ANTONIO JOSÉ VILLENA GODOY
RAFAEL ASENJO PLAZA
FRANCISCO J. CORBERA PEÑA

PRÁCTICAS DE ENSAMBLADOR BASADAS EN RASPBERRY PI

Departamento de Arquitectura de Computadores

UNIVERSIDAD DE MÁLAGA / MANUALES

Aviso

- Este material ha sido preparado por:

  Antonio José Villena Godoy
  Rafael Asenjo Plaza
  Francisco J. Corbera Peña
  Dept. de Arquitectura de Computadores. Universidad de Málaga.

-

- Debe dar crédito en la obra en la forma especificada por el autor o licenciante.

- El licenciante permite copiar, distribuir y comunicar publicamente la obra. A cambio, esta obra no puede ser utilizada con fines comerciales – a menos que se obtenga el permiso expreso del licenciante.

- El licenciante permite copiar, distribuir, transmitir y comunicar públicamente solamente copias inalteradas de la obra – no obras derivadas basadas en ella.

- Si desea enviar sugerencias, comentarios o propuestas de mejora sobre el contenido de este material, envie un correo electrónico a la dirección `asenjo@uma.es`.

Acrónimos

AAPCS	ARM Architecture Procedure Call Standard
ARM	Advanced RISC Machines
CPSR	Current Program Status Register
CPU	Central Processing Unit
CHI	system timer Counter HIgher
CLO	system timer Counter LOwer
CS	system timer Control/Status
E/S	Entrada/Salida
ETSII	Escuela Técnica Superior de Ingeniería Informática
FIQ	Fast Interrupt reQuest
GNU	GNU is Not Unix
GCC	GNU C Compiler
GDB	GNU DeBugger
GPAFEN	GPIO Pin Async. Falling Edge Detect
GPAREN	GPIO Pin Async. Rising Edge Detect
GPEDS	GPIO Pin Event Detect Status
GPFEN	GPIO Pin Falling Edge Detect Enable
GPHEN	GPIO Pin High Detect Enable
GPIO	General-Purpose Input/Output

GPL	General Public License
GPLEN	GPIO Pin Low Detect Enable
GPLEV	GPIO Pin LEVel
GPPUD	GPIO Pin High Detect Enable
GPPUDCLK	GPIO Pin High Detect Enable CLocK
GPREN	GPIO Pin Rising Edge Detect Enable
GPU	Graphics Processing Unit
IRQ	Interrupt ReQuest
LED	Light Emitting Diode
LR	Link Register
PFC	Proyecto Fin de Carrera
PC	Personal Computer
RAM	Random-Access Memory
RISC	Reduced Instruction Set Computer
ROM	Read-Only Memory
RTI	Rutina de Tratamiento de Interrupción
SoC	System on a Chip
SP	Stack Pointer
SPSR	Saved Program Status Register
UMA	Universidad de MÁlaga
VFP	Vector Floating-Point
abt	ABorT mode
mon	secure MONitor mode
svc	Supervisor mode (antiguamente SuperVisor Calls)
und	UNDefined mode

Índice

Índice de figuras

Índice de Tablas

Prólogo

El minicomputador Raspberry Pi es una placa del tamaño de una tarjeta de crédito y un precio de sólo 30€. El objetivo principal de sus creadores, la Fundación Raspberry Pi, era promover la enseñanza de conceptos básicos de informática en los colegios e institutos. Sin embargo, ha terminado convirtiéndose también en un pequeño computador de bajo coste que se destina a muy diversos usos: servidor multimedia conectado al televisor, estación base para domótica en el hogar, estaciones meteorológicas, servidor de discos en red para copias de seguridad, o como un simple ordenador que puede ejecutar aplicaciones de internet, juegos, ofimática, etc. Esto ha llegado a ser así gracias a un vertiginoso crecimiento de la comunidad de desarrolladores para Raspberry Pi, y que estos han explorado casi todas las posibilidades para sacar el máximo partido de este ordenador de 30€. Esa gran funcionalidad y el bajo coste constituyen el principal atractivo de esta plataforma para los estudiantes. Sin embargo, para los docentes del Dept. de Arquitectura de Computadores, la Raspberry Pi ofrece una excusa perfecta para hacer más amenos y atractivos conceptos a veces complejos, y a veces también áridos, de asignaturas del área.

Este trabajo se enmarca dentro del Proyecto de Innovación Educativa PIE13-082, "Motivando al alumno de ingeniería mediante la plataforma Raspberry Pi" cuyo principal objetivo es aumentar el grado de motivación del alumno que cursa asignaturas impartidas por el Departamento de Arquitectura de Computadores. La estrategia propuesta se apoya en el hecho de que muchos alumnos de Ingeniería perciben que las asignaturas de la carrera están alejadas de su realidad cotidiana, y que por ello, pierden cierto atractivo. Sin embargo, bastantes de estos alumnos han comprado o piensan comprar un minicomputador Raspberry Pi que se caracteriza por proporcionar una gran funcionalidad, gracias a estar basado en un procesador y Sistema Operativo de referencia en los dispositivos móviles. En este proyecto proponemos aprovechar el interés que los alumnos ya demuestran por la plataforma Raspberry Pi, para ponerlo a trabajar en pro del siguiente objetivo docente: facilitar el estudio de conceptos y técnicas impartidas en varias asignaturas del Departamento. Cuatro de estas asignaturas son:

- Tecnología de Computadores: Asignatura obligatoria del módulo de Formación

Común de las titulaciones de Grado en Ingeniería Informática, Grado en Ingeniería de Computadores y Grado en Ingeniería del Software. Es una asignatura que se imparte en el primer curso.

- Estructura de Computadores: Asignatura obligatoria del módulo de Formación Común de las titulaciones de Grado en Ingeniería Informática, Grado en Ingeniería de Computadores y Grado en Ingeniería del Software. Es una asignatura que se imparte en el segundo curso.

- Sistemas Operativos: Asignatura obligatoria del módulo de Formación Común de las titulaciones de Grado en Ingeniería Informática, Grado en Ingeniería de Computadores y Grado en Ingeniería del Software. Se imparte en segundo curso.

- Diseño de Sistemas Operativos: Asignatura obligatoria del módulo de Tecnologías Específicas del Grado de Ingeniería de Computadores. Se imparte en tercer curso.

En esas cuatro asignaturas, uno de los conceptos más básicos es el de gestión de interrupciones a bajo nivel. En particular, en Estructura de Computadores, esos conceptos se ilustraban en el pasado mediante prácticas en PCs con MSDOS y programación en ensamblador, pero el uso de ese sistema operativo ya no tiene ningún atractivo y además crea problemas de seguridad en los laboratorios del departamento. Sin embargo, la plataforma Raspberry Pi se convierte en una herramienta adecuada para trabajar a nivel de sistema, es económica y ya disponemos de unidades suficientes para usarlas en los laboratorios (30 equipos para ser exactos).

El principal objetivo de este trabajo es la creación de un conjunto de prácticas enfocadas al aprendizaje de la programación en ensamblador, en concreto del ARMv6 que es el procesador de la plataforma que se va a utilizar para el desarrollo de las prácticas, así como al manejo a bajo nivel de las interrupciones y la entrada/salida en dicho procesador. El aprendizaje del lenguaje ensamblador del procesador ARM usado en la Raspberry Pi se puede completar leyendo la documentación disponible en [1] y haciendo los tutoriales de [2]. Para la parte más centrada en el hardware también se puede consultar la amplia documentación disponible en internet, como por ejemplo los tutoriales disponibles en [3] y la descripción los modos de operación de los periféricos conectados al procesador ARM [4].

La presente memoria está dividida cinco capítulos y cuatro apéndices. De los 5 capítulos, el primero es introductorio. Los dos siguientes se centran en la programación de ejecutables en Linux, tratando las estructuras de control en el capítulo 2 y las subrutinas (funciones) en el capítulo 3. Los dos últimos capítulos muestran la programación en Bare Metal, explicando el subsistema de entrada/salida (puertos de entrada/salida y temporizadores) de la plataforma Raspberry Pi y su manejo a

bajo nivel en el capítulo 4 y las interrupciones en el capítulo 5. En los apéndices hemos añadido aspectos laterales pero de suficiente relevancia como para ser considerados en la memoria, como el apendice A que explica el funcionamiento de la macro ADDEXC, el apéndice B que muestra todos los detalles de la placa auxiliar, el apéndice C que nos enseña a agilizar la carga de programas Bare Metal y por último tenemos el apéndice D, que profundiza en aspectos del GPIO como las resistencias programables.

Capítulo 1

Introducción al ensamblador

Contenido

Objetivo: En esta sesión vamos a conocer el entorno de trabajo. Veremos qué aspecto tiene un programa en ensamblador, veremos cómo funcionan los tres programas que vamos a utilizar: el ensamblador, el *enlazador (linker)* y el *depurador (debugger)*. Del *debugger* sólo mostraremos unos pocos comandos, que ampliaremos en las próximas sesiones. También veremos la representación de los números naturales y de los enteros, y el funcionamiento de algunas de las instrucciones del ARM. Se repasarán también los conceptos de registros, flags e instrucciones para la manipulación de bits.

1.1. Lectura previa

1.1.1. Características generales de la arquitectura ARM

ARM es una arquitectura RISC (Reduced Instruction Set Computer=Ordenador con Conjunto Reducido de Instrucciones) de 32 bits, salvo la versión del core ARMv8-A que es mixta 32/64 bits (bus de 32 bits con registros de 64 bits). Se trata de una arquitectura licenciable, quiere decir que la empresa desarrolladora ARM Holdings diseña la arquitectura, pero son otras compañías las que fabrican y venden los chips, llevándose ARM Holdings un pequeño porcentaje por la licencia.

El chip en concreto que lleva la Raspberry Pi es el BCM2835, se trata de un SoC (System on a Chip=Sistema en un sólo chip) que contiene además de la CPU otros elementos como un núcleo GPU (hardware acelerado OpenGL ES/OpenVG/Open EGL/OpenMAX y decodificación H.264 por hardware) y un núcleo DSP (Digital signal processing=Procesamiento digital de señales) que es un procesador más pequeño y simple que el principal, pero especializado en el procesado y representación de señales analógicas. La CPU en cuestión es la ARM1176JZF-S, un chip de la familia ARM11 que usa la arquitectura ARMv6k.

Familia	**Arquitectura**	**Bits**	**Ejemplos de dispositivos**
ARM1	ARMv1	32/26	Segundo procesador BBC Micro
ARM2, ARM3, Amber	ARMv2	32/26	Acorn Archimedes
ARM6, ARM7	ARMv3	32	Apple Newton Serie 100
ARM8, StrongARM	ARMv4	32	Apple Newton serie 2x00
ARM7TDMI, ARM9TDMI	ARMv4T	32	Game Boy Advance
ARM7EJ, ARM9E, ARM10E, XScale	ARMv5	32	Samsung Omnia, Blackberry 8700
ARM11	ARMv6	32	iPhone 3G, Raspberry Pi
Cortex-M0/M0+/M1	ARMv6-M	32	
Cortex-M3/M4	ARMv7-M ARMv7E-M	32	Texas Instruments Stellaris
Cortex-R4/R5/R7	ARMv7-R	32	Texas Instruments TMS570
Cortex-A5/A7/A8/A9 A12/15/17, Apple A6	ARMv7-A	32	Apple iPad
Cortex-A53/A57, X-Gene, Apple A7	ARMv8-A	64/32	Apple iPhone 5S

Tabla 1.1: Lista de familias y arquitecturas ARM

Las extensiones de la arquitectura ARMv6k frente a la básica ARMv6 son míni-

mas por lo que a efectos prácticos trabajaremos con la arquitectura ARMv6.

Registros

La arquitectura ARMv6 presenta un conjunto de 17 registros (16 principales más uno de estado) de 32 bits cada uno.

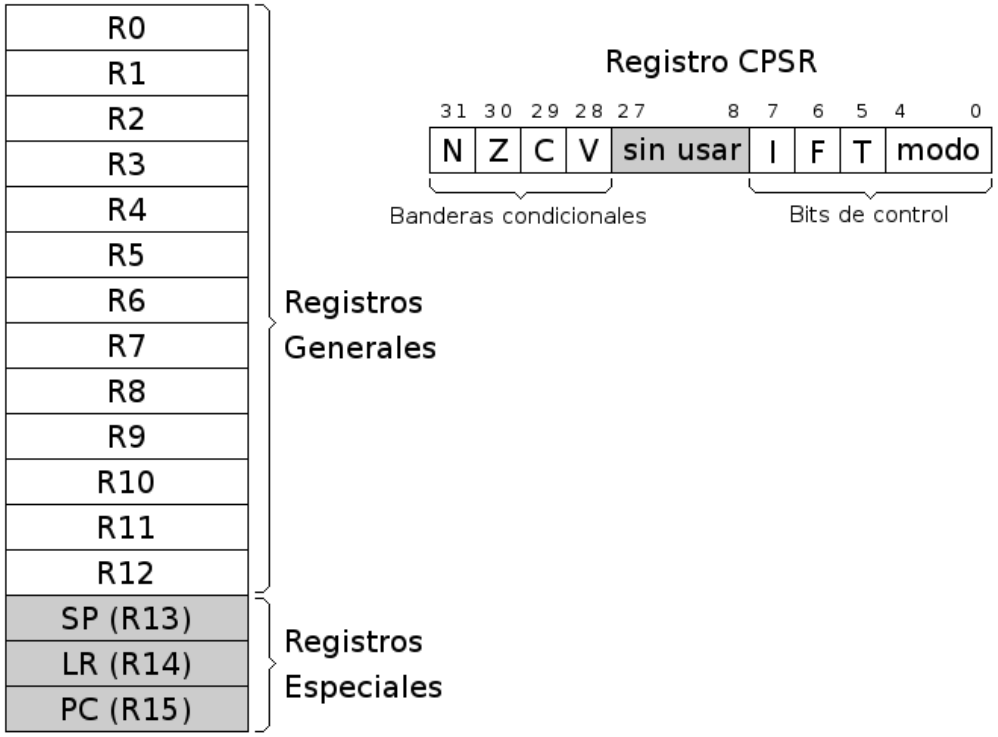

Figura 1.1: Registros de la arquitectura ARM

Registros Generales. Su función es el almacenamiento temporal de datos. Son los 13 registros que van R0 hasta R12.

Registros Especiales. Son los últimos 3 registros principales: R13, R14 y R15. Como son de propósito especial, tienen nombres alternativos.

- **SP**/R13. Stack Pointer ó Puntero de Pila. Sirve como puntero para almacenar variables locales y registros en llamadas a funciones.
- **LR**/R14. Link Register ó Registro de Enlace. Almacena la dirección de retorno cuando una instrucción BL ó BLX ejecuta una llamada a una rutina.

- **PC**/R15. Program Counter ó Contador de Programa. Es un registro que indica la posición donde está el procesador en su secuencia de instrucciones. Se incrementa de 4 en 4 cada vez que se ejecuta una instrucción, salvo que ésta provoque un salto.

Registro CPSR. Almacena las banderas condicionales y los bits de control. Los bits de control definen la habilitación de interrupciones normales (I), interrupciones rápidas (F), modo Thumb [1] (T) y el modo de operación de la CPU. Existen hasta 8 modos de operación, pero por ahora desde nuestra aplicación sólo vamos a trabajar en uno de ellos, el *Modo Usuario*. Los demás son modos privilegiados usados exclusivamente por el sistema operativo.

Desde el *Modo Usuario* sólo podemos acceder a las banderas condicionales, que contienen información sobre el estado de la última operación realizada por la ALU. A diferencia de otras arquitecturas en ARMv6 podemos elegir si queremos que una instrucción actualice o no las banderas condicionales, poniendo una "s" detrás del nemotécnico [2]. Existen 4 banderas y son las siguientes:

- **N**. Se activa cuando el resultado es negativo.
- **Z**. Se activa cuando el resultado es cero o una comparación es cierta.
- **C**. Indica acarreo en las operaciones aritméticas.
- **V**. Desbordamiento aritmético.

Esquema de almacenamiento

El procesador es *Bi-Endian*, quiere decir que es configurable entre *Big Endian* y *Little Endian*. Aunque nuestro sistema operativo nos lo limita a *Little Endian*.

Por tanto la regla que sigue es "el byte menos significativo ocupa la posición más baja". Cuando escribimos un dato en una posición de memoria, dependiendo de si es byte, half word o word,... se ubica en memoria según el esquema de la figura 1.2. La dirección de un dato es la de su byte menos significativo. La memoria siempre se referencia a nivel de byte, es decir si decimos la posición N nos estamos refiriendo al byte N-ésimo, aunque se escriba media palabra, una palabra,...

[1] Es un modo simplificado donde las instrucciones son de 16 bits en lugar de 32 y se acceden a menos registros (hasta r7), con la ventaja de que el código ocupa menos espacio.

[2] Es la forma de nombrar las instrucciones desde ensamblador, normalmente derivadas de una abreviatura del verbo en inglés. Por ejemplo la instrucción *MOV* viene de "move" (mover)

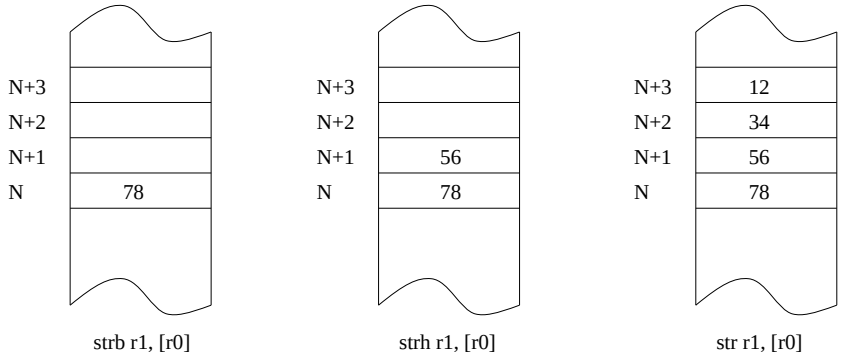

Figura 1.2: Ubicación de datos en memoria

1.1.2. El lenguaje ensamblador

El ensamblador es un lenguaje de bajo nivel que permite un control directo de la CPU y todos los elementos asociados. Cada línea de un programa ensamblador consta de una instrucción del procesador y la posición que ocupan los datos de esa instrucción.

Desarrollar programas en lenguaje ensamblador es un proceso laborioso. El procedimiento es similar al de cualquier lenguaje compilado. Un conjunto de instrucciones y/o datos forman un módulo fuente. Este módulo es la entrada del compilador, que chequea la sintaxis y lo traduce a código máquina formando un módulo objeto. Finalmente, un enlazador (montador ó *linker*) traduce todas las referencias relativas a direcciones absolutas y termina generando el ejecutable.

El ensamblador presenta una serie de ventajas e inconvenientes con respecto a otros lenguajes de más alto nivel. Al ser un lenguaje de bajo nivel, presenta como principal característica la flexibilidad y la posibilidad de acceso directo a nivel de registro. En contrapartida, programar en ensamblador es laborioso puesto que los programas contienen un número elevado de líneas y la corrección y depuración de éstos se hace difícil.

Generalmente, y dado que crear programas un poco extensos es laborioso, el ensamblador se utiliza como apoyo a otros lenguajes de alto nivel para 3 tipos de situaciones:

- Operaciones que se repitan un número elevado de veces.
- Cuando se requiera una gran velocidad de proceso.
- Para utilización y aprovechamiento de dispositivos y recursos del sistema.

1.1.3. El entorno

Los pasos habituales para hacer un programa (en cualquier lenguaje) son los siguientes: lo primero es escribir el programa en el lenguaje fuente mediante un editor de programas. El resultado es un fichero en un lenguaje que puede entender el usuario, pero no la máquina. Para traducirlo a lenguaje máquina hay que utilizar un programa traductor. Éste genera un fichero con la traducción de dicho programa, pero todavía no es un programa ejecutable. Un fichero ejecutable contiene el programa traducido más una serie de códigos que debe tener todo programa que vaya a ser ejecutado en una máquina determinada. Entre estos códigos comunes se encuentran las librerías del lenguaje. El encargado de unir el código del programa con el código de estas librerías es un programa llamado montador (*linker*) que genera el programa ejecutable (ver la figura 1.3)

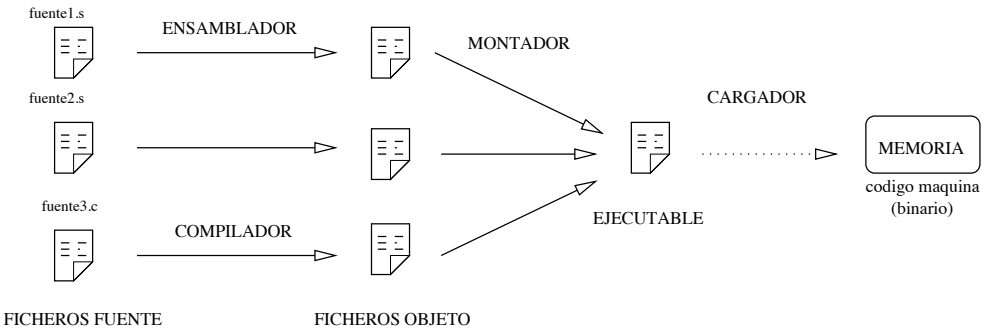

Figura 1.3: Entorno típico de programación

Durante el proceso de creación de un programa se suelen producir errores. Hay dos tipos de errores: los sintácticos o detectables en tiempo de traducción y los errores semánticos o detectables en tiempo de ejecución. Los errores sintácticos son, por ejemplo, escribir mal una instrucción o hacer una operación entre dos tipos de datos incompatibles. Estos errores son detectados por el traductor y se deben solucionar para poder generar un ejecutable.

Una vez que se tiene un programa sintácticamente correcto lo podemos ejecutar, pero ésto no implica que el programa sea correcto. Todas las instrucciones pueden ser correctas, pero se puede haber olvidado poner la condición de salida de un bucle (y que no termine nunca) o que sencillamente el programa no haga lo que queremos.

Estos errores sólo se pueden detectar en tiempo de ejecución. Para poder eliminarlos se utiliza un depurador de programas (*debugger*). El depurador nos permite ejecutar el programa instrucción a instrucción y ver todos los valores que se van a calcular, de manera que podemos encontrar los errores.

En el laboratorio utilizaremos el editor **nano** para crear y editar los módulos fuente de nuestros programas. El traductor (que en el caso de traducir de un len-

guaje ensamblador a lenguaje máquina recibe el nombre de ensamblador), el *linker* y el *debugger* son respectivamente GNU Assembler (**as**), GNU Compiler Collection (**gcc**) y GNU Debbuger (**gdb**). Todas estas herramientas forman parte de la GNU toolchain que viene instalada por defecto en la mayoría de las distribuciones basadas en Linux, en concreto Raspbian. Para obtener más información sobre estos comandos se puede recurrir a la ayuda del sistema con `man as`, `man gcc` y `man gdb`.

1.1.4. Configuración del entorno para realizar las prácticas en casa

Las instrucciones vienen detalladas en esta dirección:
http://elinux.org/RPi_Easy_SD_Card_Setup

Vamos a hacer un resumen de cómo se haría en Windows. Para otros sistemas operativos (Linux, Mac OS) seguir las instrucciones antes mencionadas.

1. Descargamos la última versión de RASPBIAN en la siguiente url:
   http://www.raspberrypi.org/downloads/
2. Extraemos del .zip el archivo de imagen, en nuestro caso se llama 2014-01-07-wheezy-raspbian.img, aunque seguramente tu versión será más moderna.
3. Insertamos una tarjeta SD en tu PC (slot SD o adaptador USB) y nos aseguramos de que funcione correctamente. Si no, la formateamos en FAT32.
4. Nos bajamos e instalamos la utilidad Win32DiskImager.
   http://sourceforge.net/projects/win32diskimager
5. Ejecutamos como Administrador la utilidad anterior.
6. Dentro de la utilidad, seleccionamos el archivo de imagen anterior, 2014-01-07-wheezy-raspbian.img
7. Seleccionamos en Device la letra de unidad que nos apareció en el paso 3. Debemos asegurarnos de que la letra sea la correcta, de lo contrario podríamos destruir los datos de nuestro disco duro.
8. Pulsamos el botón Write y esperamos a que se complete la escritura.
9. Salimos de la utilidad y extraemos la tarjeta SD.
10. Ya estamos listos para introducir la tarjeta SD en nuestra Raspberry Pi.

De forma alternativa podemos ejecutar la imagen anterior en un emulador de Raspberry Pi, y seguir gran parte de las prácticas con la comodidad de tu PC. Para ello partimos del archivo de imagen obtenido en el apartado 2 de la lista anterior, y seguimos los pasos según [5]. Los pasos son válidos para Windows y Linux, aunque nosotros mostraremos sólo los de Windows.

1. Descargamos el emulador QEMU desde aquí:
   http://lassauge.free.fr/qemu/

2. Descargamos el siguiente núcleo o kernel desde aquí:
   http://xecdesign.com/downloads/linux-qemu/kernel-qemu

3. Lanzamos la línea de comandos o ventana de MS-DOS. Esto se hace desde `Programas->Accesorios->Símbolo del sistema` o bien pulsando `Windows+R` y escribiendo "cmd". Una vez lanzada escribimos lo siguiente:

   ```
   qemu-system-armw -kernel kernel-qemu -cpu arm1176
   -m 256 -M versatilepb -no-reboot -serial stdio -append
   "root=/dev/sda2 panic=1 rootfstype=ext4 rw init=/bin/bash"
   -hda 2014-01-07-wheezy-raspbian.img
   ```

4. Aparece el emulador en una nueva ventana tipo terminal. Ya estaríamos dentro de la Raspberry emulada. Una vez se muestren los mensajes de arranque aparece el siguiente texto:

   ```
   raspberrypi login:
   ```

   Nos está pidiendo el nombre de usuario. Nosotros escribimos `pi`.

5. Luego nos piden el password, que es `raspberry`. En este caso y por motivos de seguridad no se recibe respuesta visual mientras escribimos la contraseña, ni siquiera aparecen asteriscos.

6. Una vez identificados, lo primero que hacemos es editar el archivo `/etc/ld.so.preload` con el siguiente comando:

   ```
   nano /etc/ld.so.preload
   ```

7. Dentro del editor ponemos un # al comienzo de la siguiente línea:

   ```
   #/usr/lib/arm-linux-gnueabihf/libcofi_rpi.so
   ```

8. Presionamos `Ctrl-X` y luego `y`, Enter para guardar y salir.

9. Escribimos `sudo halt` para salir limpiamente del sistema emulado.

10. Cerramos la ventana de QEMU y creamos el siguiente archivo `lanzador.bat`.

```
qemu-system-armw -kernel kernel-qemu -cpu arm1176
-m 256 -M versatilepb -no-reboot -serial stdio -append
"root=/dev/sda2 panic=1 rootfstype=ext4 rw"
-hda 2014-01-07-wheezy-raspbian.img
```

11. Ejecutamos el archivo `lanzador.bat` que acabamos de crear. Ya hemos terminado. Todos los archivos que vayamos creando se almacenan en la imagen como si se tratase de una SD real corriendo sobre una Raspberry Pi real.

1.1.5. Aspecto de un programa en ensamblador

En el listado 1.1 se muestra el código de la primera práctica que probaremos. En el código hay una serie de elementos que aparecerán en todos los programas y que estudiaremos a continuación.

Listado 1.1: Código del programa intro1.s

```
.data

var1:    .word   3
var2:    .word   4
var3:    .word   0x1234

.text
.global main

main:    ldr     r1, puntero_var1    /* r1 <- &var1    */
         ldr     r1, [r1]            /* r1 <- *r1      */
         ldr     r2, puntero_var2    /* r2 <- &var2    */
         ldr     r2, [r2]            /* r2 <- *r2      */
         ldr     r3, puntero_var3    /* r3 <- &var3    */
         add     r0, r1, r2          /* r0 <- r1 + r2  */
         str     r0, [r3]            /* *r3 <- r0      */
         bx      lr

puntero_var1:    .word   var1
puntero_var2:    .word   var2
puntero_var3:    .word   var3
```

La principal característica de un módulo fuente en ensamblador es que existe una clara separación entre las instrucciones y los datos. La *estructura más general de un módulo fuente* es:

* **Sección de datos.** Viene identificada por la directiva *.data*. En esta zona se definen todas las variables que utiliza el programa con el objeto de reservar memoria para contener los valores asignados. Hay que tener especial cuidado para que los datos estén alineados en palabras de 4 bytes, sobre todo después de las cadenas. Alinear significa rellenar con ceros el final de un dato para que el siguiente dato comience en una dirección múltiplo de 4 (con los dos bits menos significativos a cero). Los datos son modificables.

* **Sección de código.** Se indica con la directiva *.text*, y sólo puede contener código o datos no modificables. Como todas las instrucciones son de 32 bits no hay que tener especial cuidado en que estén alineadas. Si tratamos de escribir en esta zona el ensamblador nos mostrará un mensaje de error.

De estas dos secciones la única que obligatoriamente debe existir es la sección .text (o sección de código). En el ejemplo 1.1 comprobamos que están las dos.

Un módulo fuente, como el del ejemplo, está formado por instrucciones, datos, símbolos y directivas. Las instrucciones son representaciones nemotécnicas del juego de instrucciones del procesador. Un dato es una entidad que aporta un valor numérico, que puede expresarse en distintas bases o incluso a través de una cadena. Los símbolos son representaciones abstractas que el ensamblador maneja en tiempo de ensamblado pero que en el código binario resultante tendrá un valor numérico concreto. Hay tres tipos de símbolos: las etiquetas, las macros y las constantes simbólicas. Por último tenemos las directivas, que sirven para indicarle ciertas cosas al ensamblador, como delimitar secciones, insertar datos, crear macros, constantes simbólicas, etc... Las instrucciones se aplican en tiempo de ejecución mientras que las directivas se aplican en tiempo de ensamblado.

Datos

Los datos se pueden representar de distintas maneras. Para representar números tenemos 4 bases. La más habitual es en su forma decimal, la cual no lleva ningún delimitador especial. Luego tenemos otra muy útil que es la representación hexadecimal, que indicaremos con el prefijo *0x*. Otra interesante es la binaria, que emplea el prefijo *0b* antes del número en binario. La cuarta y última base es la octal, que usaremos en raras ocasiones y se especifica con el prefijo *0*. Sí, un cero a la izquierda de cualquier valor convierte en octal dicho número. Por ejemplo 015 equivale a 13 en decimal. Todas estas bases pueden ir con un signo menos delante, codificando el valor negativo en complemento a dos. Para representar carácteres y cadenas emplearemos las comillas simples y las comillas dobles respectivamente.

Símbolos

Como las etiquetas se pueden ubicar tanto en la sección de datos como en la de código, la versatilidad que nos dan las mismas es enorme. En la zona de datos, las etiquetas pueden representar variables, constantes y cadenas. En la zona de código podemos usar etiquetas de salto, funciones y punteros a zona de datos.

Las macros y las constantes simbólicas son símbolos cuyo ámbito pertenece al preprocesador, a diferencia de las etiquetas que pertenecen al del ensamblador. Se especifican con las directivas .macro y .equ respectivamente y permiten que el código sea más legible y menos repetitivo.

Instrucciones

Las instrucciones del **as** (a partir de ahora usamos **as** para referirnos al ensamblador) responden al formato general:

```
Etiqueta:   Nemotécnico   Operando/s   /* Comentario */
```

De estos campos, sólo el nemónico (nombre de la instrucción) es obligatorio. En la sintaxis del **as** cada instrucción ocupa una línea terminando preferiblemente con el ASCII 10 (LF), aunque son aceptadas las 4 combinaciones: CR, LF, CR LF y LF CR. Los campos se separan entre sí por al menos un carácter espacio (ASCII 32) o un tabulador y no existe distinción entre mayúsculas y minúsculas.

```
main:    ldr      r1, puntero_var1     /* r1 <- &var1     */
```

El Campo *etiqueta*, si aparece, debe estar formado por una cadena alfanumérica. La cadena no debe comenzar con un dígito y no se puede utilizar como cadena alguna palabra reservada del **as** ni nombre de registro del microprocesador. En el ejemplo, la etiqueta es `main:`.

El campo *Nemotécnico* (`ldr` en el ejemplo) es una forma abreviada de nombrar la instrucción del procesador. Está formado por caracteres alfabéticos (entre 1 y 11 caracteres).

El campo *Operando/s* indica dónde se encuentran los datos. Puede haber 0, 1 ó más operandos en una instrucción. Si hay más de uno normalmente al primero se le denomina destino (salvo excepciones como `str`) y a los demás fuentes, y deben ir separados por una coma. Los operandos pueden ser registros, etiquetas, valores inmediatos o incluso elementos más complejos como desplazadores/rotadores o indicadores de pre/post-incrementos. En cualquiera de los casos el tamaño debe ser una palabra (32 bits), salvo contadas excepciones como `ldr` y `str` donde puede ser media palabra (16 bits) o un byte (8 bits). En el ejemplo `r1` es el operando destino, de tipo registro, y `puntero_var1` es el operando fuente, una etiqueta. Tanto `r1` como `puntero_var1` hacen referencia a un valor de tamaño palabra (32 bits).

El campo *Comentario* es opcional (`r1 <- &var1`, en el ejemplo) y debe comenzar con la secuencia `/*` y acabar con `*/` al igual que los comentarios multilínea en C. No es obligatorio que estén a la derecha de las instrucciones, aunque es lo habitual. También es común verlos al comienzo de una función (ocupando varias líneas) para explicar los parámetros y funcionalidad de la misma.

Cada instrucción del **as** se refiere a una operación que puede realizar el microprocesador. También hay pseudoinstrucciones que son tratadas por el preprocesador como si fueran macros y codifican otras instrucciones, como `lsl rn, #x` que codifica `mov rn, rn, lsl #x`, o bien `push/pop` que se traducen instrucciones `stm/ldm` más complejas y difíciles de recordar para el programador. Podemos agrupar el conjunto de instrucciones del **as**, según el tipo de función que realice el microprocesador, en las siguientes categorías:

- *Instrucciones de transferencia de datos* Mueven información entre registros y posiciones de memoria. En la arquitectura ARMv6 no existen puertos ya que la E/S está mapeada en memoria. Pertenecen a este grupo las siguientes instrucciones: **mov, ldr, str, ldm, stm, push, pop**.
- *Instrucciones aritméticas.* Realizan operaciones aritméticas sobre números binarios o BCD. Son instrucciones de este grupo **add, cmp, adc, sbc, mul**.
- *Instrucciones de manejo de bits.* Realizan operaciones de desplazamiento, rotación y lógicas sobre registros o posiciones de memoria. Están en este grupo las instrucciones: **and, tst, eor, orr, LSL, LSR, ASR, ROR, RRX**.
- *Instrucciones de transferencia de control.* Se utilizan para controlar el flujo de ejecución de las instrucciones del programa. Tales como **b, bl, bx, blx** y sus variantes condicionales.

En esta sesión práctica se explorarán algunas de estas instrucciones. Para buscar información sobre cualquiera de ellas durante las prácticas, recuerda que puedes utilizar el manual técnico del ARM1176JZF-S [6].

Directivas

Las directivas son expresiones que aparecen en el módulo fuente e indican al compilador que realice determinadas tareas en el proceso de compilación. Son fácilmente distinguibles de las instrucciones porque siempre comienzan con un punto. El uso de directivas es aplicable sólo al entorno del compilador, por tanto varían de un compilador a otro y para diferentes versiones de un mismo compilador. Las directivas más frecuentes en el **as** son:

- *Directivas de asignación:* Se utilizan para dar valores a las constantes o reservar posiciones de memoria para las variables (con un posible valor inicial). **.byte, .hword, .word, .ascii, .asciz, .zero y .space** son directivas que indican al compilador que reserve memoria para las variables del tipo indicado. Por ejemplo:

```
a1:   .byte  1        /* tipo byte, inicializada a 1 */
var2: .byte  'A'      /* tipo byte, al caracter 'A'  */
var3: .hword 25000    /* tipo hword (16 bits) a 25000*/
var4: .word 0x12345678 /* tipo word de 32 bits       */
b1:   .ascii "hola"   /* define cadena normal        */
b2:   .asciz "ciao"   /* define cadena acabada en NUL*/
dat1: .zero  300      /* 300 bytes de valor cero     */
dat2: .space 200, 4   /* 200 bytes de valor 4        */
```

  La directiva **.equ** (ó **.set**) es utilizada para asignar un valor a una constante simbólica:

```
.equ N, -3      /* en adelante N se sustituye por -3 */
```

- *Directivas de control:* **.text y .data** sirven para delimitar las distintas secciones de nuestro módulo. **.align alineamiento** es para alinear el siguiente dato, rellenando con ceros, de tal forma que comience en una dirección múltiplos del número que especifiquemos en *alineamiento*, normalmente potencia de 2. Si no especificamos *alineamiento* por defecto toma el valor de 4 (alineamiento a palabra):

```
a1: .byte 25    /* definimos un byte con el valor 25 */
    .align      /* directiva que rellena con 3 bytes */
a2: .word 4     /* variable alineada a tamaño palabra*/
```

  .include para incluir un archivo fuente dentro del actual. **.global** hace visible al enlazador el símbolo que hemos definido con la etiqueta del mismo nombre.

- *Directivas de operando:* Se aplican a los datos en tiempo de compilación. En general, incluyen las operaciones lógicas `&`, `|`, $\sim$, aritméticas `+`, `-`, `*`, `/`, `%` y de desplazamiento $<$, $>$, $<<$, $>>$:

```
.equ pies, 9      /* definimos a 9 la constante pies */
.equ yardas, pies/3     /* calculamos las yardas= 3 */
.equ pulgadas, pies*12  /* calculamos pulgadas= 108 */
```

- *Directivas de Macros:* Una *.macro* es un conjunto de sentencias en ensamblador (directivas e instrucciones) que pueden aparecer varias veces repetidas en un

programa con algunas modificaciones (opcionales). Por ejemplo, supongamos que a lo largo de un programa realizamos varias veces la operación n^2+1 donde n y el resultado son registros. Para acortar el código a escribir podríamos usar una macro como la siguiente:

```
.macro CuadM1 input, aux, output
    mul     aux, input, input
    add     output, aux, #1
.endm
```

Esta macro se llama *CuadM1* y tiene tres parámetros (input, aux y output). Si posteriormente usamos la macro de la siguiente forma:

```
        CuadM1  r1, r8, r0
```

el ensamblador se encargará de expandir la macro, es decir, en lugar de la macro coloca:

```
        mul     r8, r1, r1
        add     r0, r8, #1
```

No hay que confundir las macros con los procedimientos. Por un lado, el código de un procedimiento es único, todas las llamadas usan el mismo, mientras que el de una macro aparece (se expande) cada vez que se referencia, por lo que ocuparán más memoria. Las macros serán más rápidas en su ejecución, pues es secuencial, frente a los procedimientos, ya que implican un salto cuando aparece la llamada y un retorno cuando se termina. La decisión de usar una macro o un procedimiento dependerá de cada situación en concreto, aunque las macros son muy flexibles (ofrecen muchísimas más posibilidades de las comentadas aquí). Esta posibilidad será explotada en sesiones más avanzadas.

