suma as dúas palabras da cima da pila e substitúeas polo resultado da suma. Os indicadores do rexistro de estado modifícanse en función do resultado. Se a pila está baleira ou se só hai un dato, a subrutina non fai nada.

5. `restap`: Resta as dúas palabras da cima da pila e substitúeas polo resultado de réstaa. Réstaa realízase subtraendo da palabra que está na cima da pila a palabra que está apilada debaixo dela. Os indicadores do rexistro de estado modifícanse en función do resultado. Se a pila está baleira ou se só hai un dato, a subrutina non fai nada.

6. `multp`: Multiplica os 16 bits menos significativos das dúas palabras da cima da pila e substitúeas polo resultado da multiplicación. Os indicadores do rexistro de estado modifícanse en función do resultado. Se a pila está baleira ou se só hai un dato, o subprograma non fai nada.

7. `swapp`: Intercambia as dúas palabras que están na cima da pila. Se a pila está baleira ou se só hai un dato, non fai nada.

Para comprobar a máquina de pila simulada, escriba un programa que calcule o resultado da operación:

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

deixando o resultado (`0x0000 7320`) no rexistro `r0`.

Solución

As operacións a realizar para obter o resultado `0x0000 7320` son: `apila 0x20`, `apila 0x40`, `apila 0x35`, `suma`, `apila 0x100`, `multiplica`, `suma`, `apila 0x200`, `swap`, `resta`, `desapila`.

Como curiosidade, podemos ver que a secuencia de operacións anteriores correspóndese coa operación orixinal escrita en notación inversa polaca.

maq_pila1.s

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

/*
    Operación de exemplo:
                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
   bl   apila     @ apila 0x200

   bl   swapp     @ swap
   bl   restap    @ resta

   bl   desap     @ desapila: resultado a r0

   wfi


/*
```

```
 initp: Subprograma para inicializar a pila
*/
initp:
   ldr   r0,=bpila    @ dirección do fondo da pila
   add   r0,r0,#4     @ unha palabra máis (bpila+4)
   ldr   r1,=ppila
   str   r0,[r1]      @ inicializamos o punteiro de pila
   mov   pc,lr        @ volvemos

/*
apila: Subprograma que almacena na pila o contido
   de r0. No caso de que non sexa posible porque a pila
   está chea, actívase o indicador de cero (Z = 1)
   Se todo OK, Z = 0.

   O indicador Z para marcar pila chea resulta conveniente
   porque é o indicador que usamos no código para
   completar ou non a operación.

*/
apila:
   ldr   r1,=ppila
   ldr   r2,=pila
   ldr   r3,[r1]    @ r3: punteiro de pila
   cmp   r3,r2      @ se iguais, pila chea e Z = 1
   beq   fina

   sub   r3,r3,#4   @ apilamos (e Z = 0)
   str   r0,[r3]
   str   r3,[r1]    @ actualizamos punteiro de pila

fina:
   mov   pc,lr

/*
desap: Subprograma que saca a palabra da cima da pila
   e déixaa en r0. No caso de que non sexa posible
   porque a pila está baleira, actívase o indicador de
   signo (N=1). Se todo OK, N=0.

   Eliximos o indicador N para marcar pila baleira porque
   resulta conveniente (é o indicador que usamos no código
   para completar ou non a operación).
*/
desap:
   ldr   r1,=ppila
   ldr   r2,=bpila
   ldr   r3,[r1]      @ r3: punteiro de pila
   cmp   r2,r3        @ se r3 > r2, pila baleira e N = 1
```

```
89      bmi   find
90
91      ldr   r0,[r3]      @ desapilamos (e N = 0)
92      add   r3,r3,#4
93      str   r3,[r1]      @ actualizamos o punteiro de pila
94
95  find:
96      mov   pc,lr
97
98  /*
99  sumap:   Suma as dúas palabras na cima da pila e
100     substitúeas polo resultado da suma. Os indicadores
101     actívanse en función do resultado da suma.
102
103     Se a pila está baleira ou se só hai un dato,
104     non fai nada.
105 */
106 sumap:
107     ldr   r1,=ppila
108     ldr   r2,=bpila
109     ldr   r3,[r1]   @ r3: punteiro de pila
110     cmp   r2,r3
111     bmi   fins      @ se r3 > r2, pila baleira
112     beq   fins      @ se r3 = r2, só un dato
113
114     ldr   r0,[r3]   @ r0: suma das dúas palabras na
115     add   r3,r3,#4  @ cima da pila,
116     ldr   r2,[r3]   @ e eliminamos o dato dabondo
117     add   r0,r0,r2  @
118     str   r0,[r3]   @ gardamos na pila
119
120     str   r3,[r1]   @ actualizamos o punteiro de pila
121
122 fins:
123     mov   pc,lr
124
125
126 /*
127 restap:   Resta as dúas palabras na cima da pila e
128     substitúeas polo resultado da suma. Indíca
129     dores actívanse de acordo co resultado de resta
130
131     Se a pila está baleira ou se só hai un dato, non fai nada.
132 */
133 restap:
134     ldr   r1,=ppila
135     ldr   r2,=bpila
136     ldr   r3,[r1]   @ r3: punteiro de pila
137     cmp   r2,r3
```

```
138     bmi   finr      @ se r3 > r2, pila baleira
139     beq   finr      @ se r3 = r2, só un dato
140
141     ldr   r0,[r3]   @ r0: resta das dúas palabras na
142     add   r3,r3,#4  @ cima da pila,
143     ldr   r2,[r3]   @ e eliminamos o dato dabondo
144     sub   r0,r0,r2  @
145     str   r0,[r3]   @ gardamos na pila
146     str   r3,[r1]   @ actualizamos o punteiro de pila
147
148 finr:
149     mov    pc,lr
150
151 /*
152 multp:   Multiplica as dúas palabras na cima da pila
153     e substitúeas polo resultado. Os indicadores
154     actívanse de acordo co resultado da multiplicación.
155
156     Se a pila está baleira ou se só hai un dato, non fai nada.
157 */
158 multp:
159     ldr   r1,=ppila
160     ldr   r2,=bpila
161     ldr   r3,[r1]   @ r3: punteiro de pila
162     cmp   r2,r3
163     bmi   finm      @ se r3 > r2, pila baleira
164     beq   finm      @ se r3 = r2, só un dato
165
166     ldr   r0,[r3]   @ r0: produto das dúas palabras na
167     add   r3,r3,#4  @ cima da pila,
168     ldr   r2,[r3]   @ e eliminamos o dato dabondo
169     mul   r0,r0,r2  @
170     str   r0,[r3]   @ gardamos na pila
171     str   r3,[r1]   @ actualizamos o punteiro de pila
172
173 finm:
174     mov   pc,lr
175
176 /*
177 swapp:   Intercambia as dúas palabras na cima da pila.
178
179     Se a pila está baleira ou se só hai un dato, non fai nada.
180 */
181 swapp:
182     ldr   r1,=ppila
183     ldr   r2,=bpila
184     ldr   r3,[r1]    @ r3: punteiro de pila
185     cmp   r2,r3
186     bmi   finw       @ se r3 > r2, pila baleira
```

```
187     beq   finw       @ se r3 = r2, só un dato
188
189     ldr   r0,[r3]    @ r0, r2: para o intercambio
190     ldr   r2,[r3,#4] @
191     str   r0,[r3,#4] @ Intercambiamos
192     str   r2,[r3]    @
193
194 finw:
195     mov   pc,lr
196     .end
```

Exercicio 4.10

Do mesmo xeito que no exercicio 4.9, o obxectivo deste exercicio é simular unha máquina de pila nun computador ARM/Thumb, pero nesta ocasión utilizando a pila do sistema. A pila, que permite albergar palabras de 32 bits, non ten definido un límite ou capacidade máxima a priori e a súa localización vén especificada polo contido do rexistro `sp` (Stack Pointer), que apunta sempre ao último dato introducido na pila. Igual que no caso anterior, a pila crece cara a direccións decrecentes de memoria.

As instrucións básicas para acceder á pila son:

- **`push {regs}`**: Apila os rexistros indicados, en orde de número, de maior a menor. Noutras palabras, o rexistro da lista con menor número quedará na cima da pila.
- **`pop {regs}`**: Saca da pila tantos valores de 32 bits como rexistros aparezan na lista de rexistros e depositaos nos devanditos rexistros, en orde de número, de menor a maior. É dicir, o valor que estaba na cima da pila depositase no rexistro con menor número.

Para simular a pila, realice os seguintes subprogramas:

1. `sumap`: Suma as dúas palabras da cima da pila e substitúeas polo resultado da suma. Os indicadores do rexistro de estado modifícanse en función do resultado.
2. `restap`: Resta as dúas palabras da cima da pila e substitúeas polo resultado de réstaa. Réstaa realízase subtraendo da palabra que está na cima da pila a palabra que está apilada debaixo dela. Os indicadores do rexistro de estado modifícanse en función do resultado.

3. `multp`: Multiplica os 16 bits menos significativos das dúas palabras da cima da pila e substitúeas polo resultado da multiplicación. Os indicadores do rexistro de estado modifícanse en función do resultado.
4. `swapp`: Intercambia as dúas palabras que están na cima da pila.

Para comprobar a máquina de pila simulada, escriba un programa que calcule o resultado da operación:

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

deixando o resultado (`0x0000 7320`) no rexistro `r0`.

Solución

De exercicio anterior, sabemos que as operacións para realizar a 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`.

Lembra que **`pop {r0, r1}`** primeiro desapila `r0` e despois `r1`, e dá igual a orde en que os rexistros aparecen na instrución, xa que se codifican utilizando unha máscara de rexistros. Noutras palabras, **`pop {r0, r1}`** é o mesmo que **`pop {r1, r0}`** ou **`pop {r0-r1}`**.

maq_pila2.s

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

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

   bl   multp      @ multiplica
   bl   sumap

   ldr  r0,=#0x200
   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 as dúas palabras na cima da pila
33   e substitúeas polo resultado da suma. Os
34   indicadores actívanse en función do resultado da suma.
35
36  */
37  sumap:
38      pop   {r0, r1}    @ cargamos os datos
39      add   r0,r0,r1    @ facemos a suma, modificando flags
40      push  {r0}        @ almacenamos o resultado (non modifica flags)
41      mov   pc,lr       @ volvemos
42
43
44  /*
45  restap:   Resta as dúas palabras na cima da pila e
46     substitúeas polo resultado da suma. Os
47     indicadores actívanse de acordo co resultado de réstaa.
48
49  */
50  restap:
51      pop   {r0, r1}    @ cargamos os datos
52      sub   r0,r0,r1    @ facemos réstaa, modificando flags
53      push  {r0}        @ almacenamos o resultado (non modifica flags)
54      mov   pc,lr       @ volvemos
55
56  /*
57  multp:   Multiplica as dúas medias palabras menos significativas
58     das dúas palabras na cima da pila, e substitúeas polo
59     resultado (unha palabra). Os indicadores actívanse
60     de acordo co resultado da multiplicación.
61
62  */
63  multp:
64      pop   {r0, r1}    @ cargamos os datos
65      mul   r0,r0,r1    @ facemos a multiplicación, modificando flags
66      push  {r0}        @ almacenamos o resultado (non modifica flags)
67      mov   pc,lr       @ volvemos
68
69  /*
70  swapp:   Intercambia as dúas palabras na cima da pila.
```

```
71
72      Se a pila está baleira ou se só hai un dato, non fai nada.
73  */
74  swapp:
75      ldr   r0,[sp,#0]    @ cargamos o primeiro dato
76      mov   r1,r0
77      ldr   r0,[sp,#4]    @ cargamos o segundo dato
78      str   r0,[sp,#0]    @ pómolo no lugar do primeiro
79      str   r1,[sp,#4]
80      mov   pc,lr
81      .end
```

Exercicio 4.11

Realizar un subprograma, etiquetado como `letras`, que dada unha secuencia de 24 caracteres ASCII na pila substitúeos por un número enteiro de 32 bits de valor 1 se todos os caracteres son letras e polo valor 0 en caso contrario (algún dos caracteres non é unha letra). A subrutina non modificará globalmente os rexistros `r4`-`r7`.

Probe o seu subprograma co programa seguinte:

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

      .text
      ldr r4, =frase
      mov r5, #20
buc:  ldr r6, [r4, r5]  @ carga 4 caracteres da secuencia en r6
      push {r6}         @ e introdúceos na pila
      cmp r5,#0         @ comprobamos se xa temos apilado todos
      beq subp
      sub r5,#4         @ se non, actualizamos o contador
      b buc             @ e repetimos cos 4 seguintes
subp:
      bl letras
      pop {r0}
      wfi
```

que devolve `r0` = `1`.

Solución

Utilizamos a pila como unha táboa de caracteres, que imos analizando un a un até detectar que un deles non é unha letra. Se iso ocorre, teremos que deixar un 0 na pila. Se terminamos con toda a táboa sen atopar ningún carácter que non fose unha letra, deixamos un 1 na pila.

Para percorrer a táboa, utilizamos `r0` para apuntar ao principio da mesma e o rexistro `r1` como índice pola táboa. Por tanto, para ler un carácter para comprobalo cargarémolo nun rexistro (p. ex. `r2`) coa instrución **`ldrb r2, [r0, r1]`**. Para apuntar `r0` ao principio da táboa o inicializamos co valor do punteiro de pila `sp`, xa que a táboa está na pila.

letras.s

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

     .text
main:
     ldr r4, =frase
     mov r5, #20
buc: ldr r6, [r4, r5] @ carga 4 caracteres da secuencia en r6
     push {r6}        @ e introdúceos na pila
     cmp r5,#0        @ comprobamos se xa temos apilado todos
     beq subp
     sub r5,#4        @ se non, actualizamos o contador
     b buc            @ e repetimos cos 4 seguintes
subp:
     bl letras
     pop {r0}
     wfi
/*
  letras: dados 24 caracteres na pila, substitúeos polo
  valor 1 se todos os caracteres son letras, e polo valor
  0 en caso contrario (algún dos caracteres non é unha letra).
*/
letras:
     mov r0,sp       @ r0: punteiro aos datos
     mov r1,#23      @ contador e índice
bucsr:
     cmp r1,#0       @ vemos se terminamos
     beq fins        @ por aquí, eran todos letras
     ldrb r2,[r0,r1] @ r2: carácter a comprobar
     cmp  r2,#'A'    @ vemos se non é unha letra
     blt  nons
     cmp  r2,#'Z'
     blt  outro
     cmp  r2,#'a'
     blt  nons
     cmp  r2,#'z'
     blt  outro

nons:
     mov  r0,#0      @ por aquí, algún non era letra
     b sair

```

```
43 outro:
44      sub r1,r1,#1
45      b   bucsr
46
47 fins: mov  r0,#1     @ todos eran letras
48 sair:
49      add  sp,#20
50      str  r0,[sp]
51      mov  pc,lr
52      .end
```

Exercicio 4.12

Realizar un subprograma, etiquetado como `conmuta`, que dados 24 caracteres na pila, devólvaos, tamén por pila, transformando as letras maiúsculas en minúsculas e a minúsculas en maiúsculas, deixando o resto de caracteres igual. A subrutina non modificará globalmente os rexistros `r4`-`r7`.

Pode usar o seguinte programa para comprobar o correcto funcionamento de `conmuta`:

```
       .data
frase: .ascii "Pensa antes de falar!!"

      .text
      ldr r4, =frase
      mov r5, #20
buc:  ldr r6, [r4, r5]  @ carga 4 caracteres da secuencia en r6
      push {r6}         @ e introdúceos na pila
      cmp r5,#0         @ comprobamos se xa habemos apilado todos
      beq subp
      sub r5,#4         @ se non, actualizamos o contador
      b buc             @ e repetimos cos 4 seguintes
subp:
      bl conmuta
      pop {r0-r5}
      wfi
```

Despois de executar o programa anterior, os valores dos rexistros quedarían como `r0` = `0x4e454970`, `r1` = `0x41204153`, `r2` = `0x5345544e`, `r3` = `0x2045420`, `r4` = `0x4c424148` e `r5` = `0x21215241`, é dicir, a frase resultante é *pensa ANTES DE FALAR!!*.

Solución

O subprograma `conmuta` analiza os 24 caracteres na frase de entrada, un a un. Primeiro comproba se se trata dunha letra (é dicir, se o seu

código ASCII está entre 'A' e 'Z' ou entre 'a' e 'z'). Se é así, conmuta devandito carácter de maiúscula a minúscula ou viceversa. Para iso ten en conta que a diferenza entre os códigos ASCII das letras maiúsculas e minúsculas está nun único bit (cf. cadro D.6 do apéndice D). Utilizando a máscara adecuada (`0x20`) e a operación lóxica ou exclusiva (**`eor`**) conmutamos unicamente o bit necesario.

conmuta.s

```
      .data
frase: .ascii "Pensa antes de falar!!"

      .text
main:
      ldr r4, =frase
      mov r5, #20
buc: ldr r6, [r4, r5] @ carga 4 caracteres da secuencia en r6
      push {r6}        @ e introdúceos na pila
      cmp r5,#0        @ comprobamos se xa habemos apilado todos
      beq subp
      sub r5,#4        @ se non, actualizamos o contador
      b buc            @ e repetimos cos 4 seguintes
subp:
      bl conmuta
      pop {r0-r5}
      wfi

/*
 conmuta: cambia maiúsculas por minúsculas,
  e viceversa. O resto dos caracteres quedan
  inalterados
*/
conmuta:
      mov r0,sp       @ r0: punteiro aos datos na pila
      mov r1,#24      @ r1: contador de caracteres
      mov r2,#0x20    @ r2: máscara para converter (conmutar bit 6)

bucs:
      ldrb r3,[r0]    @ r3: carácter da cadea
      cmp r3,#'A'     @ comprobamos se é unha letra
      blt noncam      @ e se é así a cambiamos
      cmp r3,#'Z'
      blt cam
      cmp r3,#'a'
      blt noncam
      cmp r3,#'z'
      blt cam

/* noncam: código se non hai que modificar o carácter en r3 */
```

```
42
43  noncam:
44       sub r1,r1,#1   @ decrementamos o contador
45       beq fins       @ se é cero, terminamos
46       add r0,r0,#1   @ e incrementamos o punteiro
47       b bucs
48
49  /* cam: código se hai que modificar o carácter en r3 */
50
51  cam: eor r3,r3,r2   @ ou exclusiva: 1 eor 1 = 0, e 1 eor 0 = 1
52       strb r3,[r0]   @ almacenamos o carácter cambiado
53       b noncam       @ actualizamos punteiro e contador
54
55  fins: mov pc,lr     @ volvemos do subprograma.
56       .end
```

Exercicio 4.13

Realizar un subprograma, etiquetado como `mul16`, que dados 12 números de 16 bits na pila, devólvaos, tamén por pila, substituíndo aqueles que son múltiplo de 16 por un 0. A subrutina non modificará globalmente os rexistros `r4`-`r7`.

Utiliza o programa seguinte para probar a túa solución:

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

      .text
      ldr r4,=nums      @ r4: dirección de comezo da zona de números
      sub sp,#24        @ reservamos espazo na pila para os números
      mov r5,sp         @ r5: dirección da cima dela pila
      mov r6,#22        @ r6: contador (inicializado a (12 - 1) * 2 bytes)
buc:  ldrh r7,[r4,r6]   @ r7: cargamos un número da zona
      strh r7,[r5,r6]   @ e deixámolo no seu sitio na pila
      cmp  r6,#0        @ vemos se terminamos
      beq  csp
      sub  r6,#2        @ se non, actualizamos o contador
      b buc             @ e repetimos coa seguinte media palabra
csp:  bl mul16
      pop {r0-r5}
      wfi
```

Despois de executar o programa anterior, os valores dos rexistros quedarían como `r0` = `0x00020001`, `r1` = `0x0000FFFD`, `r2` = `0x0006FFFB`, `r3` = `0x00000007`, `r4` = `0x000a0009` e `r5` = `0x0000000b`, é dicir, os números resultantes serán 1, 2, -3, 0, -5, 6, 7, 0, 9, 10, 11 e 0.

Solución

Para realizar o subprograma `mul16` percorremos a táboa de números comprobando se son múltiplos de 16. Para iso enmascaramos os 4 bits menos significativos de cada número e comprobamos se son 0.

Este procedemento funciona tamén para os números negativos, xa que o complemento a 2 dun número positivo múltiplo de 16 tamén ten os seus 4 bits menos significativos iguais a 0. Por exemplo, o complemento a 2 de `0x12345670` é `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 comezo da zona de números
      sub sp,#24        @ reservamos espazo na pila para os números
      mov r5,sp         @ r5: dirección da cima dela pila
      mov r6,#22        @ r6: contador (inicializado a (12 - 1) * 2 bytes)
buc:  ldrh r7,[r4,r6]   @ r7: cargamos un número da zona
      strh r7,[r5,r6]   @ e deixámolo no seu sitio na pila
      cmp  r6,#0        @ vemos se terminamos
      beq  csp
      sub  r6,#2        @ se non, actualizamos o contador
      b buc             @ e repetimos coa seguinte media palabra
csp:  bl mul16
      pop {r0-r5}
      wfi

/*
 mul16: a partir dunha lista de 12 números de 16 bits
   na pila, substitúe por un cero os números que
   son múltiplo de 16.

*/
mul16:
      mov r0,sp           @ r0: punteiro aos 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        @ deixamos os 4 bits menos significativos
      bne nonmul          @ se son 0, era múltiplo de 16
      strh r3,[r0]        @ substituímos por un cero

nonmul:
      sub r1,r1,#1        @ actualizamos punteiro e contador
      beq fins
```

```
38        add r0,r0,#2
39        b bucs
40
41  fins: mov pc,lr
42        .end
```

Exercicio 4.14

Realizar un subprograma, etiquetado como `altdec`, que dados 8 caracteres ASCII numéricos almacenados na pila que representan a un número decimal, substitúeos por un número enteiro de 32 bits de valor 1 se os seus díxitos alternan os seus valores (p. ex., `76767676`) e polo valor 0 en caso contrario. No caso de que os 8 caracteres ASCII sexan iguais, o subprograma devolverá o valor 1. A subrutina non modificará globalmente os rexistros `r4`-`r12`.

Supoña que os caracteres almacenados na pila son efectivamente caracteres numéricos e probe o seu subprograma co programa seguinte:

```
        .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 devolve `r0` = `1`.

Solución

O problema de comprobar se os caracteres alternan os seus valores é equivalente ao problema de comprobar se os caracteres da cadea tomados dous a dous repítense. Por exemplo, no caso de `76767676`, os caracteres alternan porque se repite o patrón `76`. Aproveitamos esta propiedade para realizar o exercicio.

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     na pila, substitúeos por un 1 se os seus díxitos
16     alternan os seus valores, e por un 0 en caso contrario.
17  */
18  altdec:
19      mov  r0,sp       @ r0: punteiro aos díxitos
20      mov  r1,#6       @ r1: contador / punteiro
21
22      ldrh r2,[r0,r1] @ r2: primeiro par de díxitos
23
24  bucsr:
25      sub  r1,r1,#2    @ r3: par de díxitos seguinte
26      ldrh r3,[r0,r1]
27      cmp  r2,r3       @ vemos se son iguais
28      bne  nons        @ se non o son, non hai alternancia
29
30      cmp  r1,#0       @ comprobamos se acabamos
31      bne  bucsr
32
33      mov  r0,#1       @ díxitos alternantes
34
35  fins: add  sp,#4     @ axustamos a pila
36      str  r0,[sp]     @ e devolvemos un 1
37      mov  pc,lr
38
39  nons: mov  r0,#0     @ non o eran. Devolvemos
40      b    fins        @ un cero.
41      .end
```

Exercicio 4.15

Realizar un subprograma, etiquetado como `lowdigit`, que dados 8 caracteres ASCII numéricos almacenados na pila que representan a un número decimal, devolve na pila unha palabra de 32 bits, que substitúe aos caracteres orixinais, cuxo valor é o valor numérico (un enteiro entre 0 e 9) do díxito de menos valor do número orixinal. A subrutina non modificará globalmente os rexistros `r4`-`r7`.

Supoña que os caracteres almacenados na pila son efectivamente caracteres numéricos e probe o seu subprograma co programa seguinte:

```
        .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 deixa `r0` = `0`.

Solución

Do mesmo xeito que no exercicio 4.11, utilizamos a pila como unha táboa de caracteres, que imos analizando un a un até detectar o que representa o díxito de menor valor. Unha vez que o temos, convertemos o carácter do díxito nun número enteiro, por exemplo, restándolle o código ASCII do carácter '0'.

Igual que no exercicio citado, para percorrer a táboa utilizamos `r0` para apuntar ao principio da mesma e o rexistro `r1` como índice pola táboa, e para ler un carácter para comprobalo cargarémolo nun rexistro (p. ex. `r3`) coa instrución **`ldrb r3, [r0, r1]`**. Para apuntar `r0` ao principio da táboa o inicializamos co valor do punteiro de pila `sp`, xa que a táboa está na pila.

lowdigit.s

```
1         .data
2  numero: .ascii "23450879"
3
4        .text
5  main: ldr r4,=numero
6        ldr r5,[r4,#0]
7        ldr r6,[r4,#4]
8        push {r5-r6}
9        bl lowdigit
10       pop {r0}
11       wfi
12
13 /*
14   lowdigit: dados 8 caracteres ASCII numéricos almacenados
15     na pila, devolve na pila unha palabra de 32 bits,
16     que substitúe aos caracteres orixinais, cuxo valor é
17     o valor numérico (un enteiro entre 0 e 9) do díxito
18     de menos valor do número orixinal.
```

```
19 */
20 lowdigit:
21       mov r0,sp       @ r0: punteiro aos díxitos
22       mov r1,#7       @ r1: contador / índice
23       ldrb r2,[r0,r1] @ r2: díxito a comprobar
24
25 bucsr:
26       sub r1,r1,#1    @ buscamos o menor
27       ldrb r3,[r0,r1] @ e mantémolo en r2
28       cmp r2,r3
29       ble  noncam
30       mov r2,r3
31
32 noncam:
33       cmp r1,#0       @ vemos se terminamos
34       bne bucsr
35
36       sub r2,r2,#'0'  @ calculamos o valor do
37       add sp,#4       @ menor díxito (cod. ASCII - cod. '0')
38       str r2,[sp]     @ axustamos a pila, e almacenamos
39       mov pc,lr
40       .end
```

Exercicio 4.16

Realizar un subprograma, etiquetado como `sumamat4`, que dados 16 números de 16 bits na pila organizados como unha matriz 4x4 serializada por filas, substitúeos por 4 números de 32 bits cada un deles representando a suma de cada unha das 4 filas. Na cima da pila quedará a suma dos elementos da primeira fila, debaixo dela a suma dos elementos da segunda fila e así sucesivamente. A subrutina non modificará globalmente os rexistros `r4`-`r7`.

Proba o subprograma utilizando o programa seguinte:

```
        .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 deixa `r0` = `10`, `r1` = `26`, `r2` = `42` e `r3` = `58`.