1.1.6. Ensamblar y *linkar* un programa

La traducción o ensamblado de un módulo fuente (**nombreprograma.s**) se realiza con el programa Gnu Assembler, con el siguiente comando:

```
as -o nombreprograma.o nombreprograma.s
```

NOTA: tanto el comando **as** como el nombre del programa son sensibles a las mayúsculas. Por tanto el comando debe ir en minúsculas y el nombre como queramos, pero recomendamos minúsculas también. Las opción **-o nombreprograma.o** puede ir después de **nombreprograma.s**.

El **as** genera un fichero nombreprograma.o.

Para montar (*linkar*) hay que hacer:

```
gcc -o nombreprograma nombreprograma.o
```

NOTA: Nuevamente, tanto **gcc** como el nombre del programa deben estar en minúsculas. Este comando es muy parecido al anterior, podemos poner si queremos **-o nombreprograma** detrás de **nombreprograma.o**. La única diferencia es que el archivo no tiene extensión, que por otro lado es una práctica muy recomendable para ejecutables en Linux.

Una vez hecho ésto, ya tenemos un fichero ejecutable (nombreprograma) que podemos ejecutar o depurar con el **gdb**.

1.2. Enunciados de la práctica

1.2.1. Cómo empezar

Recuerda que en laboratorio las raspberries no tienen monitor ni teclado, la única conexión con el mundo real es el puerto Ethernet. En el apéndice C se explica otro mecanismo para conectar.

Así que antes de nada averigua cuál es la dirección IP de la Raspberry Pi dentro de la red local. Por defecto el usuario es `pi` y la contraseña `raspberry`. Suponiendo que la dirección IP asignada es `192.168.1.42`, utilizaremos ssh:

```
ssh pi@192.168.1.42
```

Y luego introduce la contraseña. Ya conectado a la Raspberry Pi desde un PC a través de ssh. Todos los alumnos se conectarán con el mismo nombre de usuario, por lo que hay que dejar limpio el directorio de trabajo antes de terminar la sesión.

También es buena idea conectarte por ssh a la Raspberry que tengas en casa como acabamos de explicar, así te ahorras el tener que disponer de teclado y monitor extra. Desde Windows puedes bajarte putty.exe en http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html (no requiere instalación) y crear un acceso directo en el escritorio con el siguiente destino, cambiando **ruta** y **192.168.1.42** por lo que corresponda.

```
C:\ruta\putty.exe 192.168.1.42 -l pi -pw raspberry
```

Comenzaremos con el programa que hemos visto en el listado 1.1, y que se encuentra en el fichero `intro1.s`. Edítalo con el programa **nano** para verlo (y practicar un poco):

```
nano intro1.s
```

Dentro del editor tenemos una pequeña guía de comandos en la parte inferior. También podemos acceder a una ayuda más detallada pulsando **F1** dentro del editor. Estos son los atajos de teclado más comunes:

Atajo	**Función**
Ctrl-x	Salir de nano, se pide confirmación con Y/N
Ctrl-o	Salvar cambios
Ctrl-c	Muestra panel con información sobre número de línea
Alt-g	Saltar a un número de línea en concreto
Alt-a ó Ctrl-6	Seleccionar texto, mover cursores para definir región
Alt-^	Copiar selección
Ctrl-k	Cortar selección
Ctrl-u	Pegar selección
Ctrl-w	Buscar texto
Alt-w	Repetir última búsqueda
Alt-r	Buscar y reemplazar

Tabla 1.2: Lista de atajos de teclado para editor nano

Una vez que estéis familiarizados con el **nano** podemos pasar al traductor:

```
as -o intro1.o intro1.s
```

Observa que cuando se traduce, aparece una lista de errores, o bien, una indicación de que el programa se ha traducido correctamente. No se puede pasar a la etapa de montaje hasta que no se han solucionado los errores de sintaxis.

```
gcc -o intro1 intro1.o
```

De la misma forma, el montador nos informa de si todo el proceso ha sido correcto.

Una vez que hemos terminado con éxito este proceso podemos pasar el programa al **gdb** (GNU Debugger).

```
gdb intro1
```

El **gdb** ofrece muchas posibilidades, sólo explicaremos las que vamos a utilizar. Nada más ejecutar el comando anterior, el **gdb** se encuentra en modo interactivo:

```
pi@raspberrypi ~ $ gdb intro1
GNU gdb (GDB) 7.4.1-debian
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
This is free software: you are free to change and redis...
```

```
There is NO WARRANTY, to the extent permitted by law.  ...
and "show warranty" for details.
This GDB was configured as "arm-linux-gnueabihf".
For bug reporting instructions, please see:
...
Reading symbols from /home/pi/intro1...(no debugging sy...
(gdb)
```

Podemos escribir **help** para acceder a la ayuda integrada, o bien irnos a la página web de documentación del **gdb** [7]. El primer comando a aprender es:

```
(gdb) quit
```

Lanzamos de nuevo el depurador **gdb intro1**. En este momento no hay nada ejecutándose. Un primer paso es decirle al depurador que queremos lanzar el programa, esto es, cargarlo en memoria y apuntar a la primera instrucción del mismo:

```
(gdb) start
Temporary breakpoint 1 at 0x8390
Starting program: /home/pi/intro1

Temporary breakpoint 1, 0x00008390 in main ()
```

Perfecto, nos hemos saltado todos los pasos de inicialización de la librería C y estamos a punto de ejecutar la primera instrucción de nuestra función main. Veamos qué hay allí:

```
(gdb) disassemble
Dump of assembler code for function main:
=> 0x00008390: ldr r1, [pc, #24];  0x83b0 <puntero_var1>
   0x00008394: ldr r1, [r1]
   0x00008398: ldr r2, [pc, #20];  0x83b4 <puntero_var2>
   0x0000839c: ldr r2, [r2]
   0x000083a0: ldr r3, [pc, #16];  0x83b8 <puntero_var3>
   0x000083a4: add r0, r1, r2
   0x000083a8: str r0, [r3]
   0x000083ac: bx  lr
End of assembler dump.
```

Vemos que las instrucciones que hacían referencia a **puntero_varX** han cambiado. De momento lo ignoramos, ya lo explicaremos más adelante. Observen que hay una especie de flecha => apuntando a la instrucción que está apunto de ejecutarse (no lo ha hecho aún). Antes de ejecutarla, veamos los valores de algunos registros:

```
(gdb) info registers r0 r1 r2 r3
r0             0x1              1
r1             0xbefffe04       3204447748
r2             0xbefffe0c       3204447756
r3             0x8390           33680
```

Podemos modificar el valor de los registros por medio de la orden **print**, teniendo en cuenta los efectos adversos que esto podría ocasionar, ya que estamos alterando el funcionamiento de nuestro programa. En este caso no pasa nada, puesto que aún no hemos ejecutado ninguna instrucción.

```
(gdb) print $r0 = 2
$1 = 2
(gdb) info registers r0 r1 r2 r3
r0             0x2              2
r1             0xbefffe04       3204447748
r2             0xbefffe0c       3204447756
r3             0x8390           33680
```

gdb muestra $1, que es el identificador asignado al resultado. Podemos usar dicho identificador en nuevas expresiones y así ahorramos tiempo al teclear. En este ejemplo no es muy útil, pero lo será si la expresión es más compleja.

```
(gdb) print $1
$2 = 2
```

Ahora podemos usar $2, y así sucesivamente. Bueno, ya es hora de ejecutar la primera instrucción.

```
(gdb) stepi
0x00008394 in main ()
```

Veamos qué ha pasado desensamblando de nuevo.

```
(gdb) disassemble
Dump of assembler code for function main:
   0x00008390: ldr r1, [pc, #24];  0x83b0 <puntero_var1>
=> 0x00008394: ldr r1, [r1]
   0x00008398: ldr r2, [pc, #20];  0x83b4 <puntero_var2>
   0x0000839c: ldr r2, [r2]
   0x000083a0: ldr r3, [pc, #16];  0x83b8 <puntero_var3>
   0x000083a4: add r0, r1, r2
   0x000083a8: str r0, [r3]
   0x000083ac: bx  lr
End of assembler dump.
```

Observamos que la flecha => ha cambiado de posición, apuntando ahora a la segunda instrucción. Veamos qué le ha pasado a **r1**.

```
(gdb) info register r1
r1             0x10558  66904
```

Bien, ha cambiado. De hecho esta es la dirección de **puntero_var1**. Comprobémoslo usando su nombre simbólico con la sintaxis de C.

```
(gdb) print &var1
$3 = ( *) 0x10558 <puntero_var1>
```

Genial. Ahora veamos el contenido de dicha variable.

```
(gdb) print var1
$4 = 3
```

Perfecto, es lo que esperábamos. Veamos el siguiente paso.

```
(gdb) stepi
0x00008398 in main ()
(gdb) disas
Dump of assembler code for function main:
   0x00008390: ldr r1, [pc, #24];  0x83b0 <puntero_var1>
   0x00008394: ldr r1, [r1]
=> 0x00008398: ldr r2, [pc, #20];  0x83b4 <puntero_var2>
   0x0000839c: ldr r2, [r2]
   0x000083a0: ldr r3, [pc, #16];  0x83b8 <puntero_var3>
   0x000083a4: add r0, r1, r2
   0x000083a8: str r0, [r3]
   0x000083ac: bx  lr
End of assembler dump.
```

Puedes emplear **disas** (pero no **disa**) como comando abreviado. En realidad todos los comandos pueden abreviarse, sólo que no lo hemos hecho para os resulte más fácil su memorización. A partir de ahora pondré versiones abreviadas de comandos que ya hayamos mostrado.

```
(gdb) i r r1
r1             0x3  3
```

Vamos bien. Ahora ejecutamos hasta la instrucción **str**, que serían exactamente 4 pasos.

```
(gdb) si 4
0x000083a8 in main ()
(gdb) disas
```

```
Dump of assembler code for function main:
   0x00008390: ldr r1, [pc, #24];  0x83b0 <puntero_var1>
   0x00008394: ldr r1, [r1]
   0x00008398: ldr r2, [pc, #20];  0x83b4 <puntero_var2>
   0x0000839c: ldr r2, [r2]
   0x000083a0: ldr r3, [pc, #16];  0x83b8 <puntero_var3>
   0x000083a4: add r0, r1, r2
=> 0x000083a8: str r0, [r3]
   0x000083ac: bx  lr
End of assembler dump.
```

Comprobamos ahora que la intrucción **str** funciona correctamente, inspeccionando la variable **var3** antes y después.

```
(gdb) p var3
$5 = 4660
(gdb) si
0x000083ac in main ()
(gdb) p var3
$6 = 7
```

Ahora ejecutemos hasta el final.

```
(gdb) continue
Continuing.
[Inferior 1 (process 2477) exited with code 07]
```

El depurador nos indica que el código de salida es 07. Este código se lo indicamos en el registro **r0** justo antes de salir del **main**. Nos salimos del depurador y comprobamos que ejecutando el programa directamente, aunque éste no muestre ninguna salida por pantalla, podemos verificar su código de salida de la siguiente forma:

```
pi@raspberrypi ~ $ ./intro1 ; echo $?
7
pi@raspberrypi ~ $
```

Ahora que ya tenemos una idea de las posibilidades del **gdb** vamos a repasar unos cuantos conceptos con la ayuda de este programa.

1.2.2. Enteros y naturales

Recordemos que cuando se representa cualquier dato en memoria, éste tiene un valor explícito (el que tiene como dato guardado en binario) y un valor implícito (el que tiene interpretado como un tipo de dato determinado o como una instrucción). En este apartado queremos que veais la diferencia entre el valor explícito y el valor implícito interpretado como un natural y como un entero.

Ejercicio 1.1

Suponemos dos variables de longitud un byte `var1` y `var2` con los valores binarios (00110010_b) y (11000000_b), respectivamente. Completa las casillas en blanco.

	Valor explícito (binario)	Valor explícito (hexadecimal)	Valor implícito (como un natural en decimal)	Valor implícito (como un entero en decimal)
var1	00110010			
var2	11000000			

Observa que los valores son bien diferentes según la interpretación (valor implícito) que se les dé.

Ejercicio 1.2

Calcula ahora la suma de los dos números y responde en las casillas en blanco.

Valor explícito (binario)	Valor explícito (hexadecimal)	Valor implícito (como un natural en decimal)	Valor implícito (como un entero en decimal)
00110010 +11000000	=	=	=

¿Cuál es el valor final de los flags? N= Z= C= V=

¿Es el resultado final correcto interpretado como un natural?

¿Es el resultado final correcto interpretado como un entero?

Ahora es el momento de comprobar si hemos contestado correctamente. El código de esta parte se encuentra en el fichero `intro2.s` (listado 1.2).

Listado 1.2: Código del programa intro2.s

```
.data

var1:   .byte   0b00110010
        .align
var2:   .byte   0b11000000
        .align
```

```
.text
.global main

main:   ldr     r1, =var1       /* r1 <- &var1     */
        ldrsb   r1, [r1]        /* r1 <- *r1       */
        ldr     r2, =var2       /* r2 <- &var2     */
        ldrsb   r2, [r2]        /* r2 <- *r2       */
        add     r0, r1, r2      /* r0 <- r1 + r2   */
        bx      lr
```

Si os fijáis hemos hecho algunos cambios con respecto a `intro1.s`. Las variables son de tipo byte en lugar de word, lo que nos obliga a alinear con **.align** después de cada una. Las cargamos con la instrucción **ldrsb**, indicando que lo que cargamos es un byte **b** al que le extendemos su signo **s**. Hemos eliminado la variable **var3**, al fin y al cabo vamos a obtener el resultado en el registro **r0**. Por último hemos simplificado la carga de la dirección de la variables con **ldr r1, =var1**, de esta forma el ensamblador se encarga de declarar los punteros automáticamente.

Ensámblalo, móntalo y síguelo con el **gdb** tal y como se ha explicado en la primera parte de la práctica. Ejecuta sólo las 5 primeras instrucciones. Analiza el resultado del registro r0 y responde al siguiente ejercicio.

Ejercicio 1.3

Si interpretamos el resultado como byte

binario	hexa

Si interpretamos el resultado como palabra (32 bits)

binario	hexa

Ídem, pero si no hubiésemos extendido los signos (ldrb en lugar de ldrsb)

binario	hexa

Ejercicio 1.4

Repite el ejercicio anterior, pero ahora comprobando el resultado de los flags con lo que habías calculado en el Ejercicio 1.2. ¿Qué ocurre?

Si observáis, el registro cpsr no cambia, es el mismo antes y después de ejecutar la instrucción **add**.

```
(gdb) i r cpsr
cpsr           0x60000010     1610612752
```

Por cierto, desde **gdb** no hay una forma sencilla de obtener los flags por separado. Por suerte son fáciles de interpretar a partir del valor hexadecimal de **cpsr**. Convertimos a binario el nibble (dígito hexadecimal) más significativo de cpsr, en este caso 6 ->0110. Hacemos corresponder 0110 con la secuencia **NZCV** (debemos aprenderla de memoria), con lo cual tendríamos N=0, Z=1, C=1 y V=0.

La razón por la que no se actualizan los flags es que el ensamblador del ARM no lo hace a menos que se lo indiquemos con una **s** detrás de la instrucción. Cambiemos la línea 15 del archivo `intro2.s` por ésta.

```
        adds    r0, r1, r2    /* r0 <- r1 + r2  */
```

Y repetimos todos los pasos: ensamblado, enlazado y depuración. Ahora sí, comprueba que los flags se corresponden con los valores calculados.

1.2.3. Instrucciones lógicas

Ejercicio 1.5

Supón que tienes dos variables de tamaño 1 byte, `var1` y `var2`, con los valores 11110000_b y 10101010_b. Calcula el resultado de hacer una operación AND y una operación OR entre las dos variables.

Variable	Valor (binario)		Valor (hexadecimal)	Valor (binario)	
var1	11110000			11110000	
var2	10101010			10101010	
var1 AND var2			var1 OR var2		
	binario	hexa		binario	hexa

Para comprobar el resultado tenemos el programa `intro3.s`.

Listado 1.3: Código del programa intro3.s

```
.text
.global main

main:   mov     r2, #0b11110000   /* r2 <- 11110000   */
```

```
        mov     r3, #0b10101010     /* r3 <- 10101010    */
        and     r0, r2, r3          /* r0 <- r2 AND r3   */
        orr     r1, r2, r3          /* r1 <- r2 OR  r3   */
        mvn     r4, r0              /* r4 <- NOT r0      */
        mov     r0, #0x80000000
        tst     r0, #0x80000000
        tst     r0, #0x40000000
        bx      lr
```

Ejecuta las 4 primeras instrucciones y comprueba tus respuestas.

Ejercicio 1.6

El resultado de la instrucción `and` está en r0. ¿Cuál será el resultado de hacer un complemento a uno del mismo?

	binario	hexa
r0		
	binario	hexa
∼r0		

Ejecuta con el **gdb** la instrucción `mvn r4, r0` y comprueba tu respuesta.

Ejercicio 1.7

La instrucción `tst` hace la operación `and` entre un registro y una máscara y sólo actúa sobre los flags. Cumplimenta las casillas en blanco, teniendo en cuenta que el flag Z se pone a uno cuando el resultado de la `and` es cero, y se pone a cero en caso contrario. Para simplificar indicamos sólo los 16 bits menos significativos del registro r0.

	binario	hexa
r0	10000000000000000000000000000000	80000000
	`tst r0, #0x80000000`	¿Z?
	`tst r0, #0x40000000`	¿Z?

Comprueba tus respuestas con ayuda del **gdb**, y examina el resto de flags, observa qué ocurre con el flag N (flag de signo).

1.2.4. Rotaciones y desplazamientos

En este apartado veremos el funcionamiento de las instrucciones de desplamiento y rotación. Las instrucciones de desplazamiento pueden ser lógicas o aritméticas.

Los desplazamientos lógicos desplazan los bit del registro fuente introduciendo ceros (uno o más de uno). El último bit que sale del registro fuente se almacena en el flag C (figura 1.4). El desplazamiento aritmético hace lo mismo, pero manteniendo el signo (figura 1.5).

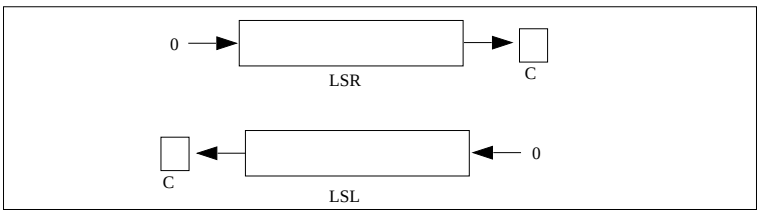

Figura 1.4: Instrucciones de desplazamiento lógico

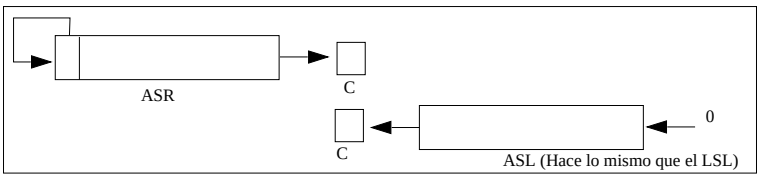

Figura 1.5: Instrucciones de desplazamiento aritmético

Las instrucciones de rotación también desplazan, pero el bit que sale del valor se realimenta. No existe ninguna instrucción para rotar hacia la izquierda `ROL`, ya que puede simularse con la de rotación a la derecha `ROR` que sí existe. En estas instrucciones el bit desplazado fuera es el mismo que el que entra, además de dejar una copia en el flag C (figura 1.6).

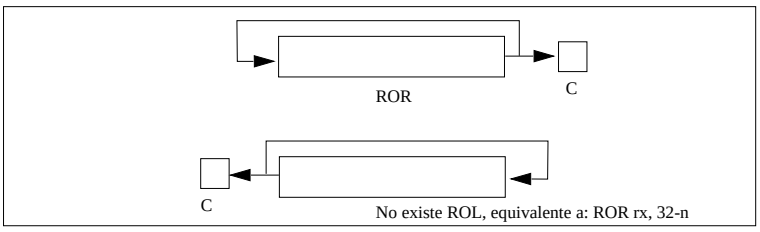

Figura 1.6: Instrucciones de rotación

Las instrucciones de rotación con el carry funcionan de manera similar, pero el bit que entra es el que había en el flag C y el que sale va a parar al flag C

(figura 1.7). Estas instrucciones sólo rotan un bit, al contrario que las anteriores que podían rotar/desplazar varios. La rotación con carry a la derecha es `RRX`, no existe la contrapartida `RLX` porque se puede sintetizar con otra instrucción ya existente `adcs`. Con `adcs` podemos sumar un registro consigo mismo, que es lo mismo que multiplicar por 2 o desplazar 1 bit hacia la izquierda. Si a esto le añadimos el bit de carry como entrada y actualizamos los flags a la salida, tendremos exactamente el mismo comportamiento que tendría la instrucción `RLX`.

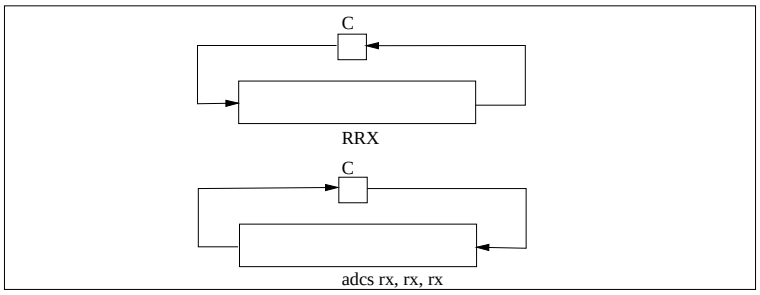

Figura 1.7: Instrucciones de rotación con *carry*

También podemos forzar el flag C o cualquier otro flag al valor que queramos con la siguiente instrucción.

```
        msr     cpsr_f, #valor
```

Donde para calcular el valor hacemos el paso inverso al explicado en **gdb**. Queremos cambiar los flags a estos valores: N=0, Z=1, C=1 y V=0. Por el orden memorizado de la secuencia **NZCV**, calculamos el nibble binario, que es 0110. Lo pasamos a hexadecimal 0110 ->6 y lo ponemos en la parte más alta de la constante de 32 bits, dejando el resto a cero.

```
        msr     cpsr_f, #0x60000000
```

Todas las instrucciones de rotación en realidad son subinstrucciones que el ensamblador traduce a una instrucción **mov**. Por esa razón las he puesto en mayúsculas, diferenciándolas de las instrucciones reales que están en minúscula. En realidad las dos siguientes instrucciones son totalmente equivalentes.

```
        LSRs    r0, r0, #1
        movs    r0, r0, LSR #1
```

Pero se tiende a escoger siempre la más sencilla, en este caso la primera. En próximas lecciones mostraremos la potencia que tienen las subistrucciones de desplazamiento/rotación (cuando están en mayúscula mezcladas con los operandos). Como adelanto, la siguiente instrucción multiplica por 5 el contenido de r0.

```
        add     r0, r0, r0, LSL #2
```

Ejercicio 1.8

Examina atentamente el programa `intro4.s` (listado 1.4). Antes de ejecutarlo completa el siguiente cuadro, después comprueba los resultados con el **gdb**. Observa la definición de variable `var1: .word 0x80000000`.

Instrucción	r1 (binario)	C
ldr r1, [r0]		
LSRs r1, r1, #1		
LSRs r1, r1, #3		

Instrucción	r2 (binario)	C
ldr r2, [r0]		
ASRs r2, r2, #1		
ASRs r2, r2, #3		

Instrucción	r3 (binario)	C
ldr r3, [r0]		
RORs r3, r3, #31		
RORs r3, r3, #31		
RORs r3, r3, #24		

Instrucción	r4 (binario)	C
ldr r4, [r0]		
msr cpsr_f, #0		
adcs r4, r4, r4		
adcs r4, r4, r4		
adcs r4, r4, r4		
msr cpsr_f, #0x2..		
adcs r4, r4, r4		

Listado 1.4: Código del programa intro4.s

```
.data

var1:    .word    0x80000000

.text
.global main
```

```
main:   ldr     r0, =var1       /* r0 <- &var1          */
        ldr     r1, [r0]        /* r1 <- *r0            */
        LSRs    r1, r1, #1      /* r1 <- r1 LSR #1      */
        LSRs    r1, r1, #3      /* r1 <- r1 LSR #3      */
        ldr     r2, [r0]        /* r2 <- *r0            */
        ASRs    r2, r2, #1      /* r2 <- r2 ASR #1      */
        ASRs    r2, r2, #3      /* r2 <- r2 ASR #3      */
        ldr     r3, [r0]        /* r3 <- *r0            */
        RORs    r3, r3, #31     /* r3 <- r3 ROL #1      */
        RORs    r3, r3, #31     /* r3 <- r3 ROL #1      */
        RORs    r3, r3, #24     /* r3 <- r3 ROL #8      */
        ldr     r4, [r0]        /* r4 <- *r0            */
        msr     cpsr_f, #0      /* C=0                  */
        adcs    r4, r4, r4      /* rotar izda carry     */
        adcs    r4, r4, r4      /* rotar izda carry     */
        adcs    r4, r4, r4      /* rotar izda carry     */
        msr     cpsr_f, #0x20000000 /* C=1              */
        adcs    r4, r4, r4      /* rotar izda carry     */
        bx      lr
```

1.2.5. Instrucciones de multiplicación

Las instrucciones de multiplicación admiten muchas posibilidades, debido a que es una operación en la cual el resultado tiene el doble de bits que cada operando. En la siguiente tabla vemos las 5 instrucciones de multiplicación que existen.

Instrucción	**Bits**	**Nombre**
mul	32=32x32	Multiplicación truncada
umull	64=32x32	Multiplicación sin signo de 32bits
smull	64=32x32	Multiplicación con signo de 32bits
smulw*	32=32x16	Multiplicación con signo de 32x16bits
smul**	32=16x16	Multiplicación con signo de 16x16bits

Tabla 1.3: Instrucciones de multiplicación

La instrucción `mul` realiza una multiplicación truncada, es decir, nos quedamos con los 32 bits inferiores del resultado. Como el signo del resultado es el bit más significativo el cual no obtenemos, esta multiplicación es válida tanto para operandos naturales (sin signo) como para enteros (con signo). En el ejemplo de abajo `r0 = parte_baja(r1*r2)`:

```
        mul     r0, r1, r2
```

Las dos siguientes multiplicaciones (`umull` y `smull`) son largas, por eso la `l` del final, donde el resultado es de 64 bits. Si los operandos son naturales escogemos la multiplicación sin signo (unsigned) `umull`. Por el contrario, si tenemos dos enteros como factores hablamos de multiplicación con signo (signed) `smull`. En ambos ejemplos la parte baja del resultado se almacena en r0, y la parte alta en r1. Para hacer que `r1:r0 = r2*r3`:

```
        umull   r0, r1, r2, r3
        smull   r0, r1, r2, r3
```

Ahora veamos `smulw*`. Es con signo, y el asterisco puede ser una **b** para seleccionar la parte baja del registro del segundo factor, o una **t** para seleccionar la alta. Según el ejemplo `r0 = r1*parte_baja(r2)`.

```
        smulwb  r0, r1, r2
```

Por último tenemos `smul**` también con signo, donde se seleccionan partes alta o baja en los dos factores, puesto que ambos son de 16 bits. En el ejemplo `r0 = parte_alta(r1)*parte_baja(r2)`.

```
        smultb  r0, r1, r2
```

En los dos últimos tipos `smulw*` y `smul**` no se permite el sufijo **s** para actualizar los flags.

Ejercicio 1.9

Completa los recuadros en blanco con los resultados en hexadecimal empleando calculadora. Luego ensambla el listado 1.5 y comprueba mediante **gdb** que los cálculos anteriores son correctos.

	Producto	Factor1	Factor2
mul			
umull			
smull			
smuwb			
smutt			

Listado 1.5: Código del programa intro5.s

```
.data
var1:   .word   0x12345678
var2:   .word   0x87654321
```

```
var3:   .word   0x00012345

.text
.global main
main:   ldr     r0, =var1           /* r0 <- &var1          */
        ldr     r1, =var2           /* r1 <- &var2          */
        ldr     r2, =var3           /* r2 <- &var3          */
        ldrh    r3, [r0]            /* r3 <- baja(*r0)      */
        ldrh    r4, [r1]            /* r4 <- baja(*r1)      */
        muls    r5, r3, r4          /* r5 <- r3*r4          */
        ldr     r3, [r0]            /* r3 <- *r0            */
        ldr     r4, [r1]            /* r4 <- *r1            */
        umull   r5, r6, r3, r4      /* r6:r5 <- r3*r4       */
        smull   r5, r6, r3, r4      /* r6:r5 <- r3*r4       */
        ldrh    r3, [r0]            /* r3 <- baja(*r0)      */
        ldr     r4, [r2]            /* r4 <- *r2            */
        smulwb  r5, r3, r4          /* r5 <- r3*baja(r4)    */
        smultt  r5, r3, r4      /* r5 <- alta(r3)*alta(r4)*/
```

Capítulo 2

Tipos de datos y sentencias de alto nivel

Contenido

Objetivo: En esta sesión repasaremos cómo se representa la información en la memoria del computador: veremos la definición en ensamblador de punteros, vectores y matrices. También veremos cómo se programan las estructuras de alto nivel del tipo `if-else` y los bucles `for` y `while`.

2.1. Lectura previa

2.1.1. Modos de direccionamiento del ARM

En la arquitectura ARM los accesos a memoria se hacen mediante instrucciones específicas `ldr` y `str` (luego veremos las variantes `ldm`, `stm` y las preprocesadas `push`

y `pop`). El resto de instrucciones toman operandos desde registros o valores inmediatos, sin excepciones. En este caso la arquitectura nos fuerza a que trabajemos de un modo determinado: primero cargamos los registros desde memoria, luego procesamos el valor de estos registros con el amplio abanico de instrucciones del ARM, para finalmente volcar los resultados desde registros a memoria. Existen otras arquitecturas como la Intel x86, donde las instrucciones de procesado nos permiten leer o escribir directamente de memoria. Ningún método es mejor que otro, todo es cuestión de diseño. Normalmente se opta por direccionamiento a memoria en instrucciones de procesado en arquitecturas con un número reducido de registros, donde se emplea la memoria como almacén temporal. En nuestro caso disponemos de suficientes registros, por lo que podemos hacer el procesamiento sin necesidad de interactuar con la memoria, lo que por otro lado también es más rápido.

Direccionamiento inmediato. El operando fuente es una constante, formando parte de la instrucción.

```
mov     r0, #1
add     r2, r3, #4
```

Direccionamiento inmediato con desplazamiento o rotación. Es una variante del anterior en la cual se permiten operaciones intermedias sobre los registros.

```
mov     r1, r2, LSL #1      /* r1 <- (r2*2) */
mov     r1, r2, LSL #2      /* r1 <- (r2*4) */
mov     r1, r3, ASR #3      /* r1 <- (r3/8) */
```

Estas instrucciones también se usan implicitamente para la creación de constantes, rotando o desplazando constantes más pequeñas de forma transparente al usuario. Como todas las instrucciones ocupan 32 bits, es técnicamente imposible que podamos cargar en un registro cualquier constante de 32 bits con la instrucción `mov`. Por esta razón cuando se necesita cargar una constante más compleja en un registro (como una dirección a una variable de memoria) no podemos hacerlo con la instrucción `mov`, tenemos que recurrir a `ldr` con direccionamiento a memoria.

Un método para determinar si una constante entra o no en una instrucción `mov` es pasar la constante a binario y quitar los ceros de la izquierda y de la derecha y contar el número de bits resultante. Si el número de bits es menor o igual que 8, la constante entra en una instrucción `mov`. Por ejemplo la constante `0x00354000` al pasarla a binario sería `00000000001101010100000000000000`. Eliminando los ceros de delante y detrás tenemos `11010101`, que son 8 bits y por tanto cabe en un `mov`.

Este método tiene excepciones. Una de ellas está en los números negativos, que en lugar de quitar ceros a izquierda y derecha quitamos unos. Por ejemplo

la constante `0xFFFBFBFF` en binario es `11111111111101111111011111111111` y quitando los unos a ambos lados queda `011111110`, que son 9 bits y por tanto este caso requiere un `ldr`.

La otra excepción está en el hecho de que las constantes de 32 bits no sólo se crean desplazando constantes de 8, el ensamblador también puede recurrir a rotaciones circulares para crearlas. En casos complejos como éste a veces es más práctico probar con `mov` y si no salta ningún error en el ensamblado es porque cabe con `mov`. Por ejemplo:

```
    mov     r1, #0x80000020
```

Ensamblamos y vemos que no da problemas. Sin embargo con esta otra.

```
    mov     r1, #0x80000040
```

El ensamblador nos muestra el siguiente error.

```
ejem.s: Assembler messages:
ejem.s:10: Error: invalid constant (80000040) after fixup
```

Direccionamiento a memoria, sin actualizar registro puntero. Es la forma más sencilla y admite 4 variantes. Después del acceso a memoria ningún registro implicado en el cálculo de la dirección se modifica.

- `[Rx, #+inmediato]`
  `[Rx, #-inmediato]`
  Simplemente añade (o sustrae) un valor inmediato al registro dado para calcular la dirección. Es muy útil para acceder a elementos fijos de un array, ya que el desplazamiento es constante. Por ejemplo si tenemos `r1` apuntando a un array de enteros de 32 bits `int a[]` y queremos poner a 1 el elemento `a[3]`, lo hacemos así:

  ```
      mov     r2, #1          /* r2 <- 1           */
      str     r2, [r1, #+12]  /* *(r1 + 12) <- r2 */
  ```

  Nótese que hemos multiplicado por 4 el desplazamiento porque cada elemento del array son 4 bytes. El desplazamiento no puede ser mayor de 12 bits, por lo que nuestro rango está límitado entre `[Rx, #-4095]` y `[Rx, #+4095]`.

- `[Rx, +Ry]`
  `[Rx, -Ry]`

  Parecido al anterior pero en lugar de un inmediato emplea otro registro. Útil en el caso de queramos mantener fijo el registro `Rx` y movernos con

`Ry`, o bien para acceder a desplazamientos mayores a 4095. El mismo ejemplo de arriba utilizando esta variante sería:

```
    mov     r2, #1          /* r2 <- 1          */
    mov     r3, #12         /* r3 <- 12         */
    str     r2, [r1, +r3]   /* *(r1 + r3) <- r2 */
```

- `[Rx, +Ry, operación_desp #inmediato]`
  `[Rx, -Ry, operación_desp #inmediato]`

  En este caso aplicamos una operación de desplazamiento o rotación sobre el segundo registro `Ry`. Muy útil en caso de arrays o estructuras con elementos de longitud potencia de 2, ya que podemos indexar directamente. El mismo ejemplo de antes:

```
    mov     r2, #1
    mov     r3, #3
    str     r2, [r1, +r3, LSL #2]
```

  Nótese cómo accedemos a `a[3]` directamente con el valor del índice, `3`.

Direccionamiento a memoria, actualizando registro puntero. En este modo de direccionamiento, el registro que genera la dirección se actualiza con la propia dirección. De esta forma podemos recorrer un array con un sólo registro sin necesidad de hacer el incremento del puntero en una instrucción aparte. Hay dos métodos de actualizar dicho registro, antes de ejecutar la instrucción (preindexado) o después de la misma (postindexado). Los tres siguientes tipos son los postindexados.

- `[Rx], #+inmediato`
  `[Rx], #-inmediato`
  Una notación muy parecida a la versión que no actualiza registro, la única diferencia es que la constante de desplazamiento queda fuera de los corchetes. Presenta el mismo límite de hasta 4095. Este ejemplo pone a cero los 3 primeros elementos `a[0], a[1], a[2]` del array:

```
    mov     r2, #0          /* r2 <- 0      */
    str     r2, [r1], #+4   /* a[0] <- r2   */
    str     r2, [r1], #+4   /* a[1] <- r2   */
    str     r2, [r1], #+4   /* a[2] <- r2   */
```

- `[Rx], +Ry`
  `[Rx], -Ry`
  Igual que antes pero con registro en lugar de inmediato.

- `[Rx], +Ry, operación_desp #inmediato`
  `[Rx], -Ry, operación_desp #inmediato`

  Nótese que en todos los modos postindexados encerramos entre llaves el primer registro, que es el que se va a utilizar en la instrucción de lectura o escritura en memoria. Es decir primero cargamos de `[Rx]` y luego actualizamos `Rx` con el valor que corresponda. Esta instrucción:

```
    ldr     r2, [r1], +r3, LSL #2
```

  Se puede desglosar en estas otras dos, cuyo comportamiento es exactamente el mismo:

```
    ldr     r2, [r1]
    add     r1, r1, r3, LSL #2
```

Ya hemos visto la notación postindexada. Veamos ahora los tres modos preindexados.

- `[Rx, #+inmediato]!`
  `[Rx, #-inmediato]!`
  La idea en todos los casos es encerrar entre corchetes la dirección que se va a usar en la instrucción. Para diferenciarlo del caso que no actualiza el registro le añadimos un `!` al final.

  Este modo es muy útil en casos que queramos reusar en una futura instrucción la dirección que hemos calculado. En este ejemplo duplicamos el valor que se encuentra en `a[3]`:

```
    ldr     r2, [r1, #+12]!
    add     r2, r2, r2
    str     r2, [r1]
```

- `[Rx, +Ry]!`
  `[Rx, -Ry]!`

  Similar al anterior pero usando `Ry` en lugar de inmediato.

- `[Rx, +Ry, operación_desp #inmediato]!`
  `[Rx, -Ry, operación_desp #inmediato]!`

  Tercer y último caso de direccionamiento preindexado. Al igual que antes, desgloso en dos instrucciones para ver el funcionamiento exacto:

```
    ldr     r2, [r1, +r3, LSL #2]!
```

Equivale a esto.

```
    add     r1, r1, r3, LSL #2
    ldr     r2, [r1]
```

O bien a esto otro.

```
    ldr     r2, [r1, +r3, LSL #2]
    add     r1, r1, r3, LSL #2
```

2.1.2. Tipos de datos

Tipos de datos básicos. En la siguiente tabla se recogen los diferentes tipos de datos básicos que podrán aparecer en los ejemplos, así como su tamaño y rango de representación.

ARM	Tipo en C	bits	Rango
`.byte`	`unsigned char`	8	0 a 255
	(`signed`) `char`	8	-128 a 127
`.hword`	`unsigned short int`	16	0 a 65.535
`.short`	(`signed`) `short int`	16	-32.768 a 32767
`.word`	`unsigned int`	32	0 a 4294967296
`.int`	(`signed`) `int`	32	-2147483648 a 2147483647
	`unsigned long int`	32	0 a 4294967296
	(`signed`) `long int`	32	-2147483648 a 2147483647
`.quad`	`unsigned long long`	64	0 a 2^{64}
	(`signed`) `long long`	64	-2^{63} a 2^{63}-1

Nótese como en ensamblador los tipos son neutrales al signo, lo importante es la longitud en bits del tipo. La mayoría de las instrucciones (salvo multiplicación) hacen la misma operación tanto si se trata de un número natural como si es entero en complemento a dos. Nosotros decidiremos el tipo mediante las constantes que pongamos o según los flags que interpretemos del resultado de la operación.

Punteros. Un **puntero** siempre ocupa 32 bits y contiene una dirección de memoria. En ensamblador no tienen tanta utilidad como en C, ya que disponemos de registros de sobra y es más costoso acceder a las variables a través de los punteros que directamente. En este ejemplo acceder a la dirección de var1 nos cuesta 2 `ldrs` a través del puntero, mientras que directamente se puede hacer con uno sólo.

```
.data
var1:           .word   3
puntero_var1:   .word   var1

.text
.global main
main:   ldr     r0, =puntero_var1
        ldr     r1, [r0]
        ldr     r2, [r1]
        ldr     r3, =var1
        bx      lr
```

Observamos cómo el valor de `r3` es el mismo que el de `r1`.

```
(gdb) ni 4
0x000083a0 in main ()
(gdb) i r r0 r1 r2 r3
r0             0x1054c  66892
r1             0x10548  66888
r2             0x3      3
r3             0x10548  66888
```

Incluso en tipos que en C están basados en punteros como las cadenas, en ensamblador no es necesario tenerlos almacenados en memoria puesto que podemos obtener dicho valor en un registro con una única instrucción `ldr`.

Vectores. Todos los elementos de un vector se almacenan en un único bloque de memoria a partir de una dirección determinada. Los diferentes elementos se almacenan en posiciones consecutivas, de manera que el elemento `i` está entre los `i-1` e `i+1` (figura 2.1). Los vectores están definidos siempre a partir de la posición 0. El propio índice indica cuántos elementos hemos de desplazarnos respecto del comienzo del primer elemento (para acceder al elemento cero hemos de saltarnos 0 elementos, para acceder al elemento 1 hemos de saltarnos un elemento, etc...; En general, para acceder al elemento con índice `i` hemos de saltarnos los `i` elementos anteriores).

Dado un vector `int v[N];`, todos los elementos se encuentran en posiciones consecutivas a partir de la dirección de `v[0]` (puesto que son `int`, en este ejemplo, cada elemento ocupa 4 bytes). Por lo tanto, el acceso al elemento `v[i]` se consigue aplicando la siguiente expresión.

$$v[i] = M_d[@v[0] + i * 4] \tag{2.1}$$

Con $@v[0]$ nos referimos a la dirección en memoria del elemento $v[0]$. Con $M_d[\ ]$ notamos el acceso a memoria para la lectura/escritura de un dato (el número de

bytes de memoria implicado dependerá del tipo de datos declarado). Cuando nos queramos refererir al acceso a memoria para la obtención de un puntero, lo notaremos como $M_{ref}[\ ]$.

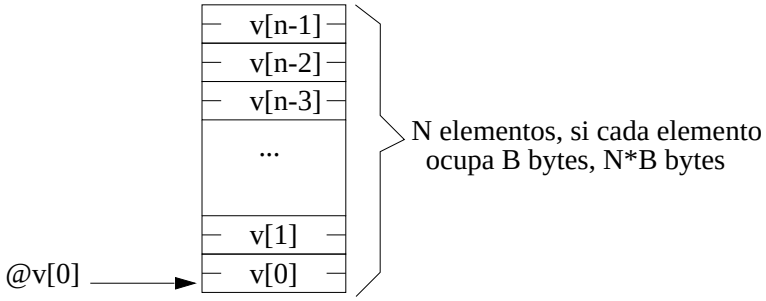

Figura 2.1: Representación de un vector en memoria

Matrices bidimensionales. Una matriz bidimensional de N×M elementos se almacena en un único bloque de memoria. Interpretaremos una matriz de N×M como una matriz con N filas de M elementos cada una. Si cada elemento de la matriz ocupa B bytes, la matriz ocupará un bloque de $M \times N \times B$ bytes (ver figura 2.2(a)).

Dentro de este bloque, los elementos se almacenan por filas. Primero se guardan todos los elementos de la fila 0, después todos los de la fila 1, etc. como se ve en la figura 2.2(b).

Por lo tanto, para acceder al elemento `mat[i][j]` hemos de saltar i filas completas (de M elementos de B bytes) y después j elementos de B bytes (suponiendo una matriz de enteros, $B = 4$ bytes). Es decir, la fórmula para obtener el elemento `mat[i][j]` será:

$$mat[i][j] = M_d[@mat + ((i * M) + j) * B] \tag{2.2}$$

2.1.3. Instrucciones de salto

Las instrucciones de salto pueden producir saltos incondicionales (`b` y `bx`) o saltos condicionales. Cuando saltamos a una etiqueta empleamos `b`, mientras que si queremos saltar a un registro lo hacemos con `bx`. La variante de registro `bx` la solemos usar como instrucción de retorno de subrutina, raramente tiene otros usos.

En los saltos condicionales añadimos dos o tres letras a la (`b/bx`), mediante las cuales condicionamos si se salta o no dependiendo del estado de los flags. Estas condiciones se pueden añadir a cualquier otra instrucción, aunque la mayoría de las veces lo que nos interesa es controlar el flujo del programa y así ejecutar o no un grupo de instrucciones dependiendo del resultado de una operación (reflejado en los flags).

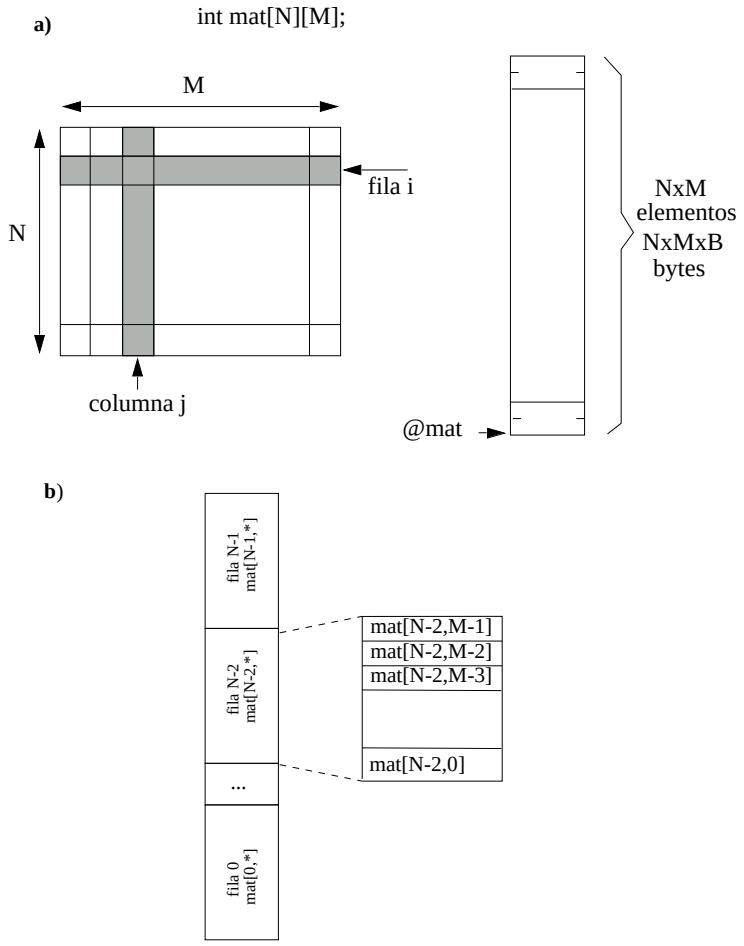

Figura 2.2: (a) Formato de una matriz C con N filas y M columnas y (b) organización por filas

La lista completa de condiciones es ésta:

- `EQ` (`e`qual, igual). Cuando `Z` está activo (`Z` vale 1).
- `NEQ` (`n`ot `e`qual, igual). Cuando `Z` está inactivo (`Z` vale 0).
- `MI` (`mi`nus, negativo). Cuando `N` está activo (`N` vale 1).
- `PL` (`pl`us, positivo o cero). Cuando `N` está inactivo (`N` vale 0).
- `CS/HS` (`c`arry `s`et/`h`igher or `s`ame, carry activo/mayor o igual). Cuando `C` está activo (`C` vale 1).

- `CC/LO` (carry clear/lower, carry inactivo/menor). Cuando `C` está inactivo (`C` vale 0).
- `VS` (overlow set, desbordamiento activo). Cuando `V` está activo (`V` vale 1).
- `VC` (overlow clear, desbordamiento inactivo). Cuando `V` está inactivo (`V` vale 0).
- `GT` (greater than, mayor en complemento a dos). Cuando `Z` está inactivo y `N=V` (`Z` vale 0, `N` vale `V`).
- `LT` (lower than, menor en complemento a dos). Cuando `N!=V` (`N` vale `not V`).
- `GE` (greater or equal, mayor o igual en complemento a dos). Cuando `N=V` (`N` vale `V`).
- `LE` (lower or equal, menor o igual en complemento a dos). Cuando `Z` está activo y `N!=V` (`Z` vale 1, `N` vale `not V`).
- `HI` (higher, mayor). Cuando `C` está activo y `Z` inactivo (`C` vale 1, `Z` vale 0).
- `LS` (lower or same, menor o igual). Cuando `C` está inactivo ó `Z` activo (`C` vale 0 ó `Z` vale 1).

Por ejemplo, la instrucción `beq destino_salto` producirá un salto a la instrucción indicada por la etiqueta `destino_salto` si y sólo si el bit de estado cero está activo (Z=1), y en caso contrario (Z=0) no interrumpirá el flujo secuencial de instrucciones. Previo a un salto condicional, el registro de flags debe ser actualizado mediante alguna instrucción aritmética (`adds`, `subs`, `cmp`, ...) o lógica (`ands`, `orrs`, `tst`, ...). En la mayoría de los casos tenemos que añadir el sufijo `s` a una instrucción normal `add`, para forzar que la nueva instrucción `adds` actualice los flags.

Un aspecto muy peculiar de la arquitectura ARM es que las llamadas a subrutinas se hacen mediante un sencillo añadido a la instrucción de salto. La instrucción `bl` (también `blx`) hace una llamada a una subrutina, mediante un salto a la subrutina y escribiendo en el registro `lr` la dirección de la siguiente instrucción.

```
main:   mov     r1, #1
        mov     r2, #2
        bl      subrut
        mov     r4, #4  /* Siguiente instrucción */
        ...

subrut: mov     r3, #3
        bx      lr
```

Si seguimos el flujo del programa primero cargamos `r1` a 1, luego `r2` a 2 y lo siguiente que hay es una llamada a subrutina. En dicha llamada el procesador carga en `lr` la dirección de la siguiente instrucción "`mov r4, #4`" y salta a la etiqueta `subrut`. Se ejecuta el "`mov r3, #3`" de la subrutina y después "`bx lr`" que vendría a ser la instrucción de retorno. Es decir, salimos de la subrutina retomando el flujo del programa principal, ejecutando "`mov r4, #4`".

Este sencillo esquema vale para un sólo nivel de subrutinas, es decir, dentro de `subrut` no podemos llamar a otra subrutina porque sobreescribimos el valor del registro `lr`. La solución para extender a cualquier número de niveles es almacenar el registro `lr` en pila con las instrucciones `push` y `pop`.

```
main:   mov     r1, #1
        mov     r2, #2
        bl      nivel1
        mov     r5, #5  /* Siguiente instrucción */
        ...

nivel1: push    {lr}
        mov     r3, #3
        bl      nivel2
        pop     {lr}
        bx      lr

nivel2: mov     r4, #4
        bx      lr
```

Como veis, en el último nivel (`nivel2`) podemos ahorrarnos el tener que almacenar y recuperar `lr` en la pila.

Las instrucciones de salto en la arquitectura ARM abarcan una zona muy extensa, hasta 64 Mb (32 Mb hacia adelante y otros 32 Mb hacia atrás). Estos límites podemos justificarlos atendiendo al formato de instrucción que podemos ver en el apéndice A. El código de operación ocupa 8 de los 32 bits, dejándonos 24 bits para codificar el destino del salto. En principio con 24 bits podemos direccionar 16 Mb $[-2^{23}-1, 2^{23}-1]$, sin embargo la arquitectura ARM se aprovecha del hecho de que las instrucciones están alineadas a direcciones múltiplo de 4 (en binario acaban en `00`), por lo que el rango real es de 64 Mb $[-2^{25}-1, 2^{25}-1]$

En caso de necesitar un salto mayor recurrimos a la misma solución de la carga de inmediatos del `mov`, solo que el registro a cargar es el `pc`.

```
        ldr pc, =etiqueta
```

2.1.4. Estructuras de control de alto nivel

En este punto veremos cómo se traducen a ensamblador las estructuras de control de alto nivel que definen un bucle (`for`, `while`, ...), así como las condicionales (`if-else`).

Las estructuras `for` y `while` se pueden ejecutar un mínimo de 0 iteraciones (si la primera vez no se cumple la condición). La traducción de las estructuras `for` y `while` se puede ver en los listados 2.1 y 2.2.

Para programar en ensamblador estas estructuras se utilizan instrucciones de salto condicional. Previo a la instrucción de salto es necesario evaluar la condición del bucle o de la sentencia `if`, mediante instrucciones aritméticas o lógicas, con el fin de actualizar los flags de estado. La traducción de la estructura `if` está en los listados 2.3 y 2.4.

Listado 2.1: Estructura del for y while en C (tipos1.c)

```
int vi, vf, i;

for ( i= vi; i<=vf; i++ ){
  /* Cuerpo del bucle */
}

i= vi;
while ( i<=vf ){
  /* Cuerpo del bucle */
  i++;
}
```

Listado 2.2: Traducción de las estructuras `for` y `while`. Hemos supuesto que el valor inicial está en la variable `vi` y el valor final en la variable `vf` y se ha utilizado el registro `r1` como índice de las iteraciones `i`.

```
        ldr     r1, =vi
        ldr     r1, [r1]
        ldr     r2, =vf
        ldr     r2, [r2]
bucle:  cmp     r1, r2
        bhi     salir
        /* Cuerpo
           del
           bucle  */
        add     r1, r1, #1
        b       bucle
salir:
```

Listado 2.3: Estructura if en C (tipos2.c)

```
  int a, b;

  if( a==b ){
    /* Código entonces */
  }
  else{
    /* Código sino */
  }
```

Listado 2.4: Traducción de la estructura `if`

```
        ldr     r1, =a
        ldr     r1, [r1]
        ldr     r2, =b
        ldr     r2, [r2]
        cmp     r1, r2
        bne     sino
entonces:
        /* Código entonces */
        b       final
sino:
        /* Código sino */
final:  ...
```

2.1.5. Compilación a ensamblador

Para acabar la teoría veamos cómo trabaja un compilador de C real. Normalmente los compiladores crean código compilado (archivos **.o**) en un único paso. En el caso de **gcc** este proceso se hace en dos fases: en una primera se pasa de C a ensamblador, y en una segunda de ensambladador a código compilado (código máquina). Lo interesante es que podemos interrumpir justo después de la compilación y ver con un editor el aspecto que tiene el código ensamblador generado a partir del código fuente en C.

Veámoslo con un ejemplo.

Listado 2.5: Código del programa tipos3.c

```
#include <stdio.h>

void main(void){

  int i;
```

```
  for ( i= 0; i<5; i++ ){
    printf("%d\n", i);
  }

}
```

Después de crear el fichero `tipos3.s`, lo compilamos con este comando.

```
        gcc -Os -S -o tipos3a.s tipos3.c
```

Con el parámetro `-S` forzamos la generación del `.s` en lugar del `.o` y con `-Os` le indicamos al compilador que queremos optimizar en tamaño, es decir que queremos código ensamblador lo más pequeño posible, sin importar el rendimiento del mismo.

El código ensamblador resultante está un poco sucio, lleno de directivas superfluas, con punteros a variables e instrucciones no simplificadas por el preprocesador. Tras limpiarlo quedaría así.

Listado 2.6: Código del programa tipos3a.s

```
.data
var1:   .asciz  "%d\012"

.text
.global main
main:   push    {r4, lr}
        mov     r4, #0
.L2:    mov     r1, r4
        ldr     r0, =var1
        add     r4, r4, #1
        bl      printf
        cmp     r4, #5
        bne     .L2
        pop     {r4, pc}
```

El carácter `\n` se ha transformado en octal `\012` puesto que el ensamblador no entiende de secuencias de escape. La instrucciones `push` y `pop` son la versión simple de `stmfd` y `ldmfd` que veremos más adelante. Nótese que la función no acaba con el típico `bx lr`. Se trata de una optimización que consigue reducir de dos instrucciones a una. Es decir, estas dos instrucciones:

```
        pop     {r4, lr}
        bx      lr
```

Se simplifican a:

```
        pop     {r4, pc}
```

En general no vamos a emplear este tipo de optimizaciones en las prácticas, puesto que dificultan la legibilidad del código.

El resto del código es sencillo de seguir. El registro `r4` hace la función del contador `i` del bucle, y la salida por pantalla se produce mediante una llamada a la función printf `bl printf`. Los parámetros se los pasamos a printf mediante `r0` y `r1` y son un puntero a la cadena a imprimir `%d\n` y el entero que le vamos a pasar. El porqué se usan estos registros para pasar parámetros (y el hecho de haber almacenado `r4` en pila) responde a la convención **AAPCS** que veremos con más detenimiento en el siguiente capítulo.

Veamos qué ocurre cuando le indicamos al compilador que queremos optimizar al máximo en velocidad (la escala va del 0 al 3) el mismo código en C:

```
        gcc -O3 -S -o tipos3b.s tipos3.c
```

Tras simplificar, el fichero en ensamblador generado sería éste:

Listado 2.7: Código del programa tipos3b.s

```
.data
var1:   .asciz  "%d\012"

.text
.global main
main:   push    {r4, lr}
        mov     r1, #0
        ldr     r4, =var1
        mov     r0, r4
        bl      printf
        mov     r0, r4
        mov     r1, #1
        bl      printf
        mov     r0, r4
        mov     r1, #2
        bl      printf
        mov     r0, r4
        mov     r1, #3
        bl      printf
        mov     r0, r4
        mov     r1, #4
        pop     {r4, lr}
        b       printf
```

Observamos que el bucle como tal ha desaparecido. En realidad lo que ha ocurrido es que el compilador ha empleado una técnica agresiva de optimización llamada *loop unrolling* o desenrollamiento de bucle, que consiste en sustituir mediante repeticiones del cuerpo del bucle, de tal forma que no perdemos tiempo comparando ni haciendo el salto condicional. En este caso empleamos tantas repeticiones como iteraciones tiene el bucle, aunque normalmente se llega hasta un límite de repeticiones. De no ser así el ejecutable se volvería excesivamente grande.

Por último señalar que de nuevo se ha optimizado el final de la función, aunque de otra forma distinta al caso anterior. La última iteración debería ser así:

```
        mov     r0, r4
        mov     r1, #4
        bl      printf
        pop     {r4, lr}
        bx      lr
```

Se deja como ejercicio explicar porqué "pop r4, lr" y "b printf" primero llama a `printf` y luego retorna al SO.

2.1.6. Ejercicios propuestos.

Ejercicio 2.1

Basándonos en los ejemplos anteriores, escribe un bucle `for` que imprima los 50 primeros números pares naturales en orden inverso (desde 100 hasta 2 en pasos de 2). Una vez hecho esto, aplica desenrollamiento de bucle de tal forma que el salto condicional se ejecute 10 veces, con 5 repeticiones cada vez.

Ejercicio 2.2

Escribe el código ensamblador correspondiente a una estructura `if` en la que no exista la rama de `else`.

Ejercicio 2.3

Escribe en ensamblador un código equivalente a éste. Primero haciendo uso de la instrucción `ands` y un registro auxiliar, luego simplifica con la instrucción `tst`.

```
for ( i= 0; i<10; i++ ){
  if( i&1 )
    printf("%d es impar\n", i);
  else
    printf("%d es par\n", i);
}
```

Ejercicio 2.4

Escribe en ensamblador la estructura de alto nivel `switch`, aplicándola al siguiente ejemplo en C.

```
for ( i= 1950; i<2015; i++ ){
  switch( i&3 ){
    case 0:  printf("En %d hubo olimpiadas\n", i);
             break;
    case 2:  printf("En %d hubo mundial de fútbol\n", i);
             break;
    default: printf("En %d no pasó nada\n", i);
  }
}
```

2.2. Enunciados de la práctica

2.2.1. Suma de elementos de un vector

En este primer apartado, estudiaremos un bucle que calcula la suma de todos los elementos de un vector. El vector se denomina `vector` y tiene 5 elementos de tipo `int` (entero de 32 bits). Los algoritmos que realizan la suma de sus elementos, tanto en C como en ensamblador, se pueden encontrar en los listados 2.8 y 2.9.

Listado 2.8: Suma de elementos de un vector (tipos4.c)

```
#include <stdio.h>

void main(void){
  int i, suma;
  int vector[5]= {128, 32, 100, -30, 124};

  for ( suma= i= 0; i<5; i++ ){
```

```
    suma+= vector[i];
  }
  printf("La suma es %d\n", suma);
}
```

Listado 2.9: Suma de elementos de un vector (tipos4.s)

```
.data

var1:   .asciz  "La suma es %d\n"
var2:   .word   128, 32, 100, -30, 124

.text
.global main

/* Salvamos registros */
main:   push    {r4, lr}

/* Inicializamos variables y apuntamos r2 a var2 */
        mov     r0, #5
        mov     r1, #0
        ldr     r2, =var2

/* Bucle que hace la suma */
bucle:  ldr     r3, [r2], #4
        add     r1, r1, r3
        subs    r0, r0, #1
        bne     bucle

/* Imprimimos resultado */
        ldr     r0, =var1
        bl      printf

/* Recuperamos registros y salimos */
        pop     {r4, lr}
        bx      lr
```

Si analizamos el código en ensamblador (listado 2.9), veremos que se recorre todo el vector con el registro `r0`, realizándose la suma sobre el registro `r1`. A diferencia de ejemplos anteriores decrementamos de 5 a 0, así nos ahorramos una comparación, ya que la instrucción `subs` detecta cuando hemos llegado al valor cero activando el flag `Z`.

En `r2` vamos recorriendo el vector elemento a elemento mediante un modo postindexado que apunta al siguiente elemento una vez leemos el actual con `ldr`. Una

vez calculada la suma en `r1`, la mostramos por pantalla mediante una llamada a `printf`.

El código del listado 2.9 está en el fichero `tipos4.s`. Compila y monta el programa con el `as` y el `gcc`. Ahora ejecuta el algoritmo con el `gdb`. Recuerda empezar con `start`. Para ver su funcionamiento, podemos ejecutar un par de iteraciones con `si` y ver cómo los valores de los registros van cambiando `i r r0 r1 r2 r3` (de vez en cuando ejecuta `disas` para saber por dónde vas). Si ejecutamos un par de iteraciones con `si` veremos que el hecho de ejecutar instrucción a instrucción resulta poco útil. Para acelerar el proceso, podemos utilizar puntos de parada o *breakpoints*.

Otro problema que tenemos es que al ejecutar un paso de una instrucción exacta `si` nos metemos dentro de la rutina `printf`, cosa que no nos interesa a no ser que queramos descubrir las interioridades de la librería. Para evitar esto ejecutamos con `ni`, que ejecutará `bl printf` de un paso sin meterse dentro de la rutina.

Para introducir un breakpoint hay varias maneras, siempre es buena idea investigar a fondo la ayuda que se nos brinda el propio depurador con `help break`. Nosotros pondremos dos puntos de ruptura.

```
(gdb) start
Temporary breakpoint 1 at 0x83cc
Starting program: /home/pi/tipos4

Temporary breakpoint 1, 0x000083cc in main ()
(gdb) break bucle
Breakpoint 2 at 0x83dc
(gdb) disas bucle
Dump of assembler code for function bucle:
   0x000083dc <+0>:     ldr     r3, [r2], #4
   0x000083e0 <+4>:     add     r1, r1, r3
   0x000083e4 <+8>:     subs    r0, r0, #1
   0x000083e8 <+12>:    bne     0x83dc <bucle>
   0x000083ec <+16>:    ldr     r0, [pc, #12]
   0x000083f0 <+20>:    bl      0x82f0 <printf>
   0x000083f4 <+24>:    pop     {r4, lr}
   0x000083f8 <+28>:    bx      lr
   0x000083fc <+32>:                    ; <UNDEFI..
   0x00008400 <+36>:    andeq   r0, r1, r4, lsr #11
End of assembler dump.
(gdb) break *0x83ec
Breakpoint 3 at 0x83ec
```

Ahora toca continuar la ejecución del programa hasta el final o hasta llegar a un punto de ruptura, y esto se hace con `continue` (de forma abreviada `cont`). También podemos mostrar la lista de puntos de ruptura, desactivar temporalmente

un breakpoint o simplemente borrarlo.

```
(gdb) info breakpoints
Num     Type           Disp Enb Address    What
2       breakpoint     keep y   0x000083dc <bucle>
3       breakpoint     keep y   0x000083ec <bucle+16>
(gdb) disable 2
(gdb) delete 3
(gdb) i b
Num     Type           Disp Enb Address    What
2       breakpoint     keep n   0x000083dc <bucle>
```

Antes de acabar nuestra sesión con `gdb` depuramos la última iteración del bucle, y luego dos instrucciones más para mostrar el texto que emite `printf`.

```
(gdb) i r r0 r1
r0             0x1      1
r1             0xe6     230
(gdb) disas
Dump of assembler code for function bucle:
=> 0x000083dc <+0>:     ldr     r3, [r2], #4
   0x000083e0 <+4>:     add     r1, r1, r3
   0x000083e4 <+8>:     subs    r0, r0, #1
   0x000083e8 <+12>:    bne     0x83dc <bucle>
   0x000083ec <+16>:    ldr     r0, [pc, #12]
   0x000083f0 <+20>:    bl      0x82f0 <printf>
   0x000083f4 <+24>:    pop     {r4, lr}
   0x000083f8 <+28>:    bx      lr
End of assembler dump.
(gdb) ni 4
0x000083ec in bucle ()
(gdb) i r r1 r3
r1             0x162    354
r3             0x7c     124
(gdb) ni 2
La suma es 354
0x000083f4 in bucle ()
```

Ahora vamos a modificar un poco el programa. Copiamos `tipos4.s` en otro fichero `tipos5.s` (con `cp tipos4.s tipos5.s`). Ahora modifica la lista de números del array, reemplazándola por esta otra.

```
var2:   .word   1600000000, -100, 800000000, -50, 200
```

Haz el ejercicio 2.5 y acábalo antes de seguir.

Ejercicio 2.5

Sabiendo que la suma de los 5 elementos del vector anterior es 2.400.000.050 completa el siguiente cuadro:

Traduce el número 2.400.000.050 a binario:

Interpreta el resultado como un entero de 32 bits y tradúcelo a decimal, ¿cuánto da?

¿Se puede representar el número entero 2.400.000.050 con 32 bits?

Si has hecho el ejercicio 2.5 puedes ahora comprobar que la suma de los valores de este vector produce un *overflow* sobre un `int`. Por tanto, el programador debería ir acumulando el resultado de la suma sobre un `long long` (64 bits), tal y como se muestra en el siguiente listado.

```
void main(void){
  int i;
  long long suma;
  int vector[5]= {1600000000, -100, 800000000, -50, 200};

  for ( suma= i= 0; i<5; i++ ){
    suma+= vector[i];
  }
  printf("La suma es %d\n", suma);
}
```

Listado 2.10: Suma de un vector de enteros largos (tipos6.s)

```
.data

var1:    .asciz  "La suma es %lld\n"
var2:    .word   1600000000, -100, 800000000, -50, 200

.text
.global main

/* Salvamos registros */
```

```
main:   push    {r4, r5, r6, lr}

/* Inicializamos variables y apuntamos r4 a var2 */
        mov     r5, #5
        mov     r2, #0
        mov     r3, #0
        ldr     r4, =var2

/* Bucle que hace la suma */
bucle:  ldr     r0, [r4], #4
        mov     r1, r0, ASR #31
        adds    r2, r2, r0
        adc     r3, r3, r1
        subs    r5, r5, #1
        bne     bucle

/* Imprimimos resultado */
        ldr     r0, =var1
        bl      printf

/* Recuperamos registros y salimos */
        pop     {r4, r5, r6, lr}
        bx      lr
```

En el código ensamblador la variable suma se almacena en los registros `r3:r2`. Como el array almacenado en memoria es de 32 bits, lo que hacemos es cargar el valor en cada iteración en `r0` y extender el signo mediante la instrucción `mov r1, r0, ASR #31` a los registros `r1:r0`. Por último hacemos la suma `r3:r2= r3:r2+r1:r0` mediante dos instrucciones de suma, en la primera `adds` sumamos los 32 bits inferiores almacenando también el posible acarreo (flag `C`), y en la segunda `adc` sumamos los 32 bits superiores más el acarreo anterior.

El número de 64 bits que le enviamos a la función `printf` debe estar en `r3:r2`, debemos guardar en pila todos los registros por encima de `r4` (incluyéndolo) y en el `push` debe haber un número par de elementos. Se explica más adelante por qué. Si tuviésemos un número impar, como es el caso, salvaremos el siguiente registro `r6` aunque no lo necesitemos en nuestra función.

Ejercicio 2.6

Dada la definición de matriz `short mat[4][6]`; ¿cuál es la fórmula para acceder al elemento `mat[i][2]`?

Listado 2.11: Matrices

```
matriz: .hword    0,    1,    2,    3,    4,    5
        .hword 0x10, 0x11, 0x12, 0x13, 0x14, 0x15
        .hword 0x20, 0x21, 0x22, 0x23, 0x24, 0x25
        .hword 0x30, 0x31, 0x32, 0x33, 0x34, 0x35
suma:   .hword 0
```

```
suma= 0;
for ( i= 0; i<4; i++ ){
  suma+= mat[i][2];
}
```

Queremos hacer un programa que sume todos los elementos de la columna 2 de la matriz (listado 2.11). Completa `tipos7.s` para que implemente este código, utilizando dentro del bucle que realiza la suma la fórmula del ejercicio 2.6. Para comprobarlo, el resultado es 104 = 0x68.

Fíjate en que en esta versión del código que recorre los elementos de una columna, para calcular la dirección de cada elemento aplicamos la ecuación de la página 38. A ésto es a lo que llamamos acceso aleatorio a los elementos de la matriz.

Sin embargo, sabemos que hay una relación entre los elementos de una fila y de la siguiente, cuando el tamaño de la columna es constante. Para hallar esta relación haz siguiente ejercicio.

Ejercicio 2.7

Calcula las fórmulas de acceso a `mat[i][2]` y `mat[i+1][2]` y halla su diferencia (resta las dos fórmulas).

Una vez hallada esta relación, que es un desplazamiento en memoria, ahora sabes que se pueden recorrer todos los elementos de una columna sin más que ir sumando ese desplazamiento. Decimos entonces que hacemos un acceso secuencial a los elementos de la matriz. Copia el fichero `tipos7.s` a `tipos8.s` e implementa sobre este último el acceso secuencial. El resultado tiene que ser el mismo que el anterior (104 = 0x68).

Capítulo 3

Subrutinas y paso de parámetros

Contenido

Objetivos: En esta sesión experimentaremos con las subrutinas. Veremos en qué consiste la convención AAPCS y cómo aplicarla tanto para llamar a funciones externas como para crear nuestras propias funciones. Escribiremos un programa en C que llame a funciones escritas en ensamblador. Por último explicaremos qué son los registros de activación y cómo aplicarlos para almacenar variables locales.

3.1. Lectura previa

3.1.1. La pila y las instrucciones `ldm` y `stm`

Se denomina pila de programa a aquella zona de memoria, organizada de forma LIFO (*Last In, First Out*), que el programa emplea principalmente para el almacenamiento temporal de datos. Esta pila, definida en memoria, es fundamental para el funcionamiento de las rutinas[1], aspecto que se desarrollará en esta práctica.

El puntero de pila es `r13` aunque por convención nunca se emplea esa nomeclatura, sino que lo llamamos `sp` (**s**tack **p**ointer o puntero de pila). Dicho registro apunta siempre a la palabra de memoria que corresponde a la cima de la pila (última palabra introducida en ella).

La pila tiene asociadas dos operaciones: `push` (meter un elemento en la pila) y `pop` (sacar un elemento de la pila). En la operación `push` primero decrementamos en 4 (una palabra son 4 bytes) el registro `sp` y luego escribimos dicho elemento en la posición de memoria apuntada por `sp`. Decimos que la pila crece hacia abajo, ya que cada vez que insertamos un dato el `sp` se decrementa en 4.

De esta forma, la instrucción `push` realmente implementa las dos siguientes instrucciones:

Listado 3.1: Operación push

```
        sub     sp, sp, #4
        str     r0, [sp]
```

Para sacar elementos de la pila tenemos la operación `pop`, que primero extrae el elemento de la pila y luego incrementa el puntero (la pila decrece hacia arriba). Por tanto, la instrucción `pop` es equivalente a:

Listado 3.2: Operación pop

```
        ldr     r0, [sp]
        add     sp, sp, #4
```

Un uso muy común de la pila es salvaguardar una serie de registros, que queremos usar para hacer las operaciones que necesitemos pero que al final tenemos que restaurar a sus valores originales. En un procesador típico escribiríamos algo así:

[1] En este texto usaremos el término rutina (o subrutina) como la implementación a bajo nivel de lo que en alto nivel se conoce como procedimientos y funciones. La diferencia entre procedimiento y función, radica en que las funciones proporcionan un valor de retorno.

```
        push    r1
        push    r2
        push    r4
        /* código que modifica los
           registros r1, r2 y r4   */
        pop     r4
        pop     r2
        pop     r1
```

Observa que el orden de recuperación de registros es inverso al de guardado. Pues bien, en ARM lo tenemos mucho más fácil. Gracias a las instrucciones de carga/escritura múltiple podemos meter los registros en una lista, empleando una única instrucción.

```
        push    {r1, r2, r4}
        /* código que modifica los
           registros r1, r2 y r4   */
        pop     {r1, r2, r4}
```

En este caso el orden no es relevante, el procesador siempre usa el orden ascendente para el `push` y el descendente para el `pop`, aunque nosotros por legibilidad siempre escribiremos los registros en orden ascendente.

Realmente, para el procesador ARM las instrucciones `push` y `pop` no existen. Sin embargo tenemos las instrucciones `stm` y `ldm` que son mucho más potentes y el ensamblador permite las pseudoinstrucciones `push` y `pop` que de forma transparente traducirá a `stm` y `ldm`.

Las instrucciones `ldm` y `stm` tienen la siguiente sintaxis.

```
ldm{modo_direc}{cond} rn{!}, lista_reg
stm{modo_direc}{cond} rn{!}, lista_reg
```

A continuación explicamos cada uno de los argumentos de `ldm/stm`

1. **modo_direc**
   - `ia`: Incrementa dirección después (**i**ncrement **a**fter) de cada transferencia. Es el modo por defecto en caso de omitirlo.
   - `ib`: Incrementa dirección antes (**i**ncrement **b**efore) de cada transferencia.
   - `da`: Decrementa después (**d**ecrement **a**fter) de cada transferencia.
   - `db`: Decrementa antes (**d**ecrement **b**efore) de cada transferencia.
2. **cond**. Es opcional, son las mismas condiciones de los flags que vimos en la sección 2.1.3 del capítulo anterior (página 38), que permiten ejecutar o no dicha instrucción.

3. **rn**. Es el registro base, el cual apunta a la dirección inicial de memoria donde se hará la transferencia. El registro más común es `sp` (`r13`), pero puede emplearse cualquier otro.

4. **!**. Es un sufijo opcional. Si está presente, actualizamos `rn` con la dirección calculada al final de la operación.

5. **lista_reg**. Es una lista de uno o más registros, que serán leídos o escritos en memoria. La lista va encerrada entre llaves y separada por comas. También podemos usar un rango de registros. En este ejemplo se almacenan los registros `r3, r4, r5, r6, r10 y r12`. Si inicialmente `r1` contiene el valor 24, después de ejecutar la instrucción siguiente `r3` se almacenará en la dirección 20, `r4` en 16, `r5` en 12, `r6` en 8, `r10` en 4 y `r12` en 0.

```
        stmdb   r1!, {r3-r6, r10, r12}
```

Si tenemos en cuenta que `push` predecrementa, que `pop` postincrementa y que ambas actualizan el registro base (que sería `sp`), la traducción de las pseudoinstrucciones `push {r4, r6}` y `pop {r4, r6}` serían respectivamente:

```
    stmdb sp!, {r4, r6}     /* push */
    ldmia sp!, {r4, r6}     /* pop  */
```

Nosotros sin embargo emplearemos los nemónicos `push/pop`, mucho más fáciles de recordar.

3.1.2. Convención AAPCS

Podemos seguir nuestras propias reglas, pero si queremos interactuar con las librerías del sistema, tanto para llamar a funciones como para crear nuestras propias funciones y que éstas sean invocadas desde un lenguaje de alto nivel, tenemos que seguir una serie de pautas, lo que se denominamos AAPCS (Procedure Call Standard for the ARM Architecture).

1. Podemos usar hasta cuatro registros (desde `r0` hasta `r3`) para pasar parámetros y hasta dos (`r0` y `r1`) para devolver el resultado.

2. No estamos obligados a usarlos todos, si por ejemplo la función sólo usa dos parámetros de tipo `int` con `r0` y `r1` nos basta. Lo mismo pasa con el resultado, podemos no devolver nada (tipo `void`), devolver sólo `r0` (tipo `int` ó un puntero a una estructura más compleja), o bien devolver `r1:r0` cuando necesitemos enteros de 64 bits (tipo `long long`).

3. Los valores están alineados a 32 bits (tamaño de un registro), salvo en el caso de que algún parámetro sea más grande, en cuyo caso alinearemos a 64 bits. Un ejemplo de esto lo hemos visto en el Ejercicio 2.5, donde necesitábamos pasar dos parámetros: una cadena (puntero de 32 bits) y un entero tipo `long long`. El puntero a cadena lo almacenábamos en `r0` y el entero de 64 bits debe empezar en un registro par (`r1` no vale) para que esté alineado a 64 bits, serían los registros `r2` y `r3`. En estos casos se emplea little endian, la parte menos significativa sería `r2` y la de mayor peso, por tanto, `r3`.