Solución

Imos percorrendo a matriz en memoria utilizando dous punteiros:

- Con `r0` navegaremos polos elementos da matriz para sumar as filas. Para cada fila, os elementos a sumar estarán nas posicións `[r0]`, `[r0, #2]`, `[r0, #4]` e `[r0, #6]`.
- Con `r1` apuntamos aos lugares onde depositaremos as sumas.

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 na pila
21     organizados como unha matriz 4x4 serializada por filas,
22     substitúeos por 4 números de 32 bits cada un deles
23     representando a suma de cada unha das 4 filas.
24 */
25 sumamat4:
26       push {r4}       @ imos necesitar usar r4
27       mov r0,sp       @ r0: punteiro aos sumandos
28       add r0,r0,#28   @ empezamos polo final
29       mov r1,sp       @ r1: punteiro para os resultados
30       add r1,r1,#32   @ aquí quedará a suma da última fila
31       mov r2,#4       @ r2: contador de iteraciones
32                       @ (4 filas)
```

```
33 bucs: ldrh r3,[r0]    @ r3: acumulador de sumas dunha file
34       ldrh r4,[r0,#2] @ primeiro e segundo elemento
35       add r3,r3,r4    @ da fila
36       ldrh r4,[r0,#4] @ terceiro elemento
37       add r3,r3,r4
38       ldrh r4,[r0,#6] @ cuarto elemento
39       add r3,r3,r4
40       str r3,[r1]     @ almacenamos o resultado
41       sub r2,r2,#1    @ actualizamos contador, e comprobamos
42       beq fin         @ se final
43
44       sub r0,r0,#8    @ movemos r0 á fila anterior
45       sub r1,r1,#4    @ e r1 ao destino da seguinte suma
46       b bucs
47
48 fin:  pop {r4}        @ recuperamos r4
49       add sp,sp,#16   @ eliminamos o que sobra da pila
50       mov pc,lr       @ e regresamos
51       .end
```

4.3 Máis sobre subprogramas

Os exercicios seguintes retoman o concepto de subprograma, esta vez con paso de parámetros e salvagarda do contexto utilizando a pila, tal como introduciuse no apartado 4.2. Tamén estudaremos as chamadas aniñadas a subprogramas e a recursividade.

Do mesmo xeito que no apartado 4.1, seguimos o convenio de chamadas a subprogramas do *ARM Architecture Procedure Call Standard* (AAPCS). Así, se un subprograma necesita máis de catro argumentos, prefírese o paso dos catro primeiros nos rexistros `r0` a `r3` e o resto na pila. No caso de modificarse algún dos rexistros `r4` a `r7`, o subprograma debe preservar o seu valor para restauralo antes de devolver o control ao programa invocante. Ademais, no caso de producirse chamadas aniñadas a subprogramas, será necesario preservar tamén o valor do rexistro `lr`, utilizando para iso a parella de instrucións **`push {regs,lr}`** e **`pop {regs,pc}`** ao principio e ao final dun suprograma respectivamente. As chamadas recursivas son un caso particular desta situación.

Propomos unha selección de exercicios dunha complexidade maior que os presentados no capítulo 4.1, que ademais fan uso das novas características citadas.

Exercicio 4.17

Realizar un subprograma, etiquetado como `hswap`, que intercambie dúas medias palabras de memoria. Os parámetros de entrada serán as direccións das medias palabras nos rexistros `r0` e `r1` respectivamente. A subrutina deixará inalterados `r0` e `r1`, e non modificará globalmente ningún rexistro `r4` – `r7`.

Pode usar o seguinte programa para comprobar o correcto funcionamento 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: @ o código do subprograma iría aquí

      .end
```

Despois de executar o programa anterior, a zona de memoria debería quedar así:

```
      .data
zona: .hword 0x1111
      .hword 0x11FF
      .hword 0x2222
      .hword 0xAA22
```

Agora, facendo uso da anterior subrutina `hswap`, programe outra subrutina, etiquetada como `ordena`, que dados 12 números enteiros de 16 bits na pila (parámetros de entrada) devolva eses mesmos enteiros na pila, pero ordenados de menor a maior. Use o algoritmo da burbulla que describimos a continuación, con `TAM = 12` neste 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]);
```

A subrutina non modificará ningún rexistro `r4` – `r7`. Use o seguinte programa para comprobar o correcto funcionamento 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 o código do subprograma
      .end
```

Despois de executar o programa anterior, a zona de memoria debería quedar ordenada

```
      .data
zona: .hword -15000, -7000, -453, 0, 1, 10
      .hword 26, 83, 777, 987, 1234, 2020
```

Solución

O subprograma `hswap` intercambia as medias palabras cuxas direccións están almacenadas nos rexistros `r0` e `r1` utilizando como almacenamento intermedio os rexistros `r2` e `r3`.

Para programar a subrutina `ordena`, basicamente traducimos a ensamblador o pseudocódigo do algoritmo da burbulla indicado no enunciado. Utilizamos o subprograma `hswap` para realizar os intercambios.

Tamén será necesario estender os enteiros de 16 bits a palabras, cargándoos coa instrución **ldsh**, para poder realizar comparacións enteiras.

burbulla.s

```
        .data
zona:   .hword 1234, 26, -453, 83, -7000, 0
        .hword -15000, 1, 777, 2020, 987, 10

        .text
/*
  Programa de proba do enunciado
*/
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, =zona
        mov r5, #0
loopf:  pop {r6}
        str r6, [r4, r5]
        cmp r5, #20
        beq endf
        add r5, #4
        b loopf
endf:   wfi

/*
  hswap: subprograma da primeira parte.

  Recibe as direccións das dúas medias palabras
  a intercambiar en r0 e r1.

*/

hswap:  ldrh r2, [r0]
        ldrh r3, [r1]
        strh r3, [r0]
        strh r2, [r1]
        mov pc, lr

/*
```

```
45  ordena: subprograma da segunda parte.
46
47  Dadas 12 medias palabras na pila
48  devolva esas mesmas medias palabras na pila ordenadas,
49  utilizando o algoritmo da burbulla.
50 */
51 ordena: push {r4-r7,lr}    @ salvagardamos os rexistros r4-r7
52         mov r7, sp
53         add r7, #20         @ r7: punteiro ao principio da 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: desprazamento na táboa 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            @ se 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            @ chamada ao subprograma do primeiro
76         pop {r0,r1}         @ exercicio
77         b for2              @ e volvemos a iterar
78
79 fin2:   add r4, #1          @ i = i + 1
80         b for1
81
82 fin1:   pop {r4-r7,pc}
83        .end
```

Exercicio 4.18

Realizar un programa que conte a cantidade de aparicións de cada un dos díxitos de '0' a '9' que hai almacenados nunha zona de memoria, cuxas direccións de inicio e final están almacenadas nas posicións de memoria etiquetadas respectivamente como `i_zona` e `f_zona`.

O número total de aparicións de cada díxito almacenarase nunha táboa de 10 elementos que comeza na dirección etiquetada como `conta`, de maneira que o número de aparicións do díxito '0' almacenaríase en `conta` e o número de aparicións do díxito '9' na dirección `(conta + 9)`.

Para realizar o programa, utilice un subprograma, etiquetado como `contad`, que conte o número de aparicións de determinado carácter nunha zona de memoria. Os parámetros de entrada de devandito subprograma son os seguintes:

- `r0`: dirección de comezo da zona de memoria.
- `r1`: dirección de comezo do último elemento na zona de memoria.
- `r2`: carácter a buscar.

O subprograma devolverá en `r0` o número de aparicións do carácter indicado dentro da zona.

Utilice as definicións seguintes:

```
        .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    @ comezo da zona
f_zona: .word   d_ult   @ fin da zona
conta: .space  10       @ táboa co número de aparicións
```

Solución

O programa principal chamará 10 veces ao subprograma `contad` sobre a mesma zona, é dicir, cos mesmos parámetros de entrada en `r0` e `r1`, pero con valores consecutivos no rexistro `r2`, comezando por `r2` = `0x30` (código ASCII do carácter '0') e terminando con `r2` = `0x39` (código ASCII do carácter '9'). Despois de cada invocación de `contad`, o programa principal almacenará o resultado en `r0` na posición correspondente da táboa de resultados almacenada a partir da dirección `cuenta`.

O subprograma `contad` analiza as posicións de memoria da zona de datos una a unha comparando o seu contido co carácter modelo en `r0`, incrementando un contador de aparicións no caso de que coincida:

```
cont = 0;
for (i = *i_zona; i >= *f_zona; i++) {
```

```
        if (*i == r0) cont = cont + 1;
    }
    r0 = cont;
```

contadix.s

```
        .data
zona:   .byte   '2', '6', '7', '3', '4', '2', '8', '5', '2', '3', '3'
d_ult:  .byte   '2'
        .balign  4
i_zona: .word   zona     @ comezo da zona
f_zona: .word   d_ult    @ fin da zona
conta: .space  10        @ táboa co número de aparicións.


   .text
main:
   ldr   r4,=conta      @ táboa de resultados
   mov   r5, #0         @ índice de díxitos
   ldr   r0,=i_zona
   ldr   r0,[r0]        @ punteiro ao principio da zona
   ldr   r1,=f_zona
   ldr   r1,[r1]        @ punteiro ao final da zona
   push  {r0, r1}       @ salvagardamos os punteiros
outro:
   mov   r2,#'0'
   add   r2,r5          @ díxito a contar
   bl    contad
   strb  r0,[r4,r5]     @ almacenamos o resultado
   cmp   r5,#9
   beq   final
   add   r5,r5,#1       @ incrementamos o índice
   ldr   r0,[sp]        @ recuperamos os punteiros ao
   ldr   r1,[sp,#4]     @ principio e final da zona
   b     outro

final:
   add   sp,sp,#8       @ balanceamos a pila
   wfi

/*
contad: subprograma para contar cantos caracteres hai nunha
  zona de memoria.
  Parámetros:
    - r0: dirección de comezo da zona de memoria.
    - r1: dirección de comezo do último elemento na zona de memoria.
    - r2: carácter a buscar.
  Resultado:
    - r0: número de ocorrencias do carácter pasado como parámetro.
*/
contad:
```

```
46      push  {r4}        @ salvagardamos r4
47      mov   r4,#0       @ r4: contador de aparicións
48  buc:
49      cmp   r0,r1       @ final da táboa?
50      bgt   finsr
51
52      ldrb  r3,[r0]     @ cargamos un carácter
53      cmp   r3,r2       @ vemos se é o carácter buscado
54      bne   nons
55
56      add   r4,r4,#1    @ se é, incrementamos o contador
57  nons:
58      add   r0,r0,#1    @ incrementamos o punteiro
59      b     buc
60
61  finsr:
62      mov   r0,r4       @ pasamos o contador ao rexistro de resultado
63      pop   {r4}        @ recuperamos r4
64      mov   pc,lr       @ retornamos
65      .end
```

Exercicio 4.19

Realice un subprograma, etiquetado como `opera`, que reciba a través da pila dous números enteiros de 16 bits e devolva, tamén a través da pila, a súa suma, a súa resta, a súa multiplicación e o cociente e resto resultado da súa división, expresados cada un deles como números enteiros de 32 bits. Só se permite traballar dentro do subprograma cos rexistros r0-r3.

Para realizar a división, utilice o subprograma do exercicio 4.2 da páxina 118.

O subprograma ha de funcionar co seguinte programa principal:

```
      .data

num1:  .hword 13
num2:  .hword -2
suma:  .space  4
resta: .space
mult:  .space  4
coc:   .space  4
resto: .space  4

      .text
main: ldr  r5, =num1
      ldr  r5, [r5]
      push {r5}
```

```
bl   opera
pop  {r0-r4}
ldr  r5, =suma
str  r0, [r5]
str  r1, [r5, #4]
str  r2, [r5, #8]
str  r3, [r5, #12]
str  r4, [r5, #16]
wfi
```

Solución

O programa extrae da pila os valores pasados como parámetros y deixa os resultados tal como se indica no enunciado. Todas as operaciones con enteiros pódense facer directamente con instruccións do repertorio ARM/Thumb, excepto a división, para o que utilizaremos o subprograma do exercicio 4.2.

Neste último caso, hai que ter en conta que o subprograma `dividir` recibe como argumentos dous números enteiros positivos de 32 bits, polo que teremos que extender os argumentos de 16 bits pasados na pila a números de 32 bits. Ademais, en caso de que teñamos un dividendo ou divisor negativo, teremos que calcular o seu valor absoluto antes de chamar ao subprograma, e despois lembrar os signos para calcular o signo do resultado:

- O signo do cociente da división de dous numeros enteiros coincide co signo da multiplicación de ditos números.
- O signo do resto coincide co signo do dividendo.

Ademais, temos que lembrar que só temos unha copia da dirección de retorno almancenada no rexistro `lr`. Este valor destruirase ao chamar ao subprograma `dividir`, xa que será sustituida pola dirección de retorno ó punto de chamada en `opera` dende o devandito subprograma. Por elo, antes de chamar a `dividir` almacenamos na pila a valor de `lr` para recuperalo ao voltar.

opera.s

```
1
2  opera:
3       pop   {r0}        @ extraemos os operandos
4       mov   r1, r0      @ e os convertimos en enteiros
5       ldr   r3, =0xFFFF @ de 32 bits
6       and   r0, r3      @ separamos os números da palabra
```

```
7       lsr   r1, #16     @ extraída da pila e extendemos
8       sxth  r1, r1      @ o signo
9       sxth  r0, r0
10
11      push  {r0, r1}   @ os almacenamos na pila
12
13      cmp   r0, #0     @ empezamos coa división
14      beq   fin_div    @ se é 0, o resultado é 0
15
16      bpl   segue1     @ se negativo, cambiamos
17      neg   r0, r0     @ o signo
18  segue1:
19      cmp   r1, #0     @ Comprobamos o divisor
20      bpl   segue2     @ Se é positivo seguimos
21      neg   r1, r1     @ Se non, cambio de signo
22  segue2:
23      push  {lr}       @ gardo lr para poder recuperalo
24      bl    dividir    @ chamamos ao subprograma
25      pop   {r2}       @ e recupero lr
26      mov   lr, r2
27  fin_div:
28      pop   {r2, r3}   @ recuperamos números orixinais
29      cmp   r2, #0     @ se o dividendo era positivo,
30      bpl   segue3     @ o resto tamén.
31      neg   r1, r1     @ Se non, cambiamos signo ao resto.
32  segue3:
33      push  {r1}       @ apilamos o resto.
34      mov   r1, r2     @ copiamos o primeiro número en r1
35      mul   r1, r3     @ calculamos a multiplicación
36      cmp   r1, #0     @ se o resultado é positivo, o
37      bpl   segue4     @ o cociente tamén
38      neg   r0, r0     @ se non, é negativo
39  segue4:
40      push {r0}        @ apilamos cociente
41      push {r1}        @ e multiplicación
42
43      sub   r1, r2, r3 @ calculamos a resta
44      add   r0, r2, r3 @ calculamos a suma
45      push  {r0, r1}   @ apilamos suma e resta
46
47      mov   pc, lr     @ e volvemos ao programa principal
48
49      .end
```

Exercicio 4.20

Realizar un subprograma, etiquetado como `setbit`, que active ou desactive determinado bit nunha matriz $M \times N$ de bits almacenada en memoria de forma serializada por filas. O subprograma recibirá os seguintes argumentos:

- En `r0`, o número M de filas na matriz.
- En `r1`, o número N de columnas na matriz.
- En `r2`, a fila i onde se atopa o bit a modificar.
- En `r3`, a columna j onde se atopa o bit a modificar.
- Na pila, a dirección a partir da cal está almacenada a matriz.
- Tamén na pila, apilado encima do valor anterior, unha palabra que indica se hai que activar (valor 1) ou desactivar (valor 0) o bit indicado polos rexistros (`r3`, `r4`).

Ademais de modificar o bit indicado, o subprograma devolverá como resultado os seguintes valores:

- En `r0`, o valor 1 se o bit modificouse correctamente e 0 se houbo un erro.
- En `r1`, o código de erro se houbo erro (se `r0` = 0) e 0 en caso contrario (se `r0` = 1)

A pila non será modificada polo subprograma.

Para probar o subprograma, utilice a matriz 4×16 do exercicio 3.24 para activar o bit indicado no devandito exercicio:

```
        .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 e 3
col:    .word 12     @ columna entre 0 e 15
```

Solución

O programa principal simplemente pasará ao subprograma `setbit` os parámetros necesarios para activar o bit indicado na matriz do exemplo:

- En `r0`, o número de filas (4).
- En `r1`, o número de columnas (16).
- En `r2`, a fila do bit a activar (2).
- En `r3`, a columna do bit a activar (12).
- Na pila, a dirección `array`.
- Tamén na pila, unha palabra con valor 1.

O subprograma `setbit` comprobará primeiro se hai erros nos parámetros de entrada, basicamente, se as coordenadas do bit a modificar referencian unha posición dentro da matriz.

A continuación, calcularemos o byte e o offset nese byte onde está o bit a activar. Primeiro calculamos o índice absoluto do bit na matriz serializada ($Indx$):

$$Indx = i \times N + j,\ 0 \leq Indx \leq (N \times M) - 1$$

Logo obtemos o byte (en `r0`) e o desprazamento dentro de devandito byte (en `r1`) a partir do índice $Indx$, como o cociente e o resto de dividir $Indx$ por 8.

Finalmente quédanos activar ou desactivar mediante a máscara adecuada o bit indicado por `r1` dentro do byte de dirección (`array` + `[r0]`).

matriz_gen.s

```
1  @
2  @ Exemplo extraído do exercicio indicado
3  @
4
5          .data
6  array:  .byte 0b10000000,0b00000000 @ fila 0
7          .byte 0b01000000,0b00000000 @ fila 1
8          .byte 0b00100000,0b00000000 @ fila 2
9          .byte 0b00001000,0b11111101 @ fila 3
10 @               ||||||||   ||||||||
11 @ columnas:     01234567   89abcdef
12 @
13         .balign 4
```

```
14  row:  .word 2       @ fila de exemplo
15  col:  .word 12      @ columna de exemplo
16
17        .text
18
19  @
20  @ definimos códigos de erro
21  @
22  .equ  nif,1         @ erro de numero de filas
23  .equ  nic,2         @ erro de numero de columnas
24  .equ  ffr,3         @ erro de fila fóra de rango
25  .equ  cfr,4         @ erro de columna fóra de rango
26
27  @
28  @ Programa principal
29  @
30  @ Parámetros: activar (1) o bit de coordenadas (2, 12)
31  @ dunha 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 da matriz
45        mov r4,#1     @ activar
46
47        push {r4,r5}  @ estes parámetros pásanse na pila
48        bl  setbit
49        add  sp,#8    @ balanceamos a pila
50
51        wfi
52
53  /*
54   setbit: subrutina para activar ou desactivar
55     o bit na posición (i,j) nunha matriz de tamaño
56     (m x n) serializada por filas e almacenada
57     a partir da dirección dirmat.
58
59     Parámetros:
60      r0: m (número de filas da matriz)
61      r1: n (número de columnas da matriz)
62      r2: i (fila do bit a activar, de 0 a m-1)
```

```
63      r3: j (columna do bit a activar, de 0 a n-1)
64      Na pila (dúas palabras, apiladas na orde indicada:
65         - dirmat (dirección do matiz).
66         - 1 para indicar activación e 0 para indicar desactivación.
67
68     Resultado: r0: 1 se o bit activouse, e 0 se houbo erro
69                r1: código de erro se houbo erro, 0 en caso contrario
70   */
71  setbit:
72  @
73  @ Comprobamos erros nos parámetros de entrada
74  @
75        cmp r0,#0
76        ble e_nif         @ erro de numero de filas (M <= 0)
77        cmp r1,#0
78        ble e_nic         @ erro de número de columnas (N <= 0)
79        cmp r2,r0
80        bge e_ffr         @ erro de fila fóra de rango (i >= M)
81        cmp r3,r1
82        bge e_cfr         @ erro de columna fóra de rango (j >= N)
83        cmp r2,#0
84        blt e_ffr         @ erro de fila fóra de rango (f < 0)
85        cmp r3,#0
86        blt e_cfr         @ erro de columna fóra de rango (c < 0)
87  @
88  @ Calculamos o byte e o offset nese byte
89  @ onde está o bit a activar. Primeiro calculamos o índice
90  @ absoluto do bit na matriz serializada e deixámolo en r2:
91  @
92  @ index = i * N + j (valor entre 0 e (N * M) - 1)
93  @
94        mul r2,r2,r1      @ r2: i * N
95        add r2,r2,r3      @ r2: index
96  @
97  @ agora calculamos byte e offset a partir de index (a r0 e r1)
98  @
99  @ r0: byte = index div 8; r1: offset: index mod 8
100
101       mov r1,#0x07      @ máscara para calcular o resto
102       and r1,r1,r2      @ resto a r1
103       lsr r0,r2,#3      @ cociente a r0
104 @
105 @ activamos ou desactivamos o bit de destino (bit r1 do byte
106 @ en [dirmat + r0]). Utilizamos como máscara de partida
107 @ 0x80, e desprazamos á dereita o offset almacenado en r1
108 @
109       ldr r2,=#0x80     @ máscara de partida para cambiar
110       lsr r2,r1         @ máscara final (co bit en offset)
111       ldr r3,[sp,#4]    @ dirección da matriz
```

```
112         add r3,r3,r0        @ r3: dir do byte a modificar
113         ldrb r1,[r3]        @ r1: byte a modificar
114         ldr r0,[sp]         @ lsb de r0: activar ou desactivar
115
116         cmp r0,#0           @ ver se activar ou desactivar
117         beq set0
118
119 set1:   orr r1,r2           @ activa o bit obxectivo
120         b segue
121
122 set0:   mvn r2,r2
123         and r1,r2           @ desactiva bir obxectivo
124
125 segue:
126         strb r1,[r3]        @ substitúe o byte
127         mov r0,#1           @ orixinal co bit obxectivo
128         b back              @ e saímos sen erros
129 @
130 @ resultados con erros
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 fóra de rango
140         b   backe
141 e_cfr:
142         mov r1,#cfr         @ fila fóra de rango
143 backe:
144         mov r0,#0
145 back:   mov pc,lr           @ volvemos
146         .end
```

Exercicio 4.21

Utilizando o subprograma do exercicio 4.20, realizar un subprograma que active ou desactive unha X nunha matriz de 8×8 leds monocolor. As aspas da X correspóndense coas dúas diagonais da matriz.

A matriz de leds está representada en memoria mediante unha matriz de bits serializada por filas. Cada bit da matriz en memoria representa un led da matriz de leds.

O subprograma recibirá como parámetros:

- En `r0`, a dirección de comezo da matriz de leds.
- En `r1`, o valor 1 para activar a X e o valor 0 para desactivala.

Para probar o subprograma, realice un programa que primeiro active e logo desactive unha X nunha matriz de leds almacenada a partir da dirección `m_leds`.

```
        .data
m_leds: .space 8 @ memoria utilizada pola matriz de leds (8 bytes)
```

Solución

O programa principal simplemente chama dúas veces ao suprograma de activación (`act_lx`), a primeira con `r1 = 1` e a segunda con `r1 = 0`.