4. El resto de parámetros se pasan por pila. En la pila se aplican las mismas reglas de alineamiento que en los registros. La unidad mínima son 32 bits, por ejemplo si queremos pasar un char por valor, extendemos de byte a word rellenando con ceros los 3 bytes más significativos. Lo mismo ocurre con los enteros de 64 bits, pero en el momento en que haya un sólo parámetro de este tipo, todos los demás se alinean a 64 bits.

5. Es muy importante preservar el resto de registros (de `r4` en adelante incluyendo `lr`). La única excepción es el registro `r12` que podemos cambiar a nuestro antojo. Normalmente se emplea la pila para almacenarlos al comienzo de la función y restaurarlos a la salida de ésta. Puedes usar como registros temporales (no necesitan ser preservados) los registros desde `r0` hasta `r3` que no se hayan empleado para pasar parámetros.

6. La pila debe estar alineada a 8 bytes, esto quiere decir que de usarla para preservar registros, debemos reservar un número par de ellos. Si sólo necesitamos preservar un número impar de ellos, añadimos un registro más a la lista dentro del `push`, aunque no necesite ser preservado.

7. Aparte de para pasar parámetros y preservar registros, también podemos usar la pila para almacenar variables locales, siempre y cuando cumplamos la regla de alinear a 8 bytes y equilibremos la pila antes de salir de la función.

Cuando programamos en Bare Metal no es necesario seguir estas reglas. Es más, podemos escribir una función sin seguir la norma incluso si trabajamos bajo Linux, pero no es recomendable ya que no podríamos reusarlas para otros proyectos.

Lo mejor para entender estas reglas es con una serie de ejemplos de menor a mayor complejidad que veremos a lo largo de este capítulo.

3.2. Ejemplos de aplicación

3.2.1. Funciones en ensamblador llamadas desde C

En este primer ejemplo crearemos nuestras propias funciones generadoras de números aleatorios, a las que llamaremos `myrand` y `mysrand` (en sustitución a las `rand` y `srand` que ya existen en la librería).

Listado 3.3: Código del programa subrut1.c

```
#include <stdio.h>

void main(void){
  int i;

  mysrand(42);
  for ( i= 0; i<5; i++ ){
    printf("%d\n", myrand());
  }
}
```

El programa principal lo hacemos en C, mientras que las funciones `myrand` y `mysrand` las haremos en ensamblador. La implementación es sencilla. Almacenamos la semilla en la variable estática `seed`. Podemos cambiar el valor de la semilla en cualquier momento con la función `mysrand`, y recibir un número pseudoaleatorio de 15 bits con la función `myrand`. En realidad `myrand` lo único que hace es aplicar una operación sencilla en la semilla (multiplicación y suma) y extraer 15 bits de esta. El secreto del algoritmo reside en que se han elegido unas constantes para la multiplicación y la suma de tal forma que la variable `seed` pasará por todos los valores de 32 bits en una secuencia que a simple vista parece aleatoria, pero que no lo es (por eso se llama pseudoaleatoria):

```
static int seed;

short myrand(void){
  seed= seed*1103515245 + 12345;
  return seed>>16 & 0x7fff;
}

void mysrand(int x){
  seed= x;
}
```

Veamos en qué se traducen estas funciones en ensamblador:

Listado 3.4: Código del programa subrut1.s

```
.data
seed:    .word   1
const1:  .word   1103515245
const2:  .word   12345

.text
.global myrand, mysrand
myrand:  ldr     r1, =seed        @ leo puntero a semilla
         ldr     r0, [r1]         @ leo valor de semilla
         ldr     r2, [r1, #4]     @ leo const1 en r2
         mul     r3, r0, r2       @ r3= seed*1103515245
         ldr     r0, [r1, #8]     @ leo const2 en r0
         add     r0, r0, r3       @ r0= r3+12345
         str     r0, [r1]         @ guardo en variable seed

/* Estas dos líneas devuelven "seed>>16 & 0x7fff".
   Con un pequeño truco evitamos el uso del AND */
         LSL     r0, #1
         LSR     r0, #17
         bx      lr

mysrand:ldr     r1, =seed
         str     r0, [r1]
         bx      lr
```

Antes de nada ensamblamos, compilamos/enlazamos y ejecutamos estos archivos para comprobar su correcto funcionamiento:

```
pi@raspberrypi ~ $ as -o subrut1.o subrut1.s
pi@raspberrypi ~ $ gcc -o subrut1 subrut1.c subrut1.o
pi@raspberrypi ~ $ ./subrut1
2929
28487
11805
6548
9708
pi@raspberrypi ~ $
```

A diferencia de ejemplos anteriores, en nuestro código ensamblador no tenemos ninguna función `main` porque ésta la hemos implementado en C. Sin embargo aparecen dos etiquetas después de la directiva `.global`, que son `myrand` y `mysrand`. Esto

es fundamental si queremos que nuestras funciones sean vistas desde el exterior, en este caso desde el programa en C.

Empecemos con la función más sencilla, `mysrand`. Consta de 3 instrucciones. En la primera de ellas apuntamos con `r1` a la dirección donde se encuentra la variable `seed`. En la segunda pasamos el primer y único parámetro de la función, `r0`, a la posición de memoria apuntada por `r1`, es decir, a la variable `seed`. Por último salimos de la función con la conocida instrucción `bx lr`. No hay más, no tenemos que devolver nada en `r0` (la función devuelve el tipo `void`), ni tenemos que preservar registros, ni crear variables locales.

La otra función es un poco más compleja. Aparte de requerir más cálculos debemos devolver un valor. Aprovechamos que las 3 variables están almacenadas consecutivamente (en realidad las dos últimas son constantes) para no tener que cargar 3 veces la dirección de cada variable en un registro. Lo hacemos la primera vez con `ldr r1, =seed`, y accedemos a las variables con direccionamiento a registro con desplazamiento (`[r1]`, `[r1, #4]` y `[r1, #8]`). Como no hay parámetros de entrada empleamos los registros `r0, r1, r2 y r3` como almacenamiento temporal, hacemos nuestros cálculos, escribimos el resultado en la variable `seed` y devolvemos el resultado en el registro `r0`.

3.2.2. Funciones en ensamblador llamadas desde ensamblador

Del ejemplo anterior vamos a pasar a ensamblador la única parte que estaba escrita en C, que era la función `main`:

Listado 3.5: Código del programa subrut2.s

```
.data

var1:   .asciz  "%d\n"
seed:   .word   1
const1: .word   1103515245
const2: .word   12345

.text

.global main

/* Salvamos registros */
main:   push    {r4, r5}

/* Llamamos a mysrand con parámetro 42 */
        mov     r0, #42
```

```
        bl      mysrand

/* Inicializamos contador de bucle en r4 */
        mov     r4, #5

/* Bucle que imprime 5 números aleatorios */
bucle:  bl      myrand      @ leo número aleatorio
        mov     r1, r0      @ paso valor a r1
        ldr     r0, =var1   @ pongo cadena en r0
        bl      printf      @ llamo a función printf
        subs    r4, r4, #1  @ decremento contador
        bne     bucle       @ salgo si llego a cero

/* Recuperamos registros y salimos */
        pop     {r4, r5}
        bx      lr

myrand: ldr     r1, =seed
        ldr     r0, [r1]
        ldr     r2, [r1, #4]
        mul     r3, r0, r2
        ldr     r0, [r1, #8]
        add     r0, r0, r3
        str     r0, [r1]
        mov     r0, r0, LSL #1
        mov     r0, r0, LSR #17
        bx      lr

mysrand:ldr     r1, =seed
        str     r0, [r1]
        bx      lr
```

Como véis ya no hace falta poner a `.global` las funciones `myrand` y `mysrand`, puesto que son de uso interno. Sin embargo sí lo hacemos con `main`, ya que ahora sí la implementamos en ensamblador. Al fin y al cabo `main` es otra función más y por tanto debe de seguir la normativa AAPCS.

Primero preservamos `r4` y `r5`. En realidad `r5` no se modifica y no haría falta preservarla, pero lo hacemos para alinear a 8 la pila. Luego llamamos a `mysrand` con el valor 42 como primer y único parámetro. Inicializamos a 5 el contador del bucle, que almacenamos en `r4` y comenzamos el bucle. El bucle consiste en llamar a `myrand` y pasar el resultado devuelto de esta función al segundo parámetro de la función `printf`, llamar a `printf`, decrementar el contador y repetir el bucle hasta que el contador llegue a cero. Una vez salimos del bucle recuperamos los registros

`r4` y `r5` y devolvemos el control al sistema: `bx lr`.

3.2.3. Funciones recursivas

El siguiente paso es implementar una función recursiva en ensamblador. Vamos a escoger la secuencia de Fibonacci por su sencillez. Trataremos de imprimir los diez primeros números de la secuencia. Se trata de una sucesión de números naturales en las que los dos primeros elementos valen uno y los siguientes se calculan sumando los dos elementos anteriores. Los diez primeros números serían los siguientes.

```
1, 1, 2, 3, 5, 8, 13, 21, 34, 55...
```

Este es el código en un lenguaje de alto nivel como C que imprime la anterior secuencia.

Listado 3.6: Código del programa subrut3.c

```
#include <stdio.h>

int fibonacci(int n){
  if( n < 2 )
    return 1;
  else
    return fibonacci(n-1) + fibonacci(n-2);
}

void main(void){
  int i;

  for ( i= 0; i<10; i++ )
    printf("%d\n", fibonacci(i));
}
```

Lo que vamos a explicar ahora es cómo crear variables locales dentro de una función. Aunque en C no necesitemos variables locales para la función fibonacci, sí nos hará falta en ensamblador, en concreto dos variables: una para acumular la suma y otra para mantener el parámetro de entrada.

Para ello vamos a emplear la pila, que hasta ahora sólo la dedicábamos para salvaguardar los registros a partir de `r4` en la función. La pila tendría un tercer uso que no hemos visto todavía. Sirve para que el llamador pase el resto de parámetros en caso de que haya más de 4. Los primeros 4 parámetros (dos en caso de parámetros de 64 bits) se pasan por los registros desde `r0` hasta `r3`. A partir de aquí si hay más parámetros éstos se pasan por pila.

Las variables locales se alojan debajo del área de salvaguarda de registros, para ello hay que hacer espacio decrementando el puntero de pila una cierta cantidad de

bytes, e incrementando `sp` en esa misma cantidad justo antes de salir de la función. En la figura 3.1 vemos el uso de la pila de una función genérica.

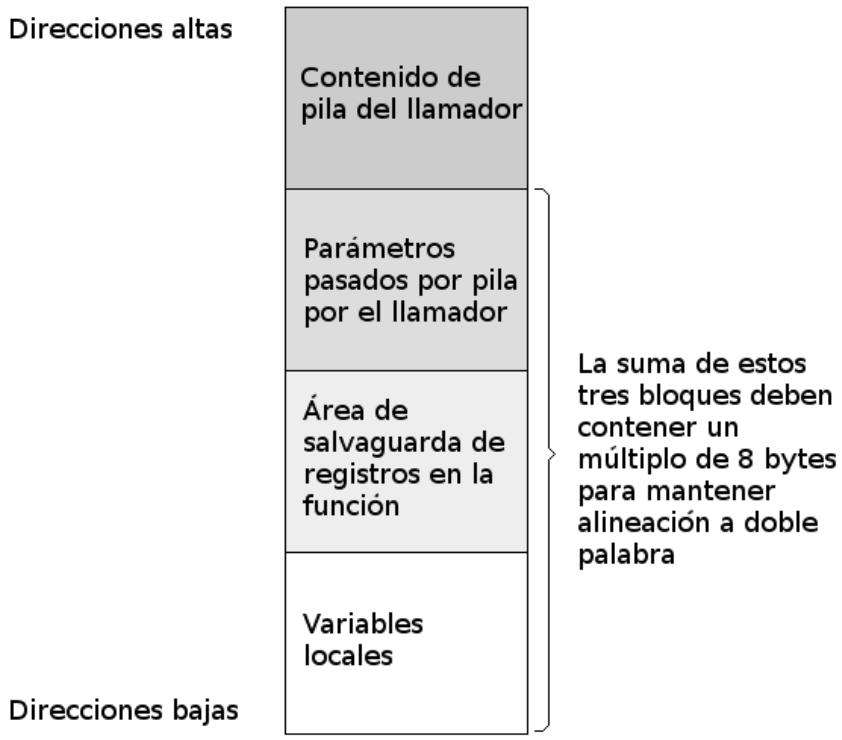

Figura 3.1: Uso de la pila en una función

Pues bien, en nuestro caso de la función fibonacci necesitamos 0 bytes para paso de parámetros, 4 bytes para salvaguarda de registros (sólo guardaremos `lr`) y 8 bytes para nuestras dos variables locales. Como la suma es de 12 bytes, que no es múltiplo de 8, redondeamos a 16 añadiendo una tercera variable local que no usaremos (también podríamos haber salvaguardado un segundo registro). Nuestro mapa particular lo podemos observar en la figura 3.2.

En teoría podemos encargarnos nosotros mismos de hacer toda la aritmética que conlleva el uso de variables locales, pero en la práctica estamos más expuestos a cometer errores y nuestro código es más ilegible. Las 3 variables locales ocupan 12 bytes, a la primera accedemos con el direccionamiento `[sp]` y a la segunda con `[sp, #4]` (la tercera no la usamos). El código quedaría como en el listado 3.7.

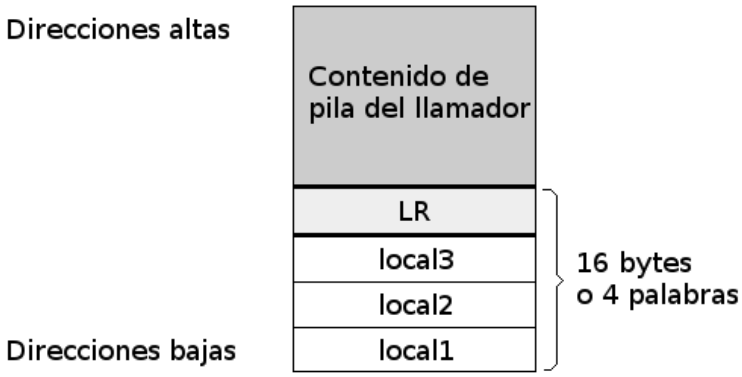

Figura 3.2: Uso de la pila en nuestra función

Listado 3.7: Función recursiva fibo (en subrut3.s)

```
fibo:   push    {lr}            @ salvaguarda lr
        sub     sp, #12         @ hago espacio para v. locales
        cmp     r0, #2          @ if n<2
        movlo   r0, #1          @ return 1
        blo     fib1

        sub     r0, #1          @ else
        str     r0, [sp]        @ salvo n-1 en [sp]
        bl      fibo            @ fibonacci(n-1)
        str     r0, [sp, #4]    @ salvo valor devuelto por fib.(n-1)
        ldr     r0, [sp]        @ recupero de la pila n-1
        sub     r0, #1          @ calculo n-2
        bl      fibo            @ fibonacci(n-2)
        ldr     r1, [sp, #4]    @ recupero salida de fib.(n-1)
        add     r0, r1          @ lo sumo a fib.(n-1)

fib1:   add     sp, #12         @ libero espacio de v. locales
        pop     {lr}            @ recupero registros (sólo lr)
        bx      lr              @ salgo de la función
```

Siguiendo el orden de la pila, primero salvaguardamos `lr` y luego hacemos espacio para 3 palabras con `sub sp, #12` en el comienzo de la función. Al salir de la rutina restauramos en orden inverso, primero restauramos los 12 bytes de las variables locales y luego recuperamos `lr`.

Nuestra función tiene dos ramas, en una se comprueba que el parámetro sea menor de 2, y si lo es devolvemos el valor 1 y salimos de la función. En la otra rama invocamos nuestra propia función recursivamente dos veces, sumamos el resultado

y devolvemos la suma al salir de la función.

El truco para hacer el código más legible es nombrando las 3 variables locales y la longitud mediante la directiva `.equ` (también nos valdría su alias `.set`). Partimos del valor 0 de desplazamiento en la primera variable local y vamos encadenando. A cada elemento le corresponde la longitud más la posición del anterior. Así, si necesitamos modificar alguna variable tan sólo tendremos en cuenta la anterior y la siguiente, no tenemos que modificar toda la estructura:

```
        .equ    local1,   0
        .equ    local2,   4+local1
        .equ    local3,   4+local2
        .equ    length,   4+local3
```

Con esta nueva filosofía el código queda menos críptico, como vemos en el listado 3.8.

Listado 3.8: Función recursiva fibo (en subrut3.s)

```
fibo:   push    {lr}                @ salvaguarda lr
        sub     sp, #length         @ hago espacio para v.locales
        cmp     r0, #2              @ if n<2
        movlo   r0, #1              @ return 1
        blo     fib1

        sub     r0, #1              @ else
        str     r0, [sp, #local1]   @ salvo n-1 en [sp]
        bl      fibo                @ fibonacci(n-1)
        str     r0, [sp, #local2]   @ salvo salida de fib.(n-1)
        ldr     r0, [sp, #local1]   @ recupero de la pila n-1
        sub     r0, #1              @ calculo n-2
        bl      fibo                @ fibonacci(n-2)
        ldr     r1, [sp, #local2]   @ recupero salida de fib(n-1)
        add     r0, r1              @ lo sumo a fib.(n-1)

fib1:   add     sp, #length         @ libero espacio de v.locales
        pop     {lr}                @ recupero registros, sólo lr
        bx      lr                  @ salgo de la función
```

Ya estamos en condiciones de mostrar el archivo completo en el listado 3.9.

Listado 3.9: Código del programa subrut3.s

```
.data
var1:   .asciz  "%d\n"

.text
```

```
.global main

/* Salvo registros */
main:   push    {r4, lr}

/* Inicializo contador del bucle a 0 en r4 */
        mov     r4, #0

/* Bucle que imprime los 10 primeros valores */
bucle:  mov     r0, r4      @ tomo contador como parámetro
        bl      fibo        @ llamo a la función
        mov     r1, r0      @ paso resultado a r1
        ldr     r0, =var1   @ pongo cadena en r0
        bl      printf      @ llamo a función printf
        add     r4, r4, #1  @ incremento contador de bucle
        cmp     r4, #10     @ comparo si es menor de 10
        bne     bucle       @ si llegamos a 10 salgo de bucle

/* Recupero registros y salgo de main */
        pop     {r4, lr}
        bx      lr

        .equ    local1,   0
        .equ    local2,   4+local1
        .equ    local3,   4+local2
        .equ    length,   4+local3

fibo:   push    {lr}              @ salvaguarda lr
        sub     sp, #length       @ hago espacio para v.locales
        cmp     r0, #2            @ if n<2
        movlo   r0, #1            @ return 1
        blo     fib1

        sub     r0, #1            @ else
        str     r0, [sp, #local1] @ salvo n-1 en [sp]
        bl      fibo              @ fibonacci(n-1)
        str     r0, [sp, #local2] @ salvo salida de fib.(n-1)
        ldr     r0, [sp, #local1] @ recupero de la pila n-1
        sub     r0, #1            @ calculo n-2
        bl      fibo              @ fibonacci(n-2)
        ldr     r1, [sp, #local2] @ recupero salida de fib(n-1)
        add     r0, r1            @ lo sumo a fib.(n-1)
```

```
fib1:   add     sp, #length         @ libero espacio de v.locales
        pop     {lr}                @ recupero registros, sólo lr
        bx      lr                  @ salgo de la función
```

Lo único que nos faltaba era la función `main`. La lista de `.equ` puede ir al comienzo, pero por claridad la ponemos justo antes de la función a la que se va a aplicar. La función `main` no tiene nada nuevo, salvo que incrementamos el contador `r4` en lugar de decrementarlo porque necesitamos dicho valor como parámetro para llamar a la función `fibo`.

Para terminar con este ejemplo vamos a hacer una sencilla optimización. Observa un momento la primera rama de la función. Si el parámetro es menor de dos tan sólo operamos con un registro, `r0`, tanto para comparar la entrada como para escribir el valor de retorno. No se toca ningún registro más, no hemos modificado `lr` porque no hemos llamado a ninguna subrutina, tampoco hemos hecho uso de las variables locales.

La optimización consiste (ver listado 3.10) en procesar la primera rama antes de las operaciones con la pila, de esta forma nos ahorramos algunos ciclos de reloj. Es un buen ejemplo para comprobar lo flexibles que pueden ser las funciones: hay funciones en las que podemos evitar tratar con la pila como en el listado 3.5, otras en las que no tenemos más remedio, y un último caso en que podemos tener una mezcla de ambas alternativas.

Listado 3.10: Parte del código del programa subrut4.s

```
fibo:   cmp     r0, #2              @ if n<2
        movlo   r0, #1              @ return 1
        bxlo    lr                  @ salgo de la función

        push    {lr}                @ salvaguarda lr
        sub     sp, #length         @ hago espacio para v.locales
        sub     r0, #1              @ r0= n-1
        str     r0, [sp, #local1]   @ salvo n-1 en [sp]
        bl      fibo                @ fibonacci(n-1)
        str     r0, [sp, #local2]   @ salvo salida de fib.(n-1)
        ldr     r0, [sp, #local1]   @ recupero de la pila n-1
        sub     r0, #1              @ calculo n-2
        bl      fibo                @ fibonacci(n-2)
        ldr     r1, [sp, #local2]   @ recupero salida de fib(n-1)
        add     r0, r1              @ lo sumo a fib.(n-1)

        add     sp, #length         @ libero espacio de v.locales
        pop     {lr}                @ recupero registros, sólo lr
        bx      lr                  @ salgo de la función
```

3.2.4. Funciones con muchos parámetros de entrada

Lo último que nos falta por ver es cómo acceder a los parámetros de una función por pila, para lo cual necesitamos una función de al menos cinco parámetros. Lo más sencillo que se nos ocurre es un algoritmo que evalue cualquier polinomio de grado 3 en el dominio de los enteros.

$$f(x) = ax^3 + bx^2 + cx + d \tag{3.1}$$

Nuestra función tendría 5 entradas, una para cada coeficiente, más el valor de la x que sería el quinto parámetro que pasamos por pila. Como siempre, comenzamos escribiendo el código en C:

Listado 3.11: Evaluador de polinomios subrut5.c

```
int poly3(int a, int b, int c, int d, int x){
  return a*x*x*x + b*x*x + c*x + d;
}

void main(void){
  printf("%d\n%d\n%d\n",
          poly3(1, 2, 3, 4, 5),
          poly3(1, -1, 1, -1, 8),
          poly3(2, 0, 0, 0, 8));
}
```

Cuya salida es la siguiente.

```
194
455
1024
```

El código completo en ensamblador se muestra en el listado 3.12.

Listado 3.12: Evaluador de polinomios subrut5.s

```
.data
var1:    .asciz   "%d\n"

.text
.global main

/* Salvo registros */
main:    push     {r4, lr}

/* Introduzco los 4 primeros parámetros vía registros */
         mov      r0, #1
```

```
        mov     r1, #2
        mov     r2, #3
        mov     r3, #4

/* Introduzco el 5o parámetro por pila */
        mov     r4, #5
        push    {r4}

/* Llamada a función poly3(1, 2, 3, 4, 5) */
        bl      poly3

/* Equilibro la pila (debido al 5o parámetro) */
        add     sp, #4

/* Paso resultado de la función a r1, cadena a
   imprimir a r0 y llamo a la función */
        mov     r1, r0
        ldr     r0, =var1
        bl      printf

/* Segunda llamada, esta vez poly3(1, -1, 1, -1, 8) */
        mov     r0, #1
        mov     r1, #-1
        mov     r2, #1
        mov     r3, #-1
        mov     r4, #8
        push    {r4}
        bl      poly3
        add     sp, #4

/* Imprimo resultado de segunda llamada */
        mov     r1, r0
        ldr     r0, =var1
        bl      printf

/* Llamo e imprimo poly3(2, 0, 0, 0, 8) */
        mov     r0, #2
        mov     r1, #0
        mov     r2, #0
        mov     r3, #0
        mov     r4, #8
        push    {r4}
        bl      poly3
```

```
        add     sp, #4
        mov     r1, r0
        ldr     r0, =var1
        bl      printf

/* Recupero registros y salgo de main */
        pop     {r4, lr}
        bx      lr

        .equ    param5,   4*1  /* r4 */

poly3:  push    {r4}              @ salvaguarda r4
        ldr     r4, [sp, #param5] @ leo r4 de pila
        smlabb  r3, r2, r4, r3    @ r3= c*x + d
        smulbb  r2, r4, r4        @ r2= x*x
        smlabb  r3, r1, r2, r3    @ r3= b*(x*x) + (c*x + d)
        smulbb  r2, r2, r4        @ r2= x*(x*x)
        smlabb  r0, r0, r2, r3    @ r0= a*x*x*x + b*x*x + c*x+d
        pop     {r4}              @ recupero r4
        bx      lr                @ salgo de la función
```

Vemos como hemos usado un `.equ` para facilitar la legibilidad del código, así accedemos al índice del quinto parámetro sin tener que hacer cálculos. El mapa de la pila quedaría así.

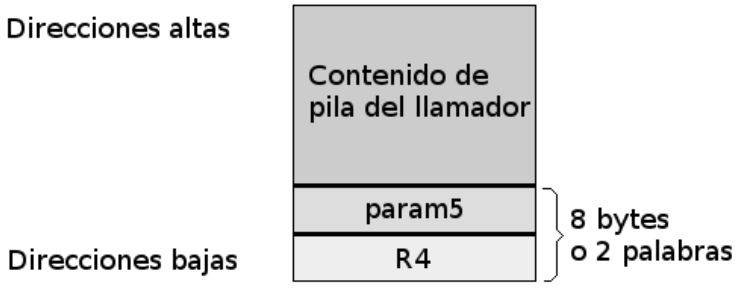

Figura 3.3: Mapa de pila de función poly3

Se pueden combinar los `.equ` de variables locales con los de parámetros por pila, por ejemplo si tuviésemos una función hipotética con 6 parámetros (dos de ellos pasados por pila), 3 variables locales y salvaguarda de 3 registros, lo haríamos de la siguiente forma.

```
      .equ    local1,   0
      .equ    local2,   4+local1
      .equ    local3,   4+local2
```

```
        .equ    length,   4+local3
        .equ    param5,   4*3+length /* r4,r5,lr */
        .equ    param6,   4+param5

func:   push    {r4, r5, lr}
        ...
```

Y éste sería el mapa de pila de nuestra hipotética función.

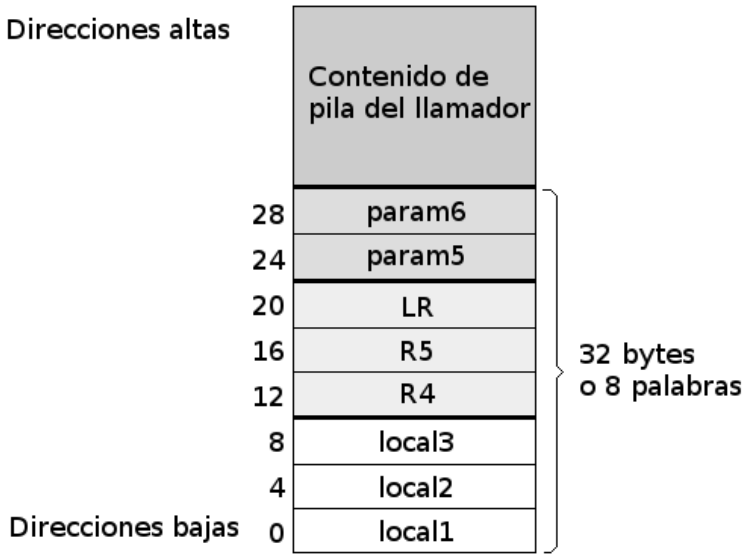

Figura 3.4: Mapa de función hipotética

Los números que hemos puesto a la izquierda de cada elemento se corresponden con las constantes que calcula el preprocesador para el desplazamiento respecto al puntero de pila. De no haber empleado la lista de `.equ` tendríamos que calcular nosotros mismos estos desplazamientos, y lo que es peor, el código perdería legibilidad. No es lo mismo poner `ldr r4, [sp, #param5]`, que por ejemplo `ldr r4, [sp, #24]`, ya que habría que revisar a qué corresponde el desplazamiento `#24` o indicarlo como comentario.

Por norma general en la arquitectura ARM se emplean muy poco las variables locales, ya que operar con éstas implica guardarlas y restaurarlas de memoria, para lo que se requieren instrucciones adicionales (recuerda que el procesador no realiza operaciones aritméticas directamente en memoria). En lugar de variables locales se suelen emplear directamente los registros que han sido salvaguardados previamente con la instrucción `push`, esto nos da juego para trabajar con hasta 10 registros (desde `r4` hasta `r12`, incluyendo `lr`) como almacén temporal para nuestras operaciones.

Veamos ahora nuestra función `poly3`. Hemos salvaguardado `r4` porque necesitamos un almacén temporal donde operar con el quinto parámetro que leeremos por pila. En esta función la pila está alineada a 8 bytes porque usamos 4 bytes en el quinto parámetro más los 4 bytes de salvaguardar `r4`, en total 8 bytes.

Todas los cálculos se condensan en 5 líneas donde se alternan las instrucciones `smlaxy` y `smulxy`. Son instrucciones que multiplican/acumulan y multiplican respectivamente números enteros. El comportamiento exacto de cada instrucción viene detallado en el datasheet[6] del procesador. [2]

```
smlabb  r3, r2, r4, r3    @ r3= d+c*x
smulbb  r2, r4, r4        @ r2= x^2
smlabb  r3, r1, r2, r3    @ r3= d+c*x+b*x^2
smulbb  r2, r2, r4        @ r2= x^3
smlabb  r0, r0, r2, r3    @ r0= d+c*x+b*x^2+a*x^3
```

Como podéis observar, las instrucciones ARM son muy potentes, permiten implementar en 5 instrucciones lo que en C nos habría costado 6 multiplicaciones y 4 sumas. Nótese cómo reusamos los registros `r2` y `r3`: al principio son parámetros de entrada, pero luego los empleamos como registros temporales a medida que no los necesitamos más.

Después de esto acaba la función con las habituales `pop r4` y `bx lr`. Ya hemos terminado la función `poly3`, que ha quedado bastante pequeña en tamaño. Todo lo contrario que la función `main`. Sin embargo, la función `main` es larga por varias razones: hacemos 3 llamadas a `poly3`, debemos introducir muchas constantes, algunas de ellas en pila, y debemos imprimir los resultados y hacer el equilibrado de pila. Este equilibrado de pila consiste en incrementar `sp` después de la llamada a la función para desalojar los parámetros que previamente habíamos introducido en la misma. Como en nuestro ejemplo pasamos por pila un único parámetro de 4 bytes, lo que hacemos es incrementar `sp` en 4 tras cada llamada a `poly3`.

Un detalle muy importante que no podemos observar en nuestro ejemplo es que los parámetros que pasamos por pila se pasan en orden inverso desde el último al quinto. Esto es así porque la pila crece hacia abajo. Es más, es aconsejable reusar los registros `r0-r3` para introducir los parámetros por pila. Si tuviésemos que pasar 6 parámetros (constantes del 1 al 6) lo haríamos así:

```
        mov     r0, #6
        push    {r0}
        mov     r0, #5
        push    {r0}
        mov     r0, #1
        mov     r1, #2
```

[2] También es fácil encontrar la especificación de una instrucción buscando su nemónico en Google, ya que suele aparecer la ayuda oficial de ARM en el primer resultado de la búsqueda

```
        mov     r2, #3
        mov     r3, #4
```

Como véis no hay una forma clara y legible de introducir los parámetros de una función. Hay que Tener cuidado con los push múltiples, ya que no importa el orden en que especifiques los registros, el procesador siempre introduce en pila el registro más alto y va hacia atrás hasta llegar al primero. Aprovechando esto podemos mejorar el ejemplo anterior:

```
        mov     r0, #5
        mov     r1, #6
        push    {r0, r1}
        mov     r0, #1
        mov     r1, #2
        mov     r2, #3
        mov     r3, #4
```

Por último vamos a mejorar un poco la velocidad de la función `poly3` de esta forma:

Listado 3.13: Parte de subrut6.s

```
poly3:  sub     sp, #4
        ldr     r12, [sp, #param5]
        smlabb  r3, r2, r12, r3
        smulbb  r2, r12, r12
        smlabb  r3, r1, r2, r3
        smulbb  r2, r2, r12
        smlabb  r0, r0, r2, r3
        add     sp, #4
        bx      lr
```

¿En qué consiste la mejora? Pues que hemos usado el registro basura `r12`, que es el único que podemos emplear sin salvaguardarlo previamente en la lista del `push`. Esto nos quitaría el `push` y el `pop`, aunque en este ejemplo lo hemos reemplazado por instrucciones `sub` y `add`. La razón es que debemos mantener el puntero de pila en un múltiplo de 8. No obstante las instrucciones que no acceden a memoria siempre son más rápidas que las que lo hacen, así que hemos ganado velocidad.

3.2.5. Pasos detallados de llamadas a funciones

Como ya hemos visto todos los casos posibles, hacemos un resumen de todo en una serie de puntos desde que pasamos los parámetros en el llamador hasta que restauramos la pila desde el llamador, pasando por la llamada a la función y la ejecución de la misma.

1. Usando los registros `r0-r3` como almacén temporal, el llamador pasa por pila los parámetros quinto, sexto, etc... hasta el último. Cuidado con el orden, especialmente si se emplea un `push` múltiple. Este paso es opcional y sólo necesario si nuestra función tiene más de 4 parámetros.

2. El llamador escribe los primeros 4 parámetros en `r0-r3`. Este paso es opcional, ya que nos lo podemos saltar si nuestra función no tiene parámetros.

3. El llamador invoca a la función con `bl`. Este paso es obligatorio.

4. Ya dentro de la función, lo primero que hace esta es salvaguardar los registros desde `r4` que se empleen más adelante como registros temporales. En caso de no necesitar ningún registro temporal nos podemos saltar este paso.

5. Decrementar la pila para hacer hueco a las variables locales. La suma de bytes entre paso de parámetros por pila, salvaguarda y variables locales debe ser múltiplo de 8, rellenar aquí hasta completar. Como este paso es opcional, en caso de no hacerlo aquí el alineamiento se debe hacer en el paso 4.

6. La función realiza las operaciones que necesite para completar su objetivo, accediendo a parámetros y variables locales mediante constantes `.equ` para aportar mayor claridad al código. Se devuelve el valor resultado en `r0` (ó en `r1:r0` si es doble palabra).

7. Incrementar la pila para revertir el alojamiento de variables locales.

8. Recuperar con `pop` la lista de registros salvaguardados.

9. Retornar la función con `bx lr` volviendo al código llamador, exactamente a la instrucción que hay tras el `bl`.

10. El llamador equilibra la pila en caso de haber pasado parámetros por ella.

3.3. Ejercicios

3.3.1. Mínimo de un vector

Dado un vector de enteros y su longitud, escribe una función en ensamblador que recorra todos los elementos del vector y nos devuelva el valor mínimo. Para comprobar su funcionamiento, haz `.global` la función y tras ensamblarla, enlázala con este programa en C.

```
#include <stdio.h>

int vect[]= {8, 10, -3, 4, -5, 50, 2, 3};

void main(void){
  printf("%d\n", minimo(vect, 8));
}
```

Hay muchas formas de calcular el mínimo de una lista de elementos, la más sencilla es comparar todos los elemento con una variable, la cual actualizamos sólo si el elemento es menor que la variable. Usa el siguiente cuadro para escribir la versión en ensamblador.

```
int minimo(int* v, int len){
  int i, min;

  min= v[0];
  for ( i= 1; i<len; i++ )
    if( v[i]<min )
      min= v[i];
  return min;
}
```

3.3.2. Media aritmética, macros y conteo de ciclos

Media aritmética

Escribe una función en ensamblador que calcule la media aritmética (truncada porque trabajamos con enteros) de dos números. Escribe también la función `main` con cinco llamadas a `media` con distintos parámetros.

Una vez hecho esto, supón que cada instrucción tarda un ciclo de reloj en ejecutarse. Cuenta manualmente el número de ciclos que tarda la ejecución completa desde la primera instrucción de `main` hasta la última `bx lr`, incluyendo ésta. En caso de una llamada a subrutina cuenta todas las instrucciones que se ejecutan metiéndote en la subrutina. La única excepción es `bl printf`, que debes contar como un único ciclo.

Haz lo mismo pero usando la herramienta `gdb` para comprobar el resultado anterior. Recuerda no meterte dentro de los printf con `ni`. En las llamadas a la función `media` usa `si`.

Macros

Hay una forma de acelerar las funciones, aunque sólo es práctica para funciones pequeñas que se utilicen mucho. Se trata de escribir el contenido de la función en lugar de llamar a la misma, y para evitar repetir siempre el mismo código utilizamos la directiva `.macro`. Con este truco nos ahorramos al menos la ejecución de las instrucciones `bl funcion` y `bx lr`. El inconveniente es que el tamaño del ejecutable será mayor.

En el listado 3.14 vemos un ejemplo que usa la función `abs`, pero que con un simple cambio empleamos la macro del mismo nombre.

Listado 3.14: Parte de subrut8.s

```
.macro  abs
  tst     r0, r0
```

```
        negmi   r0, r0
      .endm

.data
var1:   .asciz  "%d\n"

.text
.global main

/* Salvo registros */
main:   push    {r4, lr}

/* Primera llamada abs(1) */
        mov     r0, #1
        bl      abs

/* Imprimo primera llamada */
        mov     r1, r0
        ldr     r0, =var1
        bl      printf

/* Segunda llamada abs(-2) e imprimo */
        mov     r0, #-2
        bl      abs
        mov     r1, r0
        ldr     r0, =var1
        bl      printf

/* Tercera llamada abs(3) e imprimo */
        mov     r0, #3
        bl      abs
        mov     r1, r0
        ldr     r0, =var1
        bl      printf

/* Cuarta llamada abs(-4) e imprimo */
        mov     r0, #-4
        bl      abs
        mov     r1, r0
        ldr     r0, =var1
        bl      printf
        pop     {r4, lr}
        bx      lr
```

```
abs:    tst     r0, r0      @ comprueba el flag de signo
        negmi   r0, r0      @ si es negativo, negamos de nuevo
        bx      lr
```

Borra el `bl` antes del `abs` para probar la versión con macros. Dentro de `gdb` la secuencia de comandos para contar los pasos saltándose el `bl printf` junto con la cuenta es la siguiente.

```
start -> si 6 -> ni -> si 5 -> ni ->
-> si 5 -> ni -> si 5 -> ni -> si 2
```

$6 + 1 + (5 + 1) * 3 + 2 = 27$

Reescribe el ejercicio anterior de la media aritmética empleando macros en vez de funciones.