O subprograma `act_lx` fai uso do subprograma do exercicio 4.20 para activar ou desactivar os elementos (i, j) da matriz que están nas diagonais principal e 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 pola 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 a X
14       mov r1,#activar
15       bl  act_lx
16 
17       ldr r0,=m_leds   @ desactivamos a X
18       mov r1,#desactivar
19       bl  act_lx
20 
21       wfi
```

```
22
23 /*
24  act_lx: subrutina que activa unha X (as dúas diagonais) d
25  e unha matriz de bits 8 x 8.
26
27 Parámetros:
28     r0: dirección de comezo da matriz
29     r1: 1 para indicar activar e 0 para desactivar
30 */
31 act_lx:
32     push {r4,r5,lr} @ preservamos a dir. de retorno e
33                     @ rexistros por encima de r3
34     push {r0}       @ apilamos dir. matriz e acción
35     push {r1}       @ común para todas as chamadas a
36                     @ setbit
37     mov  r4,#7      @ contador de iteracións e offsets
38     mov  r5,#0      @ das diagonais
39 outro:
40     mov  r0,#8      @ activamos o bit i-ésimo
41     mov  r1,#8      @ da diagonal principal
42     mov  r2,r5
43     mov  r3,r5
44     bl   setbit
45
46     mov  r0,#8      @ activamos o bit i-ésimo
47     mov  r1,#8      @ da diagonal secundaria
48     mov  r2,r5
49     mov  r3,r4
50     bl   setbit
51
52     cmp  r4,#0      @ comprobamos se terminamos
53     beq  fin
54
55     add  r5,r5,#1   @ preparamos os dous seguintes bits
56     sub  r4,r4,#1   @ da seguinte fila
57     b    outro
58
59 fin:
60     add sp,#8       @ balanceamos a pila
61     pop {r4,r5,pc}  @ recuperamos os rexistros e volvemos
62     .end
```

Exercicio 4.22

Realizar un programa que comprobe se dúas cadeas, almacenadas respectivamente a partir das direccións `cad1` e `cad2`, teñen o mesmo número de aparicións de cada carácter que está representado na cadea.

Para iso, o programa deberá:

- Crear unha táboa para cada cadea, onde se almacene a conta de cada carácter que ten a cadea.
- Unha vez construídas as táboas, comparar se ambas as táboas son iguais.

Non se consideran caracteres que non sexan 'A'-'Z' ou 'a'-'z' e contan como iguais os caracteres en maiúsculas e minúsculas.

Para probar o programa, utilice a cadea de exemplo seguinte:

```
      .data
cad1: .asciz      "AbcDeFg  ZZZ.......ZZZ ., AbCdEFF AA,,,,,,,"
cad2: .asciz      "AbcDeFg  ZZZZCdEFF AAZZ ., Ab"
```

Solución homologas.s

```
1
2       .data
3   cad1: .asciz "AbcDeFg  ZZZ.......ZZZ ., AbCdEFF AA,,,,,,,"
4   cad2: .asciz "AbcDeFg  ZZZZCdEFF AAZZ ., Ab"
5   tab1: .space 26
6   tab2: .space 26
7
8       .text
9   main: ldr     r0, =cad1
10      ldr     r1, =tab1
11      bl      tabcar
12
13      ldr     r0, =cad2
14      ldr     r1, =tab2
15      bl      tabcar
16
17      ldr     r0, =tab1
18      ldr     r1, =tab2
19      mov     r2, #26
20      bl      compvec
21
22      wfi
23  /*
24   tabcar:
25   Subprograma para contar o numero de exemplares de cada letra
26   que ten unha cadea.
27
```

```
28        entrada: r0: dirección da cadea terminada en 0
29                 r1: dirección da táboa para o resultado
30        saída: táboa actualizada.
31  */
32
33        .equ      mascara,  0b11011111    @ para pasar a maiúsculas
34        .equ      lpost, 25               @ a táboa ten 26 slots
35                                          @ (de 0 a 25)
36  tabcar:
37  @
38  @ Inicializamos a táboa a cero
39  @
40        mov       r2, #0
41        mov       r3, #lpost
42  bini:
43        strb      r2, [r1, r3]
44        sub       r3, r3, #1
45        cmp       r3, #0
46        bne       bini
47        strb      r2, [r1]
48  @
49  @ contamos o numero de aparicións de cada letra que aparece
50  @
51        mov       r3, #mascara  @ para pasar a maiúsculas
52  bucles:
53        ldrb      r2, [r0]      @ cargamos un carácter
54        cmp       r2, #0
55        beq       terms
56
57        and       r2, r2, r3    @ pasamos todo a maiúsculas e
58        cmp       r2, #'A'      @ só consideramos letras
59        blt       nonletra
60
61        cmp       r2, #'Z'
62        bgt       nonletra
63  @
64  @ se chegamos aquí temos letras
65  @
66        sub       r2, #'A'      @ deixamos as letras no rango 0-25
67        ldrb      r3, [r1, r2]  @ e utilizámolo de índice na táboa
68        add       r3, r3, #1    @ para actualizar a conta do "slot"
69        strb      r3, [r1, r2]  @ correspondente
70  nonletra:
71        add       r0, r0, #1    @ actualizamos o punteiro á cadea
72        b         bucles
73
74  terms:
75        mov       pc, lr
76  @
```

```
@ Fin de tabcar
@
/*
 compvec : subprograma para comparar dous vectores do mesmo tamaño

      entrada: r0, dir. do primeiro vector;
      r1 = dir. do segundo vector,
      r2 = tamaño dos vectores
      saída: r0 = 1 se son iguais, 0 en caso contrario.

*/
compvec:
      push    {r4, lr}      @ preservamos r4, e de paso lr

bucc: cmp     r2, #0        @ vemos se terminamos
      beq     igual

      sub     r2, r2, #1    @ movemos o índice
      ldrb    r3, [r0, r2]  @ cargamos valores
      ldrb    r4, [r1, r2]
      cmp     r3, r4        @ e comparamos
      bne     dist
      b       bucc          @ o seguinte

igual:
      mov     r0, #1        @ indicador a "son iguais"
      b       fsr
dist:
      mov     r0, #0        @ indicador a "son distintos"
fsr:  pop     {r4, pc}      @ recuperamos r4, e de paso pc
      .end
```

Exercicio 4.23

MEDIO é unha versión simplificada do xogo de cartas UNO. Neste novo xogo, cada xogador recibe 10 cartas dunha baralla española de 40 cartas (cf. figura 4.2) e unha puntuación aleatoria inicial entre 1 e 12 puntos indicada pola denominación da primeira carta que recibe. Gaña o xogador que alcanza a maior puntuación, de acordo coa denominación das 9 cartas restantes, de acordo coa táboa seguinte (o pau da carta é indiferente e as figuras non puntúan):

Carta	Nova punt. (P)	Carta	Nova 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 non varía

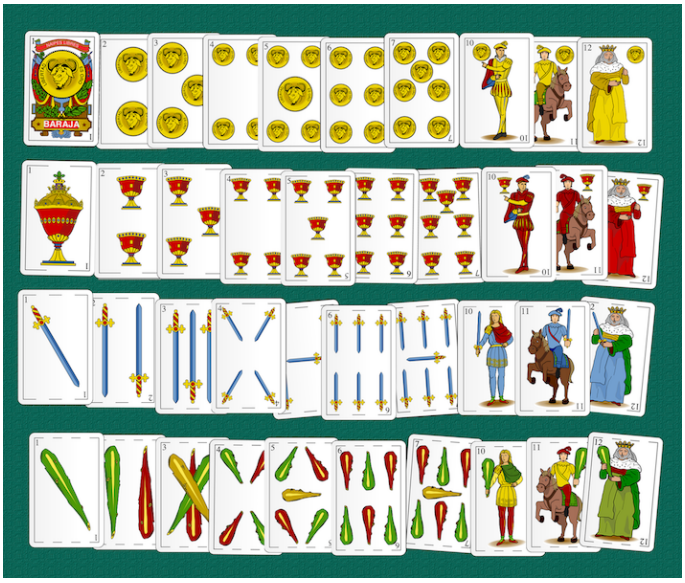


Figura 4.2. Baralla española de 40 cartas. Os paus denomínanse ouros, copas, espadas e bastos, e os naipes de numeración 10, 11 e 12 coñécense respectivamente como sota, cabalo e rei (De Basquetteur - Traballo propio, CC BY-SA 3.0, Wikipedia).

Realizar un subprograma que resolva unha xogada do xogo MEDIO pasada como parámetro. O subprograma recibirá no byte menos significativo de `r0` a denominación dunha carta (sen o pau) e en `r1` a puntuación acumulada até o momento. O subprograma devolverá en `r1` a nova puntuación tras xogar a carta.

Para probar o subprograma, utilice as definicións e o programa propostos:

```
      .data
/*
  Secuencia de cartas obtidas
*/
man:
      .byte 10, 1, 2, 3, 4, 10, 5, 6, 11, 7, 0

      .text
/*
  Programa de proba. Partimos dunha puntuación
  (en r1) de 10 indicado pola primeira carta,
  e xogamos as cartas 1 ouros,
  2 copas, 3 espadas, 4 ouros, sota de ouros,
  5 copas, 6 bastos, cabalo de ouros, 7 de ouros
```

```
  nesa orde. A puntuación final en r1
  é r1 = 55 = 0x37
*/
main: ldr  r4,=man      @ r4: dir. lista de cartas
      ldrb r1,[r4]      @ puntuación inicial
      add  r4,r4,#1     @ apuntamos á seguinte carta
buc:  ldrb r0,[r4]      @ r0: valor de carta
      cmp  r0,#0        @ vemos se terminamos
      beq  doe
      bl   comp_score   @ xogamos a carta
      add  r4,r4,#1     @ apuntamos á seguinte carta
      b  buc            @ repetimos

doe: wfi
```

Solución

Imos utilizar este exercicio para ilustrar o uso das táboas de saltos. Unha táboa de saltos é basicamente unha táboa ordenada de instrucións de salto ou de direccións. Serve para transferir o control do programa (bifurcación, salto) a un segmento de código ou a un subprograma dependendo do valor dunha variable enteira utilizado como índice sobre a táboa. A partir do valor de dicha variable, calcúlase un desprazamento desde o principio da táboa multiplicando o índice pola lonxitude da instrución ou da dirección de salto (o número de bytes en memoria ocupados por cada instrución de bifurcación ou por cada dirección).

As táboas de saltos baséanse no feito de que as instrucións de salto poden executarse de maneira extremadamente eficiente, e é máis útil cando partimos dun parámetro que pode converterse facilmente nun valor de índice secuencial.

No noso caso, utilizaremos como índice a numeración dunha carta, polo que a táboa de saltos conterá as direccións dos segmentos de código que calculan a nova puntuación dependendo da devandita numeración.

Por tanto, a dirección do segmento de código que resolve a carta de denominación i será:

$$Dir_i = Dir_{TS} + ((i-1) \times 4)$$

onde Dir_{TS} é a dirección de comezo da táboa de saltos, xa que as direccións de memoria en ARM/Thumb ocupan catro bytes.

Mediante unha táboa de saltos traducimos a ensamblador unha estrutura de control do tipo `switch`:

```
switch (carta) {
case 1:
   r1 = r1 * 5;
   break;
case 2:
   r1 = r1 - 10;
   break;
...
}
```

jump_table.s

```
      .data

@  Secuencia de cartas obtidas

man:
      .byte 1, 2, 3, 4, 10, 5, 6, 11, 7, 0

@  Táboa de saltos

      .balign 4
jump_tab:
carta1: .word s_carta1
carta2: .word s_carta2
carta3: .word s_carta3
carta4: .word s_carta4
carta5: .word s_carta5
carta6: .word s_carta6
carta7: .word s_carta7

      .text
/*
  Programa de proba. Partimos dunha puntuación
  (en r1) de 10 e xogamos as cartas 1 ouros,
  2 copas, 3 espadas, 4 ouros, sota de ouros,
  5 copas, 6 bastos, cabalo de ouros, 7 de ouros
  nesa orde. A puntuación final en r1
  é r1 = 35 = 0x23
*/
main: mov  r1,#10       @ inicializamos r1 a 10
      ldr  r4,=man      @ r4: dir. lista de cartas
buc:  ldrb r0,[r4]      @ r0: valor de carta
      cmp  r0,#0        @ vemos se terminamos
      beq  feito
      bl   comp_score   @ xogamos a carta
      add  r4,r4,#1     @ apuntamos á seguinte carta
      b  buc            @ repetimos
```

```
37
38  feito:
39        wfi
40
41  /*
42    comp_score. Xoga unha carta.
43    Parámetros:
44        - r0: valor da carta no byte menos significativo
45        - r1: valor inicial da puntuación
46    Saída:
47        - r1: novo valor da puntuación.
48  */
49  comp_score:
50        cmp r0,#0          @ se non é unha carta válida
51        ble ocs            @ terminamos
52        cmp r0,#7
53        bgt ocs
54
55  /* temos unha carta válida */
56
57        sub r0,r0,#1       @ calculamos a entrada na táboa
58        lsl r0,r0,#2       @ de saltos para esta carta:
59        ldr r2,=jump_tab   @ dir seg. código a r2:
60        ldr r2,[r2,r0]     @ r2 = jump_tab + (ro - 1) * 4
61        push {r2}          @ pasamos a dir. do código en r2
62        pop {pc}           @ ao pc
63
64  ocs:  mov pc, lr         @ terminamos. Volvemos ao programa
65                           @ principal
66  /*
67    Carta 1: r1 <- r1 * 5
68  */
69  s_carta1:
70        mov r2,r1
71        lsl r1,r1,#2
72        add r1,r1,r2
73        b   ocs
74
75  /*
76    Carta 2: r1 <- r1 - 10
77  */
78  s_carta2:
79        sub r1,r1,#10
80        b   ocs
81
82  /*
83    Carta 3: r1 <- r1 + 10
84  */
85  s_carta3:
```

```
86        add r1,r1,#10
87        b   ocs
88
89  /*
90    Carta 4: r1 <- r1 / 2
91  */
92  s_carta4:
93        lsr r1,r1,#1
94        b   ocs
95
96  /*
97    Carta 5: r1 <- r1 - 5
98  */
99  s_carta5:
100       sub r1,r1,#5
101       b   ocs
102
103 /*
104   Carta 6: r1 <- r1 * 3
105 */
106 s_carta6:
107       mov r2,r1
108       lsl r1,r1,#1
109       add r1,r1,r2
110       b   ocs
111
112 /*
113   Carta 7: r1 <- (r1 + 10) / 2
114 */
115 s_carta7:
116       add r1,r1,#10
117       lsr r1,r1,#1
118       b   ocs
119       .end
```

Exercicio 4.24

Realizar un programa que calcule o factorial dun número enteiro de 1 byte almacenado na posición de memoria etiquetada como `num` de maneira recursiva mediante un subprograma. O resultado (palabra de 4 bytes) quedará almacenado a partir da dirección de memoria etiquetada como `res`.

Para probar o programa, utilice as seguintes definicións:

```
      .data
num:  .byte 5
      .balign     4
res:  .space      4
```

Solución

Como pide o enunciado, resolvemos o problema tendo en conta que o factorial dun número pódese calcular de maneira 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 o numero
10      ldrb  r0, [r0]
11      bl    factor     @ chamamos ao subprograma
12      ldr   r1, =res
13      strb  r0, [r1]   @ gardamos o resultado
14
15      wfi
16
17
18  /*
19      factor:
20      subprograma de cálculo do factorial de maneira recursiva
21
22      entrada: r0 = número para calcular o seu factorial
23      saída:  r0 = factorial do numero
24
25      p. ex., se chamamos con r0 = 5 o subprograma devolve
26      r0 = 120 = 0x78
27
28  */
29
30  factor:
31      cmp   r0, #2         @ comprobamos se o cálculo é trivial
32      bge   maior1
33
34      mov   r0, #1         @ 0! = 1! = 1
35      mov   pc, lr         @ terminamos
36  @
37  @ fact (n) = fact(n-1) * n
38  @
39  maior1:
```

```
40    push  {r1, lr}
41    mov   r1, r0
42    sub   r0, #1
43    bl    factor          @ chamamos con n - 1
44    mul   r0, r1          @ multiplicamos o resultado por n
45    pop   {r1, pc}        @ e volvemos
46    .end
```

Exercicio 4.25

Realizar un programa que resolva o problema das torres de Hanoi. Trátase dun crebacabezas consistente en tres postes ou columnas e varios discos ensartados nos devanditos postes. Os discos teñen diferente tamaño e están ordenados de maneira que cada disco pode estar apilado unicamente sobre un disco de maior tamaño.

O xogo iníciase con todos os discos apilados, en orde, no primeiro poste. O obxectivo do xogo é traspasar todos os discos da primeira columna á terceira, mantendo a orde (de maior a menor tamaño, empezando por abaixo), tal como estaban ao principio, utilizando o poste central como columna auxiliar.

Só están permitidos os movementos seguintes:

1. Só pódese mover un disco de cada vez.
2. Un disco de maior tamaño non pode apilarse sobre un máis pequeno.
3. Só pódese desprazar dunha columna a outra o disco situado na cima.

O algoritmo recursivo para resolver o problema é o seguinte:

```
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 o crebacabezas, realice un subprograma recursivo que reciba os seguintes parámetros:

- `r0`: número de discos no poste orixe.
- `r1`: poste de orixe (1 carácter alfanumérico).
- `r2`: poste de destino (1 carácter alfanumérico).
- `r3`: poste auxiliar (1 carácter alfanumérico).

Os movementos rexistraranse nunha zona de memoria, onde se almacenarán devanditos movementos en orde crecente, un movemento en cada palabra de memoria, co formato seguinte:

`0xNNNNDDAA`

Onde `NNNN` son 16 bits que representan o número de disco que se move, entre 1 e n; `DD` representa o carácter ASCII do poste orixe e `AA` o carácter ASCII do poste destino.

Para probar o programa, resolva o problema para catro discos e tres postes, utilizando as definicións seguintes:

```
        .data
point:  .word  table   @ punteiro á táboa de movementos
table:  .space 64      @ táboa de movementos (suficiente para 4 discos)
        .equ   ORI,'A' @ nome do poste orixe
        .equ   DES,'C' @ nome do poste destino
        .equ   AUX,'B' @ nome do poste auxiliar
```

Solución

Basicamente, o noso obxectivo é traducir a código ensamblador o pseudocódigo do algoritmo recursivo do enunciado. A chamada inicial ao algoritmo será:

```
mov   r0,#4     @ 4 discos
mov   r1,#ORI   @ poste de orixe é 'A'
mov   r2,#DES   @ poste de destino é 'C'
mov   r3,#AUX   @ poste auxiliar é 'B'
bl    hanoi     @ chamamos ao subprograma
```

A partir dese momento, iranse producindo chamadas recursivas ao mesmo subprograma (`bl hanoi`) até o momento que o movemento a realizar sexa un movemento trivial do disco máis pequeno (`r0` = 1). A partir

de aí, iranse recuperando cara atrás os contextos de cada chamada aniñada, co resultado de mover un só disco en cada paso de resolución da recursividade, até recuperar o contexto da chamada inicial (hanoi(4, 'A', 'C', 'B')).

Para implementar con éxito a recursividade necesitaremos almacenar todo o contexto do subprograma na pila (os rexistros e a dirección de retorno), de maneira que se poida recuperar de maneira ordenada o contexto de cada invocación aniñada. A pila é unha estrutura de almacenamento LIFO (last-in, first-out; último en entrar, primeiro en saír), que se corresponde coa orde de recuperación do contexto das chamadas aniñadas nun subprograma que implementa un algoritmo recursivo.

hanoi.s

```
        .data
point:  .word  table   @ punteiro á táboa de movementos
table:  .space 64      @ táboa de movementos (para 4 discos)
        .equ   ORI,'A' @ nome do poste orixe
        .equ   DES,'C' @ nome do poste destino
        .equ   AUX,'B' @ nome do poste auxiliar

   .text

@ chamada inicial ao subprograma:  hanoi(4, 'A', 'C', 'B')
main:
   mov   r0,#4     @ 4 discos
   mov   r1,#ORI   @ poste de orixe é 'A'
   mov   r2,#DES   @ poste de destino é 'C'
   mov   r3,#AUX   @ poste auxiliar é 'B'
   bl    hanoi     @ chamamos ao subprograma
   wfi

/* Subprograma hanoi. Parámetros:
   r0 -> n (número de discos)
   r1 -> poste orixe  (char)
   r2 -> poste destino (char)
   r3 -> poste auxiliar (char)
*/
hanoi:
   push  {r0-r6,lr} @ algoritmo recursivo: todo o contexto
   cmp   r0,#1      @ á pila
   bne   again      @ if (n<>1) seguimos coa recursividade

@ n == 1: move (1, de, a)

   lsl   r0,#8      @ en r0 temos o número de disco
   orr   r0,r1      @ montamos o poste orixe sobre r0
```

```
35      lsl   r0,#8
36      orr   r0,r2       @ e tamén o poste destino
37
38      ldr   r4,=point   @ almacena o último movemento
39      ldr   r5,[r4]
40      str   r0,[r5]
41      add   r5,#4       @ actualizamos o punteiro á táboa
42      str   r5,[r4]
43      b     finh
44
45  @ n <> 1 : hanoi(n-1, de, aux, a); move(n, de, a);;
46  @          hanoi(n-1 ,aux ,a ,de)
47  again:
48      sub   r0,#1       @ chamada recursiva
49      mov   r4,r3       @ hanoi(n-1, de, aux, a)
50      mov   r3,r2
51      mov   r2,r4
52      bl    hanoi
53
54      mov   r6,r0       @ move (n, de, a)
55      add   r6,#1       @ recuperamos o valor de n
56      lsl   r6,#8       @ e montamos sobre r6
57      orr   r6,r1       @ n, poste orixe e poste destino
58      lsl   r6,#8
59      orr   r6,r3
60
61      ldr   r4,=point
62      ldr   r5,[r4]
63      str   r6,[r5]    @ almacena este movemento
64      add   r5,#4      @ actualizamos o punteiro
65      str   r5,[r4]
66
67      mov   r4,r1      @ chamada recursiva
68      mov   r1,r2      @ hanoi(n-1, aux, a, de)
69      mov   r2,r3
70      mov   r3,r4
71      bl    hanoi
72
73  @ recuperamos o contexto da pila
74  finh:
75      pop   {r0-r6,pc}
76      .end
```

Capítulo 5
Chegando a porto

Como xa apuntabamos no primeiro capítulo deste manual, unha das vantaxes de aprender a programar en ensamblador ARM coa Raspberry Pi, utilizando as ferramentas de programación dispoñibles baixo o sistema operativo Raspberry Pi OS, é ter á nosa disposición a biblioteca estándar da linguaxe C, así como as funcións almacenadas noutras moitas bibliotecas do sistema operativo, como aquelas que nos permitirán acceder aos portos de entrada e saída da Raspberry Pi ou controlar a liña de comunicación serie baseada no protocolo I2C, entre outras moitas.

Ademais de acceder aos dispositivos do ordenador a través das bibliotecas correspondentes, tamén é posible en moitos casos acceder directamente ao hardware de ditos dispositivos. En xeral, dende o punto de vista do programador, o método para acceder a eles non se diferencia dos accesos a memoria, xa que na arquitectura ARM os dispositivos externos teñen un conxunto de portos de entrada e saída que se corresponden con direccións do espazo de direccionamento do computador (entrada e saída con correspondencia en memoria ou *memory-mapped input/iutput*). Escribindo e lendo dos devanditos portos, utilizando as instrucións habituais de carga e almacenamento coas direccións asignadas en cada caso, poderemos programar os dispositivos, enviar información aos mesmos, obter información sobre o seu estado, ou recoller a información captada por eles.

O resto deste capítulo está dedicado a presentar unha serie de exercicios de programación que teñen como obxectivo ilustrar como podemos comunicarnos co exterior, para controlar outros dispositivos a través dos portos de entrada e saída de propósito xeral do sistema GPIO da Raspberry Pi (general-purpose input-output), ou para comunicarnos con outros sistemas a través do porto serie I2C ou da UART (Universal Asynchronous Receiver-Transmitter).

Cadro 5.1. Direccións base de E/S da Raspberry Pi

Dispositivo	Dirección base E/S
Raspberry Pi 1	`0x20000000`
Raspberry Pi 2 e 3	`0x3F000000`
Raspberry Pi 4	`0xFE000000`
Raspberry Pi Zero	`0x20000000`

Comezaremos con exemplos de programas que acceden directamente ao espazo de direccionamento de E/S tras facelo dispoñible a través da memoria virtual asignada polo sistema operativo, para a continuación presentar algúns exemplos de programas de entrada/saída utilizando las funcións das bibliotecas do sistema.

5.1 Acceso directo aos portos de entrada e saída

O cadro 5.1 recolle as direccións base da zona de entrada e saída de varios computadores Raspberry Pi. A partir de ditas direccións é posible acceder aos portos de entrada e saída dos dispositivos conectados a eles. Por exemplo, o rexistro de datos para enviar e recibir datos utilizando o interfaz serie accesible a través do GPIO correspóndese coa dirección `BASE` + `0x1000`, onde `BASE` é unha das direccións do cadro 5.1. Así, o rexistro para enviar datos pola línea serie dunha Raspberry Pi 4 estaría accesible na dirección `0xFE001000`.

Acceder aos portos directamente requirirá realizar tarefas moi próximas ao funcionamento do hardware que quedan ocultos cando utilizamos as bibliotecas do sistema. Por exemplo, para enviar un carácter polo interfaz serie teremos que comprobar explicitamente se o dispositivo está preparado lendo un dos seus portos antes de poder enviar devandito carácter.

De todos os xeitos, o sistema operativo non vai permitir de maneira xeral que os programas accedan directamente a direccións físicas de memoria como as recollidas no cadro 5.1. Por unha banda, entre as funcións do sistema operativo está garantir o acceso ordenado ós recursos do sistema, e así garantir execución concorrente de diversos módulos software sen interferencias entre eles.

Por outra banda, o sistema operativo asigna a cada programa o seu espazo de direccionamento en memoria virtual no momento da súa carga. A

consecuencia disto, se queremos acceder ao espazo de direccionamento de E/S dende un programa en execución, teremos que pedirlle ao sistema operativo que nos indique con que parte do noso espazo de direccionamento en memoria virtual se corresponde o espazo de direccionamento de E/S.

O chip Broadcom no que se basea a Raspberry Pi contén 54 pins GPIO, numerados desde GPIO0 a GPIO53 divididos en dous bancos. Os pins GPIO pódense configurar con moita flexibilidade. Cada pin pódese usar como un pin xenérico de entrada/saída, para enviar sinais de reloxo a outros sistemas, para direccionar memorias externas, etc. Ademais, cada pin pódese configurar individualmente como entrada ou como saída, e cando se configura como entrada pódese configurar á súa vez para detectar cambios no sinal de entrada ou para configurar niveis. Ademais, cada pin GPIO ten resistencias internas *pull-up* e *pull-down* configurables por software.

Os subprogramas `gpiomap` e `gpioumap` que presentamos a continuación realizan a tarefa de asociar o espazo de direccionamento físico do GPIO á memoria virtual do noso programa. O primeiro deles (`gpiomap`) accede ao dispositivo do sistema `/dev/gpiomem` para asociar os rexistros do GPIO a unha páxina de memoria virtual para así poder acceder a eles directamente desde os nosos programas. O programa devolve no rexistro `r0` a dirección base de memoria virtual asociada ao GPIO. Pola súa banda, `gpioumap` devolve ó sistema operativo a memoria virtual asignada previamente con `gpiomap`. Este subprograma recibe como parámetro a dirección base devolta pola invocación orixinal a `gpiomap`.

gpiomap.s

```
/*
  gpiomap e gpioumap.