Conteo de ciclos

Completa la siguiente tabla usando los dos tipos de conteo que acabamos de explicar.

	Ciclos contados manualmente	Ciclos contados con `gdb`	Ciclos manualmente empleando macros	Ciclos, con `gdb` y macros
media				
abs				27

Este conteo de ciclos es ilustrativo. En un procesador real sólo las instrucciones simples tardan un ciclo de reloj, siempre y cuando el resultado de la operación no se utilice en la instrucción posterior, en cuyo caso la duración es de dos ciclos. Después hay instrucciones complejas como las multiplicaciones, que necesitan 3 ciclos (más si hay que añadir la penalización anterior). Por último están los casos más complejos. Por un lado tenemos los saltos condicionales, donde el procesador hace una predicción de salto dentro de las 2 posibilidades que hay, si se produce un fallo en la predicción se penalizan ciclos. Por otro lado están los accesos a memoria, que tampoco tienen una temporización constante porque está la caché por medio. Si se produce un fallo de caché hay que añadir la penalización correspondiente.

3.3.3. Algoritmo de ordenación

Escoge un algoritmo de ordenación de entre los 4 siguientes e impleméntalo en ensamblador:

- Burbuja.
- Selección.
- Inserción.
- Quicksort.

Como ejemplo mostramos el código en C del algoritmo de la burbuja.

Listado 3.15: Parte de subrut9.c

```
#include <stdio.h>

int vect[]= {8, 10, -3, 4, -5, 50, 2, 3};

void ordena(int* v, int len){
  int i, j, aux;

  for ( i= 1; i<len; i++ )
    for ( j= 0; j<len-i; j++ )
      if( v[j] > v[j+1] )
        aux= v[j],
        v[j]= v[j+1],
        v[j+1]= aux;
}

void main(void){
  int i;

  ordena(vect, 8);
  for ( i= 0; i<8; i++ )
    printf("%d\n", vect[i]);
}
```

La lista de algoritmos está ordenada por dificultad, por lo que el algoritmo `Quicksort` es con diferencia el más difícil de implementar. Recomendamos dejarlo para el final en caso de que el alumno decida realizar los 4 algoritmos en ensamblador.

Capítulo 4

E/S a bajo nivel

Contenido

Objetivos: Hasta ahora hemos programado en ensamblador sobre la capa que nos ofrece el sistema operativo. Nosotros llamamos a una función y ésta hace todo lo demás: le dice al sistema operativo lo que tiene que hacer con tal periférico y el sistema operativo (en concreto el kernel) le envía las órdenes directamente al periférico, al espacio de memoria donde esté mapeado el mismo.

Lo que vamos a hacer en este capítulo es comunicarnos directamente con los periféricos, para lo cual debemos prescindir totalmente del sistema operativo. Este modo de acceder directamente al hardware de la máquina se denomina `Bare Metal`,

que traducido viene a ser algo como `Metal desnudo`, haciendo referencia a que estamos ante la máquina tal y cómo es, sin ninguna capa de abstracción de por medio.

Veremos ejemplos de acceso directo a periféricos, en concreto al LED de la placa auxiliar (ver apéndice B) y a los temporizadores, que son bastante sencillos de manejar.

4.1. Lectura previa

4.1.1. Librerías y Kernel, las dos capas que queremos saltarnos

Anteriormente hemos utilizado funciones específicas para comunicarnos con los periféricos. Si por ejemplo necesitamos escribir en pantalla, llamamos a la función `printf`. Pues bien, entre la llamada a la función y lo que vemos en pantalla hay 2 capas software de por medio.

Una primera capa se encuentra en la librería runtime que acompaña al ejecutable, la cual incluye sólamente el fragmento de código de la función que necesitemos, en este caso en `printf`. El resto de funciones de la librería (`stdio`), si no las invocamos no aparecen en el ejecutable. El enlazador se encarga de todo esto, tanto de ubicar las funciones que llamemos desde ensamblador, como de poner la dirección numérica correcta que corresponda en la instrucción `bl printf`.

Este fragmento de código perteneciente a la primera capa sí que podemos depurarlo mediante `gdb`. Lo que hace es, a parte del formateo que realiza la propia función, trasladar al sistema operativo una determinada cadena para que éste lo muestre por pantalla. Es una especie de traductor intermedio que nos facilita las cosas. Nosotros desde ensamblador también podemos hacer llamadas al sistema directamente como veremos posteriormente.

La segunda capa va desde que hacemos la llamada al sistema (System Call o Syscall) hasta que se produce la transferencia de datos al periférico, retornando desde la llamada al sistema y volviendo a la primera capa, que a su vez retornará el control a la llamada a librería que hicimos en nuestro programa inicialmente.

En esta segunda capa se ejecuta código del kernel, el cual no podemos depurar. Además el procesador entra en un modo privilegiado, ya que en modo usuario (el que se ejecuta en nuestro programa ensamblador y dentro de la librería) no tenemos privilegios suficientes como para acceder a la zona de memoria que mapea los periféricos.

La función `printf` es una función de la librería del lenguaje C. Como vemos en la figura, esta función internamente llama a la System Call (rutina del Kernel del SO) `write` que es la que se ejecuta en modo supervisor y termina accediendo a los

periféricos (en este caso al terminal o pantalla donde aparece el mensaje). En la figura 4.1 podemos ver el código llamador junto con las dos capas.

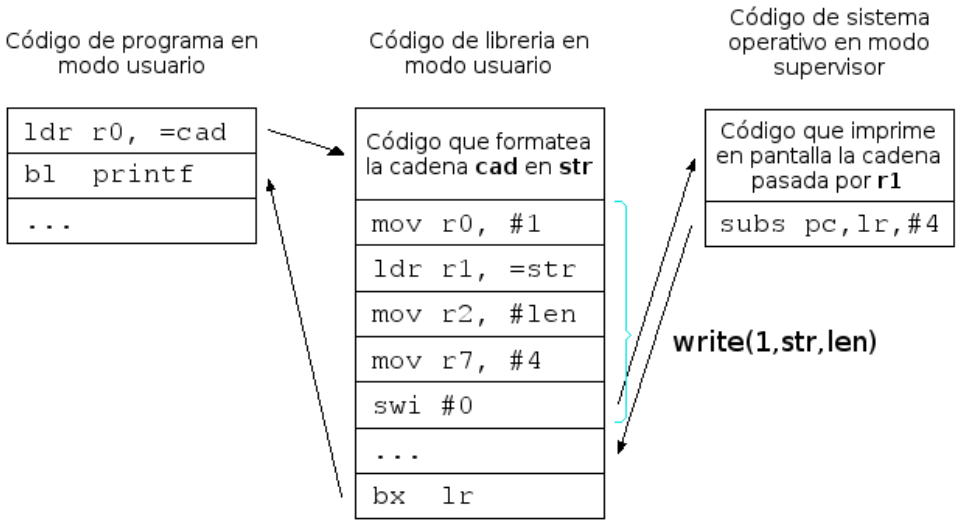

Figura 4.1: Funcionamiento de una llamada a printf

Ahora veremos un ejemplo en el cual nos saltamos la capa intermedia para comunicarnos directamente con el kernel vía llamada al sistema. En este ejemplo vamos a escribir una simple cadena por pantalla, en concreto "Hola Mundo!".

Listado 4.1: esbn1.s

```
.data

cadena: .asciz "Hola Mundo!\n"
cadenafin:

.text
.global main

main:   push    {r7, lr}            /* preservamos reg.*/
        mov     r0, #1              /* salida estándar */
        ldr     r1, =cadena         /* cadena a enviar */
        mov     r2, #cadenafin-cadena     /* longitud */
        mov     r7, #4              /* seleccionamos la*/
        swi     #0          /* llamada a sistema 'write'*/
        mov     r0, #0              /* devolvemos ok   */
        pop     {r7, lr}            /* recuperamos reg.*/
        bx      lr                  /* salimos de main */
```

La instrucción que ejecuta la llamada al sistema es `swi #0`, siempre tendrá cero como valor inmediato. El código numérico de la llamada y el número de parámetros podemos buscarlo en cualquier manual de Linux, buscando "Linux system call table" en Google. En nuestro caso la llamada `write` se corresponde con el código 4 y acepta tres parámetros: manejador de fichero, dirección de los datos a escribir (nuestra cadena) y longitud de los datos. En nuestro ejemplo, el manejador de fichero es el `1`, que está conectado con la salida estándar o lo que es lo mismo, con la pantalla.

En general se tiende a usar una lista reducida de posibles llamadas a sistema, y que éstas sean lo más polivalentes posibles. En este caso vemos que no existe una función específica para escribir en pantalla. Lo que hacemos es escribir bytes en un fichero, pero usando un manejador especial conocido como salida estándar, con lo cual todo lo que escribamos a este fichero especial aparecerá por pantalla.

Pero el propósito de este capítulo no es saltarnos una capa para comunicarnos directamente con el sistema operativo. Lo que queremos es saltarnos las dos capas y enviarle órdenes directamente a los periféricos. Para esto tenemos prescindir del sistema operativo, o lo que es lo mismo, hacer nosotros de sistema operativo para realizar las tareas que queramos.

Este modo de trabajar (como hemos adelantado) se denomina Bare Metal, porque accedemos a las entrañas del hardware. En él podemos hacer desde cosas muy sencillas como encender un LED hasta programar desde cero nuestro propio sistema operativo.

4.1.2. Ejecutar código en Bare Metal

El ciclo de ensamblado y enlazado es distinto en un programa Bare Metal. Hasta ahora hemos creado ejecutables, que tienen una estructura más compleja, con cabecera y distintas secciones en formato `ELF` [8]. Toda esta información le viene muy bien al sistema operativo, pero en un entorno Bare Metal no disponemos de él. Lo que se carga en `kernel.img` es un binario sencillo, sin cabecera, que contiene directamente el código máquina de nuestro programa y que se cargará en la dirección de RAM `0x8000`.

Lo que para un ejecutable hacíamos con esta secuencia.

```
as -o ejemplo.o ejemplo.s
gcc -o ejemplo ejemplo.o
```

En caso de un programa Bare Metal tenemos que cambiarla por esta otra.

```
as -o ejemplo.o ejemplo.s
ld -e 0 -Ttext=0x8000 -o ejemplo.elf ejemplo.o
objcopy ejemplo.elf -O binary kernel.img
```

Otra característica de Bare Metal es que sólo tenemos una sección de código (la sección .text), y no estamos obligados a crear la función `main`. Al no ejecutar ninguna función no tenemos la posibilidad de salir del programa con `bx lr`, al fin y al cabo no hay ningún sistema operativo detrás al que regresar. Nuestro programa debe trabajar en bucle cerrado. En caso de tener una tarea simple que queramos terminar, es preferible dejar el sistema colgado con un bucle infinito como última instrucción.

El proceso de arranque de la Raspberry Pi es el siguiente:

- Cuando la encendemos, el núcleo ARM está desactivado. Lo primero que se activa es el núcleo GPU, que es un procesador totalmente distinto e independiente al ARM. En este momento la SDRAM está desactivada.
- El procesador GPU empieza a ejecutar la primera etapa del bootloader (son 3 etapas), que está almacenada en ROM dentro del mismo chip que comparten ARM y GPU. Esta primera etapa accede a la tarjeta SD y lee el fichero `bootcode.bin` en caché L2 y lo ejecuta, siendo el código de `bootcode.bin` la segunda etapa del bootloader.
- En la segunda etapa se activa la SDRAM y se carga la tercera parte del bootloader, cuyo código está repartido entre `loader.bin` (opcional) y `start.elf`.
- En tercera y última etapa del bootloader se accede opcionalmente a dos archivos ASCII de configuración llamados `config.txt` y `cmdline.txt`. Lo más relevante de esta etapa es que cargamos en RAM (en concreto en la dirección 0x8000) el archivo `kernel.img` con código ARM, para luego ejecutarlo y acabar con el bootloader, pasando el control desde la GPU hacia la CPU. Este último archivo es el que nos interesa modificar para nuestros propósitos, ya que es lo primero que la CPU ejecuta y lo hace en modo privilegiado, es decir, con acceso total al hardware.

De todos estos archivos los obligatorios son `bootcode.bin`, `start.elf` y `kernel.img`. Los dos primeros los bajamos del repositorio oficial https://github.com/raspberrypi y el tercero `kernel.img` es el que nosotros vamos a generar. Estos tres archivos deben estar en el directorio raíz de la primera partición de la tarjeta SD, la cual debe estar formateada en FAT32.

El proceso completo que debemos repetir cada vez que desarrollemos un programa nuevo en Bare Metal es el siguiente:

- Apagamos la Raspberry.
- Extraemos la tarjeta SD.

- Introducimos la SD en el lector de nuestro ordenador de desarrollo.
- Montamos la unidad y copiamos (sobreescribimos) el kernel.img que acabamos de desarrollar.
- Desmontamos y extraemos la SD.
- Insertamos de nuevo la SD en la Raspberry y la encendemos.

Es un proceso sencillo para las prácticas que vamos a hacer, pero para proyectos más largos se vuelve bastante tedioso. Hay varias alternativas que agilizan el ciclo de trabajo, donde no es necesario extraer la SD y por tanto podemos actualizar el `kernel.img` en cuestión de segundos. Estas alternativas son:

- Cable JTAG con software Openocd: http://openocd.sourceforge.net
- Cable USB-serie desde el ordenador de desarrollo hacia la Raspberry, requiere tener instaladas las herramientas de compilación cruzada en el ordenador de desarrollo.
- Cable serie-serie que comunica dos Raspberries, una orientada a desarrollo y la otra para ejecutar los programas en Bare Metal. No es imprescindible trabajar directamente con la Raspberry de desarrollo, podemos acceder vía ssh con nuestro ordenador habitual, sin necesidad de tener instaladas las herramientas de compilación en el mismo.

Las dos últimas opciones están detalladas en el apéndice C. Básicamente se trata de meter en el kernel.img de la SD un programa especial (llamado bootloader) que lee continuamente del puerto serie y en el momento en que recibe un archivo del tipo `kernel.img`, lo carga en RAM y lo ejecuta.

4.2. Acceso a periféricos

Los periféricos se controlan leyendo y escribiendo datos a los registros asociados o puertos de E/S. No confundir estos registros con los registros de la CPU. Un puerto asociado a un periférico es un ente, normalmente del mismo tamaño que el ancho del bus de datos, que sirve para configurar diferentes aspectos del mismo. No se trata de RAM, por lo que no se garantiza que al leer de un puerto obtengamos el último valor que escribimos. Es más, incluso hay puertos que sólo admiten ser leídos y otros que sólo admiten escrituras. La funcionalidad de los puertos también es muy variable, incluso dentro de un mismo puerto los diferentes bits del mismo tienen distinto comportamiento.

Como cada periférico se controla de una forma diferente, no hay más remedio que leerse el datasheet del mismo si queremos trabajar con él. De ahora en adelante usaremos una placa auxiliar, descrita en el apéndice B, y que conectaremos a la fila inferior del conector GPIO según la figura 4.2. En esta sección explicaremos cómo encender un LED de esta placa auxiliar.

Figura 4.2: Colocación de la placa auxiliar

4.2.1. GPIO (General-Purpose Input/Output)

El GPIO es un conjunto de señales mediante las cuales la CPU se comunica con distintas partes de la Rasberry tanto internamente (audio analógico, tarjeta SD o LEDs internos) como externamente a través de los conectores P1 y P5. Como la mayor parte de las señales se encuentran en el conector P1 (ver figura 4.3), normalmente este conector se denomina GPIO. Nosotros no vamos a trabajar con señales GPIO que no pertenezcan a dicho conector, por lo que no habrá confusiones.

El GPIO contiene en total 54 señales, de las cuales 17 están disponibles a través del conector GPIO (26 en los modelos A+/B+). Como nuestra placa auxiliar emplea la fila inferior del conector, sólo dispondremos de 9 señales.

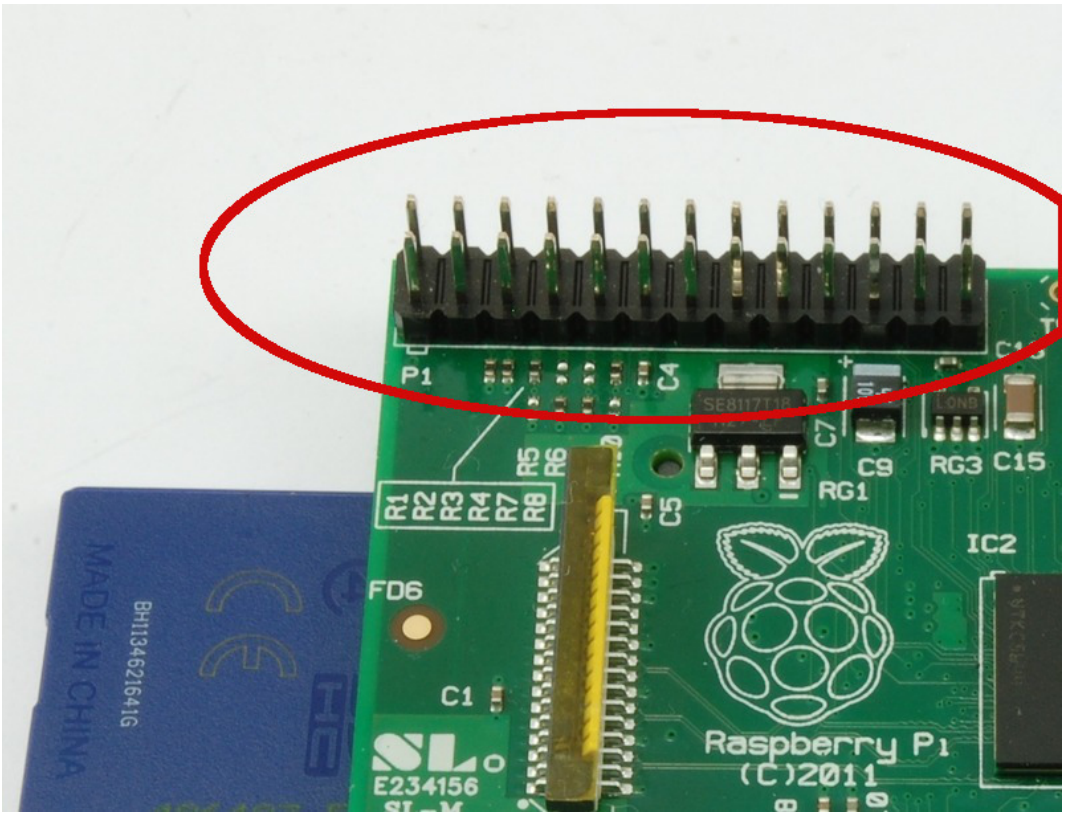

Figura 4.3: Posición del puerto GPIO

Los puertos del GPIO están mapeados en memoria, tomando como base la dirección 0x20200000. Para nuestros propósitos de esta lección nos basta con acceder a los puertos GPFSELn, GPSETn y GPCLRn. A continuación tenemos la tabla con las direcciones de estos puertos.

GPFSELn

Las 54 señales/pines las separamos en 6 grupos funcionales de 10 señales/pines cada uno (excepto el último que es de 4) para programarlas mediante GPFSELn.

El LED que queremos controlar se corresponde con la señal número 9 del puerto GPIO. Se nombran con GPIO más el número correspondiente, en nuestro caso sería GPIO 9. Nótese que la numeración empieza en 0, desde GPIO 0 hasta GPIO 53.

Así que la funcionalidad desde GPIO 0 hasta GPIO 9 se controla con GPFSEL0, desde GPIO 10 hasta GPIO 19 se hace con GPFSEL1 y así sucesivamente. Nosotros queremos encender el primer LED rojo de la placa auxiliar. En la figura B.3 vemos que el primer LED rojo se corresponde con GPIO 9. Para cambiar la funcionalidad de GPIO 9 nos toca actuar sobre GPFSEL0. Por defecto cuando arranca la Raspberry

Dirección	Nombre	Descripción	Tipo
20200000	GPFSEL0	Selector de función 0	R/W
20200004	GPFSEL1	Selector de función 1	R/W
20200008	GPFSEL2	Selector de función 2	R/W
2020000C	GPFSEL3	Selector de función 3	R/W
20200010	GPFSEL4	Selector de función 4	R/W
20200014	GPFSEL5	Selector de función 5	R/W
2020001C	GPSET0	Pin a nivel alto 0	W
20200020	GPSET1	Pin a nivel alto 1	W
20200028	GPCLR0	Pin a nivel bajo 0	W
2020002C	GPCLR1	Pin a nivel bajo 1	W

todos los pines están preconfigurados como entradas, con lo que los LEDs de nuestra placa auxiliar están apagados. Es más, aunque lo configuremos como salida, tras el reset, los pines se inicializan al valor cero (nivel bajo), por lo que podemos presuponer que todos los LEDs estarán apagados, incluso después de programarlos como salidas.

El puerto GPFSEL0 contiene diez grupos funcionales llamados FSELx (del 0 al 9) de 3 bits cada uno, quedando los dos bits más altos sin usar. Nos interesa cambiar FSEL9, que sería el que se corresponde con el primer LED rojo, el que queremos encender. Las posibles configuraciones para cada grupo son:

```
000 = GPIO Pin X es una entrada
001 = GPIO Pin X es una salida
100 = GPIO Pin X toma función alternativa 0
101 = GPIO Pin X toma función alternativa 1
110 = GPIO Pin X toma función alternativa 2
111 = GPIO Pin X toma función alternativa 3
011 = GPIO Pin X toma función alternativa 4
010 = GPIO Pin X toma función alternativa 5
```

Las funciones alternativas son para dotar a los pines de funcionalidad específicas como puertos SPI, UART, audio PCM y cosas parecidas. La lista completa está en la tabla 6-31 (página 102) del datasheet [4]. Nosotros queremos una salida genérica, así que nos quedamos con el código `001` para el grupo funcional FSEL9 del puerto `GPFSEL0` que es el que corresponde al GPIO 9.

GPSETn y GPCLRn

Los 54 pines se reparten entre dos puertos GPSET0/GPCLR0, que contienen los 32 primeros, y en GPSET1/GPCLR1 están los 22 restantes, quedando libres los 10 bits más significativos de GPSET1/GPCLR1.

Una vez configurado GPIO 9 como salida, ya sólo queda saber cómo poner un

cero o un uno en la señal GPIO 9, para apagar y encender el primer LED de la placa auxiliar respectivamente (un cero apaga y un uno enciende el LED).

Para ello tenemos los puertos GPSETn y GPCLRn, donde GPSETn pone un 1 y GPCLRn pone un 0. En principio parece enrevesado el tener que usar dos puertos distintos para escribir en el puerto GPIO, pero no olvidemos que para ahorrar recursos varios pines están empaquetados en una palabra de 32 bits. Si sólo tuviéramos un puerto y quisiéramos alterar un único pin tendríamos que leer el puerto, modificar el bit en cuestión sin tocar los demás y escribir el resultado de nuevo en el puerto. Por suerte esto no es necesario con puertos separados para setear y resetear, tan sólo necesitamos una escritura en puerto poniendo a 1 los bits que queramos setear/resetear y a 0 los bits que no queramos modificar.

En la figura 4.4 vemos cómo está hecho el conexionado de la placa auxiliar.

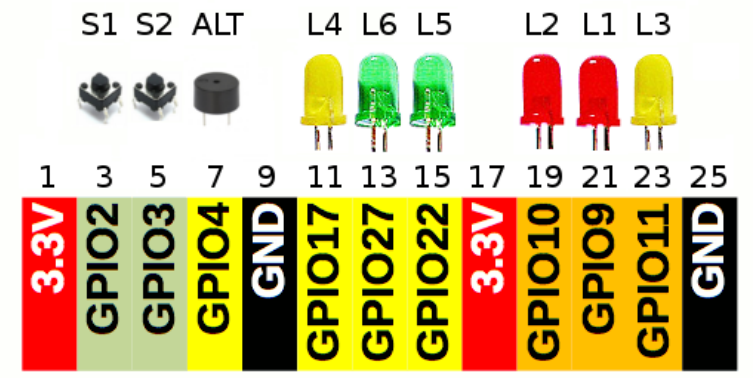

Figura 4.4: Correspondencia LEDs y GPIO

En nuestro primer ejemplo de Bare Metal sólo vamos a encender el primer LED rojo de la placa auxiliar, que como hemos dicho se corresponde con el GPIO 9 así que tendremos que actuar sobre el bit 9 del registro `GPSET0`.

Resumiendo, los puertos a los que accedemos para encender y apagar el LED vienen indicados en la figura 4.5.

El siguiente código (listado 4.2) muestra cómo hemos de proceder.

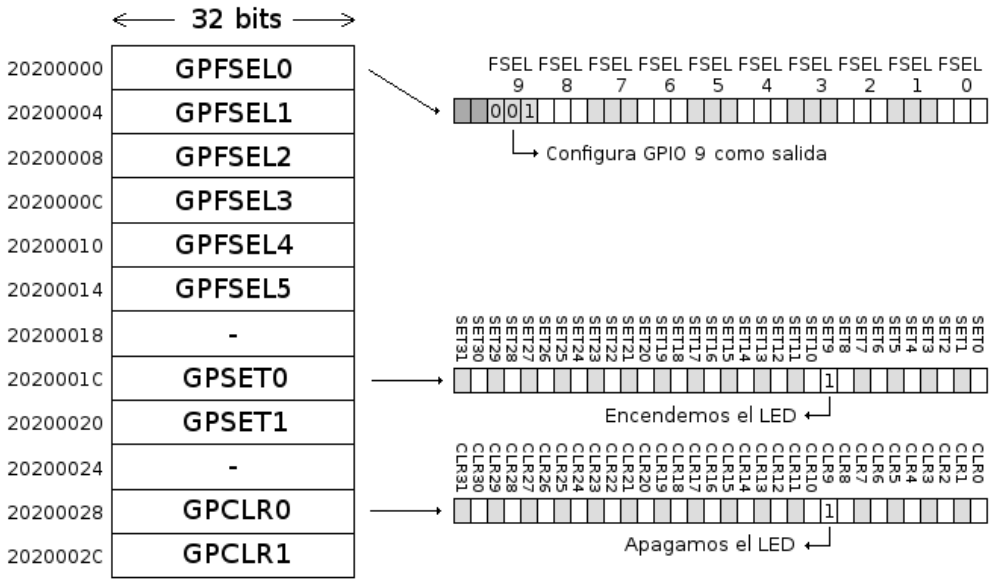

Figura 4.5: Puertos LED

Listado 4.2: esbn2.s

```
        .set    GPBASE,   0x20200000
        .set    GPFSEL0,  0x00
        .set    GPSET0,   0x1c
.text
        ldr     r0, =GPBASE
/* guia bits          xx99988877766655544433322211000*/
        mov     r1, #0b00001000000000000000000000000000
        str     r1, [r0, #GPFSEL0]  @ Configura GPIO 9
/* guia bits          10987654321098765432109876543210*/
        mov     r1, #0b00000000000000000000001000000000
        str     r1, [r0, #GPSET0]   @ Enciende GPIO 9
infi:   b       infi
```

El acceso a los puertos lo hemos hecho usando la dirección base donde están mapeados los periféricos `0x20200000`. Cargamos esta dirección base en el registro `r0` y codificamos los accesos a los puertos E/S con direccionamiento a memoria empleando distintas constantes como desplazamiento en función del puerto al que queramos acceder.

El código simplemente escribe dos constantes en dos puertos: `GPFSEL0` y `GPSET0`. Con la primera escritura configuramos el LED como salida y con la segunda escritura lo encendemos, para finalmente entrar en un bucle infinito con `infi: b infi`.

Otros puertos

Ya hemos explicado los puertos que vamos a usar en este capítulo, pero el dispositivo GPIO tiene más puertos.

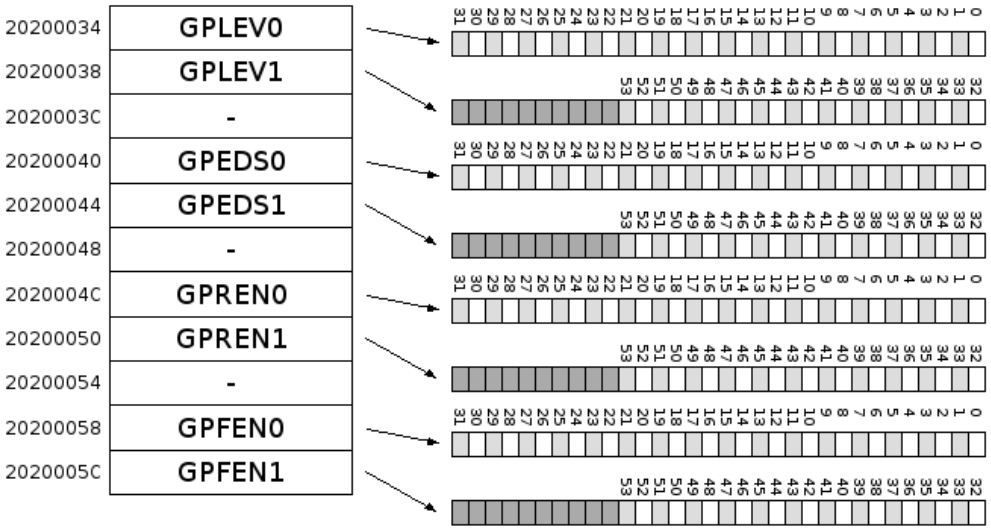

Figura 4.6: Otros puertos del GPIO (1ª parte)

En la figura 4.6 tenemos los siguientes:

- `GPLEVn`. Estos puertos devuelven el valor del pin respectivo. Si dicho pin está en torno a 0V devolverá un cero, si está en torno a 3.3V devolverá un 1.
- `GPEDSn`. Sirven para detectar qué pin ha provocado una interrupción en caso de usarlo como lectura. Al escribir en ellos también podemos notificar que ya hemos procesado la interrupción y que por tanto estamos listos para que nos vuelvan a interrumpir sobre los pines que indiquemos.
- `GPRENn`. Con estos puertos enmascaramos los pines que queremos que provoquen una interrupción en flanco de subida, esto es cuando hay una transición de 0 a 1 en el pin de entrada.
- `GPFENn`. Lo mismo que el anterior pero en flanco de bajada.

El resto de puertos GPIO se muestran en la figura 4.7.
Estos registros son los siguientes:

- `GPHENn`. Enmascaramos los pines que provocarán una interrupción al detectar un nivel alto (3.3V) por dicho pin.

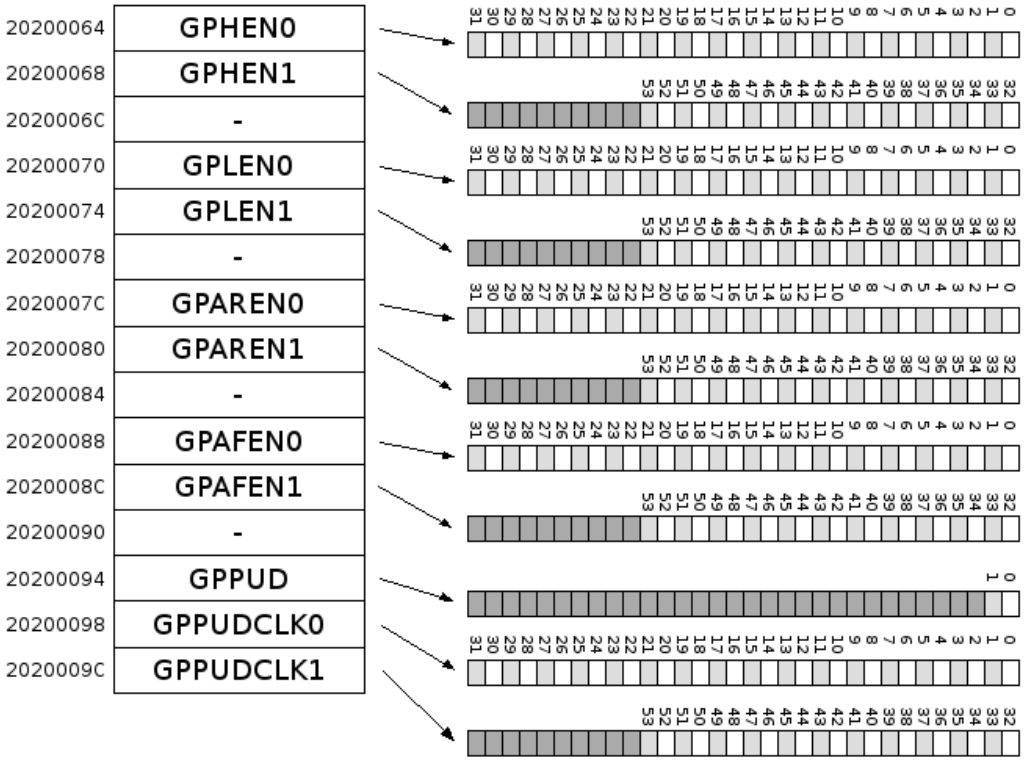

Figura 4.7: Otros puertos del GPIO (2ª parte)

- `GPLENn`. Lo mismo que el anterior pero para un nivel bajo (0V).
- `GPARENn` y `GPAFENn`. Tienen funciones idénticas a `GPRENn` y `GPFENn`, pero permiten detectar flancos en pulsos de poca duración.
- `GPPUD` y `GPPUDCLKn`. Conectan resistencias de pull-up y de pull-down sobre los pines que deseemos. Para más información ver el último ejemplo del siguiente capítulo.

4.2.2. Temporizador del sistema

El temporizador del sistema es un reloj que funciona a 1MHz y en cada paso incrementa un contador de 64bits. Este contador viene muy bien para implementar retardos o esperas porque cada paso del contador se corresponde con un microsegundo. Los puertos asociados al temporizador son los de la figura 4.8. Básicamente encontramos un contador de 64 bits y cuatro comparadores. El contador está dividido en dos partes, la parte baja `CLO` y la parte alta `CHI`. La parte alta no nos resulta

interesante, porque tarda poco más de una hora (2^{32} µs) en incrementarse y no va asociado a ningún comparador.

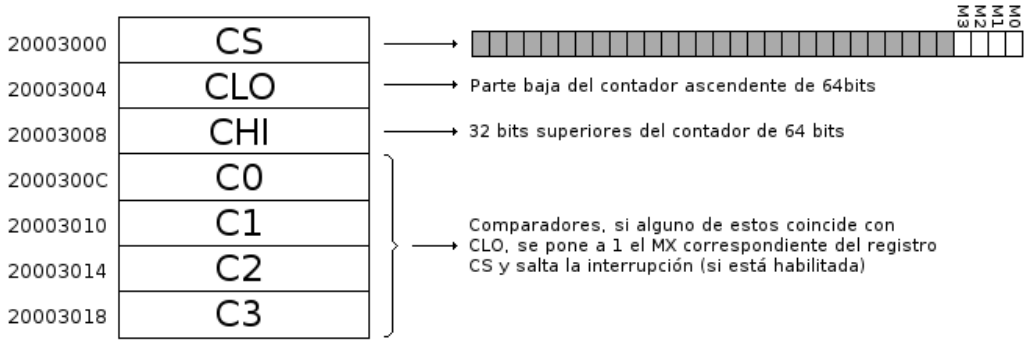

Figura 4.8: System Timer

Los comparadores son puertos que se pueden modificar y se comparan con `CLO`. En el momento que uno de los 4 comparadores coincida y estén habilitadas las interrupciones para dicho comparador, se produce una interrupción y se activa el correspondiente bit `Mx` asociado al puerto `CS` (para que en la rutina de tratamiento de interrupción o RTI sepamos qué comparador ha provocado la interrupción). Los comparadores `C0` y `C2` los emplea la GPU internamente, por lo que nosotros nos ceñiremos a los comparadores `C1` y `C3`.

Las interrupciones las veremos en la siguiente lección. Por ahora sólo vamos a acceder al puerto `CLO` para hacer parpadear un LED a una frecuencia determinada. El esquema funcional del `System Timer` se muestra en la figura 4.9.

4.3. Ejemplos de programas Bare Metal

4.3.1. LED parpadeante con bucle de retardo

La teoría sobre encender y apagar el LED la sabemos. Lo más sencillo que podemos hacer ahora es hacer que el LED parpadee continuamente. Vamos a intruducir el siguiente programa en la Raspberry, antes de probarlo piensa un poco cómo se comportaría el código del listado 4.3.

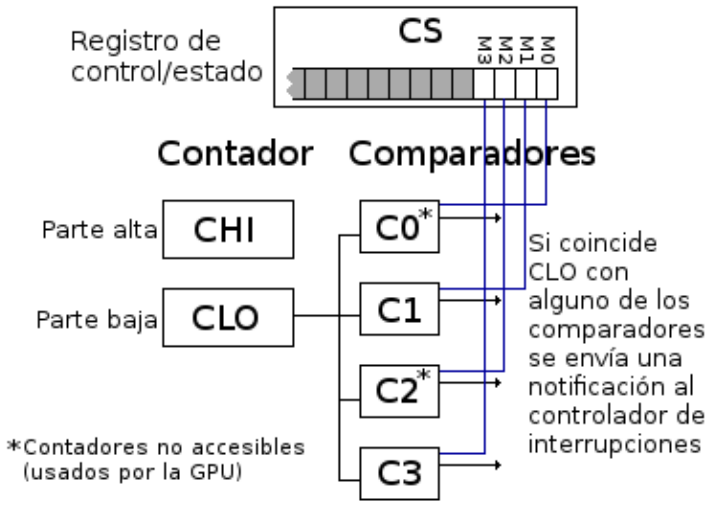

Figura 4.9: Esquema funcional del System Timer

Listado 4.3: esbn3.s

```
        .set    GPBASE,   0x20200000
        .set    GPFSEL0,  0x00
        .set    GPSET0,   0x1c
        .set    GPCLR0,   0x28
.text
        ldr     r0, =GPBASE
/* guia bits          xx99988877766655544433322211000*/
        mov     r1, #0b00001000000000000000000000000000
        str     r1, [r0, #GPFSEL0]  @ Configura como salida
/* guia bits          10987654321098765432109876543210*/
bucle:  mov     r1, #0b00000000000000000000001000000000
        str     r1, [r0, #GPSET0]   @ Enciende
        mov     r1, #0b00000000000000000000001000000000
        str     r1, [r0, #GPCLR0]   @ Apaga
        b       bucle
```

Para compilar y ejecutar este ejemplo sigue los pasos descritos en 4.1.2. Al ejecutar el kernel.img resultante comprobamos que el LED no parpadea sino que está encendido con menos brillo del normal. En realidad sí que lo hace, sólo que nuestro ojo es demasiado lento como para percibirlo. Lo siguiente será ajustar la cadencia del parpadeo a un segundo para que podamos observar el parpadeo. La secuencia sería apagar el LED, esperar medio segundo, encender el LED, esperar otro medio segundo y repetir el bucle. Sabemos que el procesador de la Raspberry corre a 700MHz por lo que vamos a suponer que tarde un ciclo de este reloj en ejecutar cada

instrucción. En base a esto vamos a crear dos bucles de retardo: uno tras apagar el LED y otro tras encenderlo de 500ms cada uno. Un bucle de retardo lo único que hace es esperar tiempo sin hacer realmente nada.

Si suponemos que cada instrucción consume un ciclo y teniendo en cuenta que el bucle de retardo tiene 2 instrucciones, cada iteración del bucle consume 2 ciclos. A 700 MHz (7×10^8 ciclos/segundo) un ciclo consume $1/(7\times10^8)$ segundos que es igual a $1{,}42\times10^{-9}s$ (aproximadamente 1,5 ns). Así que cada iteración en principio consume 3 ns y para consumir 500 ns necesitamos $500\times10^{-3}/(3\times10^{-9}) = 166{,}66\times10^6$, es decir más de 166 millones de iteraciones.

Si usamos ese número de iteraciones observaremos como la cadencia del LED es más lenta de lo esperado, lo que quiere decir que cada iteración del bucle de retardo tarda más de los dos ciclos que hemos supuesto. Probamos con cronómetro en mano distintos valores para las constantes hasta comprobar que con 7 millones de iteraciones del bucle se consigue más o menos el medio segundo buscado. Haciendo cuentas nos salen 50 ciclos por iteracción, bastante más de los 2 ciclos esperados. Esto se debe a una dependencia de datos (ya que el flag que altera la orden `subs` es requerido justo después por la instrucción `bne`) y que los saltos condicionales suelen ser lentos.

Listado 4.4: Parte de esbn4.s

```
        .set    GPBASE,   0x20200000
        .set    GPFSEL0,  0x00
        .set    GPSET0,   0x1c
        .set    GPCLR0,   0x28
.text
        ldr     r0, =GPBASE
/* guia bits          xx99988877766655544433322211000*/
        mov     r1, #0b00001000000000000000000000000000
        str     r1, [r0, #GPFSEL0]  @ Configura GPIO 9
/* guia bits          10987654321098765432109876543210*/
        mov     r1, #0b00000000000000000000001000000000

bucle:  ldr     r2, =7000000
ret1:   subs    r2, #1              @ Bucle de retardo 1
        bne     ret1
        str     r1, [r0, #GPSET0]   @ Enciende el LED
        ldr     r2, =7000000
ret2:   subs    r2, #1              @ Bucle de retardo 2
        bne     ret2
        str     r1, [r0, #GPCLR0]   @ Enciende el LED

        b       bucle               @ Repetir para siempre
```

4.3.2. LED parpadeante con temporizador

Viendo lo poco preciso que es el temporizar con el bucle de retardo, vamos a sincronizar leyendo continuamente el valor del `System Timer`. Como el temporizador va a 1MHz, para temporizar medio segundo lo único que tenemos que hacer es esperar a que el contador se incremente en medio millón. El código final quedaría así:

Listado 4.5: esbn5.s

```
        .set    GPBASE,   0x20200000
        .set    GPFSEL0,        0x00
        .set    GPSET0,         0x1c
        .set    GPCLR0,         0x28
        .set    STBASE,   0x20003000
        .set    STCLO,          0x04
.text
        ldr     r0, =GPBASE
/* guia bits          xx99988877766655544433322211000*/
        mov     r1, #0b00001000000000000000000000000000
        str     r1, [r0, #GPFSEL0]  @ Configura GPIO 9
/* guia bits          10987654321098765432109876543210*/
        mov     r1, #0b00000000000000000000001000000000
        ldr     r2, =STBASE

bucle:  bl      espera        @ Salta a rutina de espera
        str     r1, [r0, #GPSET0]
        bl      espera        @ Salta a rutina de espera
        str     r1, [r0, #GPCLR0]
        b       bucle

/* rutina que espera medio segundo */
espera: ldr     r3, [r2, #STCLO]  @ Lee contador en r3
        ldr     r4, =500000
        add     r4, r3            @ r4= r3+medio millón
ret1:   ldr     r3, [r2, #STCLO]
        cmp     r3, r4            @ Leemos CLO hasta alcanzar
        bne     ret1              @ el valor de r4
        bx      lr
```

4.3.3. Sonido con temporizador

Este ejemplo es exactamente el mismo que el anterior, tan sólo hemos cambiado el pin del LED (GPIO 9) por el pin asociado al altavoz de nuestra placa de expan-

sión (GPIO 4). También modificamos el tiempo de espera para producir un sonido audible.

Vamos a producir un tono de 440 Hz. Para ello generamos una onda cuadrada por dicho pin, que no es más que una serie de ceros y unos consecutivos de idéntica duración. A esta duración la llamamos semi-periodo, y es la que queremos calcular. Como el periodo es el inverso de la frecuencia, tenemos que $periodo = 1/(440s^{-1}) = 2{,}272 \times 10^{-3}s$, por lo que el semi-periodo buscado es $2{,}272 \times 10^{-3}s/2 = 1{,}136 \times 10^{-3}s$ o lo que es lo mismo, 1136 microsegundos.

Listado 4.6: Parte de esbn6.s

```
        ldr     r0, =GPBASE
/* guia bits           xx99988877766655544433322211000*/
        mov     r1, #0b00000000000000000001000000000000
        str     r1, [r0, #GPFSEL0]  @ Configura GPIO 4
/* guia bits           10987654321098765432109876543210*/
        mov     r1, #0b00000000000000000000000000010000
        ldr     r2, =STBASE

bucle:  bl      espera          @ Salta a rutina de espera
        str     r1, [r0, #GPSET0]
        bl      espera          @ Salta a rutina de espera
        str     r1, [r0, #GPCLR0]
        b       bucle

/* rutina que espera 1136 microsegundos */
espera: ldr     r3, [r2, #STCLO]  @ Lee contador en r3
        ldr     r4, =1136
        add     r4, r3            @ r4= r3 + 1136
ret1:   ldr     r3, [r2, #STCLO]
        cmp     r3, r4            @ Leemos CLO hasta alcanzar
        bne     ret1              @ el valor de r4
        bx      lr
```

4.4. Ejercicios

4.4.1. Cadencia variable con bucle de retardo

Usando la técnica del bucle de retardo haz que el LED parpadee cada vez más rápido, hasta que la cadencia sea de 1/4 de segundo. Una vez llegues a esta cadencia salta de golpe a la cadencia original de 1 segundo. El tiempo que se tarda en pasar de una cadencia a otra puede ser el que quieras, siempre que sea suficiente para poder apreciar el efecto.

4.4.2. Cadencia variable con temporizador

Repite el ejercicio anterior pero empleando el temporizador interno. Durante los 10 primeros segundos aumentamos la cadencia del LED desde 1 segundo hasta los 250ms, y en los últimos 10 segundos disminuimos la cadencia al mismo ritmo de tal forma que el ciclo completo se repite cada 20 segundos.

4.4.3. Escala musical

Escribe un programa que haga sonar el altavoz con las notas Do, Mi y Sol (de la quinta octava) durante tres segundos cada una de ellas. Las frecuencias de estas notas son:

Nota	**Frecuencia**
Do	523 Hz
Mi	659 Hz
Sol	784 Hz

Capítulo 5

Interrupciones hardware

Contenido

Objetivos: En esta sesión vamos a realizar programas que utilizan dispositivos de E/S haciendo uso del sistema de interrupciones hardware. Para poder programar los distintos parámetros que configuran el entorno de las interrupciones es necesario

conocer de forma detallada cómo funcionan los puertos asociados, ya que éste es el mecanismo típico mediante el cual el procesador se comunica con los periféricos.

Hacemos incapié en lo de *hardware* porque las *interrupciones software* no son más que las llamadas a sistema que vimos en el capítulo anterior. Ambas comparten vector de interrupciones, pero las *interrupciones software* son más bien llamadas a subrutinas.

5.1. Lectura previa

El microprocesador se encuentra en un entorno donde existen otros componentes. La forma de comunicación más usual entre estos componentes y el microprocesador se denomina interrupción. Básicamente, una interrupción es una petición que se hace a la CPU para que detenga temporalmente el trabajo que esté realizando y ejecute una rutina determinada.

5.1.1. El sistema de interrupciones del ARM

Decimos que las interrupciones del ARM son **autovectorizadas**. Cada tipo de interrupción lleva asociado un número (que llamamos **número de interrupción**, NI) que identifica el tipo de servicio a realizar. En total hay 8 tipos de interrupciones. A partir de dicho número se calcula la dirección a la que salta la CPU para atender dicha interrupción. A diferencia de otras arquitecturas donde los vectores contienen las direcciones de las rutinas de tratamiento, en ARM no tenemos direcciones sino instrucciones. Cada vector contiene normalmente un salto a la rutina de tratamiento correspondiente. Dicha rutina se suele llamar RTI (**Rutina de Tratamiento de Interrupción**). En la arquitectura ARMv6 todos los vectores de interrupción se almacenan en una zona de memoria llamada **tabla de vectores de interrupción**. Esta tabla comienza en la dirección física 0x00000000 (aunque puede cambiarse por 0xffff0000) y acaba en 0x0000001f y contiene en total 8 vectores de interrupción. Cuando termina de ejecutarse una RTI, el procesador continúa ejecutando la instrucción siguiente a la que se estaba ejecutando cuando se produjo la interrupción.

Existen dos tipos de interrupciones: hardware y software. Las **interrupciones hardware** son aquellas en las que su activación está condicionada por el hardware del sistema, ya sea por: 1) *excepciones* provocadas en la ejecución de alguna instrucción o error grave, o 2) provocadas por la placa base o por cualquier tarjeta implicada en un canal de E/S.
La lista del vector de interrupciones es la siguiente.

Excepción	**Tipo**	**Desplaz.**	**Modo**
Reset	Interrupción	`0x00`	Supervisor
Instrucción no definida	Excepción	`0x04`	Indefinido
Interrupción software	Int. software	`0x08`	Supervisor
Error en prefetch	Excepción	`0x0C`	Abort
Error en datos	Excepción	`0x10`	Abort
Reservado	-	`0x14`	Reservado
IRQ	Interrupción	`0x18`	IRQ
FIQ	Interrupción	`0x1C`	FIQ

Tabla 5.1: Vector de interrupciones

La última columna se refiere al *Modo de operación* que comentamos en el primer capítulo y que forma parte del registro `cpsr` (ver figura 5.1). Es un estado en el que se encuentra el procesador con una serie de privilegios con respecto a otros modos y que gracias a ellos podemos construir un sistema operativo con diferentes capas.

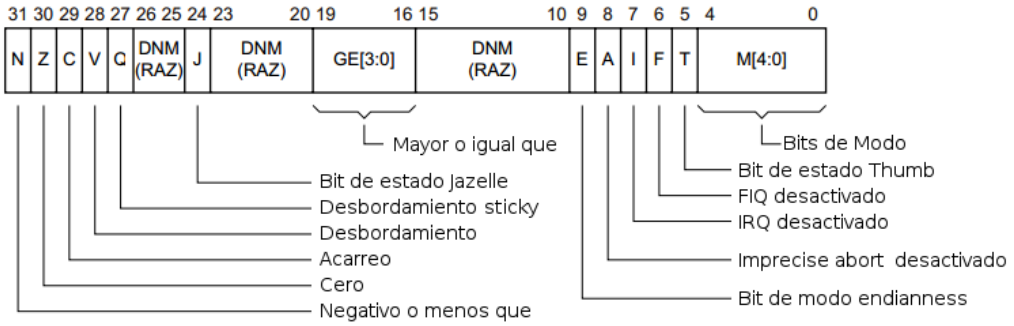

Figura 5.1: Registro cpsr

Cada modo tiene sus propios registros `sp`, `lr` y `spsr` (Saved Program Status Register) de tal forma que no alteramos la pila ni los flags de la secuencia de programa que interrumpimos. Incluso el modo `FIQ` tiene 5 registros generales propios (desde `r8` hasta `r12`), de esta forma si los empleamos en nuestra rutina de tratamiento no tendremos que salvaguardarlos en pila. En la figura 5.2 observamos los registros propios mencionados marcados con un triángulo.

- *Reset* es la excepción que permitiría un reset en caliente dentro de la CPU. Desgraciadamente en la Raspberry no hay forma conocida de forzar esta excepción. Los pines del conector `P6` en la Raspberry 2.0 y `RUN` en el modelo B+ (ver figura C.3) provocan una secuencia completa de arranque, la que comien-

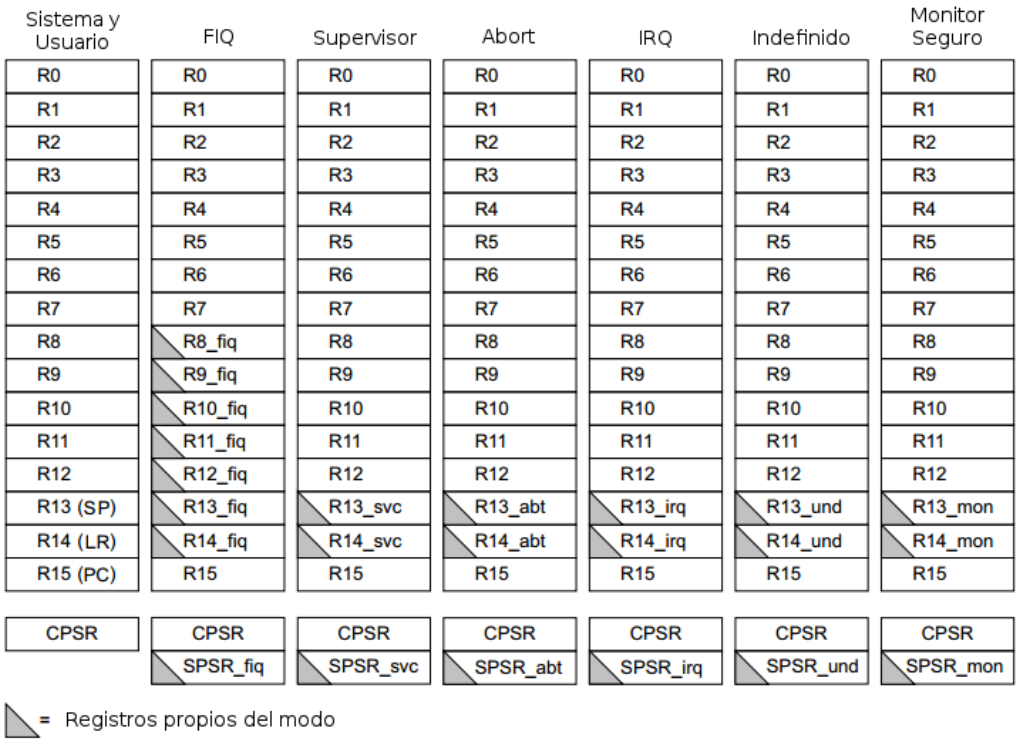

Figura 5.2: Registros según modo de operación

za con el bootloader de la GPU y acaba cediendo el control a la CPU en la dirección 0x8000.

- *Instrucción no definida* se produce cuando en el flujo de instrucciones nos encontramos un código de operación que no se corresponde con ninguna instrucción. Normalmente esto se produce por una corrupción en la memoria de programa o bien que hemos saltado erróneamente a una zona donde hay datos. También se puede dar el caso de que intentemos ejecutar código ARM para una plataforma más moderna y nos encontremos con una instrucción no soportada por el procesador que tenemos.

- Las *interrupciones software* son subrutinas que se incluyen en el sistema operativo y que son llamadas desde el programa ejecutando la instrucción `swi #n`. No interrumpen realmente nada, pero se denominan así porque el mecanismo de funcionamiento es el mismo que en las hardware.

- Luego tenemos los errores en *prefetch* y *datos*. Un error de *prefetch* se produce cuando tratamos de ejecutar una instrucción que momentos antes hemos modificado. Es poco frecuente y para que se produzca debemos escribir código

que sea automodificable, que es una práctica no deseable y apenas utilizada en dicha arquitectura. Los errores de *datos* son generados normalmente por el manejador de memoria y responden a fallos de alineación, de traslación, de dominio o de permisos.

- Lo siguiente es una entrada reservada, no tiene ninguna funcionalidad ahora pero es probable que en futuras extensiones sí que la tenga.
- Por último están las excepciones que nos interesan y que trataremos en este capítulo, que son las interrupciones normales `IRQ` y las interrupciones rápidas `FIQ`.

Puesto que cada interrupción, NI, lleva asociada una rutina, de alguna forma, debe haber una correspondencia entre este NI y la ubicación del vector asociado, que contiene la instrucción de salto a la rutina que debe ejecutar cuando se produce la interrupción. La forma de hacerlo es multiplicar por cuatro el número de interrupción para obtener un desplazamiento (NI*4). Se multiplica por 4 porque cada vector de excepción ocupa 4 bytes (es lo que ocupa una instrucción en ARM).

Cuando se activa una interrupción, la CPU detiene su trabajo para atenderla. Después, continúa su trabajo donde lo dejó. Los pasos a seguir para que esto sea posible son:

1. Cuando se activa la interrupción, se termina la ejecución de la instrucción en curso. A continuación se hace una copia `cpsr` en el registro propio `spsr` correspondiente. De esta forma se recuerdan los flags de estado y el modo que había antes de la interrupción. Una vez hecha la copia se procede a cambiar `cpsr`, conmutando al modo correspondiente según la tabla 5.1.

2. Seguidamente a lo anterior se almacena en `lr` (en su registro propio del modo) el contenido de `pc+8` (salvo si es un `error en datos` que sería `pc+12`). La razón de estos desplazamientos es puramente técnica, debido al segmentado de la CPU en el momento de hacer la copia el registro `pc` se ha incrementado en 1 ó 2 instrucciones.

PC	**Ins.**	**Nombre**
x-4	i-1	Instrucción anterior
x	i	Instrucción interrumpida
x+4	i+1	Instrucción siguiente
x+8	i+2	...
x+12	i+3	...

Podemos observarlo gráficamente en la figura 5.3.

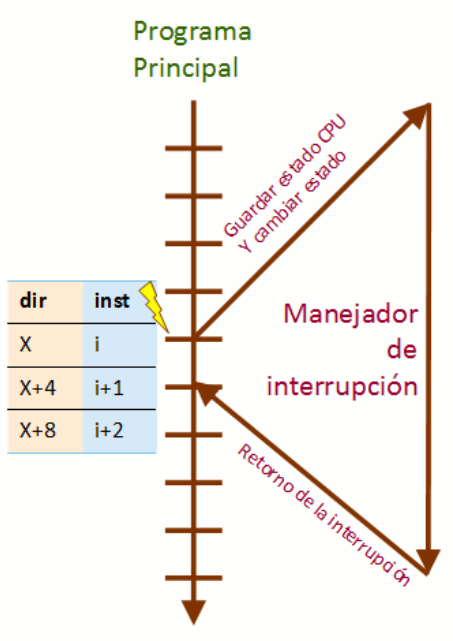

Figura 5.3: Diagrama de una interrupción

3. Luego se salta al vector correspondiente (NI*4). En esa posición del vector se encuentra una instrucción de salto a la RTI correspondiente.

4. Se ejecuta la rutina.

5. La última instrucción de la rutina es `subs pc, lr, #4`, que se encarga de restaurar los flags originales y el modo copiando `spsr` en `cpsr`. Además volvemos al punto donde se interrumpió copiando de `lr` a `pc` (con el desplazamiento correspondiente).

6. Se continúa ejecutando la tarea inicial.

El registro `cpsr` contiene 3 flags globales mediante los cuales podemos habilitar o inhabilitar las interrupciones: uno para `Abort` llamado `A`, otro para `IRQ` llamado `I` y el último para `FIQ` denominado `F`.

El manejo de estos flags corre a cuenta del usuario, en ningún momento la CPU enmascara dichos flags. Por esta razón, si queremos dar prioridad a una interrupción en particular para no ser interrumpidos nuevamente, debemos enmascarar dichos flags al comienzo de su RTI.

5.1.2. Rutina de tratamiento de interrupción

Es el segmento de código que se ejecuta para atender a una interrupción. Una vez se haya ejecutado dicha rutina, retomamos la ejecución normal de nuestro programa, justo después de la instrucción donde lo habíamos interrumpido. Cada rutina de tratamiento debe atender a todas las posibles fuentes de interrupción de su mismo tipo, con lo que al comienzo de la interrupción se suelen acceder a los puertos asociados para detectar qué periférico ha causado la interrupción y actuar en consecuencia.

Si nos interesan `IRQ` y `FIQ`, a lo sumo tendremos que escribir dos rutinas de tratamiento distintas. Si se produce una `IRQ`, se ejecutará el código que se encuentre en la dirección `0x0018`, mientras que si lo que salta es una `FIQ`, la dirección a ejecutar será `0x001C`. La diferencia entre una `IRQ` y una `FIQ` es que esta última tiene sus propios registros desde `r8` hasta `r12` asociados al modo de operación, con lo que podemos prescindir del salvado y recuperación de estos registros en la RTI, ahorrando un tiempo que en determinadas aplicaciones de tiempo real puede ser decisivo.

El esqueleto de una RTI es el siguiente.

```
irq_handler:
        push    {lista registros}
        ...
        pop     {lista registros}
        subs    pc, lr, #4
```

Vemos que a diferencia de las subrutinas donde salíamos con `lr`, en una RTI salimos con `lr-4` (si es un `error en datos` sería `lr-8`), a ello se debe que la última instrucción sea `subs` en lugar de `movs`. ¿Y porqué hay un sufijo `s` al final de la instrucción `sub`? Pues porque se trata de instrucción especial que sirve para restaurar el registro `cpsr` que había antes de la interrupción (copia `spsr_irq` o `spsr_fiq` en `cpsr`).

Imaginemos que el programa principal está en modo supervisor y que la interrupción que esperamos es del tipo IRQ. Cada modo de operación (en particular el modo IRQ) tiene 3 registros replicados: `sp`, `lr` y `spsr`. Para evitar confusiones los nombramos con los sufijos de modo `_svc` y `_irq` correspondientes. Cuando ocurre una interrupción pasamos de modo supervisor a modo IRQ, pero antes hemos guardado el registro `cpsr` en `spsr_irq`.

Los registros `sp_svc` y `lr_svc` no se tocan para nada, con lo que no alteramos ni la pila ni el registro de retorno del modo supervisor. El registro `lr_irq` se carga apuntando a la instrucción `i+2` siguiente a la que fue interrumpida, `pc+8`. El resto de registros debemos salvarlos en pila si tenemos la intención de modificarlos en nuestra RTI, al tener registro propio `sp_irq` se trata de una pila independiente que no interfiere con la principal `sp_svc`. Luego se ejecuta el código particular de la RTI,

empleando a nuestro antojo los registros previamente salvados, y antes de acabar la RTI recuperamos con su `pop` correspondiente.

Al terminar la interrupción restauramos `pc` partiendo de `lr_irq` y `cpsr` del registro `spsr_irq`. Esto último fuerza un cambio de modo de IRQ a supervisor, conmutando `sp` y `lr` a sus registros propios `sp_svc` y `lr_svc`. Con todo esto conseguimos volver exactamente al punto del que partíamos minimizando las operaciones que tiene que hacer la RTI y por tanto el retardo asociado. En otras arquitecturas además de delegar en la RTI este trabajo, se usa la misma pila de programa, lo que puede ocasionar problemas si nos importa lo que hay debajo de ésta.

5.1.3. Pasos para configurar las interrupciones

Nosotros vamos a tratar un caso sencillo de programa principal en el cual hacemos las inicializaciones correspondientes para luego meternos en un bucle infinito y que las interrupciones hagan su trabajo. Las cosas se pueden complicar metiendo código en el programa principal concurrente con las interrupciones. Un ejemplo de esto sería una rutina que dibuja la pantalla en el programa principal, mientras que se aceptan interrupciones para registrar las pulsaciones del teclado.

Sin embargo nuestro programa principal tras la inicialización será una instrucción que salta a sí misma continuamente, `bucle: b bucle`.

El orden recomendado es el siguiente, aunque se puede cambiar el mismo salvo el último punto.

1. Escribimos en el vector de interrupciones la instrucción de salto necesaria a nuestra RTI. Nosotros emplearemos una macro llamada `ADDEXC` que tiene 2 parámetros, vector y dirección de la RTI. La macro genera y escribe el código de operación del salto, para ver los detalles consultar apéndice A. En nuestros ejemplos tendremos IRQs (`0x18`) y FIQs (`0x1c`), por lo que como mucho haremos dos invocaciones a dicha macro (para dos RTIs distintas).

```
.macro    ADDEXC  vector, dirRTI
  ldr     r1, =(\dirRTI-\vector+0xa7ffffb)
  ROR     r1, #2
  str     r1, [r0, #\vector]
.endm
```

2. Inicializamos el puntero de pila (registro `sp`) en todos los modos de operación. Al cambiar el modo de operación hay que tener cuidado de no modificar la máscara global de interrupciones, ya que comparten el mismo byte bajo de `cpsr`. Como sabemos que al comienzo estaban deshabilitadas, las mantenemos igual (bits `I` y `F` a 1. Los punteros tienen que alojar la pila en zonas distintas donde sepamos que no habrá conflictos con la memoria de programa. En los

ejemplos en los que usemos FIQ e IRQ inicializamos la pila de FIQ a `0x4000`, la de IRQ a `0x8000` y la del modo Supervisor a `0x8000000`. Como la memoria de programa empieza en `0x8000` y la pila crece hacia abajo, tendremos 16K de pila en modo IRQ, otros 16K en modo FIQ y 128Mb a compartir entre programa principal y pila de programa. El mapa de memoria sería el indicado en la figura 5.4

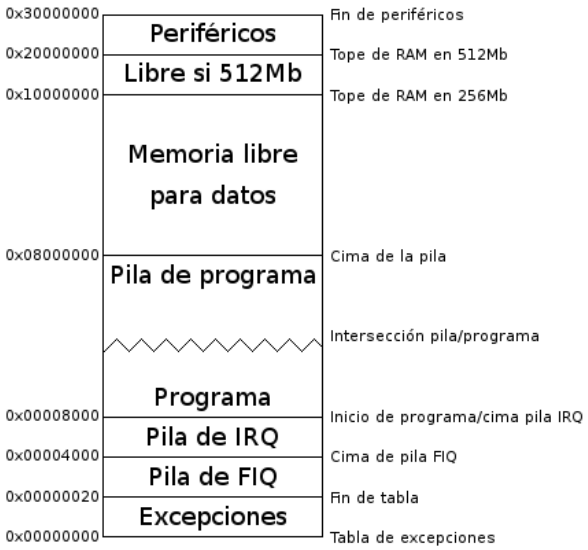

Figura 5.4: Mapa de memoria en nuestros ejemplos

3. Escribimos código de inicialización ajeno al proceso de interrupción, como por ejemplo configurar los GPIOs a salidas donde queramos que actúe un LED.

4. Ahora viene la inicialización de las interrupciones. Aquí le decimos al sistema qué fuentes pueden provocar interrupciones, escribiendo en los puertos asociados.

5. El último paso es habilitar las interrupciones globalmente escribiendo en el registro `cpsr`. Lo hacemos indirectamente vía otro registro, y la instrucción tiene otro nombre pero hace lo mismo que un `mov`. En concreto se llama `msr`, y también hay otra equivalente `mrs` si lo que queremos es leer de `cpsr` a un registro.

6. Después de esto se acaba la inicialización y tendríamos el bucle infinito del que consta nuestro programa principal. Si todo ha ido bien las rutinas de tratamiento de interrupción se encargarán de hacer funcionar nuestro programa como queramos.

5.1.4. El controlador de interrupciones

Los puertos que componen el controlador de interrupciones son los siguientes.

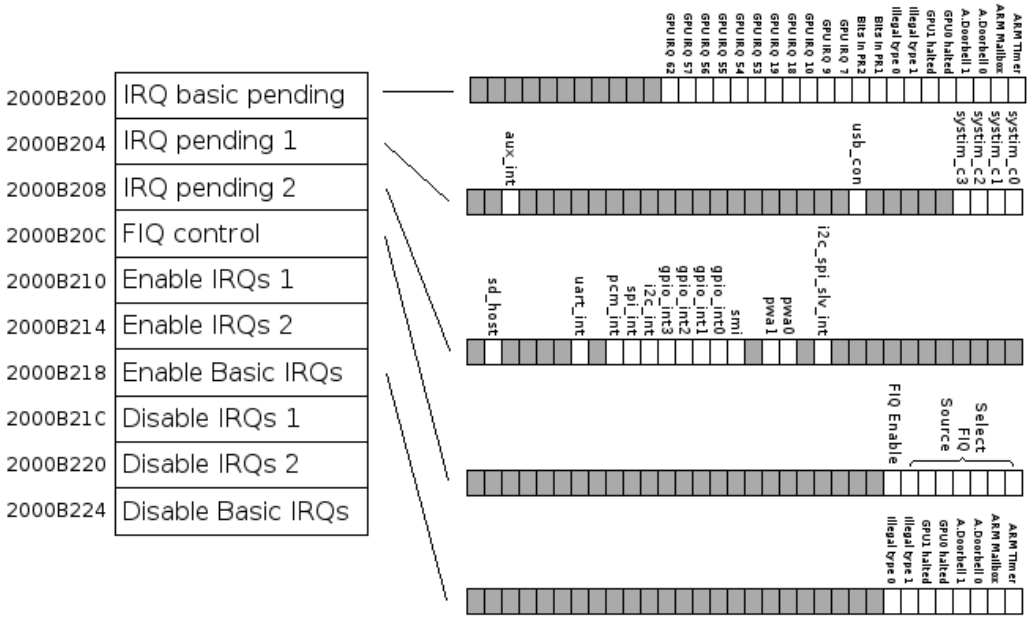

Figura 5.5: Interrupciones

Las `FIQs` sólo tienen un puerto de control asociado, quedando todo el detalle en las `IRQs`. Hay tres grupos de tres puertos cada uno. El primer grupo (Pending) sirve para indicar que hay una interrupción pendiente, el segundo (Enable) es para habilitar las interrupciones y el tercero (Disable) para deshabilitarlas. Dentro de cada grupo tenemos un puerto básico que tiene un resumen sobre el mapa de interrupciones y otros dos puertos que indican con más detalle la fuente de la interrupción. En el puerto básico hay fuentes individuales `GPU IRQ x` y bits que engloban a varias fuentes `Bits in PR1`, que por ejemplo indica que el origen hay que buscarlo en el puerto 1. En el puerto 1 están las primeras 32 posiciones del mapa de interrupciones, mientras que en el puerto 2 están las 32 últimas.

La documentación oficial sobre el mapa de interrupciones está incompleta, pero buscando un poco por internet se puede encontrar que las interrupciones asociadas al `System Timer` se controlan con los 4 primeros bits de la tabla (uno para cada comparador).

En la figura 5.6 vemos los puertos ordenados en grupos.

La forma habitual de trabajar es usar el puerto apropiado del grupo `Enable` para habilitar la fuente de interrupción que queramos que nos interrumpa. Luego en el caso de ser interrumpidos podemos detectar cuál ha sido la fuente leyendo el mismo

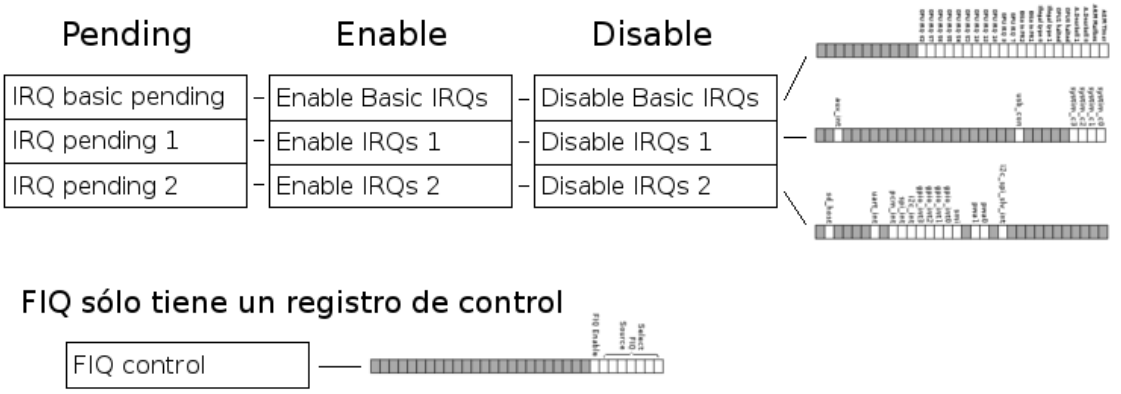

Figura 5.6: Agrupación de puertos de interrupciones

Índice	**Fuente**
0-63	Interrupciones IRQ 1 y 2 (ver figura 5.5)
64	ARM Timer
65	ARM Mailbox
66	ARM Doorbell 0
67	ARM Doorbell 1
68	GPU0 detenida
69	GPU1 detenida
70	Acceso ilegal de tipo 1
71	Acceso ilegal de tipo 2

bit del grupo `Pending` y finalmente, si pasamos a otra sección del programa donde no queremos que nos interrumpa más dicha fuente la desactivamos con el grupo `Disable`.

A parte del controlador de interrupciones, cada dispositivo tiene su propio mecanismo de habilitar/deshabilitar y detectar/notificar la fuente de interrupción. En el caso del GPIO tenemos los puertos `GPRENn`, `GPFENn`, `GPHENn`, `GPLENn`, `GPARENn` y `GPAFENn` para habilitar/deshabilitar. Para detectar/notificar están los `GPEDSn`.

Para el temporizador tenemos que `STCS` hace las funciones de detección y notificación. No existen puertos específicos para habilitar/deshabilitar ya que el controlador de interrupciones permite habilita/deshabilitar cada comparador por separado.

El único puerto que nos falta por ver es `FIQ control ó INTFIQCON` que hemos mostrado en la figura 5.5. Antes mostraremos la lista de fuentes de interrupción aplicables a este puerto.

Son las mismas fuentes que en IRQ pero condensadas en un único puerto. De 0 a 31 coincide con la tabla `IRQ 1`, de 32 a 63 con `IRQ 2` y de 64 en adelante con `IRQ`

`Basic`.

El puerto `INTFIQCON` se programa con los 8 bits inferiores, indicando en el bit 7 si queremos habilitar la fuente, y en los bits del 0 al 6 ponemos el índice de la fuente que se corresponde con la lista. A diferencia de las IRQ, con las FIQ sólo podemos atender a una fuente de interrupción.

5.1.5. Ejemplo. Encender LED rojo a los 4 segundos

Se trata de programar el comparador y las interrupciones para que transcurrido un tiempo determinado se produzca una interrupción, dentro de la cual se encienda el LED. Es un caso muy sencillo porque sólo se va a producir una interrupción que viene de una sola fuente, por lo que en la RTI lo único que haremos es encender el LED.

El diagrama que vamos a usar es el siguiente.

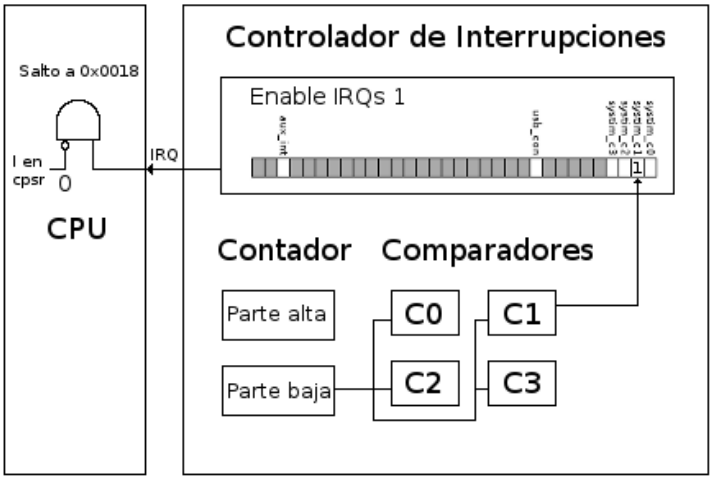

Figura 5.7: Interrupciones

1. Escribimos en el vector de interrupciones

Invocamos la macro para una IRQ, pasándole la etiqueta de nuestra RTI `irq_handler`.

```
        ADDEXC  0x18, irq_handler
```

2. Inicializamos punteros de pila

La única forma de acceder a los registros `sp_irq` y `sp_fiq` es cambiando de modo y modificando el registro `sp` correspondiente.

El modo viene indicado en la parte más baja del registro `cpsr`, el cual modificaremos con la instrucción especial `msr`. En la figura 5.1 vemos el contenido completo del registro `cpsr`. Como `cpsr` es un registro muy heterogéneo, usamos sufijos para acceder a partes concretas de él. En nuestro caso sólo nos interesa cambiar el byte bajo del registro, añadimos el sufijo `_c` llamándolo `cpsr_c`, para no alterar el resto del registro. Esta parte comprende el modo de operación y las máscaras globales de las interrupciones. Otra referencia útil es `cpsr_f` que modifica únicamente la parte de flags (byte alto). Las otras 3 referencias restantes apenas se usan y son `cpsr_s` (Status) para el tercer byte, `cpsr_x` (eXtended) para el segundo byte y `cpsr_csxf` para modificar los 4 bytes a la vez.

En la siguiente tabla vemos cómo se codifica el modo de operación.

Hex	**Binario**	**Modo de operación**
0x10	10000	Usuario
0x11	10001	FIQ
0x12	10010	IRQ
0x13	10011	Supervisor
0x16	10110	Monitor seguro
0x17	10111	Abort
0x1B	11011	Indefinido
0x1F	11111	Sistema

Como las interrupciones globales de IRQ y FIQ están desactivadas (estado por defecto tras el reset), mantenemos a 1 dichos bits.

El código que inicializa los punteros de pila es el siguiente:

```
        mov     r0, #0b11010010    @ Modo IRQ, FIQ&IRQ desact
        msr     cpsr_c, r0
        mov     sp, #0x8000
        mov     r0, #0b11010011    @ Modo SVC, FIQ&IRQ desact
        msr     cpsr_c, r0
        mov     sp, #0x8000000
```

En concreto a `0x8000` y `0x8000000` para los modos `IRQ` y `Supervisor` respectivamente.

3. Código de inicialización ajeno a interrupciones

En el ejemplo que tenemos entre manos se trata de configurar los puertos GPIO de entrada y de salida, inicializar temporizadores. En casos más complejos tendríamos que inicializar estructuras de datos, rellenar las tablas que sean precalculadas y en general cualquier tarea de inicialización requerida para hacer funcionar nuestro programa.