  Ensamblado:

  as -o gpiomap.o gpiomap.s

   Para utilizar este programa xunto a outros, enlazaremos
  o ficheiro obxecto cos ficheiros obxecto dos devanditos
  programas e a libraría estándar de C.
*/

/*
  flags para a chamada a open
*/
    .equ    O_RDWR,   00000002       @ flags para abrir
    .equ    O_SYNC,   04100000       @ gpiomem en modo
    .equ    O_FLAGS,  O_RDWR|O_SYNC  @ L/E e SYNC
```

```
20
21  /*
22    Direccións base de entrada e saída da Raspberry Pi 3B+
23  */
24
25      .equ    MMIO_BASE, 0x3f000000
26      .equ    GPIO_OFST, 0x200000
27      .equ    GPIO_BASE, MMIO_BASE + GPIO_OFST
28
29  /*
30    Tamaño do bloque (1 páxina)
31  */
32      .equ    BLOCK, 4096
33
34      .code16
35      .align  2
36
37      .data
38  gpiod:
39      .asciz  "/dev/gpiomem"  @ dispositivo do sistema
40
41      .global gpiomap
42      .type   gpiomap, %function
43
44      .align  2
45      .text
46
47  /*
48
49    dir_base = gpiomap()
50
51    Asocia os rexistros do GPIO da Raspberry Pi
52    a unha páxina de memoria virtual para poder acceder
53    a eles desde os programas de usuario.
54
55    Devolve en r0 a dirección base da zona
56  */
57
58  gpiomap:
59      push    {r4, lr}
60
61  /*
62    Abrimos /dev/gpiomem para lectura/escritura e sync
63  */
64
65      ldr     r0, =gpiod      @ dirección da cadea
66      ldr     r1, =O_FLAGS    @ flags para acceder ao gpiod
67      blx     open
68      mov     r4, r0          @ handler a r4
```

```
69
70  /*
71    Asociamos os rexistros do GPIO a unha páxina de memoria
72    virtual para poder acceder a eles cunha chamada a mmap
73  */
74      sub   sp, sp, #8      @ deixamos sitio para os argumentos
75      str   r4, [sp, #0]    @ handler de mem
76      ldr   r0, =GPIO_BASE
77      str   r0, [sp, #4]    @ dirección basee do GPIO
78      mov   r0, #0          @ que o kernel elixa localización
79      ldr   r1, =BLOCK      @ cunha páxina chega
80      mov   r2, #0x3        @ modo lectura/escritura
81      mov   r3, #0x01       @ espazo compartido
82      blx   mmap            @ dirección base en r0
83
84      add   sp, sp, #8      @ balanceamos a pila
85      pop   {r4, pc}        @ recuperamos os rexistros e volvemos
86
87      .global gpioumap
88      .type   gpioumap, %function
89  /*
90     void gpioumap(dir_base)
91
92     Desasocia a memoria virtual asociada á memoria E/S do GPIO
93     Argumento: r0: dirección base da memoria do gpio en memoria virtual
94  */
95  gpioumap:
96      push  {r4, lr}
97                            @ r0: dirección de comezo da zona
98      ldr   r1,=BLOCK       @ r1: tamaño da zona.
99      blx   munmap          @ chamamos a munmap
100
101     pop   {r4, pc}        @ recuperamos os rexistros e volvemos
102
103     .end
```

Exercicio 5.1

Este exercicio ten como obxectivo desenvolver un programa, etiquetado como `setleds`, que acenda e apague dous leds conectados aos pins 23 e 24 do GPIO da Raspberry Pi. Os leds encenderanse cando estean pulsados sendos botóns conectados aos pins 20 e 21 do GPIO respectivamente, e apagaranse cando ditos botóns estean desactivados. O programa terminará se se pulsa outro botón conectado ao pin 22 do GPIO.

No noso caso, para conectar os botóns á Raspberry Pi utilizamos un circuíto *pull-up* externo como o da figura 5.1 para evitar que as entradas asocia-

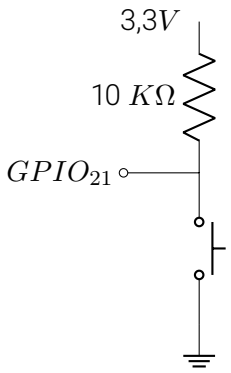


Figura 5.1. Circuíto de conexión dos botóns. Exemplo para o pin 21.

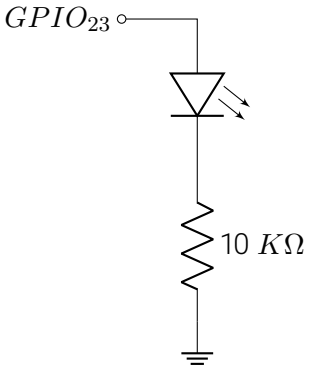


Figura 5.2. Circuíto de conexión dos leds. Exemplo para o pin 23.

das queden flotantes. Devandito cirtuito fixa o nivel de entrada do pin a 1 (3,3 voltios) se o botón está aberto. Ao pulsar o botón, leva a entrada a masa e o valor de entrada do pin pasa a 0.

Utilizar un circuíto externo evitarannos ter que configurar as resistencias de *pull-up* internas.

Para conectar os leds, unimos o seu polo positivo a un dos pins do GPIO utilizados como entradas (23 e 24 neste exercicio), e poñemos o polo negativo a masa a través dunha resistencia (cf. figura 5.2).

Solución

Neste programa accederemos directamente aos pins do GPIO a través dos correspondentes portos de entrada e saída. Para elo teremos que coñecer a dirección base de memoria virtual asignada polo sistema operativo ao espazo de direccionamento de E/S e a ubicación dos portos necesarios en dito espazo de direccionamento.

A continuación, configuramos os pins 20, 21 e 22 como entradas e os pins 23 e 24 como saídas. Unha vez que temos configurados os pins, entramos nun bucle que le o estado dos pins 21 e 22 e o transfire aos pins 23 e 24. Se detectamos un cambio de estado no pin 22, saímos do bucle e o programa termina. Para detectar que un botón está pulsado, comprobamos se o valor de entrada é 0.

A organización xeral do programa é a seguinte:

```
Configurar o funcionamiento dos pins 20, 21, 22, 23 e 24.
Le pin 22
while (pin 22 inactivo) {
      Le pins 20 e 21
      Transfiere o estado aos pins 23 e 24
      Le pin 22
}
```

SetLedRaw.s

```
/*
  Activar leds nunha Raspberry Pi 3B+.

  Acende e apaga dous leds conectados aos pins 23 e 24
  do GPIO da  Raspberry Pi. Os leds encenderanse cando
  estean pulsados sendos botóns conectados aos pins
  20 e 21 do GPIO, e apagaranse cando ditos botóns
  estean desactivados. O programa terminará se se pulsa
  outro botón conectado ao pin 22 do GPIO.

  Para conectar os botóns á Raspberry Pi utilizamos
  un circuíto pull-up para evitar que as entradas asociadas
  queden flotantes (mon facemos uso dos pull-up internos).
  Devandito cirtuito fixa o nivel de entrada do pin a 1
  (3,3 voltios) se o botón está aberto. Ao pulsar o botón,
  leva a entrada a masa e o valor de entrada do pin pasa a 0.

  Utilizamos os subprogramas gpiomap e gpioumap de gpiomap.s

  Ensamblado:

  as -g -o SetLedRaw.s -o SetLedRaw.o
  gcc -g -o SetLedRaw SetLedRaw.o gpiomap.o
```

```
24
25  */
26
27
28  /*
29    Rexistros do GPIO a partir da dirección base de entrada e saída
30    do módulo GPIO
31  */
32        .equ    GPFSEL2,    0x0008      @ pins 20 - 29
33        .equ    GPSET0,     0x001C      @ activar pins
34        .equ    GPCLR0,     0x0028      @ desactivar pins
35        .equ    GPLEV0,     0x0034      @ estado dos pins
36
37  /*
38    Máscaras para definir as direccións dos pins 20, 21, 22 (entradas)
39    23 e 24 (saídas) segundo a configuración de GPFSEL2 (pins 20 a 29).
40    En binario:
41
42              29  28  27  26  25  24  23  22  21  20
43        0bxx xxx xxx xxx xxx xxx 001 001 000 000 000
44
45    Desactivar os que deben quedar a 0 con and:
46
47        0b11 111 111 111 111 111 001 001 000 000 000
48        0b1111 1111 1111 1111 1001 0010 0000 0000
49        0xFFFF93C0
50
51    Activar os que deben quedar a 1 con orr:
52
53        0b00 000 000 000 000 000 001 001 000 000 000
54        0b0000 0000 0000 0000 0001 0010 0000 0000
55        0x00001200
56  */
57        .equ    ANDGPIOM, 0xFFFF9200
58        .equ    ORRGPIOM, 0x00001200
59
60  /* máscaras dos pins 20, 21, 22, 23 e 24 */
61
62        .equ    PIN20,      0X00100000
63        .equ    PIN21,      0X00200000
64        .equ    PIN22,      0X00400000
65        .equ    PIN23,      0X00800000
66        .equ    PIN24,      0X01000000
67
68        .text
69  /*
70  -------------------------------------------------------------
71      CODIGO THUMB
72  -------------------------------------------------------------
```

```
*/
      .code16
      .align 2
/*
 Subprograma para demostrar o uso dos pins do GPIO 20, 21 e 22
 como pins de entrada e os pins 23 e 24 como pins
 de saída.
*/

setledr:
      push {r4-r6, lr}
      bl   gpiomap           @ asociamos a memoria de E/S
      mov  r3, r0            @ a unha páxina de memoria virtual

@ Configuramos as direccións dos pines 20, 21, 22, 23 e 24

      ldr   r0, [r3, #GPFSEL2]
      ldr   r1, =ANDGPIOM      @ máscara para desactivar bits
      ldr   r2, =ORRGPIOM      @ máscara para activar bits
      and   r0, r0, r1
      orr   r0, r0, r2
      str   r0, [r3, #GPFSEL2] @ pomos as direccións

@ entramos nun bucle ata pulsar o botón de terminar

      ldr   r4,=PIN22          @ para comprobar final
bucle:
      mov   r1,#0              @ para activar leds
      mov   r2,#0              @ para desactivar leds

      ldr   r0,[r3,#GPLEV0]    @ lemos os botóns

      tst   r0,r4              @ miramos o botón de finalizar
      beq   sair               @ se activo, terminamos

      ldr   r5,=PIN23
      ldr   r6,=PIN20
      tst   r0,r6              @ estado do pin 20
      bne   desled1
      orr   r1,r1,r5           @ se activo, activar led 23
      b     bot22
desled1:
      orr   r2,r2,r5           @ se non activo, desactivar led 23

bot22:
      ldr   r5,=PIN24
      ldr   r6,=PIN21
      tst   r0,r6              @ estado do pin 21
      bne   desled2
```

```
122       orr   r1,r1,r5          @ se activo, activar led 24
123       b     daleds
124 desled2:
125       orr   r2,r2,r5          @ se non activo, desactivar led 24
126
127 daleds:
128       str   r1,[r3,#GPSET0]   @ activamos leds
129       str   r2,[r3,#GPCLR0]   @ desactivamos leds
130
131       b     bucle
132
133 /*
134   Terminamos. Desasociamos a zona de memoria e recuperamos os rexistros
135 */
136 sair: mov   r0,r3
137       bl gpioumap
138       pop  {r4-r6, pc}
139
140 /*
141 -------------------------------------------------------------
142     FIN DO CODIGO THUMB
143 -------------------------------------------------------------
144 */
145
146 /*
147    Programa principal de entrada desde o SO
148 */
149
150       .arm
151       .align 4
152       .global main
153 main:
154       push {r4, lr}
155       blx  setledr
156       pop  {r4, lr}
157       bx   lr
158
159      .end
```

No caso de que decidísemos utilizar as resistencias de *pull-up* internas, teriamos que engadir o código de configuración que aparece a continuación.

pullup.s

```
1  /*
2    Subprograma para activar a resistencia pullup
3    dun pin GPIO
4
5    Supomos que en r0 temos a dirección basee en memoria
```

```
  virtual do GPIO, obtida con gpiomap e en r1
  o pin GPIO cuxa resitencia pullup queremos activar

  void pup_gpio(base_addr, pin)

*/
@
@ rexistros involucrados na activación
@
      .equ   GPPUD,       0x0094
      .equ   GPPUDCLK0,   0x0098
@
@ outras constantes necesarias
@
      .equ   SHORT_DELAY, 50
      .equ   PULLUP,      2

      .text
      .code16
      .align 2

pup_gpio:
@
@ indicamos no rexistro correspondente
@ que imos activar unha resistencia pullup
@
      mov   r2, #PULLUP
      str   r2, [r0,#GPPUD]
@
@ esperamos 150 ciclos de reloxo
@
      mov   r2,#SHORT_DELAY
esp1: subs  r2,r2,#1
      bcc   esp1
@
@ activamos a resistencia de pullup do
@ pin GPIO solicitado (en r1) no
@ rexistro correpondiente
@
      mov   r2,#1
      lsl   r2,r2,r1
      str   r2,[r0,#GPPUDCLK0]
@
@ esperamos outros 150 ciclos de reloxo
@
      mov   r2,#SHORT_DELAY
esp2: subs  r2,r2,#1
      bcc   esp2
@
```

```
55 @ quitamos os sinais de activación nos
56 @ dous rexistros
57 @
58       mov   r2,#0
59       str   r2,[r0,##GPPUD]
60       str   r2,[r0,#GPPUDCLK0]
61
62       mov   pc, lr
63       .end
```

Exercicio 5.2

Neste exercicio imos ilustrar o uso dunha das UART que inclúe o chip Broadcom da Raspberry Pi para enviar e recibir información. Para iso, escribiremos un programa, etiquetado como `uart0_tst`, que envíe a mensaxe *Hai alguén do outro lado?* pola UART0, e reciba a resposta a continuación. Utilizaremos o carácter ASCII `EOT` (código `0x03`) para indicar o final das mensaxes.

Configuraremos ademais a UART0 cos seguintes parámetros:

- Velocidade de transmisión de 115200 bits por segundo.
- Control de paridade, con paridade par.
- Memoria intermedia FIFO de transmisión de 4 bytes.
- Emisión e recepción de caracteres mediante espera activa.

Solución

Neste programa accederemos directamente aos pins dunha das UART da Raspberry Pi (UART0) a través dos correspondentes portos de entrada e saída. UART é un protocolo e un dispositivo de comunicación serie asíncrona, é dicir, transmite e recibe bytes de datos enviando e recibindo os bits individuais de xeito secuencial por unha liña de comunicación con dous fíos (transmisión e recepción). A transmisión asíncrona permite transmitir información sen que o emisor teña que enviar unha sinal de sincronización (sinal de reloxo) ao receptor. Pola contra, emisor e receptor acordan previamente a velocidade de transmisión (i.e., o tempo dedicado a transmitir cada bit), e se engade a cada byte uns bits especiais chamados *bits de arranque* para facilitar a sincronización entre eles. Podemos usar a UART da Raspberry Pi, entre outras posibilidades, para comunicarnos con outros dispositivos con unha interfaz compatible, como as placas Arduino, ESP9266, ou o microcontrolador de Raspbery Pi RP2040.

Parac conectar outros dispositivos á UART0 o faremos a través dos pins GPIO14 e GPIO15, que teñen respectivamente as funcións alternativas TXD (transmisión) e RXD (recepción) para a UART0. Do mesmo xeito que ocorría co GPIO no exercicio 5.1, teremos que coñecer a dirección base de memoria virtual asignada polo sistema operativo ao espazo de direccionamento de E/S da UART0 e a ubicación dos portos necesarios en dito espazo de direccionamento. Para elo utilizaremos o código que presentamos de seguido.

uart0map.s

```
/*
  uart0map y uart0umap.

  Ensamblado:

  as -o uart0map.o uart0map.s

   Para utilizar este programa xunto a outros, enlazaremos
  o ficheiro obxecto cos ficheiros obxecto dos devanditos
  programas e a libraría estándar de C.
*/

/*
  flags para a chamada a open
*/
    .equ    O_RDWR,   00000002       @ flags para abrir
    .equ    O_SYNC,   04100000       @ gpiomem en modo
    .equ    O_FLAGS,  O_RDWR|O_SYNC  @ L/E e SYNC

/*
  Direccións base de entrada e saída da Raspberry Pi 3B+
*/

    .equ   MMIO_BASE, 0x3f000000
    .equ   UART_OFST, 0x201000
    .equ   UART_BASE, MMIO_BASE + UART_OFST

/*
  Máscara de aliñamento
*/

    .equ   ALG_MASK, 0x0FFF

/*
  Tamaño do bloque (1 páxina)
*/
    .equ   BLOCK, 4096

```

```
39      .code   16
40      .align  2
41
42      .data
43  uart0d:
44      .asciz  "/dev/mem"  @ dispositivo do sistema (xeral, "mem")
45
46      .global uart0map
47      .type   uart0map, %function
48
49      .text
50      .code16
51      .align2
52
53  /*
54
55    dir_base = uart0map()
56
57    Asocia os rexistros da UART0 da Raspberry Pi
58    a unha páxina de memoria virtual para poder acceder
59    a eles desde os programas de usuario.
60
61    Devolve en r0 a dirección base da zona
62  */
63
64  uart0map:
65      push    {r4-r6, lr}
66
67  /*
68    Abrimos /dev/mem para lectura/escritura e sync
69  */
70
71      ldr   r0, =uart0d    @ dirección da cadea
72      ldr   r1, =O_FLAGS   @ flags para acceder a uart0d
73      blx   open
74      cmp   r0,#0          @ miramos si OK
75      bge   aberto         @ se todo OK, aberto
76
77      mov   r0,#0          @ se fallo, devolvemos 0
78      pop   {r4-r6, pc}    @ recuperamos e volvemos
79
80  aberto:
81      mov   r4, r0         @ handler a r4
82
83  /*
84    Asociamos os rexistros da UART0 a unha páxina de memoria
85    virtual para poder acceder a eles cunha chamada a mmap
86  */
87      sub   sp, sp, #8     @ deixamos sitio para os argumentos
```

```
88      str   r4, [sp, #0]   @ apilamos handler de mem
89
90      ldr   r5, =UART_BASE @ dirección base da UART0
91      mov   r1, r5         @ también en r1
92
93      ldr   r2,=ALG_MASK   @ alineamos la dirección física
94      and   r1,r1,r2       @ a un límite de páxina
95      bic   r5,r5,r2
96      str   r5, [sp, #4]   @ apilamos dir. física alineada
97      mov   r0, #0         @ que o kernel elixa localización
98      ldr   r1, =BLOCK     @ cunha páxina chega
99      mov   r2, #0x3       @ modo lectura/escritura
100     mov   r3, #0x01      @ espazo compartido
101
102     blx   mmap           @ dirección base en r0
103     mvn   r0, r0
104     cmp   r0, #1         @ vemos si error
105     beq   acabado
106     mvn   r0, r0
107     add   r0, r0, r6     @ Si OK, añadimos offset a límite
108                          @ de páxina
109 acabado:
110     add   sp, sp, #8     @ balanceamos a pila
111     pop   {r4-r6, pc}    @ recuperamos os rexistros e volvemos
112
113     .global uart0umap
114     .type   uart0umap, %function
115 /*
116    void gpioumap(dir_base)
117
118    Desasocia a memoria virtual asociada á memoria E/S da UART0
119    Argumento: r0: dirección base da memoria do gpio en memoria virtual
120 */
121 uart0umap:
122     push  {r4, lr}
123                          @ r0: dirección de comezo da zona
124     ldr   r1,=BLOCK      @ r1: tamaño da zona.
125     blx   munmap         @ chamamos a munmap
126
127     pop   {r4, pc}       @ recuperamos os rexistros e volvemos
128
129     .end
```

A continuación, configuraremos a UART0 cos parámetros solicitados, e enviaremos mediante espera activa a mensaxe indicada. Unha vez que enviásemos a mensaxe, porémonos a ler caracteres ata que detectemos un carácter de fin de texto `EOT`. Finalmente, imprimimos a

mensaxe recibida pola saída estándar utilizando a función `printf` da biblioteca estándar.

A organización xeral do programa é a seguinte:

```
Configurar ou funcionamento da UART0
while (caracteres por enviar) {
      Tomar carácter da mensaxe
      Enviar carácter
}
while (non se reciba EOT e non haxa erro) {
      Recibir carácter
      Almacenalo na memoria de mensaxes
}
Imprimir mensaxe recibida pola saída estándar.
```

UART0TxRx.s

```
/*
Programa para probar a UART0 da Raspberry Pi.

Enviamos a frase "Hai alguén do outro lado?" e esperamos unha resposta

  Utilizamos os subprogramas uart0map e uart0umap de uart0map.s

  Ensamblado:

  as -g -o UART0TxRx.s -o UART0TxRx.o
  gcc -g -o UART0TxRx UART0TxRx.o uart0map.o


*/

@
@ Rexistros da UART0 e outras constantes para o control da UART
@
      .equ    UART_IBRD,    0x24  @ integer baud rate divisor
      .equ    UART_FBRD,    0x28  @ fractional baud rate divisor
      .equ    UART_LCRH,    0x2C  @ line control register
      .equ    UART_IMSC,    0x38  @ interrupt mask set clear
      .equ    UART_CR,      0x30  @ control register

      .equ    UART_FR,      0x18  @ flag register
      .equ    UART_DR,      0x00  @ data register

      .equ    BIT_CONF,     0x78  @ 2 bytes, FIFOs, 2 stop bits, par.
      .equ    BIT_ACTF,     0x301 @ RXE, TXE, UARTEN (activar UART0)
      .equ    UART_TXFF,    0x20  @ transmit FIFO full
      .equ    UART_RXFE,    0x10  @ receive FIFO empty
      .equ    UART_OE,      0x800 @ overrun error bit
      .equ    UART_PE,      0x200 @ parity error bit
```

```
34        .equ    UART_FE,      0x100 @ framing error bit
35        .equ    UART_BE,      0x400 @ break error bit
36
37        .equ    ERR_C,        0xFF  @ valor para sinalar un erro
38        .equ    EOT_C,        0x03  @ carácter de final de texto (EOT)
39
40        .equ    B_SIZE,       20    @ tamaño do búffer dew recepción
41
42        .data
43  mesaxe:
44        .ascii "Hai alguén do outro lado?"
45        .byte  EOT_C
46
47  buffer:
48        .space B_SIZE
49
50        .text
51  /*
52  -------------------------------------------------------------
53      CODIGO THUMB
54  -------------------------------------------------------------
55  */
56        .code16
57        .align 2
58  /*
59    Inicializamos a UART
60
61    uart0_init(dir. base)
62  */
63  uart0_init:
64  @
65  @ ponemos o divisor da tasa en baudios
66  @ (3Mhz / (115200 * 16)) = 1.62760416667 = 0b1.101000
67  @
68        mov   r1, #1
69        str   r1,[r0,#UART_IBRD]
70        mov   r1, #0x28
71        str   r1,[r0,#UART_FBRD]
72  @
73  @ fixamos a paridade, tamaño de palabra e
74  @ activamos buffers
75  @
76        mov   r1, #BIT_CONF
77        str   r1,[r0,#UART_LCRH]
78  @
79  @ enmascaramos todas as interrupcións
80  @
81        mov   r1, #0
82        str   r1,[r0,#UART_IMSC]
```

```
@
@ activamos a uart0 para recepción e transmisión
@
      ldr  r1, =BIT_ACTF
      str  r1,[r0,#UART_CR]

      mov  pc, lr

/*
  Enviamos un byte mediante espera activa.

  uart0_put(dir_base, byte)

*/
uart0_put:
@
@ Lemos o estado da UART0 e miramos se estamos listos
@ para transmitir. Se non, esperamos.
@
      ldr  r2,[r0,#UART_FR]
      mov  r3,#UART_TXFF
      tst  r2,r3
      bne  uart0_put         @ esperamos a estar listos

      str  r1,[r0,#UART_DR]  @ enviamos o dato
      mov  pc, lr

/*
  Recibimos un byte mediante espera activa.

  byte = uart0_get(dir_base)

  se hay calquera erro, devolvemos o valor ERR_C

*/
uart0_get:
@
@ Lemos o estado da UART0 e miramos se estamos listos
@ para recibir. Se non, esperamos.
@
      ldr  r2,[r0,#UART_FR]
      mov  r3,#UART_RXFE
      tst  r2,r3
      bne  uart0_get         @ esperamos a estar listos

      ldr r1,[r0,#UART_DR]   @ lemos o dato
@
@ miramos se error de overrun
@
```

```
132       ldr r3,=UART_OE
133       tst r1,r3
134       bne get_ok1
135       ldr r0,=ERR_C
136       b   sair_get
137
138 get_ok1:
139 @
140 @ miramos se error de paridade
141 @
142       ldr r3,=UART_PE
143       tst r1,r3
144       bne get_ok2
145       ldr r0,=ERR_C
146       b   sair_get
147
148 get_ok2:
149 @
150 @ miramos se error de framing
151 @
152       ldr r3,=UART_FE
153       tst r1,r3
154       bne get_ok3
155       ldr r0,=ERR_C
156       b   sair_get
157
158 get_ok3:
159 @
160 @ miramos se error de break
161 @
162       ldr r3,=UART_BE
163       tst r1,r3
164       bne get_ok4
165       ldr r0,=ERR_C
166       b   sair_get
167
168 get_ok4:
169       mov   r0, r1    @ pasamos o carácter leído a r0
170
171 sair_get:
172       mov  pc, lr
173
174 /*
175   uart0_tst
176
177   Enviamos unha mensaxe pola UART0 e recibimos outra mensaxe
178   e a pintamos na saída estándar
179
180 */
```

```
uart0_tst:
      push {r4-r6, lr}
      bl   uart0map   @ dirección base da UART0 en r0
      bl   uart0_init @ inicializamos a UART0
@
@ Enviamos mensaxe. Enviamos caracteres ata enviar un EOT
@
      ldr  r6,=mesaxe

outro_p:
      ldrb r1,[r6]
      bl   uart0_put
      cmp  r1,#EOT_C
      beq  put_f
      add  r6,#1
      b    outro_p
@
@ recibimos mensaxe. Leemos caracteres ata que se produza un erro
@ ou se reciba o carácter EOT.
@
put_f:
      mov  r4,r0
      ldr  r5,=ERR_C

      ldr  r6,=buffer
outro_g:
      bl   uart0_get
      cmp  r0, r5
      beq  get_f
      cmp  r0, #EOT_C
      beq  get_f

      strb r0,[r6]
      add  r6,#1
      mov  r0,r4
      b    outro_g
@
@ feito. Pintamos calquera cousa que teñamos recibido
@
get_f:
     mov  r0,#0
     str  r0,[r6]
     ldr  r0,=buffer
     blx  printf
@
@ Liberamos a UART
@
     mov   r0, r4
     bl    uart0umap
```

```
230
231      pop {r4-r6, pc}
232
233  /*
234  ---------------------------------------------------------
235      FIN DO CODIGO THUMB
236  ---------------------------------------------------------
237  */
238
239  /*
240     Programa principal de entrada desde o SO
241  */
242
243        .arm
244        .align 4
245        .global main
246  main:
247        push {r4, lr}
248        blx  uart0_tst
249        pop  {r4, lr}
250        bx   lr
251
252      .end
```

5.2 Entrada e saída coas bibliotecas do sistema

Nos nosos programas en ensamblador ARM/Thumb, teremos que invocar as funcións das bibliotecas como subprogramas ou subrutinas, pasándolles en cada caso os parámetros necesarios do xeito e na orde esperada. Unha vez que o subprograma invocado realizou o seu labor, depositará os resultados no lugar acordado antes de devolvernos o control. O apéndice B describe como debemos pasar os parámetros e recoller os resultados nos nosos programas en ensamblador.

O comando `man` para acceder ao manual do sistema operativo é unha fonte de información excelente para obter detalles do uso de todas esas funcións, e máis concretamente sobre o prototipo ou signatura delas, é dicir, de que tipo e en que orde van os argumentos, e de que tipo é o resultado. En cada un dos exercicios de programación propostos no presente capítulo introduciremos brevemente as funcións utilizadas, indicando en que biblioteca atópanse, pero convidamos o lector a que consulte a correspondente entrada do manual para obter detalles interesantes das devanditas funcións e doutras funcións

relacionadas. As funcións de biblioteca atópanse nos capítulos 2 e 3 do manual. Por exemplo, o comando:

```
$ man 3 printf
```

proporciónanos todos os detalles da función `printf` para enviar información a un dispositivo de saída, por exemplo unha cadea de caracteres á pantalla do computador. Para obter detalles adicionais do propio comando `man`, podemos executar o comando:

```
$ man man
```

Entre outras cousas, a saída do devandito comando proporciónanos unha relación dos distintos capítulos do manual, ou das diferentes maneiras en que podemos formatear a información proporcionada para facela máis lexible.

A continuación presentamos unha serie de exercicios de entrada/saída utilizando as bibliotecas asociadas ós dispositivos da Raspberry Pi.

Exercicio 5.3

Realizar un programa, etiquetado como `lepin` que lea o estado dunha liña do GPIO da Raspberry PI. En concreto, o programa deberá ler 20 veces a liña 23 cunha cadencia de 1 segundo, e informar cada vez do estado do pin pola saída estándar.

Solución

Utilizamos os seguintes subprogramas da biblioteca `libgpiod`:

- `gpiod_chip_open_by_name(chipname)`. Devolve en `r0` o manejador do dispositivo GPIO, a partir do seu nome. O nome do GPIO da Raspberry Pi podémolo obter co comando `gpioinfo`. `chipname` (en `r0`) é a dirección de memoria dunha cadea terminada en 0 co nome do dispositivo, no noso caso `gpiochip0`.
- `gpiod_chip_get_line(chip, offset)`. Devolve en `r0` o manexador dunha liña do GPIO, a partir do seu número (`offset`). `chip` (`r0`) é unha palabra co manexador do dispositivo, e `offset` (`r1`) é unha palabra co número da liña. Podemos obter os números das liñas do GPIO e a súa relación cos pines físicos co comando `gpio readall` ou o comando `pinout`.

- `gpiod_line_request_input(line, consumer)` Configura unha liña do GPIO como entrada. Devolve en `r0` o valor 0 se o resultado foi correcto, e un número negativo se houbo erro. `line` (`r0`) é o manexador da liña e `consumer` (`r1`) a dirección dunha cadea de caracteres co identificador do consumidor.
- `gpiod_line_get_value(line)`. Espera por un evento do tipo rexistrado con `gpiod_line_request_rising_edge_events`, devolvendo o valor 0 se o resultado foi correcto, e un número negativo se houbo erro. `line` (`r0`) é o manexador da liña.
- `gpiod_line_release(line)`. Libera unha liña do GPIO. `line` (`r0`) é o manexador da liña a liberar.
- `gpiod_chip_close(chip)`, Pecha o dispositivo. `chip` (`r0`) é unha palabra co manexador do dispositivo.

Ademais utilizamos os subprogramas `printf` e `sleep` da biblioteca estándar. `printf` recibe en `r0` a dirección da cadea terminada en 0 que queremos imprimir xunto cos códigos de formato, así como os valores adicionais a imprimir, por esta orde, en `r1`, `r2`, `r3` e a pila. `sleep` realiza unha espera durante o tempo en segundos indicado en `r0`.

Input.s

```
/*

Programa en ARM/Thumb para ler o estado dun pin do GPIO da
Raspberry Pi

Ensamblado:

$ as -g Input.s -ou Input.ou
$ gcc -Wall -lgpiod -ou Input Input.ou

*/

/*
Cadeas utilizadas para acceder ao GPIO
*/
      .data
      .align      4
gpiochip:
      .asciz      "gpiochip0"
err_chip:
      .asciz      "Erro abrindo chip.\n"
err_pin:
      .asciz      "Erro accedendo ao pin %d.\n"
```

```
24  cons:
25        .asciz      "Consumer"
26  err_out:
27        .asciz      "Erro de pin %d como entrada (1).\n"
28  err_out2:
29        .asciz      "Erro de pin %d como entrada (2).\n"
30  msg_out:
31        .asciz      "Pin %d con valor %d.\n"
32
33  chip: .space 4            @ handler do chip
34  line: .space 4            @ handler do pin
35
36  /* Pin obxectivo */
37
38        .equ   PIN_OBJ, 23
39        .equ   N_BUC, 20
40
41        .text
42  /*
43  -------------------------------------------------------------
44      CODIGO THUMB
45  -------------------------------------------------------------
46  */
47        .code16
48        .align 2
49
50  /*
51   lepin: Le o pin PIN_OBJ do GPIO N_BUC veces
52  */
53
54  lepin:
55        push {r4, lr}                 @ apilamos seguindo o APCS
56                                      @ (múltiplos de 8 bytes)
57        ldr r0, =gpiochip             @ IDE do GPIO
58        blx gpiod_chip_open_by_name   @ abrimos o GPIO
59        cmp r0, #0                    @ se 0, erro
60        bne chip_ok
61        ldr r0, =err_chip             @ mensaxe de erro
62        blx printf                    @ e terminamos
63        b   final
64
65  /* Abrimos o chip, agora imos a por o pin PIN_OBJ */
66
67  chip_ok:
68        ldr r3, =chip                 @ gardamos o chip
69        str r0, [r3]                  @ r0 = handler do chip
70        mov r1, #PIN_OBJ              @ r1 = led en pin PIN_OBJ
71        blx gpiod_chip_get_line       @ abrimos o pin
72        cmp r0, #0                    @ se 0, erro
```

```
73        bne pin_ok
74        ldr r0, =err_pin              @ mensaxe de erro
75        mov r1, #PIN_OBJ
76        blx printf                    @ pechamos o chip
77        b   pecha_chip                @ e terminamos
78
79  /* Temos o pin PIN_OBJ a punto. Configurámolo como entrada */
80
81  pin_ok:
82        ldr r3, =line                 @ cargamos o handler do pin
83        str r0, [r3]                  @ r0 = handler do pin
84        ldr r1, =cons                 @ r1 = IDE do consumer
85        blx gpiod_line_request_input
86        cmp r0, #0                    @ se < 0, erro
87        bge output_ok
88        ldr r0, =err_out              @ mensaxe de erro
89        mov r1, #PIN_OBJ
90        blx printf                    @ pechamos o pin
91        b   pecha_pin                 @ e o chip, e terminamos
92
93  /* Temos o pin PIN_OBJ activado como entrada. Lémolo */
94
95  output_ok:
96        mov r4, #N_BUC                @ contador a N_BUC
97        ldr r3, =line
98        ldr r0, [r3]                  @ r0 = pin
99
100 outro: blx gpiod_line_get_value     @ lemos o pin
101       cmp r0, #0                    @ se < 0, erro
102       bge todook
103       ldr r0,=err_out2              @ mensaxe de erro
104       mov r1, #PIN_OBJ
105       blx printf                    @ pechamos o pin
106       b   pecha_pin                 @ e o chip, e terminamos
107
108 /* Imprimimos a lectura e preparamos outra iteración */
109
110 todook:
111       mov r2, r0                    @ lectura a r2 para printf
112       ldr r0, =msg_out
113       mov r1, #PIN_OBJ
114       blx printf
115       mov r0, #1                    @ facemos unha pausa de 1 segundo
116       blx  sleep
117       sub r4, #1                    @ decrementamos o contador
118       beq pecha_pin                 @ se terminamos, pechamos todo
119       ldr r3, =line                 @ r0 = pin
120       ldr r0, [r3]
121       b   outro
```

```
122
123 pecha_pin:
124       ldr r3, =line
125       ldr r0, [r3]                  @ r0 = número de pin
126       blx gpiod_line_release
127
128 pecha_chip:
129       ldr r3, =chip
130       ldr r0, [r3]                  @ r0 = número de chip
131       blx gpiod_chip_close
132
133 final:
134       pop {r4, pc}                  @ recuperamos a pila
135
136 /*
137 -------------------------------------------------------------
138     FIN DO CODIGO THUMB
139 -------------------------------------------------------------
140 */
141
142 /*
143       Programa principal de entrada desde o SO
144 */
145
146       .arm
147       .align 4
148       .global main
149 main:
150       push {r4, lr}
151       blx lepin
152       pop {r4, lr}
153       bx lr
154
155      .end
```

Exercicio 5.4

Realizar un programa, etiquetado como `setled` que acenda e apague un led conectado ao pin 23 do GPIO da Raspberry Pi. O programa deberá acender e apagar un led conectado ao pin 23 un total de 20 veces, cunha cadencia de 1 segundo entre cambios (20 segundos de intermitencia en total).

Solución

Utilizamos os seguintes subprogramas da biblioteca `libgpiod`:

- `gpiod_chip_open_by_name(chipname)` e `gpiod_chip_get_line (chip, offset)` descritos anteriormente (cf. exercicio 5.3).

- `gpiod_line_request_output(line, consumer, inival)`. Reserva un pin do GPIO como saída, devolve en `r0` o valor 0 se o resultado foi correcto, e un número negativo se houbo erro. `line` (`r0`) é o manexador (*handler*) da liña, `consumer` (`r1`) a dirección dunha cadea de caracteres co identificador do consumidor e `inival` (`r2`) o valor inicial do pin (1 ou 0).
- `gpiod_line_set_value(line, val)`. Pon unha liña do GPIO a 1 ou a 0, devolvendo o valor 0 se o resultado foi correcto, e un número negativo se houbo erro. `line` (`r0`) é o manexador da liña e `val` (`r1`) o novo valor do pin (1 ou 0).
- `gpiod_line_release(line)` e `gpiod_chip_close(chip)`, para liberar a liña e pechar o chip, xa descritos.

Ademais utilizamos os subprogramas `printf` e `sleep` da biblioteca estándar, xa descritos no exercicio 5.3.

SetLed.s

```
/*

Programa en ARM/Thumb para activar un led conectado
ao pin 23 do GPIO da Raspberry Pi 3B+.

Ensamblado:

$ as -g SetLed.s -ou SetLed.ou
$ gcc -Wall -lgpiod -ou SetLed SetLed.ou

*/

/*
Cadeas utilizadas para acceder ao GPIO
*/
      .data
      .align      4
gpiochip:
      .asciz      "gpiochip0"
err_chip:
      .asciz      "Erro abrindo chip.\n"
err_pin:
      .asciz      "Erro accedendo ao pin 23.\n"
cons:
      .asciz      "Consumer"
err_out:
      .asciz      "Erro de pin 23 como output (1).\n"
err_out2:
      .asciz      "Erro de pin 23 como output (2).\n"
```

```
30  msg_out:
31        .asciz      "Toggle do pin 23.\n"
32
33  chip: .space 4            @ handler do chip
34  line: .space 4            @ handler do pin
35
36        .text
37  /*
38  -------------------------------------------------------------
39      CODIGO THUMB
40  -------------------------------------------------------------
41  */
42        .code16
43        .align 2
44
45  /*
46   setled: Activa e desactiva un led no pin 23
47      do GPIO 20 veces
48  */
49
50  setled:
51        push {r4-r6, lr}              @ apilamos seguindo o APCS
52                                      @ (múltiplos de 8 bytes)
53        ldr r0, =gpiochip             @ IDE do GPIO
54        blx gpiod_chip_open_by_name   @ abrimos o GPIO
55        cmp r0, #0                    @ se 0, erro
56        bne chip_ok
57        ldr r0, =err_chip             @ mensaxe de erro
58        blx printf                    @ e terminamos
59        b   final
60
61  /* Abrimos o chip, agora imos a por o pin 23 */
62
63  chip_ok:
64        ldr r3, =chip                 @ gardamos o chip
65        str r0, [r3]                  @ r0 = handler do chip
66        mov r1, #23                   @ r1 = led en pin 23
67        blx gpiod_chip_get_line       @ abrimos o pin
68        cmp r0, #0                    @ se 0, erro
69        bne pin_ok
70        ldr r0, =err_pin              @ mensaxe de erro
71        blx printf                    @ pechamos o chip
72        b   pecha_chip                @ e terminamos
73
74  /* Temos o pin 23 a punto. Configurámolo como saída
75      e pómolo a 0 */
76
77  pin_ok:
78        ldr r3, =line                 @ cargamos o handler do pin
```

```
79        str r0, [r3]                  @ r0 = handler do pin
80        ldr r1, =cons                 @ r1 = IDE do consumer
81        mov r2, #0                    @ r2 = 0 (valor inicial)
82        blx gpiod_line_request_output
83        cmp r0, #0                    @ se < 0, erro
84        bge output_ok
85        ldr r0, =err_out              @ mensaxe de erro
86        blx printf                    @ pechamos o pin
87        b   pecha_pin                 @ e o chip, e terminamos
88
89  /* Temos o pin 23 activado como saída, posto a 0.
90     O toggleamos 0 -> 1 -> 0 ... 20 veces */
91
92  output_ok:
93        mov r5, #20                  @ contador a 20
94        ldr r3, =line
95        ldr r0, [r3]                 @ r0 = pin
96        mov r6, #1                   @ r1 = 1 (novo valor)
97        mov r1, r6                   @ novo valor
98        mov r4, #1                   @ r4: máscara para toggle con eor
99
100 outro: blx gpiod_line_set_value    @ toggle led (pin 23)
101       cmp r0, #0                   @ se < 0, erro
102       bge todook
103       ldr r0,=err_out2             @ mensaxe de erro
104       blx printf                   @ pechamos o pin
105       b   pecha_pin                @ e o chip, e terminamos
106
107 /* Preparamos outra iteración, cambiando o valor anterior do pin */
108
109 todook:
110       mov r0, #1                   @ facemos unha pausa de 1 segundo
111       blx  sleep
112       sub r5, #1                   @ decrementamos o contador
113       beq pecha_pin                @ se terminamos, pechamos todo
114       ldr r0,=msg_out              @ sacamos mensaxe do cambio
115       blx printf
116       ldr r3, =line                @ r0 = pin
117       ldr r0, [r3]
118       eor r6, r4
119       mov r1, r6                   @ r1 (t) = -r1(t-1)
120       b   outro
121
122 pecha_pin:
123       ldr r3, =line
124       ldr r0, [r3]                 @ r0 = número de pin
125       blx gpiod_line_release
126
127 pecha_chip:
```

```
128       ldr r3, =chip
129       ldr r0, [r3]                  @ r0 = número de chip
130       blx gpiod_chip_close
131
132 final:
133       pop {r4-r6, pc}               @ recuperamos a pila
134
135 /*
136 -------------------------------------------------------------
137     FIN DO CODIGO THUMB
138 -------------------------------------------------------------
139 */
140
141 /*
142       Programa principal de entrada desde o SO
143 */
144
145       .arm
146       .align 4
147       .global main
148 main:
149       push {r4, lr}
150       blx setled
151       pop {r4, lr}
152       bx lr
153
154      .end
```

Exercicio 5.5

Realizar un programa, etiquetado como `event` que detecte o cambio de estado dun pin do GPIO da Raspberry PI. En concreto, o programa deberá detectar un evento de flanco ascendente do pin 23 nun prazo máximo de 1 segundo. Se no devandito prazo o evento non se produce, o programa termina notificando pola saída estándar que devandito evento non se produciu.

Solución

Utilizamos os seguintes subprogramas da biblioteca `libgpiod`:

- `gpiod_chip_open_by_name(chipname)` e `gpiod_chip_get_line (chip, offset)` descritos anteriormente (cf. exercicio 5.3).
- `gpiod_line_request_rising_edge_events(line, consumer)`. Solicita as notificacións dun cambio nunha liña do GPIO. Devolve en `r0` o valor 0 se o resultado foi correcto, e un número negativo

se houbo erro. `line` (`r0`) identifica o manexador (*handler*) da liña e `consumer` (`r1`) a dirección dunha cadea de caracteres co identificador do consumidor.

- `gpiod_line_event_wait(line, ts)`. Espera por un evento do tipo rexistrado con `gpiod_line_request_rising_edge_events`, devolvendo o valor 0 se o resultado foi correcto, e un número negativo se houbo erro. `line` (`r0`) é o manexador da liña e `ts` (`r1`) a dirección de memoria dunha estrutura de 2 palabras para especificar un límite de tempo. A primeira palabra indica o número de segundos e a segunda o número de nanosegundos.
- `gpiod_line_event_read(line, event)`. Le o último evento nunha liña, devolvendo o valor 0 se o resultado foi correcto, e un número negativo se houbo erro. `line` (`r0`) é o manexador da liña e `event` (`r1`) a dirección de memoria dunha estrutura onde se depositará información do evento, composta por tres palabras. As dúas primeiras indican a estimación temporal do momento do evento (a primeira palabra indica segundos e a segunda nanosegundos), e a terceira indica o tipo de evento (ascendente ou descendente).
- `gpiod_line_release(line)` e `gpiod_chip_close(chip)`, para liberar a liña e pechar o chip, xa descritos no exercicio 5.3.

Ademais utilizamos o subprograma `printf` descrito no exercicio 5.3.

Event.s

```
/*

Programa en ARM/Thumb para detectar o cambio de estado dun pin do
GPIO da Raspberry Pi.

Ensamblado:

$ as -g Event.s -ou Event.ou
$ gcc -Wall -lgpiod -ou Event Event.ou
*/

/*
Cadeas utilizadas para acceder ao GPIO
*/
      .data
      .align      4
gpiochip:
      .asciz      "gpiochip0"
err_chip:
      .asciz      "Erro abrindo chip.\n"
```

```
21 err_pin:
22       .asciz      "Erro accedendo ao pin %d.\n"
23 cons:
24       .asciz      "Consumer"
25 err_nreq:
26       .asciz      "Fallou a solicitude de evento en liña %d.\n"
27 err_evnw:
28       .asciz      "Fallou a espera por un evento no pin %d.\n"
29 err_evnr:
30       .asciz      "Fallou a lectura do evento no pin %d\n"
31 msg_tout:
32       .asciz      "Timeout sen flanco ascendente no pin %d\n"
33 msg_evnt_ts1:
34       .asciz      "Evento (ascendente) en pin %d en (%d, %d). "
35 msg_evnt_ts2:
36       .asciz      "Info do evento: %d\n"
37
38 @ Variables usadas
39
40 chip: .space 4            @ handler do chip
41 line: .space 4            @ handler do pin
42
43 timespec:                 @ estrutura para timeot
44 secs: .word  1            @ * número de segundos
45 nsec: .word  0            @ * número de nanosegundos
46
47 evnt:                     @ info do evento lido:
48 etse: .space 4            @ * tempo do evento (secs)
49 etns: .space 4            @ * tempo do evento (nsecs)
50 etip: .space 4            @ * tipo de evento
51
52 /* Pin obxectivo */
53
54       .equ   PIN_OBJ, 23
55
56       .text
57 /*
58 -------------------------------------------------------------
59     CODIGO THUMB
60 -------------------------------------------------------------
61 */
62       .code16
63       .align 2
64
65 /*
66  event: Espera por un flanco ascendente no pin PIN_OBJ
67 */
68
69 event:
```

```
70        push {r4, lr}                 @ apilamos seguindo o APCS
71                                      @ (múltiplos de 8 bytes)
72        ldr r0, =gpiochip             @ IDE do GPIO
73        blx gpiod_chip_open_by_name   @ abrimos o GPIO
74        cmp r0, #0                    @ se 0, erro
75        bne chip_ok
76        ldr r0, =err_chip             @ mensaxe de erro
77        blx printf                    @ e terminamos
78        b   final
79
80  /* Abrimos o chip, agora imos a por o pin PIN_OBJ */
81
82  chip_ok:
83        ldr r3, =chip                 @ gardamos o chip
84        str r0, [r3]                  @ r0 = handler do chip
85        mov r1, #PIN_OBJ              @ r1 = led en pin PIN_OBJ
86        blx gpiod_chip_get_line       @ abrimos o pin
87        cmp r0, #0                    @ se 0, erro
88        bne pin_ok
89        ldr r0, =err_pin              @ mensaxe de erro
90        mov r1, #PIN_OBJ
91        blx printf                    @ pechamos o chip
92        b   pecha_chip                @ e terminamos
93
94  /* Temos o pin PIN_OBJ a punto. Pedimos que se nos notifiquen
95     os eventos de flanco ascendente no pin PIN_OBJ */
96
97  pin_ok:
98        ldr r3, =line                 @ cargamos o handler do pin
99        str r0, [r3]                  @ r0 = handler do pin
100       ldr r1, =cons                 @ r1 = IDE do consumer
101       blx gpiod_line_request_rising_edge_events
102       cmp r0, #0                    @ se r0 < 0, erro
103       bge nreq_ok
104       ldr r0, =err_nreq             @ mensaxe de erro
105       mov r1, #PIN_OBJ
106       blx printf                    @ pechamos o pin
107       b   pecha_pin                 @ e o chip, e terminamos
108
109 /* Rexistramos a notificación dos eventos de flanco
110    ascendente no pin PIN_OBJ. Pómonos á espera
111    do seguinte flanco ascendente */
112
113 nreq_ok:
114       ldr r3, =line
115       ldr r0, [r3]                @ r0 = pin
116       ldr r1, =timespec           @ r1 = timeout (1 sec.)
117       blx gpiod_line_event_wait   @ esperamos polo flanco
118       cmp r0, #0                  @ se r0 < 0, erro
```