El código para asignar el sentido al pin GPIO 9 es el siguiente:

```
        ldr     r0, =GPBASE
/* guia bits          xx99988777666555444333222111000*/
        mov     r1, #0b00001000000000000000000000000000
        str     r1, [r0, #GPFSEL0]
```

Luego programamos el comparador para que salte la interrupción a los 4,19 segundos:

```
        ldr     r0, =STBASE
        ldr     r1, [r0, #STCLO]
        add     r1, #0x400000     @4,19 segundos
        str     r1, [r0, #STC1]
```

4. Inicializamos interrupciones localmente

Consiste en escribir en los puertos asociados dependiendo de las fuentes que querramos activar. En este primer ejemplo habilitamos el comparador `C1` del temporizador como fuente de interrupción:

```
        ldr     r0, =INTBASE
        mov     r1, #0b0010
        str     r1, [r0, #INTENIRQ1]
```

5. Habilitamos interrupciones globalmente

Se trata de poner a cero el bit correspondiente en `cpsr`. El siguiente código habilita interrupciones del tipo IRQ:

```
        mov     r0, #0b01010011   @ Modo SVC, IRQ activo
        msr     cpsr_c, r0
```

6. Resto del programa principal

Como hemos adelantado, en todos nuestros ejemplos será un bucle infinito:

```
bucle:  b       bucle
```

A continuación mostramos el listado del ejemplo completo:

Listado 5.1: inter1.s

```
        .include  "inter.inc"
.text
/* Agrego vector interrupción */
        ADDEXC  0x18, irq_handler

/* Inicializo la pila en modos IRQ y SVC */
        mov     r0, #0b11010010   @ Modo IRQ, FIQ&IRQ desact
        msr     cpsr_c, r0
        mov     sp, #0x8000
        mov     r0, #0b11010011   @ Modo SVC, FIQ&IRQ desact
        msr     cpsr_c, r0
        mov     sp, #0x8000000

/* Configuro GPIO 9 como salida */
        ldr     r0, =GPBASE
/* guia bits           xx99988777666555444333222111000*/
        mov     r1, #0b00001000000000000000000000000000
        str     r1, [r0, #GPFSEL0]

/* Programo contador C1 para futura interrupción */
        ldr     r0, =STBASE
        ldr     r1, [r0, #STCLO]
        add     r1, #0x400000     @4,19 segundos
        str     r1, [r0, #STC1]

/* Habilito interrupciones, local y globalmente */
        ldr     r0, =INTBASE
        mov     r1, #0b0010
        str     r1, [r0, #INTENIRQ1]
        mov     r0, #0b01010011   @ Modo SVC, IRQ activo
        msr     cpsr_c, r0

/* Repetir para siempre */
bucle:  b       bucle

/* Rutina de tratamiento de interrupción */
irq_handler:
        push    {r0, r1}          @ Salvo registros

        ldr     r0, =GPBASE
/* guia bits           10987654321098765432109876543210*/
        mov     r1, #0b00000000000000000000001000000000
```

```
        str     r1, [r0, #GPSET0] @ Enciendo LED

        pop     {r0, r1}          @ Recupero registros
        subs    pc, lr, #4        @ Salgo de la RTI
```

Observamos que la RTI es muy sencilla, aparte del esqueleto tenemos tres instrucciones encargadas de encender el LED en cuestión.

5.1.6. Ejemplos de aplicación

Vamos a crear un archivo `inter.inc` donde guardaremos las constantes asociadas a los puertos y también la macro `ADDEXC`, esta última se explica en detalle en el apéndice A. De esta forma evitamos escribir siempre las mismas constantes, haciendo el código más sencillo de mantener.

Listado 5.2: inter.inc

```
      .macro    ADDEXC  vector, dirRTI
        ldr     r1, =(\dirRTI-\vector+0xa7ffffb)
        ROR     r1, #2
        str     r1, [r0, #\vector]
      .endm
        .set    GPBASE,   0x20200000
        .set    GPFSEL0,        0x00
        .set    GPFSEL1,        0x04
        .set    GPFSEL2,        0x08
        .set    GPSET0,         0x1c
        .set    GPCLR0,         0x28
        .set    GPEDS0,         0x40
        .set    GPFEN0,         0x58
        .set    GPPUD,          0x94
        .set    GPPUDCLK0,      0x98
        .set    STBASE,   0x20003000
        .set    STCS,           0x00
        .set    STCLO,          0x04
        .set    STC1,           0x10
        .set    STC3,           0x18
        .set    INTBASE,  0x2000b000
        .set    INTFIQCON,     0x20c
        .set    INTENIRQ1,     0x210
        .set    INTENIRQ2,     0x214
```

El método para incluir el código fuente de un fichero dentro de otro es mediante la macro `.include`, todos nuestros ficheros comienzarán con lo siguiente.

```
        .include  "inter.inc"
```

5.1.7. Parpadeo de todos los LEDs

Sería hacer lo mismo que en la lección anterior pero empleando interrupciones y aplicando la salida simultáneamente a los 6 LEDs en lugar de sólo al primero. La novedad en lo que a interrupciones se refiere consiste en reprogramar el comparador C1 cada vez que se produzca una interrupción, de esta forma conseguimos interrupciones periódicas en lugar de una única interrupción.

Veamos el código:

Listado 5.3: inter2.s

```
        .include  "inter.inc"
.text
/* Agrego vector interrupción */
        ADDEXC  0x18, irq_handler

/* Inicializo la pila en modos IRQ y SVC */
        mov     r0, #0b11010010   @ Modo IRQ, FIQ&IRQ desact
        msr     cpsr_c, r0
        mov     sp, #0x8000
        mov     r0, #0b11010011   @ Modo SVC, FIQ&IRQ desact
        msr     cpsr_c, r0
        mov     sp, #0x8000000

/* Configuro GPIOs 9, 10, 11, 17, 22 y 27 como salida */
        ldr     r0, =GPBASE
        mov     r1, #0b00001000000000000000000000000000
        str     r1, [r0, #GPFSEL0]
/* guia bits           xx99988877766655544433322211000*/
        ldr     r1, =0b00000000001000000000000000001001
        str     r1, [r0, #GPFSEL1]
        ldr     r1, =0b00000000001000000000000001000000
        str     r1, [r0, #GPFSEL2]

/* Programo contador C1 para dentro de 2 microsegundos */
        ldr     r0, =STBASE
        ldr     r1, [r0, #STCLO]
        add     r1, #2
        str     r1, [r0, #STC1]
```

```
/* Habilito interrupciones, local y globalmente */
        ldr     r0, =INTBASE
        mov     r1, #0b0010
        str     r1, [r0, #INTENIRQ1]
        mov     r0, #0b01010011   @ Modo SVC, IRQ activo
        msr     cpsr_c, r0

/* Repetir para siempre */
bucle:  b       bucle

/* Rutina de tratamiento de interrupción */
irq_handler:
        push    {r0, r1, r2}

/* Conmuto variable de estado del LED */
        ldr     r0, =ledst    @ Leo puntero a v. ledst
        ldr     r1, [r0]      @ Leo variable
        eors    r1, #1        @ Invierto bit 0, act. flag Z
        str     r1, [r0]      @ Escribo variable

/* Enciendo o apago todos los LEDs en función del flag Z */
        ldr     r0, =GPBASE
/* guia bits           10987654321098765432109876543210*/
        ldr     r1, =0b00001000010000100000111000000000
        streq   r1, [r0, #GPSET0]
        strne   r1, [r0, #GPCLR0]

/* Reseteo estado interrupción de C1 */
        ldr     r0, =STBASE
        mov     r1, #0b0010
        str     r1, [r0, #STCS]

/* Programo siguiente interrupción medio segundo después */
        ldr     r1, [r0, #STCLO]
        ldr     r2, =500000       @1 Hz
        add     r1, r2
        str     r1, [r0, #STC1]

/* Recupero registros y salgo */
        pop     {r0, r1, r2}
        subs    pc, lr, #4
```

```
/* Ubicación de la variable ledst */
ledst:  .word   0
```

Y vamos enumerando, por orden, los pasos que hemos seguido. En primer lugar apuntamos a nuestra RTI en el vector de interrupciones:

```
        ADDEXC  0x18, irq_handler
```

Luego inicializamos los punteros de pila:

```
        mov     r0, #0b11010010   @ Modo IRQ, FIQ&IRQ desact
        msr     cpsr_c, r0
        mov     sp, #0x8000
        mov     r0, #0b11010011   @ Modo SVC, FIQ&IRQ desact
        msr     cpsr_c, r0
        mov     sp, #0x8000000
```

Lo siguiente es configurar los pines GPIO asociados a los 6 LEDs como salidas:

```
        ldr     r0, =GPBASE
        mov     r1, #0b00001000000000000000000000000000
        str     r1, [r0, #GPFSEL0]
/* guia bits           xx99988777666555444333222111000*/
        ldr     r1, =0b00000000001000000000000000001001
        str     r1, [r0, #GPFSEL1]
        ldr     r1, =0b00000000001000000000000001000000
        str     r1, [r0, #GPFSEL2]
```

Preparamos el comparador `C1` para que al cabo de dos microsegundos nos proporcione la primera interrupción:

```
        ldr     r0, =STBASE
        ldr     r1, [r0, #STCLO]
        add     r1, #2
        str     r1, [r0, #STC1]
```

Para después habilitar las interrupciones asociadas al comparador `C1`:

```
        ldr     r0, =INTBASE
        mov     r1, #0b0010
        str     r1, [r0, #INTENIRQ1]
```

Y finalmente habilitar las interrupciones `IRQ` globalmente, entrando luego en el bucle infinito:

```
        mov     r0, #0b01010011   @modo SVC, IRQ activo
        msr     cpsr_c, r0
bucle:  b       bucle
```

Ya hemos terminado con el programa principal, que como veremos más adelante va a ser siempre muy parecido.

Lo interesante está en la RTI, que es donde hacemos parpadear los LEDs y configuramos el comparador para la siguiente interrupción.

El estado de los LEDs (si están apagados o encendidos) lo guardamos en la variable `ledst`, que conmutamos entre cero y uno mediante un `OR` exclusivo. Al actualizar los `flags` tras esta operación, tenemos que si el resultado fue cero nos lo indica el `flag Z` activo, mientras que estará inactivo en el caso contrario (resultado 1). Mediante las instrucciones de ejecución condicional `streq` y `strne` enviamos la orden al puerto que enciende los LEDs o al puerto que los apaga, respectivamente:

```
irq_handler:
        push    {r0, r1, r2}

        ldr     r0, =ledst      @ Leo puntero a v. ledst
        ldr     r1, [r0]        @ Leo variable
        eors    r1, #1          @ Invierto bit 0, act. flag Z
        str     r1, [r0]        @ Escribo variable

        ldr     r0, =GPBASE
/* guia bits            10987654321098765432109876543210*/
        ldr     r1, =0b00001000010000100000111000000000
        streq   r1, [r0, #GPSET0]
        strne   r1, [r0, #GPCLR0]
```

Luego escribimos un 1 en el `M1` de `CS`, para resetear el estado de coincidencia, ya que de lo contrario el gestor de interrupciones verá el bit siempre a 1 y no lanzará más interrupciones. Resulta confuso tener que escribir un 1 en el puerto para almacenar un 0, pero de un modo similar a lo que ocurre con el puerto `GPCLR0` del GPIO es para ahorrar operaciones y que la RTI sea más rápida. Así no hay que leer el puerto, aplicar una máscara y volver a escribir en el mismo puerto, con una escritura es suficiente:

```
        ldr     r0, =STBASE
        mov     r1, #0b0010
        str     r1, [r0, #STCS]
```

Luego tenemos que actualizar el puerto comparador, de lo contrario tardará poco más de una hora en cambiar de estado el LED (es lo que tarda el contador en dar una vuelta completa). Para ello leemos el contador (`CLO`) y le añadimos 500000 al valor leído. Como cada cuenta equivale a un microsegundo, este añadido al contador supone medio segundo, lo que nos da la cadencia de un segundo que buscamos. El resultado de la suma lo escribimos en el comparador (`C1`):

```
        ldr     r1, [r0, #STCLO]
        ldr     r2, =500000         @1 Hz
        add     r1, r2
        str     r1, [r0, #STC1]
```

Por último restauramos los registros utilizados y salimos de la RTI. Más abajo tenemos la definición de la variable `ledst`, como no tenemos sección de datos aparte la ponemos al final del código:

```
        pop     {r0, r1, r2}
        subs    pc, lr, #4

ledst:  .word   0
```

5.1.8. Control de LEDs rojos con pulsadores

En este ejemplo cambiamos de fuente de interrupción, en lugar del temporizador empleamos los pulsadores. Queremos que al pulsar un botón se encienda el LED rojo del mismo lado del pulsador, dejando el otro apagado.

El esquema sería el de la figura 5.8.

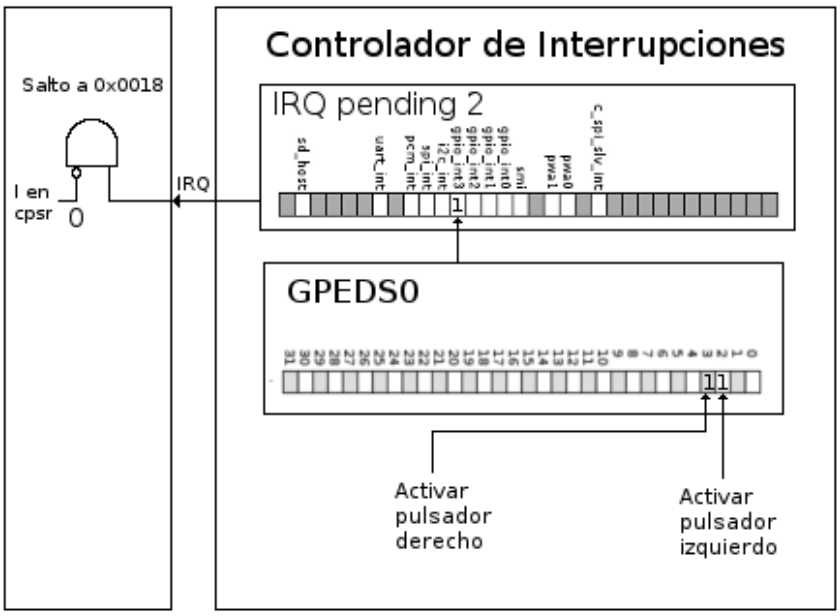

Figura 5.8: Interrupciones

Y el código fuente lo vemos a continuación:

Listado 5.4: inter3.s

```
        .include  "inter.inc"
.text
/* Agrego vector interrupción */
        ADDEXC  0x18, irq_handler

/* Inicializo la pila en modos IRQ y SVC */
        mov     r0, #0b11010010   @ Modo IRQ, FIQ&IRQ desact
        msr     cpsr_c, r0
        mov     sp, #0x8000
        mov     r0, #0b11010011   @ Modo SVC, FIQ&IRQ desact
        msr     cpsr_c, r0
        mov     sp, #0x8000000

/* Configuro GPIOs 9 y 10 como salida */
        ldr     r0, =GPBASE
        mov     r1, #0b00001000000000000000000000000000
        str     r1, [r0, #GPFSEL0]
/* guia bits           xx99988877766655544433322211000*/
        mov     r1, #0b00000000000000000000000000000001
        str     r1, [r0, #GPFSEL1]

/* Enciendo LEDs       10987654321098765432109876543210*/
        mov     r1, #0b00000000000000000000011000000000
        str     r1, [r0, #GPSET0]

/* Habilito pines GPIO 2 y 3 (botones) para interrupciones*/
        mov     r1, #0b00000000000000000000000000001100
        str     r1, [r0, #GPFEN0]
        ldr     r0, =INTBASE

/* Habilito interrupciones, local y globalmente */
        mov     r1, #0b00000000000100000000000000000000
/* guia bits           10987654321098765432109876543210*/
        str     r1, [r0, #INTENIRQ2]
        mov     r0, #0b01010011   @ Modo SVC, IRQ activo
        msr     cpsr_c, r0

/* Repetir para siempre */
bucle:  b       bucle
```

```
/* Rutina de tratamiento de interrupción */
irq_handler:
        push    {r0, r1}
        ldr     r0, =GPBASE
/* Apago los dos LEDs rojos  54321098765432109876543210*/
        mov     r1, #0b00000000000000000000011000000000
        str     r1, [r0, #GPCLR0]
/* Consulto si se ha pulsado el botón GPIO2 */
        ldr     r1, [r0, #GPEDS0]
        ands    r1, #0b00000000000000000000000000000100
/* Sí: Activo GPIO 9; No: Activo GPIO 10 */
        movne   r1, #0b00000000000000000000001000000000
        moveq   r1, #0b00000000000000000000010000000000
        str     r1, [r0, #GPSET0]
/* Desactivo los dos flags GPIO pendientes de atención
   guia bits                 54321098765432109876543210*/
        mov     r1, #0b00000000000000000000000000001100
        str     r1, [r0, #GPEDS0]
        pop     {r0, r1}
        subs    pc, lr, #4
```

Obviamos los dos primeros pasos (apuntar a RTI e inicialización de punteros de pila) puesto que son idénticos al ejemplo anterior.

Lo siguiente que tenemos es configurar e inicializar los puertos del GPIO. Por un lado ponemos los correspondientes a los LEDs rojos (GPIO 9 y GPIO 10) como salida. Por otro lado escribimos un uno en ambos LEDs, para que al arrancar veamos los dos LEDs encendidos. Así sabemos que el programa está cargado a la espera de que activemos los pulsadores:

```
        ldr     r0, =GPBASE
        ldr     r1, =0b00001000000000000000000000000000
        str     r1, [r0, #GPFSEL0]
/* guia bits           xx99988777666555444333222111000*/
        ldr     r1, =0b00000000000000000000000000000001
        str     r1, [r0, #GPFSEL1]
/* guia bits           10987654321098765432109876543210*/
        mov     r1, #0b00000000000000000000011000000000
        str     r1, [r0, #GPSET0]
```

Luego habilitamos las interrupciones particulares del GPIO en el puerto `GPFEN0`, en concreto las que entran por los pulsadores (GPIO 2 y GPIO 3). Para que las peticiones se propaguen desde el GPIO al controlador de interrupciones habilitamos el bit 20 del puerto `INTENIRQ2`:

```
        mov     r1, #0b00000000000000000000000000001100
        str     r1, [r0, #GPFEN0]
        ldr     r0, =INTBASE
/* guia bits            10987654321098765432109876543210*/
        mov     r1, #0b00000000000010000000000000000000
        str     r1, [r0, #INTENIRQ2]
```

Para terminar activando globalmente las IRQ y metiéndonos en el bucle infinito:

```
        mov     r0, #0b01010011    @ Modo SVC, IRQ activo
        msr     cpsr_c, r0
bucle:  b       bucle
```

Veamos ahora el aspecto que tiene la RTI. Lo primero es poner los LEDs susceptibles de encenderse (los LEDs rojos) a cero:

```
irq_handler:
        push    {r0, r1}
        ldr     r0, =GPBASE
/* Apaga los dos LEDs rojos  54321098765432109876543210*/
        mov     r1, #0b00000000000000000000011000000000
        str     r1, [r0, #GPCLR0]
```

Testeamos cuál de los dos pulsadores se ha activado, indicándolo en el flag Z:

```
/* Consulto si se ha pulsado el botón GPIO2 */
        ldr     r1, [r0, #GPEDS0]
        ands    r1, #0b00000000000000000000000000000100
```

En función del flag Z encendemos uno u otro LED:

```
/* Sí: Activo GPIO 9; No: Activo GPIO 10 */
        movne   r1, #0b00000000000000000000001000000000
        moveq   r1, #0b00000000000000000000010000000000
        str     r1, [r0, #GPSET0]
```

Y finalmente desactivamos los dos flags GPIO pendientes de atención:

```
        mov     r1, #0b00000000000000000000000000001100
        str     r1, [r0, #GPEDS0]
        pop     {r0, r1}
        subs    pc, lr, #4
```

5.1.9. Parpadeo secuencial de LEDs con sonido por altavoz

En este ejemplo vamos a trabajar con el temporizador, pero esta vez vamos a complicar un poco las cosas. En lugar de una fuente vamos a atender simultáneamente las peticiones de los comparadores `C1` y `C3`.

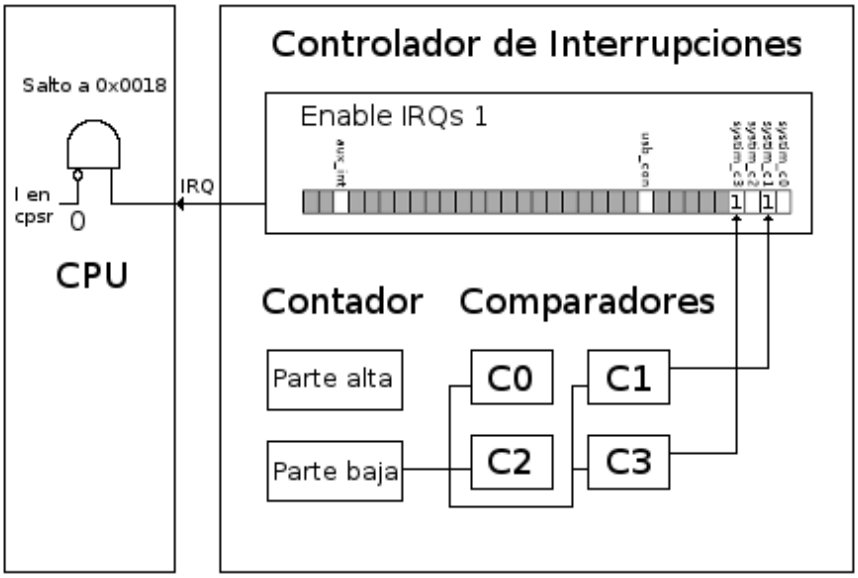

Figura 5.9: Interrupciones

Con esta segunda fuente vamos a controlar el altavoz, como podemos observar en la figura 5.9. Sacar un tono puro por el altavoz es equivalente a hacer parpadear un LED, lo único que cambia es que usamos otro pin distinto `GPIO 4` y aumentamos la frecuencia para que sea audible (a 1 Hz el oído humano no captaría sonido alguno). Utilizaremos la frecuencia estándar de afinación de 440 Hz, que coincide con el tono de espera de marcado en telefonía fija.

Por otro lado en lugar de hacer parpadear todos los LEDs lo que haremos es repetir una secuencia de 6 posiciones en la que en todo momento sólo uno de los 6 LEDs está encendido, que va cambiando de izquierda a derecha (aparentando movimiento) y cuando se llegue al sexto LED comenzamos de nuevo desde el primero. Para dar más sensación de movimiento disminuimos el periodo a 200 milisegundos.

La clave de todo está en saber cuál de los dos comparadores ha producido la interrupción (se puede dar el caso en que salten los dos a la vez). Ésto se puede hacer de dos formas distintas: o bien leemos el bit asociado `systim_cx` en el puerto `IRQ pending 1`, o bien leemos el `Mx` del puerto `CS`. Elegimos el segundo caso, así no gastamos otro puerto más para almacenar `INTBASE`.

El código completo del ejemplo es el siguiente:

Listado 5.5: inter4.s

```
        .include  "inter.inc"
.text
/* Agrego vector interrupción */
        ADDEXC  0x18, irq_handler

/* Inicializo la pila en modos IRQ y SVC */
        mov     r0, #0b11010010   @ Modo IRQ, FIQ&IRQ desact
        msr     cpsr_c, r0
        mov     sp, #0x8000
        mov     r0, #0b11010011   @ Modo SVC, FIQ&IRQ desact
        msr     cpsr_c, r0
        mov     sp, #0x8000000

/* Configuro GPIOs 4, 9, 10, 11, 17, 22 y 27 como salida */
        ldr     r0, =GPBASE
        ldr     r1, =0b00001000000000000000010000000000000
        str     r1, [r0, #GPFSEL0]
/* guia bits           xx99988877766655544433322211000*/
        ldr     r1, =0b00000000001000000000000000001001
        str     r1, [r0, #GPFSEL1]
        ldr     r1, =0b00000000001000000000000001000000
        str     r1, [r0, #GPFSEL2]

/* Programo C1 y C3 para dentro de 2 microsegundos */
        ldr     r0, =STBASE
        ldr     r1, [r0, #STCLO]
        add     r1, #2
        str     r1, [r0, #STC1]
        str     r1, [r0, #STC3]

/* Habilito interrupciones, local y globalmente */
        ldr     r0, =INTBASE
        mov     r1, #0b1010
        str     r1, [r0, #INTENIRQ1]
        mov     r0, #0b01010011   @ Modo SVC, IRQ activo
        msr     cpsr_c, r0

/* Repetir para siempre */
bucle:  b       bucle
```

```
/* Rutina de tratamiento de interrupción */
irq_handler:
        push    {r0, r1, r2, r3}

/* Leo origen de la interrupción */
        ldr     r0, =STBASE
        ldr     r1, =GPBASE
        ldr     r2, [r0, #STCS]
        ands    r2, #0b0010
        beq     sonido

/* Si es C1, ejecuto secuencia de LEDs */
        ldr     r2, =cuenta
/* guia bits           10987654321098765432109876543210*/
        ldr     r3, =0b00001000010000100000111000000000
        str     r3, [r1, #GPCLR0] @ Apago todos los LEDs
        ldr     r3, [r2]          @ Leo variable cuenta
        subs    r3, #1            @ Decremento
        moveq   r3, #6            @ Si es 0, volver a 6
        str     r3, [r2]          @ Escribo cuenta
        ldr     r3, [r2, +r3, LSL #2] @ Leo secuencia
        str     r3, [r1, #GPSET0] @ Escribo secuencia en LEDs

/* Reseteo estado interrupción de C1 */
        mov     r3, #0b0010
        str     r3, [r0, #STCS]

/* Programo siguiente interrupción en 200ms */
        ldr     r3, [r0, #STCLO]
        ldr     r2, =200000       @ 5 Hz
        add     r3, r2
        str     r3, [r0, #STC1]

/* ¿Hay interrupción pendiente en C3? */
        ldr     r3, [r0, #STCS]
        ands    r3, #0b0100
        beq     final             @ Si no, salgo

/* Si es C3, hago sonar el altavoz */
sonido: ldr     r2, =bitson
        ldr     r3, [r2]
        eors    r3, #1            @ Invierto estado
        str     r3, [r2]
```

```
        mov     r3, #0b10000        @ GPIO 4 (altavoz)
        streq   r3, [r1, #GPSET0] @ Escribo en altavoz
        strne   r3, [r1, #GPCLR0] @ Escribo en altavoz

/* Reseteo estado interrupción de C3 */
        mov     r3, #0b1000
        str     r3, [r0, #STCS]

/* Programo interrupción para sonido de 440 Hz */
        ldr     r3, [r0, #STCLO]
        ldr     r2, =1136           @ Contador para 440 Hz
        add     r3, r2
        str     r3, [r0, #STC3]

/* Recupero registros y salgo */
final:  pop     {r0, r1, r2, r3}
        subs    pc, lr, #4

bitson: .word   0               @ Bit 0 = Estado del altavoz
cuenta: .word   1               @ Entre 1 y 6, LED a encender
/* guia bits      76543210987654321098765432 10*/
secuen: .word   0b10000000000000000000000000000
        .word   0b00000100000000000000000000000
        .word   0b00000000001000000000000000000
        .word   0b00000000000000001000000000000
        .word   0b00000000000000000010000000000
        .word   0b00000000000000000001000000000
```

Como es muy parecido al ejemplo de antes, sólo vamos a comentar las diferencias que encontremos. La primera de ellas es que además de los 6 GPIOs de los LEDs, configuramos como salida un séptimo pin, el GPIO 4, para manejar el altavoz:

```
        ldr     r0, =GPBASE
        ldr     r1, =0b00000100000000000000001000000000000
        str     r1, [r0, #GPFSEL0]
```

El siguiente código es para incluir el comparador `C3` (además del `C1` que había anteriormente), tanto para proporcionar la primera interrupción como para habilitarla individualmente:

```
        ldr     r0, =STBASE
        ldr     r1, [r0, #STCLO]
        add     r1, #2
        str     r1, [r0, #STC1]
        str     r1, [r0, #STC3]
```

```
        ldr     r0, =INTBASE
        mov     r1, #0b1010
        str     r1, [r0, #INTENIRQ1]
```

Ya hemos acabado con el programa principal, veamos ahora la RTI. Primero mostramos la estructura del código y luego las rutinas individuales tanto para el manejo de LEDs como para el altavoz:

```
irq_handler:
        push    {r0, r1, r2, r3}
        ldr     r0, =STBASE
        ldr     r1, =GPBASE
        ldr     r2, [r0, #STCS]
        ands    r2, #0b0010
        beq     sonido

        [ manejo de LEDs ]

        ldr     r3, [r0, #STCS]
        ands    r3, #0b0100
        beq     final
sonido:
        [ manejo de altavoz ]

final:  pop     {r0, r1, r2, r3}
        subs    pc, lr, #4
```

Los registros `r0` y `r1` los hacemos apuntar a la base del `System Timer` y del `GPIO` y no tocamos dichos valores durante toda la interrupción, vamos a estar constantemente leyendo y escribiendo puertos y resulta incómodo tener que cargar la base cada vez.

Es un error muy habitual suponer que la fuente de la interrupción sólo ha sido una, aunque la gran mayoría de las veces sea así se puede dar el caso de que coincidan los dos comparadores a la vez. De la misma forma si sabemos que sólo hay dos fuentes y una de ellas no ha provocado la interrupción, por descarte ha tenido que ser la otra, podemos ahorrarnos la comprobación.

El flujo sería el siguiente: leemos `M1` para ver si la interrupción la ha provocado el comparador de `C1`, si ha sido así ejecutamos el código de manejo de LEDs; si no, saltamos directamente al manejo del altavoz (sabemos seguro que la fuente viene de ahí).

Tras el código del manejo de LEDs leemos `M3` para saber si además de `C1` ha saltado también el comparador `C3`. Si no ha saltado, lo más normal, salimos por `final`; si lo ha hecho, procesamos la interrupción con el código de manejo del altavoz

para luego salir de la RTI.

Estos programas no son fáciles de crear y nunca funcionan a la primera. Es una buena práctica hacer funcionar por separado el código de los LEDs y el código del altavoz, y una vez comprobemos que funcionan, aglutinarlo en una única RTI. De esta forma aislamos lo máximo posible los errores que podamos cometer, es muy fácil equivocarse en una tontería y estar dándole vueltas al código sin encontrar el fallo. A diferencia de los primeros capítulos que disponíamos de `gdb`, en Bare Metal no tenemos acceso a ningún depurador.

Prosigamos ahora con el código de manejo de LEDs. Recordemos que hemos complicado un poco las cosas para emitir una secuencia en lugar de un simple parpadeo. Para ello mostramos el código seguido de las variables empleadas en el mismo:

```
        ldr     r2, =cuenta
/* guia bits           10987654321098765432109876543210*/
        ldr     r3, =0b00001000010000100000111000000000
        str     r3, [r1, #GPCLR0] @ Apago todos los LEDs
        ldr     r3, [r2]          @ Leo variable cuenta
        subs    r3, #1            @ Decremento
        moveq   r3, #6            @ Si es 0, volver a 6
        str     r3, [r2]          @ Escribo cuenta
        ldr     r3, [r2, +r3, LSL #2] @ Leo secuencia#2]
        str     r3, [r1, #GPSET0] @ Escribo secuencia en LEDs
        mov     r3, #0b0010
        str     r3, [r0, #STCS]
        ldr     r3, [r0, #STCLO]
        ldr     r2, =200000       @ 5 Hz
        add     r3, r2
        str     r3, [r0, #STC1]
        [...]
cuenta: .word   1             @ Entre 1 y 6, LED a encender
/* guia bits      76543210987654321098765432l0*/
secuen: .word   0b10000000000000000000000000000
        .word   0b00000100000000000000000000000
        .word   0b00000000001000000000000000000
        .word   0b00000000000000001000000000000
        .word   0b00000000000000000010000000000
        .word   0b00000000000000000001000000000
```

En la variable `cuenta` almacenamos un contador que va desde 6 hasta 1, que actua como índice para el array `secuen`. Al decrementar aprovechamos la propia instrucción de resta para comprobar que se ha llegado al final de la cuenta (0), y en dicho caso restablecemos la cuenta a 6 mediante la instrucción de ejecución condicional `moveq`.

En el array secuen tenemos almacenadas las posiciones que corresponden a los LEDs dentro del puerto `GPSET0`, cada posición del array es para encender un LED en concreto. Antes de esto hemos apagado todos los LEDs enviando el valor que codifica todos los LEDs al puerto `GPCLR0`.

A parte de sacar la secuencia correspondiente debemos especificar cuándo será la siguiente interrupción. Como hicimos en el ejemplo anterior, esto se resuelve leyendo el valor del puerto `STCLO`, sumándole 200000 (200 milisegundos) y escribiéndolo en el comparador `STC1`.

Acabado el código de manejo de LEDs, ya sólo falta por explicar el manejo del altavoz:

```
sonido: ldr     r2, =bitson
        ldr     r3, [r2]
        eors    r3, #1              @ Invierto estado
        str     r3, [r2]
        mov     r3, #0b10000        @ GPIO 4 (altavoz)
        streq   r3, [r1, #GPSET0]   @ Escribo en altavoz
        strne   r3, [r1, #GPCLR0]   @ Escribo en altavoz
        mov     r3, #0b1000
        str     r3, [r0, #STCS]
        ldr     r3, [r0, #STCLO]
        ldr     r2, =1136           @ Contador para 440 Hz
        add     r3, r2
        str     r3, [r0, #STC3]
        [...]
bitson: .word   0
```

Es un calco de la rutina que hacía parpadear todos los LEDs, cambiando el valor que se envia a `GPCLR0/GPSET0`, el comparador que es `C3` en lugar de `C1`, y el valor que sumamos al temporizador, que se corresponde a 440 Hz en vez de a 1 Hz.

5.1.10. Manejo de FIQs y sonidos distintos para cada LED

Este ejemplo es muy parecido al anterior pero con cambios sutiles. El hecho de cambiar una de las dos IRQs por una FIQ incluso simplifica el código, ya que tienen distintas RTIs y en cada una la fuente de interrupción es única, por lo que no hay que comprobar nada ni hacer saltos.

Empecemos con el programa principal. Aquí sí que hay cambios porque tenemos que agregar un elemento nuevo al vector de interrupciones, inicializar el puntero de pila del modo FIQ y activar la fuente de interrupción FIQ local y globalmente:

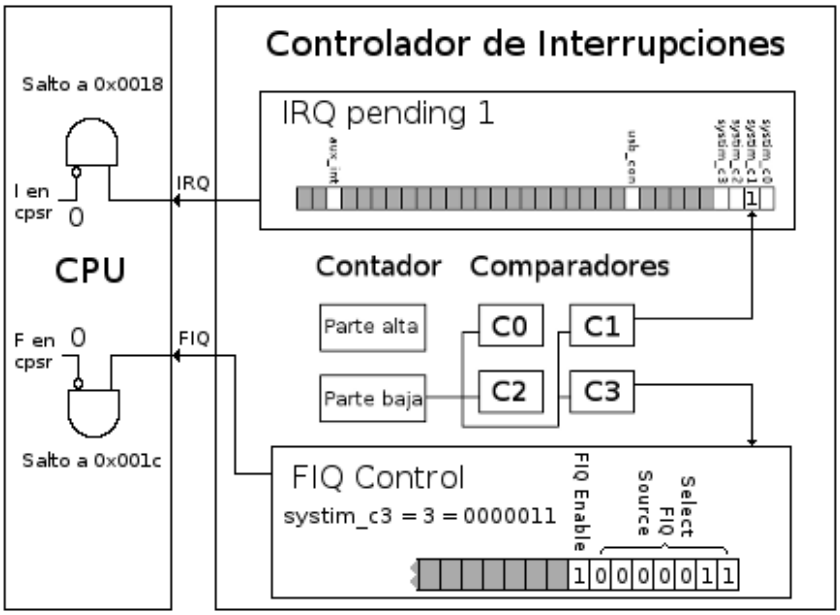

Figura 5.10: Interrupciones

Listado 5.6: Programa principal de inter5.s

```
/* Agrego vectores de interrupción */
        ADDEXC  0x18, irq_handler
        ADDEXC  0x1c, fiq_handler

/* Inicializo la pila en modos FIQ, IRQ y SVC */
        mov     r0, #0b11010001   @ Modo FIQ, FIQ&IRQ desact
        msr     cpsr_c, r0
        mov     sp, #0x4000
        mov     r0, #0b11010010   @ Modo IRQ, FIQ&IRQ desact
        msr     cpsr_c, r0
        mov     sp, #0x8000
        mov     r0, #0b11010011   @ Modo SVC, FIQ&IRQ desact
        msr     cpsr_c, r0
        mov     sp, #0x8000000

/* Configuro GPIOs 4, 9, 10, 11, 17, 22 y 27 como salida */
        ldr     r0, =GPBASE
        ldr     r1, =0b00001000000000000001000000000000
        str     r1, [r0, #GPFSEL0]
/* guia bits           xx99988777666555444333222111000*/
        ldr     r1, =0b00000000001000000000000000001001
```

```
        str     r1, [r0, #GPFSEL1]
        ldr     r1, =0b00000000001000000000000001000000
        str     r1, [r0, #GPFSEL2]

/* Programo C1 y C3 para dentro de 2 microsegundos */
        ldr     r0, =STBASE
        ldr     r1, [r0, #STCLO]
        add     r1, #2
        str     r1, [r0, #STC1]
        str     r1, [r0, #STC3]

/* Habilito C1 para IRQ */
        ldr     r0, =INTBASE
        mov     r1, #0b0010
        str     r1, [r0, #INTENIRQ1]

/* Habilito C3 para FIQ */
        mov     r1, #0b10000011
        str     r1, [r0, #INTFIQCON]

/* Habilito interrupciones globalmente */
        mov     r0, #0b00010011   @ Modo SVC, FIQ&IRQ activo
        msr     cpsr_c, r0

/* Repetir para siempre */
bucle:  b       bucle
```

Queremos que FIQ se active con `C3`, que es el bit 3 del `IRQ 1`, por tanto índice 3 para la fuente `FIQ`. Como veis, la única pega que tienen las FIQs es que sólo admiten una fuente de interrupción. Además del índice ponemos el bit 7 a uno para indicar que queremos habilitar dicha fuente, siendo la constante `0b10000011`.