```
119       bge todook
120       ldr r0,=err_evnw            @ mensaxe de erro
121       mov r1, #PIN_OBJ
122       blx printf                  @ pechamos o pin
123       b   pecha_pin               @ e o chip, e terminamos
124
125 /* Vemos se volvemos por timeout ou por un flanco*/
126
127 todook:
128       bne flanco                  @ se r0 > 0 pillamos flanco
129       ldr r0,=msg_tout            @ sacamos mensaxe do timeout
130       mov r1, #PIN_OBJ
131       blx printf
132       b   pecha_pin               @ pechamos todo e terminamos
133
134 /* Pillamos flanco. Lemos o evento  */
135
136 flanco:
137       ldr r3, =line
138       ldr r0, [r3]                @ r0 = pin
139       ldr r1, =evnt               @ r1 = dirección do evento
140       blx gpiod_line_event_read
141       cmp r0, #0                  @ se r0 < 0, erro
142       bge todook2
143       ldr r0, =err_evnr           @ sacamos mensaxe de erro
144       mov r1, #PIN_OBJ
145       blx printf
146       b   pecha_pin               @ e o chip, e terminamos
147
148 /* Sacamos unha mensaxe de evento lido
149    correctamente, con datos, e terminamos */
150
151 todook2:
152       ldr r0, =msg_evnt_ts1        @ sacamos mensaxe do timeout
153       mov r1, #PIN_OBJ
154       ldr r4, =evnt                @ e terminamos
155       ldr r2, [r4]
156       ldr r3, [r4, #4]
157       blx printf
158       ldr r0, =msg_evnt_ts2
159       ldr r1, [r4, #8]
160
161 pecha_pin:
162       ldr r3, =line
163       ldr r0, [r3]                @ r0 = número de pin
164       blx gpiod_line_release
165
166 pecha_chip:
167       ldr r3, =chip
```

```
168       ldr r0, [r3]                @ r0 = número de chip
169       blx gpiod_chip_close
170
171 final:
172       pop {r4, pc}                @ recuperamos a pila
173
174 /*
175 -------------------------------------------------------------
176     FIN DO CODIGO THUMB
177 -------------------------------------------------------------
178 */
179
180 /*
181       Programa principal de entrada desde o SO
182 */
183
184       .arm
185       .align 4
186       .global main
187 main:
188       push {r4, lr}
189       blx event
190       pop {r4, lr}
191       bx lr
192
193      .end
```

Exercicio 5.6

Realizar un programa, etiquetado como `i2ctst`, que reciba 4 bytes do dispositivo de dirección `0x5a` conectado ao porto I2C da Raspberry Pi, e que despois envíe 4 bytes ao mesmo dispositivo.

Solución

Utilizamos os seguintes subprogramas da biblioteca `ioctl`:

- `open(disp, modo)`. Para abrir o interfaz I2C. Devolve en `r0` o manexador do dispositivo I2C, a partir do seu nome. O nome do interfaz I2C da Raspberry Pi é `/dev/23c-1`. `disp` (en `r0`) é a dirección de memoria dunha cadea terminada en 0 co nome do dispositivo, no noso caso `/dev/23c-1`, e modo (`r1`) tomará o valor 0 para abrilo en modo lectura; 1 para abrilo en modo escritura e 2 para abrilo en modo lectura/escritura.
- `ioctl(handler, I2C_SLAVE, dir)`. Subprograma da biblioteca estándar para manipular os parámetros de dispositivos. En

nuesto caso, utilizámola para tomar o interfaz I2C e conectarnos cun dispositivo escravo na dirección especificada. Para iso pasamos o comando `I2C_SLAVE` en `r1` e a dirección do escravo en `r2`. Devolve en `r0` o valor 0 se o resultado foi correcto, e un número negativo se houbo erro.

- `read(hanlder, buffer, tamaño)` Recibe `tamaño` caracteres do bus I2C e deposítaos na zona de memoria indicada por `buffer`. O manexador pásase en `r0`, a dirección da zona de memoria en `r1`, e o tamaño en `r2`. Devolve en `r0` o número de caracteres recibidos.
- `write(hanlder, buffer, tamaño)`. Envía `tamaño` caracteres ao bus I2C desde a zona de memoria indicada por `buffer`. O manexador pásase en `r0`, a dirección da zona de memoria en `r1`, e o tamaño en `r2`. Devolve en `r0` o número de caracteres enviados.
- `close(handler)`. Libera o bus I2C. `handler` (`r0`) é o manexador obtido co comando `open`.

Ademais utilizamos o subprograma `printf` da biblioteca estándar (cf. exercicio 5.3).

I2C.s

```
/*

Exemplo de lectura e escritura con I2C.

Ensamblado:

$ as -g -ou I2C.ou I2C.s
$ gcc -g -Wall -ou I2C I2C.ou

Primeiro hai que activar os portos I2C con raspi-config.

Para comprobar os I2C da raspi ytilizamos i2cdetect (ver man)

$ súo i2cdetect -e 0
$ súo i2cdetect -e 1

Para atopar a dirección do escravo, vemos o resultado
de i2cdetect.

*/


/*
Cadeas utilizadas para acceder ao I2C
```

```
*/
      .data
      .align      4

i2c_dev:
      .asciz      "/dev/i2c-1"
err_dev:
      .asciz      "Erro abrindo o bus I2C.\n"
err_slave:
      .asciz      "Erro contactando co escravo na dir %X.\n"
err_lect:
      .asciz      "Erro lendo bytes (lidos: %d).\n"
msg_out:
      .asciz      "Lectura correcta. Bytes lidos: %s.\n"
err_esc:
      .asciz      "Erro escribindo bytes (escritos: %d).\n"
msg_ok_esc:
      .asciz      "Escritura correcta.\n"

@ Variables usadas

i2c_hdrl:
      .space 4                    @ handler do bus

      .equ   SLAVE_ADDR, 0x5a     @ Dirección I2C do escravo
      .equ   TX_LON, 4            @ bytes a transmitir
      .equ   O_RDWR, 2            @ L/E (L: 0; E: 1; L/E: 2)

buffer:
      .space TX_LON
      .byte  0

/*
-------------------------------------------------------------
    CODIGO THUMB
-------------------------------------------------------------
*/
      .code16
      .align 2

/*
i2ctst: Proba a ler e escribir do bus I2C
*/
i2ctst:
      push {r4, lr}                @ apilamos seguindo o APCS
                                   @ (múltiplos de 8 bytes)
      ldr r0, =i2c_dev             @ device do bus I2C
      mov r1, #O_RDWR              @ modo lectura/escritura (L/E)
      blx open                     @ abrimos o bus I2C
```

```
75        cmp r0, #0                    @ se menor que 0, erro
76        bge dev_ok
77        ldr r0, =err_dev              @ mensaxe de erro
78        blx printf                    @ e terminamos
79        b   final
80
81  dev_ok:
82        ldr r3, =i2c_hdrl             @ gardamos o handler
83        str r0, [r3]                  @ r0 = handler do chip
84
85  /* Xa temos o bus I2C aberto.
86     Vemos se podemos acceder ao escravo. */
87
88        mov r1, #I2C_SLAVE            @ Contactar co escravo
89        mov r2, #SLAVE_ADDR           @ dirección do escravo
90        blx ioctl                     @ contactamos con iocl
91        cmp r0, #0                    @ se menor que 0, erro
92        bge slave_ok
93        ldr r0, =err_slave            @ mensaxe de erro
94        mov r1, #SLAVE_ADDR
95        blx printf                    @ pechamos o I2C
96        b   close_i2c                 @ e terminamos
97
98  /* Contactamos co escravo.
99     Agora imos ler uns cuantos bytes */
100
101 slave_ok:
102       ldr r3, =i2c_dev
103       ldr r0, [r3]
104       ldr r1, =buffer
105       mov r2, #TX_LON
106       blx read
107       cmp r0, #TX_LON
108       beq lect_ok
109       mov r1, r0
110       ldr r0, =err_lect             @ mensaxe de erro
111       blx printf                    @ pechamos o I2C
112       b   close_i2c                 @ e terminamos
113
114 /* Imprimimos os bytes lidos */
115
116 lect_ok:
117       ldr r0, =msg_out
118       ldr r1, =buffer
119       blx printf
120
121 /* Lectura correcta. Escribimos */
122
123       ldr r3, =i2c_dev
```

```
124       ldr r0, [r3]
125       ldr r1, =buffer
126       mov r2, #TX_LON
127       blx write
128       cmp r0, #TX_LON
129       beq esc_ok
130       mov r1, r0
131       ldr r0, =err_esc              @ mensaxe de erro
132       blx printf                    @ pechamos o I2C
133       b   close_i2c                 @ e terminamos
134
135 esc_ok:
136      ldr  r0, =msg_ok_esc
137      blx printf
138
139 close_i2c:
140      ldr r3, =i2c_dev
141      ldr r0, [r3]
142      blx close
143
144 final:
145      pop {r4, pc}                   @ recuperamos a pila
146
147 /*
148 -------------------------------------------------------------
149     FIN DO CODIGO THUMB
150 -------------------------------------------------------------
151 */
152
153 /*
154       Programa principal de entrada desde o SO
155 */
156
157       .arm2
158       .align 4
159       .global main
160 main:
161       push {r4, lr}
162       blx i2ctst
163       pop {r4, lr}
164       bx lr
165
166      .end
```

5.3 Recapitulación

Nun sistema microprocesador, todas as operacións de comunicación co exterior realízanse utilizando dispositivos de entrada e saída. Existen multitude de dispositivos con características e labores moi diferentes. Temos periféricos como o teclado, a pantalla ou a impresora; dispositivos de almacenamento como discos duros ou de estado sólido; interfaces de comunicación inalámbrica baseados en protocolos como WiFi, Bluetooth ou RFID; sensores e actuadores que nos permiten interactuar cunha enorme variedade de sistemas en automóbiles, procesos de fabricación, electrodomésticos, aeronaves... Como consecuencia desta variedade, cada dispositivo ten o seu propio conxunto de rexistros e o seu propio repertorio de comandos e configuracións.

O programador necesita comprender como funciona cada dispositivo que utiliza nos seus proxectos e o cometido e formato de cada rexistro, para poder así progamarlo no baixo nivel. Para iso, os fabricantes de dispositivos proporcionan documentación con toda a información necesaria (follas de características, manuais de referencia, notas de aplicación, etc.).

Existen dous grandes ámbitos onde é necesario programar dispositivos no baixo nivel. Por unha banda, os programadores de sistemas operativos necesitan acceder ao funcionamento detallado dos dispositivos ao programar os manexadores de dispositivos ou *device drivers*. Os sistemas operativos proporcionan unha capa de abstracción que permite aos programadores de aplicacións utilizar os dispositivos seguindo un modelo común, máis sinxelo, a través das chamadas ao sistema ou das bibliotecas de entrada e saída, como puidemos ver nos exercicios deste capítulo. De todos os xeitos, para proporcionar ese nivel de abstracción é necesario que o programador de sistemas, encargado de desenvolver un sistema operativo, aplique os seus coñecementos sobre o funcionamento detallado do dispositivo para escribir esas bibliotecas.

Outro ámbito de aplicación onde é necesaria a programación de dispositivos de entrada e saída a baixo nivel é o deseño de sistemas embebidos, por exemplo moitos sistemas do ámbito IoT. En sistemas como estes, xeralmente de pequeno tamaño e con limitacións en canto a memoria e recursos, é habitual non dispor de bibliotecas de acceso ou de manexadores para os dispositivos de entrada e saída que queiramos utilizar. Nestes casos, será necesario acceder directamente ao dispositivo.

Mesmo cando o sistema operativo proporciona un manexador, pode ser conveniente acceder ao dispositivo sen intermediarios. Por exemplo, algúns dispositivos poden dispor de características ou soportar modos de funcionamento que non foron implementados no manexador en aras dunha maior simplicidade para o programador de aplicacións ou para garantir certa compatibilidade entre dispositivos semellantes. Como vimos neste capítulo, os sistemas operativos adoitan incluír algún mecanismo para que o programador poida facer visibles os rexistros de baixo nivel dun dispositivo a través do espazo de memoria asignado aos programas de usuario.

Apéndice A
Exercicios introdutorios para Raspberry Pi

Neste apéndice reelaboramos os exercicios introdutorios do capítulo 1 para o caso da Raspberry Pi. Propomos unha serie de exercicios sinxelos orientados a familiarizarse coas ferramentas de programación de GNU dispoñibles en Raspberry Pi OS. Ademais, repasamos algúns conceptos básicos relacionados coa representación e almacenamento de información na arquitectura ARM T32/Thumb.

O apéndice C inclúe, a modo de referencia rápida, información sobre as instrucións (cf. cadros C.4 a C.8) e as directivas do ensamblador de GNU utilizadas neste libro (cf. cadro C.9).

Exercicio A.1

O seguinte programa inicializa os rexistros `r0` a `r3` con 4 valores numéricos en decimal, hexadecimal, octal e binario, respectivamente. Edita o programa utilizando o editor da túa preferencia, ensámblalo e enlázao tal como descríbese no apartado 1.2, carga o teu programa en GDB e responde as preguntas.

e_intro_1.s

```
      .text
      .code16
      .align 2

ex1:  mov r0, #30
      mov r1, #0x42
      mov r2, #0102
      mov r3, #0b1000010
fin:  bx lr

      .arm
      .align 4
      .global main
main:
```

```
    push {r4, lr}
    blx ej1
    pop {r4, lr}
    bx lr
.end
```

1. Como se mostran os números anteriores ao listar o programa en GDB? Están na mesma base que no código orixinal? En que base están representados?

2. Executa o programa paso a paso. Que números almacénanse nos rexistros `r0` a `r3`?

   Solución

   Os datos inmediatos no código desensamblado represéntanse en decimal, co que ao observar o código en GDB obteriamos algo como:

   ```
   0x103d2 <ex1>    movs   r0, #30
   0x103d2 <ex1+2>  movs   r1, #66
   0x103d4 <ex1+4>  movs   r2, #66
   0x103d6 <ex1+6>  movs   r3, #66
   0x103d8 <fin>    bx     lr
   ```

   En canto á representación do contido dos rexistros, en xeral represéntase en hexadecimal, salvo que se solicite expresamente un formato diferente (por exemplo, co comando `x` de GDB). Así, ao executar o programa o contido dos rexistros represéntase como segue tras executar o comando `info registers` de GDB:

   ```
   r0   0x1e
   r1   0x42
   r2   0x42
   r3   0x42
   ```

Exercicio A.2

Edita o código presentado a continuación, ensámblao, analízao con GDB e responde as preguntas.

e_intro_2.s

```
      .data
word1: .word 11
word2: .word 0x11
```

```
word3: .word 011
word4: .word 0b11

       .text
       .code16
       .align 2

nada:   bx lr

       .arm
       .align 4
       .global main
main:
        push {r4, lr}
        blx nada
        pop {r4, lr}
        bx lr
       .end
```

1. Identifica as posicións de memoria onde se almacenaron os datos definidos no programa. Identifica os catro valores en hexadecimal.

2. En que direccións almacénanse os catro valores? Por que estas direccións de memoria son múltiplos de catro en lugar de ser direccións consecutivas?

3. Cales son os valores das etiquetas `word1`, `word2`, `word3` e `word4`?

Solución

O resultado obtido con GDB é o seguinte:

```
[0x00021024]  0x0000000B
[0x00021028]  0x00000011
[0x0002102C]  0x00000009
[0x00021030]  0x00000003
```

As direccións, no caso da Raspberry Pi, son direccións de memoria virtual asignadas polo sistema operativo no momento de cargar o programa, polo que poden variar dun computador a outro, ou mesmo dunha sesión a outra. En calquera caso, o importante é decatarnos de que cada palabra ocupa 4 bytes e de que o convenio de almacenamento ou *endianness* é o extremista menor (*little endian*), é dicir, o byte menos

significativo da palabra almacénase na posición de memoria de dirección máis baixa.

Por tanto, os valores das etiquetas `word1`, `word2`, `word3` e `word4` serán os indicados a continuación:

Etiqueta	Valor
`word1`	`0x00021024`
`word2`	`0x00021028`
`word3`	`0x0002102C`
`word4`	`0x00021030`

Exercicio A.3

Ensambla o seguinte código:

e_intro_3.s

```
      .data
wrds: .word 11, 0x11, 011, 0b11

      .text
      .code16
      .align 2

nada:  bx lr

      .arm
      .align 4
      .global main
main:
       push {r4, lr}
       blx nada
       pop {r4, lr}
       bx lr
      .end
```

Hai algún cambio nos valores almacenados na memoria con respecto aos almacenados no caso do programa anterior? Están no mesmo lugar?

Solución

Non hai ningún cambio no que respecta ao almacenamento dos datos anteriores. O único cambio sería que agora definimos unha única etiqueta (`wrds`) con valor `0x00021024`.

Exercicio A.4

Ensambla o seguinte código:

e_intro_4.s

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

      .text
      .code16
      .align 2

nada:  bx lr

      .arm
      .align 4
      .global main
main:
       push {r4, lr}
       blx nada
       pop {r4, lr}
       bx lr
      .end
```

1. Que valores almacenáronse na memoria? En que posicións?
2. Cal é o valor da etiqueta `bys`?

   Solución

   Almacénase un byte en cada posición de memoria, comezando na dirección `0x00021024`. A etiqueta `bys` tomará devandito valor.

Exercicio A.5

Agora ensambla o seguinte código:

e_intro_5.s

```
      .data
strg: .ascii "abác"
byte: .byte  0xff

      .text
      .code16
      .align 2

nada:  bx lr

      .arm
      .align 4
      .global main
main:
       push {r4, lr}
       blx nada
```

```
    pop {r4, lr}
    bx lr
.end
```

1. Que rango de posicións de memoria reservouse para a variable etiquetada como `strg`?
2. Como se representa en memoria a cadea `abác`? Como se representa o carácter `á`?
3. Pescuda cal é o sistema de codificación de caracteres da túa contorna de traballo (UTF-8, ASCII, ISO Latin 9 ...).
4. Cal é o valor da etiqueta `byte`?

Solución

Trátase dunha cadea de caracteres. Raspberry Pi OS baséase no ensamblador de GNU, que utiliza UTF-8 como sistema de representación de caracteres na maioría das plataformas, incluída a propia Raspberry Pi.

Por tanto, para a cadea etiquetada como `streg` reservaranse 5 posicións e a representación en memoria byte a byte será:

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

Vemos que o carácter `á` represéntase co código de dous bytes (`0xc3`, `0xa1`) en UTF-8, de acordo co descrito no apéndice D.

Finalmente, o valor da etiqueta `byte` será `0x00021029`.

Exercicio A.6

Ensambla e analiza o seguinte código:

e_intro_6.s

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

      .text
      .code16
```

```
      .align 2

nada:  bx lr

      .arm
      .align 4
      .global main
main:
       push {r4, lr}
       blx nada
       pop {r4, lr}
       bx lr
      .end
```

1. Cantas posicións de memoria resérvanse para a variable `gap`?
2. Poderían lerse ou escribirse os catro bytes utilizados pola variable `gap` coma se fosen unha palabra? Por que?
3. Cal é o valor da etiqueta `b1`? E da etiqueta `b2`?
4. Cal é o valor da etiqueta `bign`? Poderían lerse ou escribirse os catro bytes que comezan na dirección `bign` coma se fosen unha palabra? Por que?
5. Agrega unha directiva `.balign` ao código anterior para que a variable etiquetada como `bign` alíñese cun límite de palabra (é dicir, cunha dirección múltiplo de catro).

Solución

Para a variable `gap` resérvanse 4 posicións de memoria. Non poderemos acceder en lectura ou escritura aos 4 bytes de `gap` coma se fosen unha palabra porque os accesos a palabra en ARM teñen que estar aliñados a unha dirección múltiplo de 4.

Para poder acceder a `bign` como unha palabra teriamos que engadir unha directiva de aliñación `.balign 4` ou ben `.align 2` antes da definición da devandita etiqueta. En ambos os casos aliñaríase o espazo etiquetado por `bign` a unha dirección múltiplo de 4.

O código é o seguinte:

e_intro_7.s

```
1       .data
2  b1:  .hword  0x11       @ isto ocupa dous bytes
3  gap: .space  4          @ catro bytes adicionais
```

```
4  b2:    .byte   0x22        @ un byte adicional (total 7 bytes)
5         .balign 4           @ aliñamos a un múltiplo de 4
6  bign:  .word   0x33445566 @ agora podemos definir unha palabra
7
8         .text
9         .code16
10        .align 2
11
12 nada:   bx lr
13
14        .arm
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

As chamadas ao sistema e o paso de parámetros

O sistema operativo Raspberry Pi OS é un sistema operativo baseado na distribución Debian de Linux. Como todo sistema operativo, Raspberry Pi OS é un conxunto de módulos software que serve para xestionar os diferentes elementos que constitúen o computador como sistema: programas en execución, arquivos, dispositivos de rede, pantallas, impresoras, cámaras, temporizadores, portos de entrada e saída, etc.

Desde o punto de vista dun programa en ensamblador, podemos interpretar que o sistema operativo é á vez un gran provedor de servizos e un vixiante que toma medidas cando algo sae mal ou cando tentamos facer algo que non estamos autorizados. É un programa fundamental que sempre está preparado para atender as solicitudes dos programas executados polos usuarios, de xeito que se garanta a convivencia ordenada dos distintos programas en execución. Raspberry Pi OS é un sistema operativo da familia Unix, e por tanto comparte moitas características coa longa liñaxe de sistemas operativos da devandita familia.

En xeral, un programa en execución solicita servizos ao sistema operativo realizando *chamadas ao sistema*. Unha chamada ao sistema é conceptualmente o mesmo que chamar a un subprograma, pero máis complexo de realizar na práctica. Isto é así porque o sistema operativo impón unha serie de requisitos e maneiras de facer consecuencia de que é unha parte crítica do computador que ten que protexerse de posibles erros ou fallos. Como consecuencia destas restricións, necesitamos o apoio da arquitectura (no noso caso ARM) para implementar de forma segura o mecanismo de chamadas ao sistema.

En Raspberry Pi OS realizaríamos as chamadas ao sistema utilizando a instrución **`swi`** para xenerar unha interrupción software ou chamada a supervisor. En xeral, esta instrución recibiría un operando de 24 bits transparente para o procesador, pero que podería ser utilizado polo sistema operativo para indicarlle que servizo solicitamos. Nos sistemas operativos baseados en Linux, establécese con todo que devandito operando tomará sempre o valor 0 e o servizo solicitado indícase a través do rexistro `r7`, así que en Raspberry Pi OS as chamadas ao sistema serían sempre a través da instrución **`swi`** `#0`.

Do mesmo xeito que calquera outra función ou subprograma, as chamadas ao sistema poden requirir parámetros. No caso de Linux, ningunha chamada ao sistema necesita máis de 7 argumentos, e estes pasaríanse nos rexistros `r0` – `r6`. Se a chamada ao sistema devolve algún valor, devolveríase no rexistro `r0`. Por exemplo, o código que presentamos a continuación serviría para enviar á saída estándar unha cadea de caracteres utilizando a chamada ao sistema `WRITE` (cf. exemplo da figura 1.8 da páxina 26).

swi-ex.s

```
1
2  /*
3   Exemplo de chamada ao sistema
4  */
5      .equ    STDOUT,1
6      .equ    WRITE,4
7
8      .align  2
9  ommsg:
10     .asciz  "Ola Mundo!\n"
11     .equ    omlon,.-ommsg
12
13     .text
14     .code   16
15     .align  2
16
17 ola:
18     push    {r4,r7,lr}
19
20     mov     r0, #STDOUT     @ saída estándar
21     ldr     r1, =ommsg      @ dirección da mensaxe
22     mov     r2, #omlon      @ tamaño en bytes da mensaxe
23     mov     r7, #WRITE      @ chamada ao sistema: WRITE
24     svc     0               @ Facemos a chamada
25
26     mov     r0, #0          @ devolvemos 0;
27     pop     {r4,r7,pc}
28
29 /* punto de entrada ARM completo */
```

```
30      .code 32
31      .align 4
32      .global main
33  main:
34      push {r4, lr}
35
36      blx ola
37
38      pop {r4, lr}
39      bx lr
40     .end
```

De todos os xeitos, o modelo que acabamos de describir é incompatible co convenio AAPCS (Arm Ltd., 2020), polo que no noso caso envólvense as chamadas ao sistema nun envoltorio de código que si cumpre devandito convenio (cf. epígrafes 4.1 e 4.3 dedicadas ás chamadas a subprograma). É dicir, as chamadas ao sistema desde os nosos programas en ARM/Thumb serán chamadas a subprogramas de acordo co indicado no capítulo 1 e nos exercicios do capítulo 4, que seguen o convenio AAPCS.

Este é precisamente o obxectivo principal da biblioteca estándar da linguaxe C. En Raspberry Pi OS, do mesmo xeito que noutros sistemas Linux, esta biblioteca adoita coñecerse como GNU Libc, pero podemos utilizar ademais outras bibliotecas con outros cometidos. En conxunto, estas bibliotecas ocultan a complexidade adicional de realizar chamadas ao sistema dándolles a aparencia dunha chamada a subprograma normal, e proporcionan servizos adicionais para manexar calquera elemento ou recurso do que puidese dispor o noso sistema hardware-sofware.

Unha función ou subprograma debe utilizar os rexistros do sistema tal como recóllese na táboa B.1. A primeira columna enumera os rexistros, en cor máis escura os dispoñibles no modo Thumb/T32. A seguinte columna indica se se debe preservar o contido orixinal do rexistro correspondente (N significa que non é necesario preservar o seu valor, e S significa que si é necesario preservalo, normalmente na pila). Así, as funcións non teñen por que preservar o contido dos rexistros `r0` – `r3`. A última columna indica o propósito de cada rexistro dentro dunha función.

Ademais, cando un subprograma ou función chama a outro, só os primeiros catro argumentos pásanse en rexistros. Se enumeramos os argumentos arg_1, arg_2, $\ldots arg_4$ dunha función de esquerda a dereita, pásanse en rexistros tal como indícase na táboa B.2.

Cadro B.1. Utilización dos rexistros nos subprogramas.

Rexistro	P	Cometido
`r0`	N	argumentos / resultados
`r1`	N	argumentos / resultados
`r2`	N	argumentos / resultados
`r3`	N	argumentos / resultados
`r4`	S	variables locais
`r5`	S	variables locais
`r6`	S	variables locais
`r7`	S	variables locais
`r8`	S	variables locais
`r9`	S	depende da plataforma
`r10`	S	variables locais
`r11/fp`	S	punteiro de marco
`r12/ip`	N	temp. intra-procedemento
`r13/sp`	S	punteiro de pila
`r14/lr`	N	rexistro de ligazón
`r15/pc`	N	contador de programa

Cadro B.2. Orde de paso de argumentos en rexistros.

Argumento	Rexistro
arg_1	`r0`
arg_2	`r1`
arg_3	`r2`
arg_4	`r3`

Se unha función necesita máis de 4 argumentos, estes pásanse na pila, deixando na cima arg_5, debaixo del (direccións de memoria seguintes) arg_6 e así sucesivamente (cf. Fig. B.1).

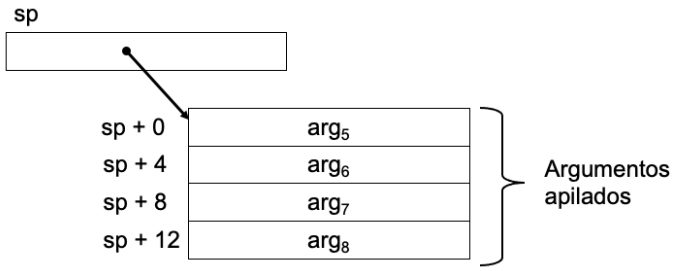


Figura B.1. Exemplo de paso de parámetros nunha chamada a un subprograma con 8 argumentos. Os catro primeiros argumentos $arg_1, \ldots arg_4$ pásanse nos rexistros, tal como se indica no cadro B.2.

A pila ten que estar aliñada cun límite de palabra de 32 bits, ou o que é o mesmo, o contido do punteiro de pila ten que ser sempre un múltiplo de 4. No caso de interfaces públicos, a pila ten que estar aliñada cun límite de dobre palabra, é dicir, o contido do `sp` ten que ser un múltiplo de 8.

Polo xeral, cada argumento que se pasa a unha función ou subprograma, a través da pila ou dos rexistros `r0` – `r3` ocupa unha palabra de 32 bits. Cando o tipo de datos requirido por unha función ou un subprograma ten un tamaño menor de 32 bits, alíñase á dereita da palabra de 32 bits. Por exemplo, un byte pasado nun rexistro estará situado no byte menos significativo dun rexistro, ou no byte menos significativo da palabra depositada na pila[1].

Cando o tipo de datos ten un tamaño maior de 32 bits, pásase en varios rexistros consecutivos ou en varias palabras de memoria consecutivas na pila. No caso de que se pase en rexistros consecutivos, o argumento completo pásase en rexistros, é dicir, non se pasa parte en rexistros e parte na pila. Por exemplo, unha dobre palabra (64 bits) pódese pasar en `r0` e `r1` ou en `r2` e `r3` ou na pila. Un vector de 128 bits pasaríase nos catro rexistros `r0` – `r3`, ou en catro palabras consecutivas na pila.

No caso de que o argumento ocupe máis de 4 bytes (e.g., tipo composto, unión, estrutura, matriz, etc.), ou cando non se pode determinar de antemán o tamaño tanto polo subprograma chamante como polo subprograma chamado, deposítase nunha área de memoria e no seu lugar pásase como argu-

[1] Tendo en conta o convenio de almacenamento en memoria ou *endiannes* por defecto, que é o extremista menor ou *litte endian*, o byte menos significativo dunha palabra en memoria depositada na pila será o que ten a dirección máis baixa dos catro bytes que forman a palabra.

mento un punteiro á devandita zona de memoria. Se o tamaño do argumento non é un múltiplo de 4 bytes, o seu tamaño redondéase por arriba ao seguinte múltiplo de 4 bytes.

No segmento de código seguinte ilústrase o paso de parámetros a unha hipotética chamada ao sistema que ten oito argumentos (cf. figura B.1), donde os sete primeiros ($arg_1, \ldots, arg_7$) son palabras de 32 bits e o último (arg_8) unha estrutura de 12 bytes. Vemos que pasamos no lugar correspondente (rexistros ou pila) os valores dos argumentos, excepto no caso de arg_8, que pasamos a dirección de memoria a partir da cal está almacenada a estrutura.

sc-8args.s

```
/*
      Argumentos de exemplo, almacenados
      nunha táboa nalgures na zona de datos
      Son 7 palabras e una estrutura de 12 bytes.
*/

        .data
arg1:   .word       0x12341111
arg2:   .word       0x23452222
arg3:   .word       0x34563333
arg4:   .word       0x45674444
arg5:   .word       0x56785555
arg6:   .word       0x67896666
arg7:   .word       0x789a7777
arg8:   .space      12

        .text

/* Antes iría o resto do programa. Só incluímos o segmento de
   código que prepara a chamada ó sistema
*/

      push {r4,r7}      @ salvagardamos os rexistros r7 e r4

      ldr  r4,=arg5     @ pasamos os parámetros que van na pila
      ldr  r0,[r4]      @ arg5
      ldr  r1,[r4, #4]  @ arg6
      ldr  r2,[r4, #8]  @ arg7
      ldr  r3,=arg8     @ arg8 (enderezo da zona de 12 bytes)

      push {r0-r3}      @ os poñemos na pila

      ldr  r0,=arg1     @ pasamos os parámetros que van en rexitros
      ldr  r1,[r0, #4]  @ arg2
      ldr  r2,[r0, #8]  @ arg3
      ldr  r4,[r0, #12] @ arg4
```

```
37        ldr   r0,[r0]       @ arg1
38
39        mov   r7,#SCCODE    @ chamada ao sistema
40        swi   #0            @ ó volver, o restultado estará en r0
41
42        add   sp, #16       @ balanceamos a pila
43        pop   {r4,r7}       @ recuperamos r4 e r7
44
45  /* O resto do programa seguiría a continuación */
```

En canto ao valor de retorno dunha función ou subprograma, en xeral ocupa 32 bits e deposítase en `r0`. No caso doutros tamaños:

- No caso de que o resultado sexa menor de 32 bits (e.g., un carácter), complétase con ceros ou co signo no caso dun número negativo e devólvese en `r0`.
- No caso dunha dobre palabra, devólvese en `r0` e `r1`.
- No caso dun vector de 128 bits, devólvese en `r0` – `r3`.
- No caso dun tipo composto de máis de 4 bytes, ou cando non se pode determinar de antemán o tamaño tanto polo subprograma chamante como polo subprograma chamado, deposítase nunha área de memoria cuxa dirección se pasa como un argumento extra ao invocar a rutina ou subprograma.

En canto aos tipos os argumentos e os resultados indicados nas signaturas das funcións da libraría estándar e doutras librarías do sistema recollidas no manual do sistema `man` e nas cabeceiras, establécese a correspondencia do cadro B.3 entre tipos da linguaxe C e tipos do procesador.

Cadro B.3. Correspondencia entre tipos C e tipos do procesador

Tipo C	Tip do Proc.
`char`	byte sen signo
`unsigned char`	byte sen signo
`signed char`	byte con signo
`bool`	byte sen signo
`short`	media palabra con signo
`unsigned short`	media palabra sen signo
`int`	palabra con signo
`unsigned int`	palabra sen signo
`size_t`	palabra sen signo
`long`	palabra con signo
`unsigned long`	palabra sen signo
`long long`	dobre palabra con signo
`unsigned long long`	dobre palabra sen signo

Apéndice C
Modelo do programador

Thumb, coñecido como T32 a partir da versión 8 da arquitectura ARM, é un estado dos microprocesadores da devandita arquitectura que proporciona unha versión compacta do repertorio de instrucións, onde se combinan instrucións de 16 bits con instrucións de 32 bits. T32/Thumb dispón de versións de 16 bits das instrucións que se usan con máis frecuencia, e no caso de aplicacións que non necesiten toda a potencia expresiva orixinal de ARM, os programas escritos co repertorio Thumb ocuparán menos espazo na memoria que os programas equivalentes escritos na versión completa do repertorio de ARM (repertorio A32 de 32 bits).

Neste apéndice presentamos unha breve introdución ao modelo do programador Thumb de 16 bits, a modo de referencia e ferramenta de consulta rápida á hora de afrontar os exercicios e problemas propostos no libro. En primeiro lugar describimos de maneira sucinta a organización da memoria e os rexistros cando o microprocesador funciona en estado T32/Thumb. A continuación, enumeramos os modos de direccionamento dispoñibles, así como as instrucións utilizadas no libro. Finalmente, identificamos as directivas do ensamblador de GNU utilizadas nos programas incluídos neste manual.

C.1 Rexistros do programador

A arquitectura ARM ten 16 rexistros de propósito xeral de 32 bits, dos cales adóitase asignar funcións especiais ao rexistro `r13`, que funciona como punteiro de pila (`sp`), e ao rexistro `r14`, que funciona como rexistro de ligazón

para almacenar a dirección de retorno tras unha chamada a subprograma (`lr`). Ademais, `r15` fai as funcións de contador de programa (`pc`)[1].

No caso do estado T32/Thumb, só disponse dos oito primeiros rexistros como rexistros de propósito xeral (cf. figura C.1) e, como veremos máis adiante, determinadas instrucións como **`push`** no caso de `r13` ou **`bl`** no caso de `r14` utilizan implicitamente estes dous rexistros como punteiro de pila e rexistro de ligazón respectivamente.

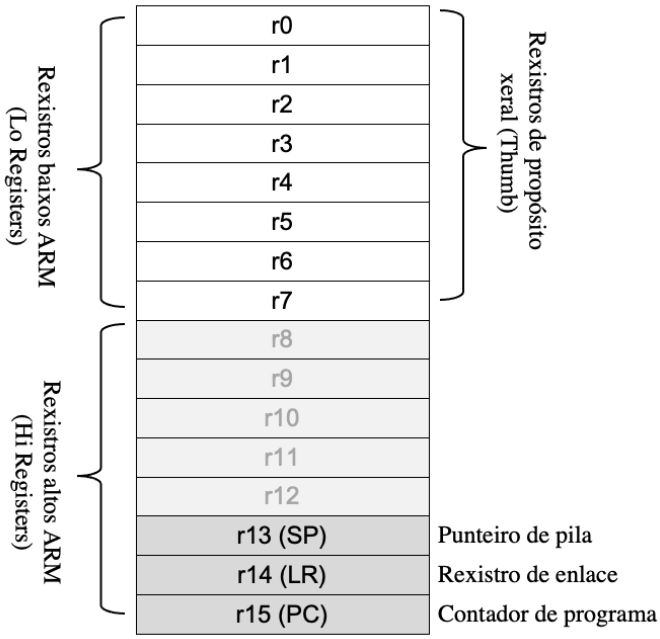


Figura C.1. Rexistros do programador T32/Thumb: oito rexistros de propósito xeral (`r0` – `r7`), máis o punteiro de pila, o rexistro de ligazón e o contador de programa (`r13` – `r15`).

A limitación no uso dos rexistros de propósito xeral aos rexistros `r0` a `r7`, xunto coa interpretación implícita da función dos rexistros `r13`, `r14` e `r15`, que por tanto non necesitan codificación adicional máis aló do propio código de operación, permite codificar unha ampla variedade de instrucións con tan só 16 bits por instrución.

[1] Ademais do uso especial dos rexistros `r13 - r15`, os rexistros `r9 - r12` utilízanse cun cometido específico dentro do convenio AAPCS de chamadas a subprograma e paso de parámetros (cf. apéndice B). En calquera caso, estes rexistros non forman parte do modelo do programador Thumb/T32 e por tanto quedan fóra do ámbito deste manual.

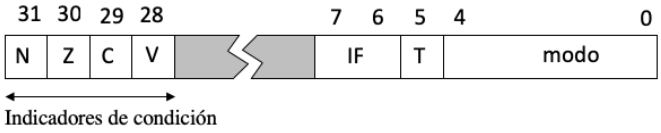


Figura C.2. Rexistro de estado (CPSR - Current Processor Status Register). O catro bits máis significativos son os indicadores de condición.

En canto ao rexistro de estado (cf. figura C.2), os catro bits de maior peso almacenan os indicadores de condición (*condition flags*), mentres que os oito bits de menor peso conteñen información do sistema relacionada co estado do procesador e os mecanismos de tratamento das interrupcións. Os indicadores de condición utilízanse, entre outros labores, para tomar decisións sobre o fluxo de programa mediante as instrucións de salto condicional. Os códigos de condición das instrucións de salto teñen o significado indicado no cadro C.1 .

Cadro C.1. Significado dos códigos de condición (Cond.) en función dos indicadores `N`, `Z`, `C` e `V`.

Cond.	Ind.	Significado
mi	N	Minus - Negativo
eq	Z	Equal - Igual / Cero
cs	C	Carry set - Carraxe
vs	V	oVerflow set - Desbordamento
hi	$C\overline{Z}$	Higher - Maior sen signo
gt	$\overline{Z}(N = V)$	Greater - Maior que
ge	$N = V$	Greater or equal - Maior ou igual
pl	$\overline{N}$	Plus - Positivo
ne	$\overline{Z}$	Not equal - Distinto (de cero)
cc	$\overline{C}$	Carry clear - Non carraxe
vc	$\overline{V}$	oVerflow clear - Non desbordamiento
ls	$\overline{C} + Z$	Lower or same - Menor ou igual sen signo
le	$Z + (N \neq V)$	Less or equal - Menor ou igual que
lt	$N \neq V$	Less than - Menor que

C.2 Organización da memoria

A memoria organízase en palabras de 4 bytes e é direccionable a nivel de byte. Isto implica que hai unha dirección de memoria distinta para cada byte que forma parte da memoria do computador. As instrucións que acceden a

palabras en memoria, necesitan que estas estean aliñadas en direccións de memoria múltiplo de 4.

O convenio para a orde de almacenamento das palabras en memoria por defecto é o extremista menor (*little-endian* en inglés), é dicir, o byte de menor peso da palabra almacénase na dirección de memoria máis baixa.

Ademais de traballar con bytes e palabras de 4 bytes, algunhas instrucións traballan con medias palabras (16 bits, 2 bytes). No caso das instrucións que utilizan medias palabras almacenadas en memoria, as súas direccións deben ser múltiplo de 2.

As direccións de memoria son de 32 bits, polo que o rango de direccionamento directo é de 0 a $2^{32} - 1 =$ 4.294.967.295 bytes, é dicir, o tamaño máximo da memoria direccionable é de 4 GB.

C.3 Modos de direccionamento

Unha instrución en ensamblador codifica que operación débese realizar, con que operandos fonte hai que realizar dita operación e onde se debe gardar o resultado. No ámbito da arquitectura de computadores, coñécese como modos de direccionamento ás distintas formas nas que pode indicarse a dirección efectiva dos operandos e do resultado das instrucións dun procesador.

O cadro C.2 recolle os modos de direccionamento soportados no estado T32/Thumb.

C.3.1 Direccionamento directo a rexistro

O direccionamento directo a rexistro utilízase na maior parte das instrucións, tanto de transferencia, como de transformación de datos, para algúns ou todos os seus operandos. En Thumb utilízase este modo, por exemplo, para especificar os dous operandos fonte e o destino das instrucións de suma e resta. Así, a instrución `add r0, r2, r3` sumaría os contidos dos rexistros `r2` e `r3` deixando o resultado no rexistro `r0` e a instrución `sub r0, r1, r2` restaría o contido do rexistro `r2` do rexistro `r1` deixando o resultado no rexistro `r0`.

Cadro C.2. Modos de direccionamento no estado T32/Thumb.

Modo	Dirección efectiva
Directo a rexistro	DE = Rexistro (dato en rexistro)
Inmediato	DE = Instrución (dato na instrución)
Indirecto con desp.	DE = R + desprazamento (5 bits)
Relativo a PC	DE = PC + desprazamento (8 bits)
Relativo a SP	DE = SP + desprazamento (8 bits)
Indirecto con RD	DE = R + RD

Modo	Exemplo
Directo a rexistro	`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]`

C.3.2 Direccionamento inmediato

No caso do direccionamento inmediato, un dos operandos codifícase na propia instrución. O rango de valores de devandito operando pode variar dependendo de cada instrución concreta, utilizándose para a súa codificación entre 3 e 8 bits (cf. cadro C.3). Como exemplo de instrucións que usen este modo de direccionamento están as instrucións aritméticas, como por exemplo as instrucións:

- **`add r2,r1,#Inm3`** (R2 ← R1 + Inm3).
- **`sub r2,#Inm8`** (R2 ← R2 - Inm8).
- **`sub SP,#Inm9`** (SP ← SP - Inm9).
- **`add r5,SP,#Imm10`** (R5 ← SP + Inm10).

Debido a que as palabras de 32 bits teñen que estar aliñadas a direccións múltiplo de catro e as medias palabras teñen que estar aliñadas a direccións pares, non é necesario codificar todos os bits do dato inmediato se se coñece de antemán que devandito dato é unha palabra ou unha media palabra. Por

exemplo, a instrución **`add SP, #Inm9`**, suma a constante `#Inm9` ao punteiro de pila (cf. figura C.3). Como o punteiro de pila almacena unha dirección, o resultado da operación ten que ser tamén unha dirección. Por iso, para codificar os nove bits de `#Inm9` só necesitamos almacenar os sete máis significativos, xa que os dous bits menos significativos teñen que ser necesariamente 0.

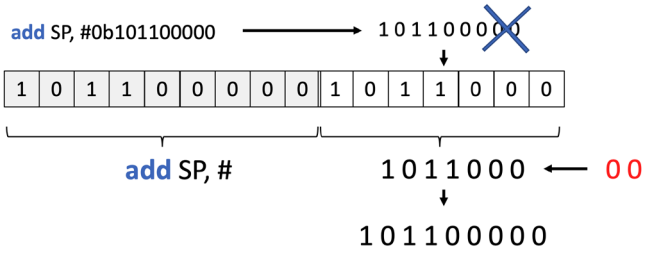


Figura C.3. Codificación da instrución **`add SP,#0x160`**. A instrución codifica só os sete bits máis significativos do dato inmediato (os dous bits menos significativos toman sempre o valor 0).

O mesmo ocorre con `#Inm10` en **`add Rd,SP,#Imm10`**. Para codificar o dato inmediato necesitamos almacenar unicamente oito bits, sendo os dous menos significativos sempre cero.

En definitiva, a codificación e o rango dos datos inmediatos difiren naquelas instrucións que utilizan o dato inmediato para calcular a dirección dunha palabra ou dunha media palabra, é dicir, para sumar ou restar unha constante ao punteiro de pila (cf. cadro C.4), ou ben para calcular a dirección efectiva nunha instrución de carga ou almacenamento de palabras ou medias palabras (cf. cadro C.6) con direccionamento indirecto con desprazamento.

Estas situacións nas que o rango de valores do dato inmediato (p. ex., de 0 a 510) non coincide co número de elementos codificados represéntanse en negriña na táboa C.3.

C.3.3 Direccionamento indirecto a rexistro con desprazamento

Este modo utilízase nas instrucións **`ldr`**/**`ldrh`**/**`ldrb`** e **`str`**/**`strh`**/**`strb`** de carga e almacenamento, para o operando fonte e o destino respectivamente. A dirección efectiva calcúlase sumando unha constante ao contido dun rexistro de dirección, cuxo rango dependerá do tamaño do operando ou do destino. En todos os casos, utilízanse 5 bits para codificar un desprazamento.

Cadro C.3. Posibles valores dos datos inmediatos. Aparecen en negriña aqueles casos nos que o rango de valores codificados é maior que o representable co número de bits reservados na instrución. Neses casos, un ou dous dos bits que codifican o dato inmediato toman o valor 0 de maneira implícita, en función de se se codifican palabras ou medias palabras. Por exemplo Inm5 codifícase con 5 bits e **Imn9** codifícase con 7 bits (os dous bits menos significativos son cero). x2 e x4 significan respectivamente que o dato inmediato é múltiplo de 2 ou múltiplo de 4.

Rango dos 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		

- No caso de **`ldr`** e **`str`**, cuxos operandos están aliñados a palabra, a instrución define un desprazamento de 7 bits, onde os 5 bits máis significativos do desprazamento están codificados na devandita instrución e os 2 bits menos significativos toman valor 0. Desta maneira, o rango do desprazamento é de (0 – 124) bytes ou (0 – 31) palabras.
- No caso de **`ldrh`** e **`strh`**, cuxos operandos están aliñados con medias palabras, a instrución define un desprazamento de 6 bits, onde os 5 bits máis significativos están codificados na instrución e o bit menos significativo toma valor 0. Así, o rango do desprazamento é de (0 – 62) bytes ou (0 – 31) medias palabras.
- No caso de **`ldrb`** e **`strb`**, con operandos aliñados a byte, defínese un desprazamento de 5 bits codificados na instrución, cun rango de desprazamento de (0 – 31) bytes.

Por exemplo, a instrución de carga **`ldr r0, [r7, #0b1010100]`** realiza a operación R0 ← [R7 + 0b1010100], pero na instrución codifícanse soamente os 5 bits máis significativos do desprazamento (10101).

Este modo coñécese como direccionamento indirecto con desprazamento porque, se no direccionamento directo a rexistro o dato estaba no rexistro codificado na instrución, neste modo o dato está na dirección indicada polo devandito rexistro, e por tanto trátase dun direccionamento indirecto.

A instrución **`bx`** de salto e intercambio (cf. apartado 1.2 do capítulo 1 e apéndice A) supón un caso particular deste modo de direccionamento. Neste caso, a dirección efectiva defínese mediante os 31 bits máis significativos do rexistro codificado na instrución (o bit menos significativo da dirección efectiva ten que ser 0, xa que as instrucións teñen que comezar sempre en direc-

cións pares). Trataríase por tanto dun modo de direccionamento indirecto a rexistro (sen desprazamento). Por exemplo, o seguinte código:

```
ldr r0,=0x28000000
bx  r0
```

provocaría un salto á dirección `0x28000000` (e ademais o procesador pasaría ao estado ARM A32 completo).

C.3.4 Direccionamento relativo ao contador de programa

O direccionamento relativo ao contador de programa é unha variante do direccionamento indirecto con desprazamento no que o rexistro de dirección é o contador de programa (`pc`, `r15`). Este modo especifica a dirección efectiva do operando ou dun salto como a suma do contido do contador de programa e un desprazamento codificado na propia instrución.

No caso das instrucións de carga e almacenamento, este modo de direccionamento só está dispoñible con operandos de 32 bits (instrucións **`ldr`** e **`str`** unicamente), e o desprazamento ten un rango de 10 bits dos cales codifícanse na instrución 8 (os 2 bits menos significativos teñen valor 0). Desta maneira, o rango do desprazamento é de (0 – 1020) bytes ou (0 – 255) palabras. No caso das pseudoinstrucións **`ldr r0,=etiqueta`** ou análogas, onde `etiqueta` é unha etiqueta que referencia unha dirección de memoria nun programa, o ensamblador substitúe dita pseudoinstrución por unha instrución equivalente con direccionamento relativo a PC, como vimos no capítulo 1 (cf. figura 1.3).

O modo de direccionamento relativo a contador de programa é especialmente útil para acceder a posicións de memoria que se atopan nas inmediacións da instrución que se está executando. En Thumb de 16 bits, para que as instrucións de salto póidanse codificar en unicamente dous bytes e o código sexa directamente reubicable na súa maior parte, en lugar de saltos absolutos recórrese a utilizar saltos relativos ao PC. No caso da instrución de salto incondicional **`b`**, o desprazamento é un número de 12 bits en complemento a 2 que proporciona un rango de desprazamento de $\pm$ 2048 bytes. Deses 12 bits, codifícanse os 11 bits máis significativos, xa que o bit menos significativo ten que ser 0 ao estar as instrucións aliñadas a direccións pares. As instrucións de salto condicional teñen un rango de desprazamento de 9 bits ($\pm$ 256 bytes), dos cales codifícanse 8 na instrución.

No caso da instrución de chamada a subprograma **`bl`** o desprazamento codifícase dunha maneira especial. Neste caso, o desprazamento é un número de 23 bits en complemento a dous, dos cales codifícanse 22 bits, proporcionando un rango de $\pm$ 4 Mbytes. Obviamente, é imposible codificar os 22 bits do desprazamento utilizando unha soa instrución de 16 bits, polo que ao ensamblarse dita instrución xéranse dúas instrucións **`bl`** consecutivas, onde a primeira delas codifica os 11 bits máis significativos do desprazamento e a segunda os 11 bits menos significativos. Noutras palabras, a instrución sempre se codifica como un par de instrucións, e así é posible codificar un *salto afastado* cun desprazamento de 23 bits. Por exemplo, a instrución **`bl`** `eti`, onde `eti` apunta a unha dirección situada `0x0670fe` posicións máis adiante, codifícase como dúas instrucións **`bl`**, onde a primeira codifica a parte alta do offset como `0x06e`, e a segunda a parte baixa do offset como `0x07f` (cf. figura C.4).

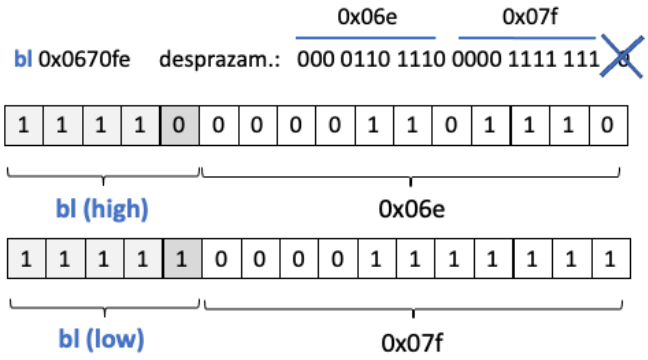


Figura C.4. Codificación da instrución **`bl`** `0x0670fe`. A instrución codifícase como dúas instrucións **`bl`** consecutivas, onde cada unha delas codifica a metade dos bits do desprazamento.

Como indicamos antes para as instrucións de carga e almacenamento, ao programar en ensamblador non é necesario calcular manualmente o desprazamento a sumar ao contador de programa para obter a dirección efectiva dun salto. Para iso utilizaremos nas instrucións de salto unha etiqueta que identifica a localización da dirección efectiva (e.g., **`bcc`** `eti`). O ensamblador calculará o desprazamento correcto tendo en conta a localización da instrución e a dirección de destino.

C.3.5 Direccionamento relativo ao punteiro de pila

O direccionamento relativo ao punteiro de pila é similar ao relativo ao contador de programa, pero utilizando como rexistro de direccionamento o punteiro de pila (`sp`, `r14`). A dirección efectiva é a suma do contido do punteiro de pila e un desprazamento de 10 bits codificado na propia instrución con 8 bits, cun rango do desprazamento de (0 – 1020) bytes ou (0 – 255) palabras.

O direccionamento relativo ao punteiro de pila utilízase unicamente nas instrucións **`ldr`** e **`str`**, e de maneira implícita nas instrucións **`push`** e **`pop`** de acceso á pila.

C.3.6 Direccionamento indirecto a rexistro con rexistro de desprazamento

O modo de direccionamento indirecto con rexistro de desprazamento é moi similar ao indirecto con desprazamento. A diferenza radica en que o desprazamento se obtén dun rexistro en lugar de ser un dato inmediato. Un exemplo sería a instrución **`str`** `r0, [r1, r3]`, que cargaría no rexistro `r0` a palabra cuxa dirección de memoria é a suma dos contidos de `r1` e `r3` (R0 ← [R1 + R3]).

Este modo está dispoñible para as instrucións de carga e almacenamento de bytes, palabras e medias palabras, e ademais para as instrucións de carga de bytes e palabras con extensión de signo **`ldsb`** e **`ldsh`** (cf. táboa C.6).

C.4 Repertorio de instrucións

As táboas seguintes recollen as instrucións utilizadas no libro. Trátase basicamente do repertorio T32/Thumb do núcleo ARM7TDMI, a primeira versión do repertorio, que incluía unicamente as instrucións de 16 bits. Para unha descrición detallada destas instrucións consúltese o capítulo 5 do manual de referencia do devandito núcleo[2] (Amtel Corporation, 1999), un documento facilmente localizable en Internet.

[2] O núcleo ARM7TDMI (ARM7 + **T**humb de 16 bits + **D**epuración JTAG + **M**ultiplicador rápido + emulador **I**n-circuit mellorado) é un dos núcleos ARM máis usados até o ano 2009. Podémolo atopar en miles de sistemas embebidos. Entre os usos notables do núcleo ARM7 temos o móbil Nokia 6110, as consolas Dreamcast, Game Boy Advance e Nintendo DS, o reprodutor Zune HD, o iPod, e a aspiradora Roomba.

Existen tres instrucións na referencia anterior que non utilizamos neste manual introductorio. Trátase de **ldmia**, **stmia** e **swi**. As instrucións **ldmia** e **stmia** teñen un comportamento similar a **pop** e **push** respectivamente, só que o rexistro utilizado como punteiro pode ser calquera rexistro de datos `r0` – `r7`. Por exemplo, a instrución **ldmia** `r1!,{r0,r3-r5}` le da memoria a palabra indicada polo rexistro `r1` e os tres seguintes, almacenando o seu contido nos rexistros `r0`, `r3`, `r4` e `r5`, e incrementa `r1` en 16 unidades (4 palabras), e **stmia** `r14!,{r0,r3-r5}` é equivalente a **push** `{r0,r3-r5}`. Utilizar estas instrucións engadiría unha complexidade innecesaria aos exercicios dun texto introductorio como este, que por outra banda xa ilustra o uso das instrucións **push** e **pop**.

A instrución **swi** dispara unha interrupción software. Ao tratarse dun texto introductorio sobre programación en ensamblador, non tratamos en detalle as interrupcións, polo que a instrución **swi** non tiña cabida nos nosos programas.

Como vimos antes, a instrución **bx** da táboa C.8 utilízase unicamente no apartado 1.2 do capítulo 1, no capítulo 5 e no apéndice A para devolver o control ao sistema operativo desde os nosos programas en Thumb para a Raspberry Pi.

Ademais das instrucións Thumb do núcleo ARM7TDMI recollidas nas táboas seguintes, nos exercicios propostos neste libro utilizamos a instrución do repertorio Thumb-2 **wfi** para marcar a terminación do noso código ao utilizar o simulador QtARMSim. Nunha contorna multitarea real, a instrución **wfi** (wait for interrupt) pon ao procesador en modo de espera ata que se produce unha interrupción, e nalgúns microprocesadores para aplicacións embebidas ademais pásase ao modo de baixo consumo. No momento de producirse a interrupción, a rutina de servizo correspondente tomará o control do procesador para ceder o devandito control a unha nova tarefa.

Cadro C.4. Instrucións aritméticas. Obsérvese que o rexistro punteiro de pila **sp** ten un tratamento especial, con instrucións específicas de suma e resta.

Mover			
Inmediato 8 bits	`mov Rd, #Inm8`	Rd ← Inm8	NZ · ·
Rexistros	`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
Rexistros	`add Rd, Ro1, Ro2`	Rd ← Ro1 + Ro2	NZCV
Ao SP (`r13`)	`add SP, #Inm9`	SP ← SP + Inm9	· · · ·
SP a Reg.	`add Rd, SP, #Inm10`	Rd ← SP + Inm10	· · · ·
Con carraxe	`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
Rexistros	`sub Rd, Ro1, Ro2`	Rd ← Ro1 − Ro2	NZCV
Ao SP (`r13`)	`sub SP, #Inm9`	SP ← SP − Inm9	· · · ·
Con carraxe	`sbc Rd, Rd, Ro`	Rd ← Rd − Ro − $\overline{C}$	NZCV
Negar	`neg Rd, Ro`	Rd ← − Ro	NZCV
Multiplicar			
Rexistros	`mul Rd, Ro, Rd`	Rd ← Ro ∗ Rd	NZCV
Comparar			
Rexistros	`cmp Rn, Rm`	Rn − Rm	NZCV
Rexistros negado	`cmn Rn, Rm`	Rn + Rm	NZCV
Inmediato 8 bits	`cmp Rn, #Inm8`	Rn − Inm8	NZCV

Cadro C.5. Instrucións lóxicas e de desprazamento.

Lóxicas			
AND	`and Rd, Rd, Ro`	Rd ← Rd AND Ro	NZ · ·
Borrar bits	`bic Rd, Rd, Ro`	Rd ← Rd AND NOT(Ro)	NZ · ·
OR	`orr Rd, Rd, Ro`	Rd ← Rd OR Ro	NZ · ·
XOR	`eor Rd, Rd, Ro`	Rd ← Rd XOR Ro	NZ · ·
NOT	`mvn Rd, Ro`	Rd ← NOT(Ro)	NZ · ·
Test	`tst Rn, Rm`	Rn AND Rm	NZ · ·
Desprazamento			
Lóxico	`lsl Rd, Ro, #Inm5`	Rd ← Ro << Inm5	NZC ·
á esquerda	`lsl Rd, Rd, Rs`	Rd ← Rd << $[Rs]_{7:0}$	NZC ·
Lóxico	`lsr Rd, Ro, #Inm5`	Rd ← Ro $>>_l$ Inm5	NZC ·
á dereita	`lsr Rd, Rd, Rs`	Rd ← Rd $>>_l$ $[Rs]_{7:0}$	NZC ·
Aritmético	`asr Rd, Ro, #Inm5`	Rd ← Ro $>>_a$ Inm5	NZC ·
á dereita	`asr Rd, Rd, Rs`	Rd ← Rd $>>_a$ $[Rs]_{7:0}$	NZC ·
Rotación á dereita	`ror Rd, Rd, Rs`	Rd ← Rd ROR $[Rs]_{7:0}$	NZC ·

Cadro C.6. Instrucións de carga e almacenamento.

Cargar		
Ind. Desp., palabra	`ldr Rd, [Ri, #Inm7]`	Rd ← [Rn + Inm7]
– media palabra	`ldrh Rd, [Ri, #Inm6]`	Rd ← Ext0($[Rn + Inm6]_{15:0}$)
– byte	`ldrb Rd, [Ri, #Inm5]`	Rd ← Ext0($[Rn + Inm5]_{7:0}$)
Ind. Reg., palabra	`ldr Rd, [Ri, Rm]`	Rd ← [Rn + Rm]
– media pal.	`ldrh Rd, [Ri, Rm]`	Rd ← Ext0($[Rn + Rm]_{15:0}$)
– media pal., signo	`ldsh Rd, [Ri, Rm]`	Rd ← ExtS($[Rn + Rm]_{15:0}$)
– byte	`ldrb Rd, [Ri, Rm]`	Rd ← Ext0($[Rn + Rm]_{7:0}$)
– byte, signo	`ldsb Rd, [Ri, Rm]`	Rd ← ExtS($[Rn + Rm]_{7:0}$)
Relativo a PC	`ldr Rd, [PC, #Inm10]`	Rd ← [PC + Inm10]
– a SP	`ldr Rd, [SP, #Inm10]`	Rd ← [SP + Inm10]
Almacenar		
Ind. Desp., palabra	`str Ro, [Ri, #Inm7]`	[Rn + Inm7] ← Ro
– media palabra	`strh Ro, [Ri, #Inm6]`	$[Rn + Inm6]_{15:0}$ ← $Ro_{15:0}$
– byte	`strb Ro, [Ri, #Inm5]`	$[Rn + Inm5]_{7:0}$ ← $Ro_{7:0}$
Ind. Reg., palabra	`str Rd, [Ri, Rm]`	[Rn + Rm] ← Ro
– media pal.	`strh Ro, [Ri, Rm`	$[Rn + Rm]_{15:0}$ ← $Ro_{15:0}$
– byte	`strb Ro, [Ri, Rm]`	$[Rn + Rm]_{7:0}$ ← $Ro_{7:0}$
Relativo a SP	`str Ro, [SP, #Inm10]`	[SP + Inm10] ← Ro

Ext0: recheo con ceros á esquerda; ExtS: recheo co signo á esquerda.

Cadro C.7. Instrucións de manexo da pila.

Xestión da pila		
Apilar	`push {Regs}`	SP ← SP − Tam(Regs) [SP] ← Regs
Enlazar e apilar	`push {Regs, lr}`	SP ← SP − Tam(Regs, LR) [SP] ← Regs,LR
Desapilar	`pop {Regs}`	Regs ← [SP] SP ← SP + Tam(Regs)
Desapilar e retorno	`pop {Regs, pc}`	Regs, PC ← [SP] SP ← SP + Tam(Regs, PC)

Cadro C.8. Instrucións de salto. Os códigos de condición `cond` da instrución de salto condicional son os recollidos no cadro C.1.

Saltos			Rango
Incondicional	`b eti`	PC ← eti	± 2048 B
Condicional	`b{xx} eti`	Se {xx}, PC ← eti	± 256 B
A subprograma	`bl eti`	LR ← dir. sig. instrución PC ← eti	± 4 MB
E intercambiar	`bx Ra`	PC ← (Ra AND `0xFFFFFFFE`)^i	4 GB

i: Se Ra_0 = 0, cambio de estado T32/Thumb a estado ARM. Ra contén unha dirección absoluta.

C.5 Selección de directivas do ensamblador

Neste traballo seguimos a notación e os convenios do ensamblador de GNU. Todas as directivas teñen nomes que comezan cun punto (?.?) e o resto da directiva está composto por letras minúsculas.

O cadro C.9 recolle as directivas utilizadas neste manual.

Cadro C.9. Directivas do ensamblador Thumb.

Directivas do ensamblador	
`.align N`	Seguinte dato empeza en dir. múltiplo de 2^N.
`.arm`	O que segue é código ARM/A32.
`.ascii “cadeaa”`	Inicializa unha zona de memoria cos caracteres UTF-8 de `cadea`.
`.asciz “cadea”`	Inicializa unha zona de memoria cos caracteres UTF-8 de `cadea`, termina con 0.
`.balign N`	Seguinte dato empeza en dir. múltiplo de N.
`.byte valor`	Inicializa un byte a `valor`.
`.code16`	O que segue é código Thumb/T32.
`.code32`	Equivalente a `.arm`.
`.data`	Ensambla o que segue na zona de datos.
`.end`	Non hai máis instrucións.
`.equ símbolo, expr`	Asigna o valor de `expr` a `símbolo`.
`.equiv símbolo, expr`	Como `.equ`, pero dá erro se existe `símbolo`.
`.eqv`	Equivalente a `.equiv`.
`.hword valor`	Inicializa unha media palabra a `valor`.
`.global etiqueta`	Indica que `etiqueta` é una variable global.
`.quad valor`	Inicializa unha dobre 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.
`.text`	Ensambla o que segue na zona de código.
`.type etiqueta,%function`	`etiqueta` é unha función.
`.unreq símbolo`	Cancela o alias `símbolo`.
`.word valor`	Inicializa unha palabra a `valor`.

Apéndice D
Táboas UTF-8

UTF-8 (8-bit Unicode Transformation Format) é un formato de codificación de caracteres que utiliza secuencias de lonxitude variable. Correspóndese co estándar RFC 3629 da Internet Engineering Task Force (IETF) e entre as súas vantaxes está a de incluír directamente os caracteres ASCII tradicionais de 7 bits, polo que calquera mensaxe ASCII represéntase sen cambios.

UTF-8 tamén serve para codificar con 2, 3 ou 4 bits (cf. cadro D.1) todos os códigos do estándar ISO/IEC 10646) que define un conxunto de caracteres universal (Unicode) coa a práctica totalidade dos símbolos de uso común, os caracteres de calquera alfabeto (latino, cirílico, chinés, xaponés, coreano, etc.), os emoxis ou os símbolos matemáticos. No caso concreto do galego e doutras linguas romances, os caracteres acentuados (á, é, ó, etc.) e os que utilizan outros símbolos diacríticos (ç, ñ, ü, etc.) se codifican con dous bytes.

Cadro D.1. Codificacións UTF-8 dos puntos de código ISO/IEC 10646 (Unicode).

Bits	1o PC	Ult. PC	Bytes	Byte 1	Byte 2	Byte 3	Byte 4
7	U+0000	U+007F	1	0xxxxxxx			
11	U+0080	U+07FF	2	110xxxxx	10xxxxxx		
16	U+0800	U+FFFF	3	1110xxxx	10xxxxxx	10xxxxxx	
21	U+10000	U+10FFFF	4	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx

A continuación, presentamos como referencia as táboas de caracteres UTF-8 correspondentes aos puntos Unicode `U+0000` – `U+00FF` (dúas primeiras filas da táboa D.1). A codificación destes 256 primeiros caracteres Unicode en UTF-8 leva a cabo de acordo coas seguintes regras:

Rango Unicode	UTF-8	Comentarios
`U+0000-U+007F`	`xxxxxxx`	Caracteres ASCII de 7 bits.
`U+0080-U+00FF`	`110yyyyy 10xxxxxx`	Caracteres acentuados e outros símbolos de uso común.

Os caracteres do primeiro grupo (cf. cadros D.2 e D.3) correspóndense cos caracteres ASCII orixinais de 7 bits. Trátase pois de símbolos dun único byte cuxo bit máis significativo é 0. Agrúpanse á súa vez da seguinte maneira:

Codificación	Grupo
`0b000X XXXX`	Caracteres de control.
`0b001X XXXX`	Díxitos e signos de puntuación.
`0b010X XXXX`	Maiúsculas e caracteres especiais.
`0b011X XXXX`	Minúsculas e caracteres especiais.

Os caracteres do segundo grupo (cf. cadros D.4 e D.5) son caracteres que adoitaban definirse como caracteres ASCII estendidos, codificados nun byte co bit máis significativo a 1 (i.e., valores de `0x80` a `0xFF`) e que, como vimos, correspóndense cos puntos Unicode entre `U+0080` e `U+00FF`. A codificación UTF-8 en 2 bytes realízase da seguinte maneira:

`0b0000 0000 XXXX YYYY` → `0b110 00 00XX 10XX YYYY`

Por exemplo, a letra "ñ" cuxo punto Unicode é `U+00F1` codificaríase como (cf. cadro D.5 na páxina 263):

`0b0000 0000 1111 0001` → `0b110 00 0011 1011 0001`

É dicir, como `0xC3B1`. A razón desta transformación é para garantir outra das propiedades do sistema de codificación UTF-8 chamada *non superposición*, que consiste en que os conxuntos de valores que pode tomar cada byte dun carácter multibyte son disxuntos, polo que non é posible confundilos entre si. Noutras palabras, non é posible por exemplo confundir ningún símbolo UTF-8 dun byte co primeiro ou o segundo byte dun símbolo de dous bytes. Ademais, nunha transmisión de símbolos UTF-8 é posible determinar o inicio de cada símbolo sen reiniciar a lectura desde o principio da transmisión.

Nas táboas D.4 e D.5 substitúese o valor en decimal polo valor do punto Unicode en hexadecimal, xa que neste caso este dato é moito máis informativo que devandito valor en decimal.

Finalmente, os cadros D.6 e D.7 recollen os caracteres alfabéticos utilizados en galego, portugués e castelán.

Cadro D.2. Caracteres UTF-8 de `0x00` a `0x3F`. Caracteres de control, díxitos e signos de puntuación. A codificación CP437 utilizada polo PC de IBM orixinal, utilizaba os caracteres de control de (soh) a (us) para definir os caracteres imprimibles da táboa. O carácter correspondente a (nul) non formaba parte do CP437 orixinal.

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	☼	(se)	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	?

Cadro D.3. Caracteres UTF-8 de `0x40` a `0x7F`. Maiúsculas, minúsculas e caracteres especiais.

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	^	126	0x7E	~
95	0x5F	_	127	0x7F	⌂

Cadro D.4. Caracteres UTF-8 de `0xC280` a `0xC2BF`. Símbolos varios. Correspóndense co segundo bloque Unicode, tamén coñecido como Controis C1 e Suplemento Latin 1. Os primeiros 32 caracteres son caracteres de control non imprimibles. A columna Dif representa as diferenzas entre ISO Latin 1 (columna Sim) e ISO Latin 9.

Uni	UTF-8	Ctl	Uni	UTF-8	Sim	Dif
0x80	0xC280	(pad)	0xA0	0xC2A0		
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	¿	

Cadro D.5. Caracteres UTF-8 de `0xC380` a `0xC3BF`. Caracteres acentuados. Correspóndense co segundo bloque Unicode, tamén coñecido como Suplemento Latin 1. Podemos observar que as maiúsculas e minúsculas diferéncianse nun só bit, o mesmo que no caso dos 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	Đ	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

Cadro D.6. Caracteres UTF-8 alfabéticos utilizados en galego, portugués e castelán (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

Cadro D.7. Caracteres UTF-8 alfabéticos utilizados en galego, portugués e castelán (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

Bibliografía

Amtel Corporation (1999). *ARM7TDMI (Thumb) Datasheet*. Volumen 0673B. Amtel Corporation. Capítulo 5.

Arm Ltd. (2020). *Procedure Call Standard for the Arm Architecture*. IHI 0042J. Arm Ltd.

Fernández, Gregorio Fernández (2004). *Curso de Ordenadores. Conceptos básicos de arquitectura y sistemas operativos*. 5.ª edición. Fundación Rogelio Segovia para el Desarrollo de las Telecomunicaciones. ISBN: 978-84-7402312-1.

Fernández-Iglesias, Manuel José y col. (2020). *Arquitectura de Ordenadores. Ejercicios prácticos de ARM/Thumb*. Universidade de Vigo. ISBN: 978-84-8158-855-2.

Hennessy, John L y David A Patterson (1995). *Organización y diseño de computadores : la interfaz hardware/software*. McGraw-Hill. ISBN: 978-84-481-1829-7.

— (2016). *Computer Organization and Design ARM Edition: The Hardware Software Interface*. Morgan Kaufmann. ISBN: 978-012801733-3.

Knaggs, Peter J. (2016). *ARM Assembly Language Programming*. Peter J. Knaggs.

Mir, Sergio Barrachina y col. (2014). *Prácticas de Introducción a la arquitectura de computadores con QtARMSim y Arduino*. Universitat Jaume I.

— (2018). *Introducción a la arquitectura de computadores con QtARMSim y Arduino*. Sapientia 129. Universitat Jaume I. ISBN: 978-84-16546-76-3. DOI: 10.6035/Sapientia129.

Naimi, Sepehr, Srmad Naime y Muhammad Ali Mazidi (2020). *Arm Cortex-M Assembly Programming for Embedded Programmers: Using Keil*. Mazidi & Naimi. ISBN: 978-197005413-2.

Nistal, Martín Llamas, Fernando A. Mikic Fonte y Manuel J. Fernández-Iglesias (2012). *Arquitectura de Ordenadores: Problemas y Cuestiones de Teoría*. Andavira. ISBN: 978-84-8408663-5.

NXP Semiconductors (2017). *Kinetis KL02 32 KB Flash 48 MHz Cortex-M0+ Based Microcontroller*. KL02P32M48SF0. NXP Semiconductors.

Planting, C. Arlen (2017). *Microprocessor Fundamentals: Using Qemu Emulation of Arm Architecture, the Gnu Toolchain, and Gdb*. Amazon Digital Services. ISBN: 978-1548370527.

Sato, Mitsuhisa y col. (2020). "Co-Design for A64FX Manycore Processor and Fugaku". En: *Procs. of SC20 International Conference for High Performance Computing, Networking, Storage and Analysis*. DOI: 10.1109/SC41405.2020.00051.

Smith, Bruce (2013). *Raspberry Pi Assembly Language*. CreateSpace Independent Publishing Platform. ISBN: 978-149213528-9.

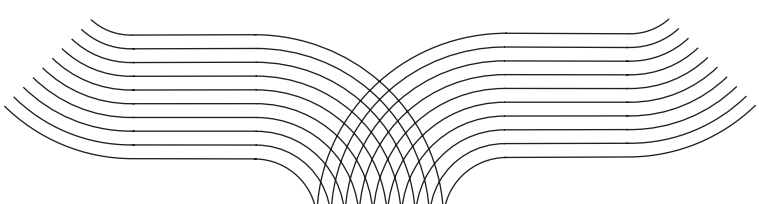
Stallings, William (2006). *Organización y arquitectura de computadoras*. Grupo Anaya Publicaciones Generales. ISBN: 978-84-8966082-3.

— (2015). *Computer Organization and Architecture*. Global edition. Pearson Education. ISBN: 978-0-13035119-7.

Tanenbaum, Andrew S. (2000). *Organizacion de Computadoras - Un Enfoque Estructurado*. Pearson Educacion. ISBN: 978-970170399-1.

— (2016). *Structured Computer Organization*. Pearson Education. ISBN: 978-0-13291652-3.

Índice de materias



Manuais

Serie de manuais didácticos

Últimas publicacions na colección

Elaboración de TFG, TFM e Teses: Claves para o éxito (2021)
Laura Novelle López

Gestión del circulante. Una aplicación práctica para la PYME (2021)
Javier Lorenzo Paniagua, Pablo Cabanelas Lorenzo e Pedro González Santamaría

Las ecuaciones del océano: Teoría y problemas resueltos (2020)
Gabriel Rosón Porto

Design Thinking: Guía de iniciación (2020)
Manuel José Fernández Iglesias, Manuel Caeiro Rodríguez, Íñigo Cuiñas Gómez, Enrique Costa Montenegro, Francisco Javier Díaz Otero e Perfecto Mariño Espiñeira

Implementación e desenvolvemento de aulas de xeometría euclídea e diferencial en SAGE (2020)
Francisco de Arriba Pérez, Alberto Castejón Lafuente, Eusebio Corbacho Rosas, M.ª Carmen Somoza López e Ricardo Vidal Vázquez

9 788481 589085

Manual de Programación en Ensamblador

Unha achega teórico-práctica

Este texto ten como obxectivo servir de referencia para aprender a programar en ensamblador mediante exemplos. Utilizando como instrumento a linguaxe ensambladora da versión T32/Thumb da arquitectura ARM, desenvólvese un conxunto de supostos de programación de complexidade crecente, elixidos polo seu valor pedagóxico. Comezamos coa programación das estruturas básicas de control e a trasfega de información utilizando os rexistros e a memoria principal, resolvendo tarefas como contar, sumar e comparar números. A continuación, tratamos o procesado de cadeas de caracteres a utilización combinada do repertorio de instrucións aritméticas, lóxicas e de desprazamento. Máis adiante introducimos o concepto de subprograma, para presentar despois o concepto de pila e o paso de parámetros a través da pila, o aniñamento de chamadas a subprograma e a recursividade. Finalmente tratamos a comunicación do ordenador coa súa contorna, é dicir, á resolución de problemas de programación de entrada e saída, ben directamente ou a través de outros módulos software ou do sistema operativo. O libro inclúe tamén un resumo do modelo do programador T32/Thumb, e unha descrición detallada do sistema de codificación UTF8. Como ferramentas de traballo, utilizamos o ensamblador de GNU, o simulador QtARMSim e o ordenador pedagóxico Raspberri Pi.

Servizo de Publicacións
Universidade Vigo