Ahora veamos el manejador IRQ (la RTI) que, como hemos adelantado, es más sencilla que en el ejemplo anterior:

```
/* Rutina de tratamiento de interrupción IRQ */
irq_handler:
        push    {r0, r1, r2}
        ldr     r0, =GPBASE
        ldr     r1, =cuenta

/* Apago todos LEDs    10987654321098765432109876543210*/
        ldr     r2, =0b00001000010000100000111000000000
        str     r2, [r0, #GPCLR0]
```

```
        ldr     r2, [r1]            @ Leo variable cuenta
        subs    r2, #1              @ Decremento
        moveq   r2, #6              @ Si es 0, volver a 6
        str     r2, [r1], #-4       @ Escribo cuenta
        ldr     r2, [r1, +r2, LSL #3] @ Leo secuencia
        str     r2, [r0, #GPSET0] @ Escribo secuencia en LEDs

/* Reseteo estado interrupción de C1 */
        ldr     r0, =STBASE
        mov     r2, #0b0010
        str     r2, [r0, #STCS]

/* Programo siguiente interrupción en 500ms */
        ldr     r2, [r0, #STCLO]
        ldr     r1, =500000         @ 2 Hz
        add     r2, r1
        str     r2, [r0, #STC1]

/* Recupero registros y salgo */
        pop     {r0, r1, r2}
        subs    pc, lr, #4
```

Observamos que al acceder a la tabla `secuen` multiplicamos el índice por 8 en lugar de por 4. Esto es así porque hemos incluído en dicha tabla el valor de la longitud de onda (inverso de la frecuencia) con la que queremos que suene cada LED, la zona de datos es ésta:

```
bitson: .word   0               @ Bit 0 = Estado del altavoz
cuenta: .word   1               @ Entre 1 y 6, LED a encender
secuen: .word   0b10000000000000000000000000000000
        .word   716             @ Retardo para nota 6
        .word   0b00000100000000000000000000000000
        .word   758             @ Retardo para nota 5
/* guia bits      76543210987654321098765432 10*/
        .word   0b00000000001000000000000000000000
        .word   851             @ Retardo para nota 4
        .word   0b00000000000000000100000000000000
        .word   956             @ Retardo para nota 3
/* guia bits      76543210987654321098765432 10*/
        .word   0b00000000000000000010000000000000
        .word   1012            @ Retardo para nota 2
        .word   0b00000000000000000001000000000000
        .word   1136            @ Retardo para nota 1
```

Serían las notas puras que van después del LA estándar de 440 Hz (1136), cuyos semitonos se obtienen multiplicando la frecuencia por raíz duodécima de 2, que es aproximadamente 1,05946. Las notas serían, en hercios: LA (440), SI (493,88), DO (523,25), RE (587,33), MI (659,26) y FA (698,46).

Finalmente tenemos el manejador de FIQ asociado al altavoz. La elección de la fuente de interrupción no es arbitraria, hemos escogido FIQ para el altavoz porque se ejecutará más veces que el cambio de LEDs, concretamente 220 veces más con la nota más grave. En estos ejemplos no importa, pero en casos reales donde el tiempo de CPU es un recurso limitado, los ciclos que nos ahorramos con una FIQ en un proceso crítico pueden ser determinantes:

```
/* Rutina de tratamiento de interrupción FIQ */
fiq_handler:
        ldr     r8, =GPBASE
        ldr     r9, =bitson

/* Hago sonar altavoz invirtiendo estado de bitson */
        ldr     r10, [r9]
        eors    r10, #1
        str     r10, [r9], #4

/* Leo cuenta y luego elemento correspondiente en secuen */
        ldr     r10, [r9]
        ldr     r9, [r9, +r10, LSL #3]

/* Pongo estado altavoz según variable bitson */
        mov     r10, #0b10000     @ GPIO 4 (altavoz)
        streq   r10, [r8, #GPSET0]
        strne   r10, [r8, #GPCLR0]

/* Reseteo estado interrupción de C3 */
        ldr     r8, =STBASE
        mov     r10, #0b1000
        str     r10, [r8, #STCS]

/* Programo retardo según valor leído en array */
        ldr     r10, [r8, #STCLO]
        add     r10, r9
        str     r10, [r8, #STC3]


/* Salgo de la RTI */
        subs    pc, lr, #4
```

El código sería idéntico al de una IRQ si no fuera porque empleamos registros a partir de `r8` en lugar de a partir de `r0`, y no los salvaguardamos con las instrucciones `push/pop`. La razón es que el modo de operación FIQ es el único que tiene sus propios registros `r8-r12` (ver figura 5.2) con el objetivo de no perder el tiempo guardando y recuperando datos de la pila. En situaciones más críticas podemos incluso ubicar la RTI justo al final del vector de interrupciones. Esta tabla no contiene datos, sino instrucciones, y lo que hace la CPU cuando ocurre una excepción es saltar (ejecutar) a la dirección asociada en dicho vector. Así que cada elemento es una instrucción de salto que apunta a su RTI correspondiente, de no ser un salto se solaparía con el siguiente elemento del vector. Excepto el último elemento del vector, que no se solaparía con nada y que corresponde a las interrupciones FIQ. Se ha escogido intencionalmente así para ahorrarse el salto inicial.

5.1.11. Control de luces/sonido con pulsadores en lugar temporizadores

Los pulsadores izquierdo y derecho de nuestra placa externa están asociados a los puertos `GPIO 2` y `GPIO 3` respectivamente. Veremos cómo se genera una interrupción al pulsar cualquiera de los mismos.

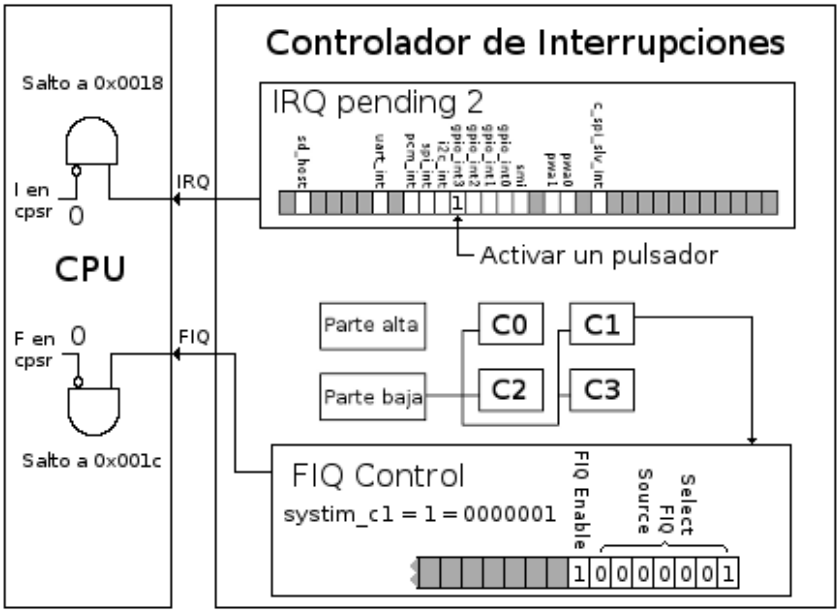

Figura 5.11: Interrupciones

El programa principal sería el siguiente:

Listado 5.7: Programa principal de inter6.s

```
/* Agrego vectores de interrupción */
        ADDEXC  0x18, irq_handler
        ADDEXC  0x1c, fiq_handler

/* Inicializo la pila en modos FIQ, IRQ y SVC */
        mov     r0, #0b11010001   @ Modo FIQ, FIQ&IRQ desact
        msr     cpsr_c, r0
        mov     sp, #0x4000
        mov     r0, #0b11010010   @ Modo IRQ, FIQ&IRQ desact
        msr     cpsr_c, r0
        mov     sp, #0x8000
        mov     r0, #0b11010011   @ Modo SVC, FIQ&IRQ desact
        msr     cpsr_c, r0
        mov     sp, #0x8000000

/* Configuro GPIOs 4, 9, 10, 11, 17, 22 y 27 como salida */
        ldr     r0, =GPBASE
        ldr     r1, =0b00001000000000000001000000000000
        str     r1, [r0, #GPFSEL0]
/* guia bits           xx99988877766655544433322211000*/
        ldr     r1, =0b00000000001000000000000000001001
        str     r1, [r0, #GPFSEL1]
        ldr     r1, =0b00000000001000000000000001000000
        str     r1, [r0, #GPFSEL2]

/* Enciendo LEDs       10987654321098765432109876543210*/
        mov     r1, #0b00000000000000000000001000000000
        str     r1, [r0, #GPSET0]

/* Habilito pines GPIO 2 y 3 (botones) para interrupciones*/
        mov     r1, #0b00000000000000000000000000001100
        str     r1, [r0, #GPFEN0]

/* Programo C1 para dentro de 2 microsegundos */
        ldr     r0, =STBASE
        ldr     r1, [r0, #STCLO]
        add     r1, #2
        str     r1, [r0, #STC1]

/* Habilito GPIO para IRQ */
```

```
        ldr     r0, =INTBASE
/* guia bits           10987654321098765432109876543210*/
        mov     r1, #0b00000000000010000000000000000000
        str     r1, [r0, #INTENIRQ2]

/* Habilito C1 para FIQ */
        mov     r1, #0b10000001
        str     r1, [r0, #INTFIQCON]

/* Habilito interrupciones globalmente */
        mov     r0, #0b00010011   @modo SVC, FIQ&IRQ activo
        msr     cpsr_c, r0

/* Repetir para siempre */
bucle:  b       bucle
```

Lo nuevo que vemos aquí es una escritura en el puerto `GPFEN0`. De esta forma le decimos al controlador de interrupciones que esos pines del GPIO serán los únicos que provoquen interrupciones, concretamente flancos con de bajada síncronos (justo en el momento en que el botón toca fondo).

El manejador FIQ es idéntico al del ejemplo anterior, saca el sonido que corresponde al LED por el altavoz, cambiando `C3` por `C1`.

Lo más relevante de este ejemplo está en la RTI asociada a la IRQ, que es la siguiente:

```
irq_handler:
        push    {r0, r1, r2}
        ldr     r0, =GPBASE
        ldr     r1, =cuenta

/* Apago todos LEDs    10987654321098765432109876543210*/
        ldr     r2, =0b00001000010000100000111000000000
        str     r2, [r0, #GPCLR0]

/* Leo botón pulsado */
        ldr     r2, [r0, #GPEDS0]
        ands    r2, #0b00000000000000000000000000001000
        beq     incre

/* Si es botón izquierdo, decrementar */
        str     r2, [r0, #GPEDS0] @ Reseteo flag b. izq
        ldr     r2, [r1]          @ Leo variable cuenta
        subs    r2, #1            @ Decremento
        moveq   r2, #6            @ Si es 0, volver a 6
```

```
        b       conti               @ Salto a conti

/* Si es botón derecho, incrementar */
incre:  mov     r2, #0b00000000000000000000000000000100
        str     r2, [r0, #GPEDS0] @ Reseteo flag b. der
        ldr     r2, [r1]          @ Leo variable cuenta
        add     r2, #1            @ Incremento
        cmp     r2, #7            @ Comparo si llego a 7
        moveq   r2, #1            @ Si es 7, volver a 1

/* Actualizo variable, enciendo LED y salgo */
conti:  str     r2, [r1], #-4     @ Escribo variable cuenta
        ldr     r2, [r1, +r2, LSL #3] @ Leo secuencia
        str     r2, [r0, #GPSET0] @ Escribo secuencia en LEDs
        pop     {r0, r1, r2}      @ Recupero registros
        subs    pc, lr, #4        @ Salgo RTI
```

Tenemos una bifurcación (saltos condicionales) debido a que cada botón es una fuente distinta de interrupción y tenemos que distinguir qué botón se ha pulsado. Aquí por suerte tenemos un puerto totalmente análogo al `STCS` de los temporizadores. Se llama `GPEDS0` (también hay otro `GPEDS1` para los GPIOs de 32 a 53 que no necesitamos) y sirve tanto para saber qué fuente ha producido la interrupción como para resetear su estado (y así permitir volver a ser interrumpidos por el mismo pin GPIO).

Con la instrucción `ands` comprobamos si un determinado bit está a 1 y lo indicamos en el flag Z. También podría valer la instrucción `tst`, que tiene la ventaja de no destruir el registro a la salida (de la misma forma que `cmp` es el equivalente no destructivo de `subs`).

Y por último debemos sacar la secuencia inversa a la que teníamos para que al pulsar el botón izquierdo las luces vayan hacia la izquierda y que con el botón derecho vayan en el otro sentido. Si la secuencia de izquierda a derecha era (6, 5, 4, 3, 2, 1, 6, 5, 4...), la inversa sería (1, 2, 3, 4, 5, 6, 1...). Es decir, incrementamos y cuando llegamos a 7 lo convertimos en 1. Ésto se hace con el siguiente fragmento:

```
        add     r2, #1            @ Incremento
        cmp     r2, #7            @ Comparo si llego a 7
        moveq   r2, #1            @ Si es 7, volver a 1
```

Nótese que aquí la opción destructiva `subs` (en lugar de `cmp`) no nos vale porque necesitamos el valor del registro después. Sí que podemos cambiarlo por un `teq` (la alternativa no destructiva de `eors`).

5.2. Ejercicios

5.2.1. Todo con IRQs

Modifica el último ejemplo (`inter5.s`) para controlar el altavoz también con IRQs, prescindiendo totalmente de las interrupciones FIQs.

5.2.2. Alargar secuencia a 10 y parpadeo

Partiendo de `inter5.s` (o del resultado del ejercicio anterior) haz las siguientes modificaciones. Si te resulta más cómodo, realízalas por orden.

- Sacar de la secuencia el LED 6 (el de más a la derecha) y ponerlo a parpadear continuamente con una cadencia de un segundo. En este momento tendrás que acortar la secuencia a 5.
- Duplica la secuencia a 10. Para ello utiliza el código Morse aplicado a los dígitos (todos tienen longitud 5). Cambia el punto (tono corto) por LED apagado y el guión (tono largo) por LED encendido. Por supuesto los nuevos códigos tendrán su sonido asociado, sigue las notas (saltándote sostenidos y bemoles) para completar la tabla.

5.2.3. Tope de secuencia y limitar sonido

Partiendo de `inter5.s` (o del resultado del ejercicio anterior) haz las siguientes modificaciones. Si te resulta más cómodo, realízalas por orden.

- Hasta ahora si llegamos al límite de la secuencia hemos comenzado por el principio, haciendo que la secuencia sea circular tanto en un sentido como en otro. Pues bien, ahora tienes que detectar dichos límites (tanto superior como inferior), poniendo una especie de tope al llegar al límite, que impida avanzar más. En caso de intentar avanzar en el sentido prohibido al llegar a un tope, en lugar de sacar el sonido que corresponda por el altavoz, auméntalo una escala (tope superior) o disminúyelo también una escala (tope inferior).
- Como habrás observado el sonido continuo resulta un tanto molesto después de un tiempo. Y con la indicación de los LEDs tenemos información suficiente para saber en qué posición de la secuencia estamos. Altera el programa para que sólamente suene el altavoz mientras el botón está pulsado, o lo que es lo mismo, para el sonido del altavoz cuando detectes un flanco de bajada en la señal GPIO correspondiente.

5.2.4. Reproductor de melodía sencilla

Escoge una melodía sencilla y trata de interpretarla. Emplea los LEDs a tu gusto para que cambien según la nota que esté sonando. Implementa las siguientes funciones en los pulsadores.

- **Pulsador izquierdo.** Cambio de tempo. La melodía debe comenzar a tempo normal (llamémoslo 1), y variar desde tempo lento (0) y tempo rápido (2) según la secuencia (0, 1, 2, 0...) cada vez que pulsemos dicho botón.
- **Pulsador derecho.** Iniciar/Parar/Reanudar. La melodía tiene una duración determinada y cuando acaba deja de sonar, no suena en modo bucle todo el tiempo. Si pulsamos dicho botón cuando está en silencio después que haya sonado la melodía, la función correspondiente sería la de iniciarla. Si lo pulsamos durante la reproducción actuaría a modo de pause (los LEDs se quedan congelados en el estado en el que estén), parando y reanudando la reproducción de la música.

En este ejemplo puedes profundizar todo lo que quieras. Por ejemplo empieza codificando los silencios, éstos son muy importantes y también forman parte de la melodía. Un segundo paso sería codificar la duración de las notas, si no lo has hecho ya. También es posible tener varios instrumentos sonando a la vez, aunque sólo dispongamos de un altavoz, busca por internet `1-bit music` o `beeper music` si quieres saber cómo se hace.

Apéndice A

Funcionamiento de la macro ADDEXC

Contenido

A.1. Finalidad y tipos de salto

Queremos implementar una macro que nos permita codificar las instrucciones de salto dentro del vector de interrupciones. Como en la arquitectura el bus de direcciones es de 32 bits no podemos codificar una instrucción de salto a cualquier dirección con una instrucción de 32 bits, puesto que no nos queda espacio para el código de operación.

Por esta razón existen dos tipos de salto, los saltos cortos ($\pm 32Mb$) y los saltos largos (todo el espacio de memoria). Un salto corto (aplicable también a saltos condicionales) en ensamblador se escribiría así.

```
        b       etiqueta
```

Los saltos largos no tienen instrucción propia, se realizan mediante la carga del registro `pc` partiendo de un dato en memoria.

```
        ldr     pc, a_etiq
        [...]
a_etiq: .word   etiqueta
```

Ésto en código máquina siempre se traduce a un direccionamiento relativo a `pc`, si estuviésemos depurando veríamos algo como esto.

```
        ldr     pc, [pc, #0x24]
```

Donde `0x24` es el desplazamiento (hemos elegido un valor arbitrario) donde se encontraría `a_etiq`. De hecho esto mismo ocurre cuando utilizamos `ldr` con el operador `=`, podríamos haber escrito esta otra instrucción con idéntico resultado.

```
        ldr     pc, =etiqueta
```

Es la forma de escribirlo en alto nivel, se produce exactamente el mismo código máquina que en el caso anterior. Debemos recordar que las instrucciones donde aparece el operador `=` ocupan 8 bytes, 4 para la propia instrucción y otros 4 para el dato que generará de forma transparente el propio ensamblador.

A.2. Elección: salto corto

Si buscamos código por internet lo más normal es encontrar tablas de excepciones completas que usan el salto largo en lugar del corto. Esto nos obliga a rellenar una tabla en la que la mayor parte de vectores no se usan y a que dicha tabla sea estática. Por esa razón nosotros emplearemos nuestro propio método basado en el salto corto.

Una desventaja es que tenemos que traducir una dirección (la de la RTI) al código máquina de un salto corto. Y la complicación viene más que nada porque el salto corto es relativo, es decir, depende del valor que tenga `pc` en el momento del salto.

La otra desventaja es que no podemos saltar más allá de 32Mb, pero para esto tendríamos que estar metidos en un proyecto bastante grande como para necesitar más de 32Mb de código, y aún así podemos solventarlo ubicando las RTI al principio.

A.3. Escribir una macro

En la primera línea ponemos la directiva `.macro` seguida del nombre de la macro `ADDEXC` y de los parámetros `vector, dirRTI` separados por coma.

```
      .macro    ADDEXC  vector, dirRTI
```

Luego escribiríamos el código de la macro, indicando los parámetros con `\vector` y `\dirRTI` para acabar con `.endm`.

A.4. Codificación de la instrucción de salto

Como las instrucciones son de 32 bits y siempre están alineadas a direcciones múltiplos de 4, en lugar de codificar el desplazamiento en bytes se hace en número de instrucciones (o grupos de 4 bytes). En código máquina una instrucción de salto incondicional tiene el formato indicado en la figura A.1.

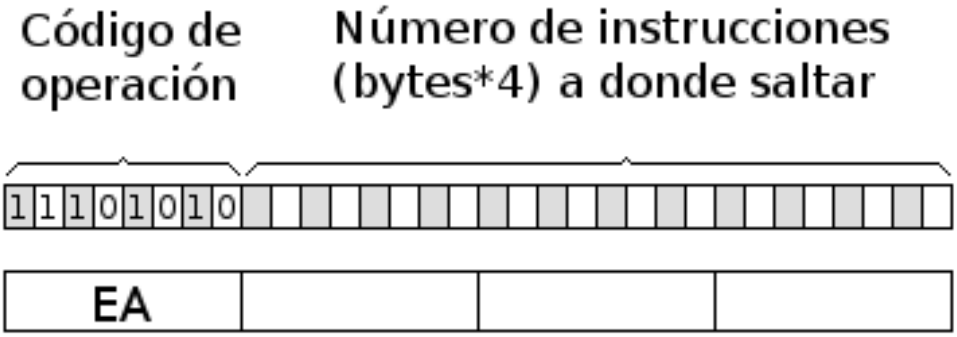

Figura A.1: Formato de instrucción de salto

Los pasos para calcular la instrucción de salto serían.

- Restar la dirección a saltar a la dirección actual
- Dividir entre 4
- Añadir `0xEA` al byte alto

Como todo son constantes en teoría podríamos implementar la macro con dos instrucciones. Desgraciadamente el preprocesador que usamos no es muy potente y si un operando es una etiqueta sólo nos permite operar con sumas y restas. No podemos hacer las divisiones o desplazamientos que necesitamos, con lo que emplearemos una tercera instrucción para hacer el desplazamiento.

La dirección actual es `\vector`, la de la RTI es `\dirRTI` y hay que restarle 8 por el segmentado de la CPU (ver figura A.2).

$instrucción = (\backslash dirRTI - \backslash vector - 8)/4 + 0xEA000000$
$instrucción = (\backslash dirRTI - \backslash vector)/4 + 0xE9FFFFFE$
$instrucción = (\backslash dirRTI - \backslash vector + 3A7FFFFF8)/4$
$instrucción = (\backslash dirRTI - \backslash vector + A7FFFFFB)ROR2$

Vemos cómo en el último paso hemos transformado una división en una rotación, donde los 2 bits menos significativos (ambos a 1) pasan a ser los más significativos tras la rotación.

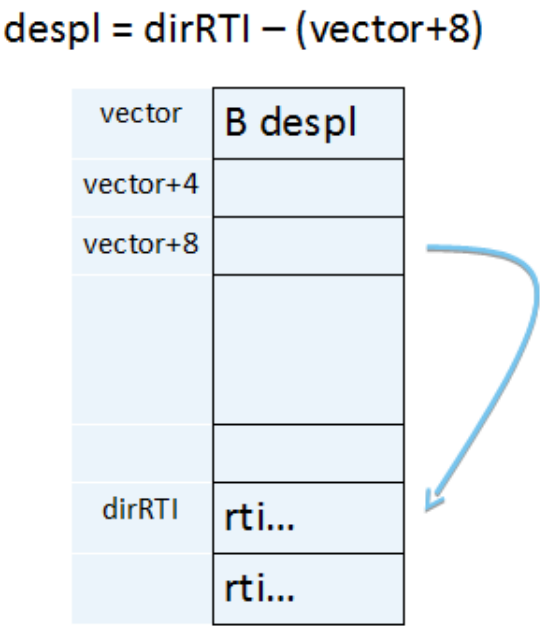

Figura A.2: Cálculo del desplazamiento

A.5. Resultado

El código final queda como sigue.

```
.macro    ADDEXC  vector, dirRTI
  ldr     r1, =(\dirRTI-\vector+0xa7ffffb)
  ROR     r1, #2
  str     r1, [r0, #\vector]
.endm
```

Como la arquitectura no nos permite escribir en una dirección absoluta, antes de invocar la macro debemos asegurarnos de que `r0` apunte al vector de interrupciones, es decir que valga `0`. En caso de usar esta macro como primera instrucción del programa Bare Metal podemos omitir la inicialización de `r0`, ya que en las especificaciones de carga del `kernel.img` se establece este valor.

Apéndice B

Funcionamiento de la placa auxiliar

Contenido

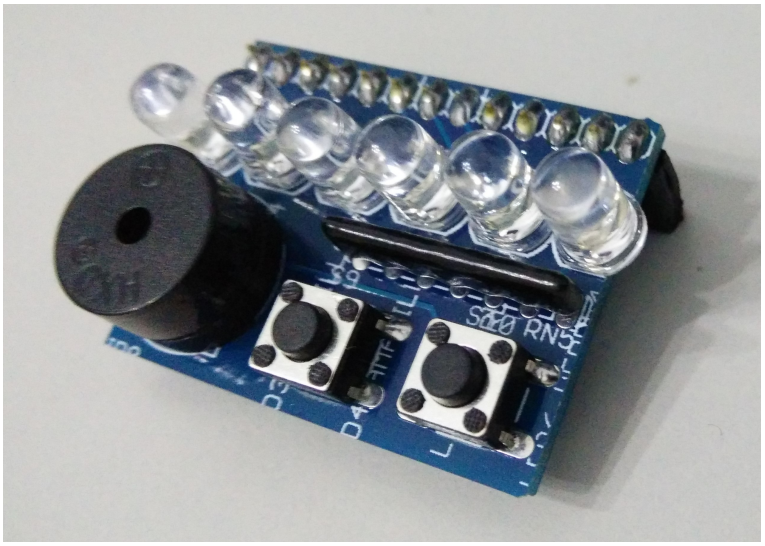

Figura B.1: Placa auxiliar

B.1. Esquema

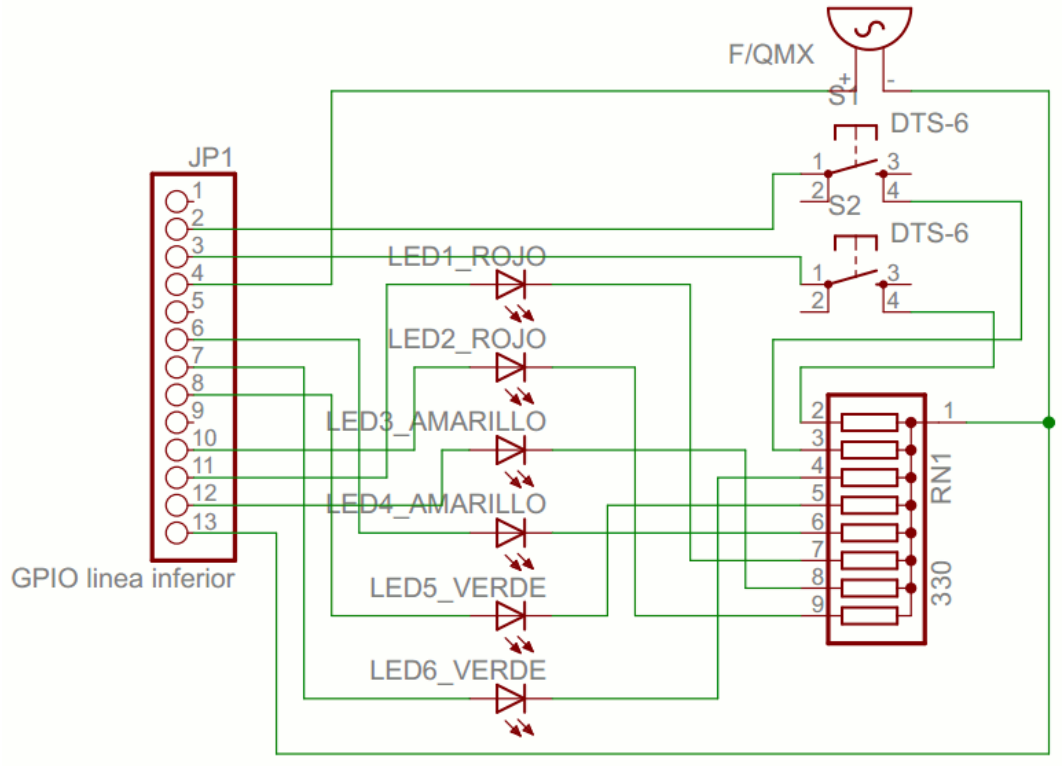

Figura B.2: Esquema del circuito

Es un circuito sencillo y se puede montar en una protoboard sin problemas, el esquema es el de la figura B.2. Se conecta en la fila inferior del conector GPIO, dejando libre la superior para el puerto serie y otros propósitos.

B.2. Pinout

El puerto GPIO varía ligeramente dependiendo del modelo de Raspberry. En nuestro caso la mayor diferencia está entre la revisión 1 y la 2, ya que el modelo B+ es compatible. Al ser idénticos los primeros 26 pines, cualquier periférico diseñado para la revisión 2 es compatible con el modelo B+ (pero no al contrario).

La zona marcada con un recuadro verde (en la figura B.3) es donde conectaremos nuestra placa auxiliar.

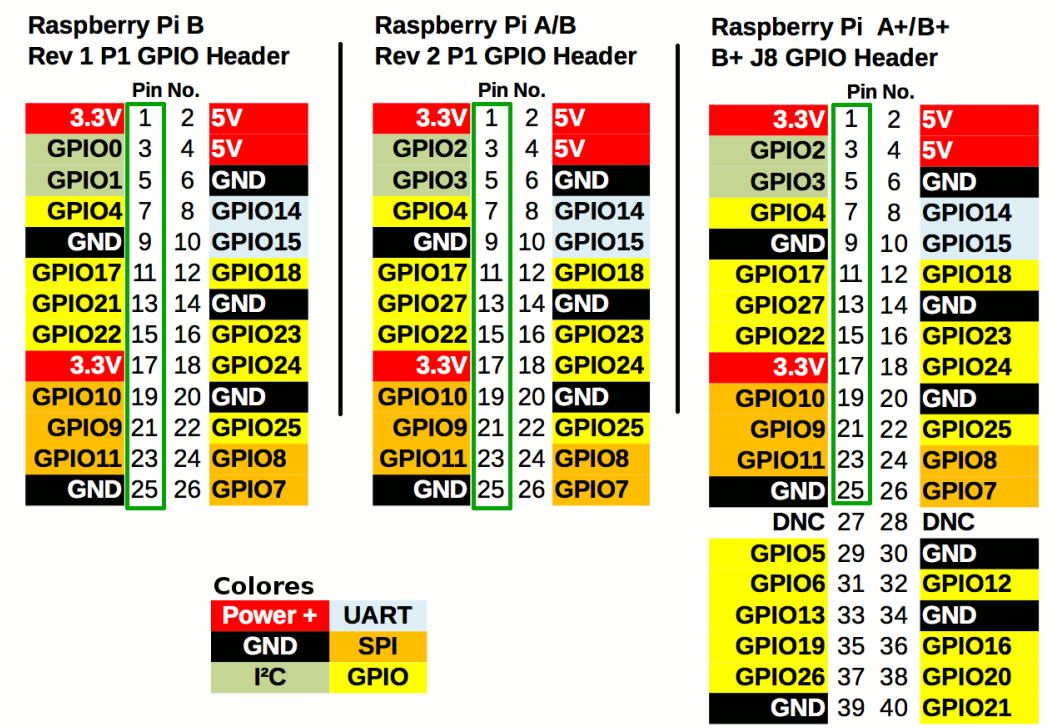

Figura B.3: Pinout del puerto GPIO

B.3. Correspondencia

En la siguiente tabla vemos la correspondencia entre puertos del GPIO y componentes. Los componentes son: 2 pulsadores, 6 LEDs y un altavoz piezoeléctrico. Los números marcados con asterisco tienen otra correspondencia en la revisión 1.

Nombre	GPIO	Tipo	Descripción
LED1	9	Salida	Diodo led color rojo
LED2	10	Salida	Diodo led color rojo
LED3	11	Salida	Diodo led color amarillo
LED4	17	Salida	Diodo led color amarillo
LED5	22	Salida	Diodo led color verde
LED6	27*	Salida	Diodo led color verde
BOT1	2*	Entrada	Pulsador izquierdo
BOT2	3*	Entrada	Pulsador derecho
ALT	4	Salida	Altavoz piezoeléctrico

Tabla B.1: Correspondencia entre pines y componentes

B.4. Funcionamiento

Los LEDs son salidas que se activan (encienden) cuando escribimos un 1 en el puerto correspondiente. Cuando están a 0 permanecen apagados. Podemos jugar con los tiempos de encendido/apagado para simular intensidades de luz intermedias.

El altavoz piezoeléctrico es otra salida, conectada al puerto GPIO 4. A diferencia de los LEDs no basta un 0 ó un 1 para activarlo, necesitamos enviar una onda cuadrada al altavoz para que éste suene. Es decir, hay que cambiar rápidamente de 0 a 1 y viceversa, además a una frecuencia que sea audible (entre 20 y 20000 Hz).

Por último tenemos los pulsadores. Eléctricamente son interruptores que conectan el pin a masa cuando están presionados. Cuando están en reposo entran en juego unas resistencias internas de la Raspberry (de pull-up) que anulan el comportamiento de las de pull-up/pull-down que se cambian por software. De esta forma los pulsadores envian un 0 lógico por el pin cuando están pulsados y un 1 cuando están en reposo.

Los pulsadores y el LED verde de la derecha se corresponden con distintos puertos según el modelo de Raspberry. Podemos hacer que nuestro programa sea compatible con todos los modelos, comprobando a la vez en las distintas entradas en el caso de los pulsadores, o escribiendo a la vez en ambas salidas en el caso del LED verde.

En la figura B.4 tenemos la correspondencia entre pines, componentes y puertos GPIO.

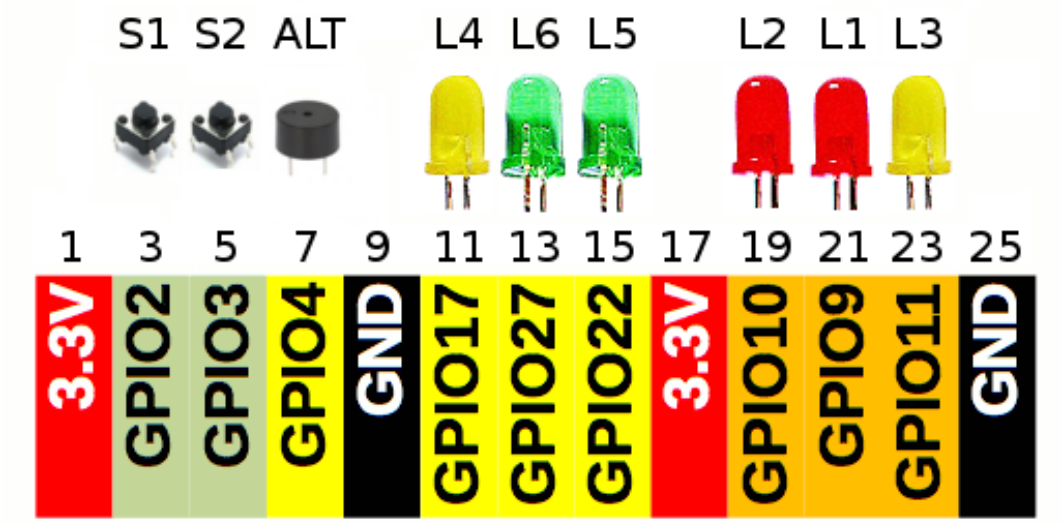

Figura B.4: Correspondencia LEDs y GPIO

B.5. Presupuesto

El presupuesto que mostramos a continuación es haciendo un pedido de 30 unidades, que son las necesarias para cubrir los puestos del laboratorio. En la tabla ponemos el precio unitario, para que sea fácil extrapolar los datos a otras situaciones. Cada puesto consta de un PC, con monitor, teclado y ratón conectado en una red local y con Linux instalado.

Componente	**Tienda**	**Precio**
Raspberry Pi Modelo A+	RS Online	17,26 €
USB-Serie con DTR	Ebay	1,44 €
PCB placa auxiliar	Seeedstudio	0,20 €
Altavoz	Ebay	0,08 €
Array resistencias	Aliexpress	0,06 €
2 pulsadores	Ebay	0,02 €
6 LEDs	Ebay	0,17 €
Conector hembra	Ebay	0,06 €
Total		19,29 €

Tabla B.2: Presupuesto unitario por puesto

En dicho presupuesto hemos incluido la Raspberry Pi, la placa auxiliar y el conversor USB-Serie para comunicar el PC con la Raspberry.

B.6. Diseño PCB

El diseño de la PCB se ha hecho con la versión de evaluación de la herramienta Cadsoft EAGLE, disponible en http://www.cadsoftusa.com/download-eagle

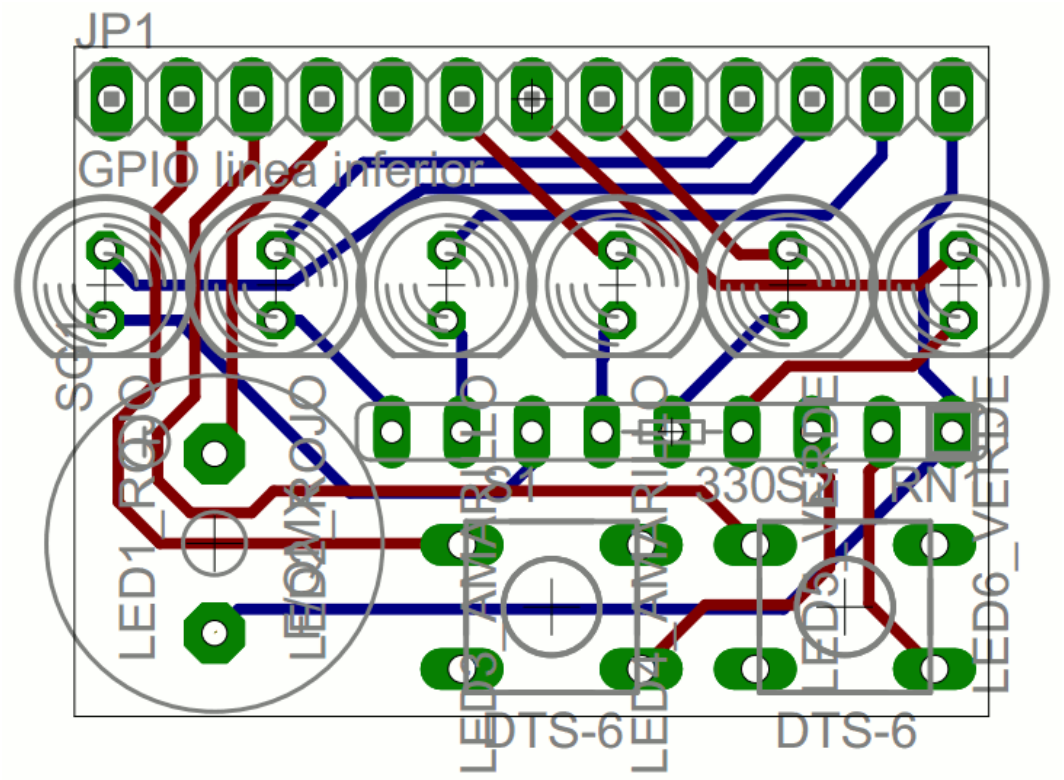

Figura B.5: Diseño PCB del circuito

Apéndice C

Cable serie y bootloaders

Contenido

C.1. Introducción

En esta sección profundizamos sobre dos métodos para cargar programas en Bare Metal sin necesidad de insertar y extraer continuamente la tarjeta SD. Existe un tercer método que no explicamos aquí, el del cable JTAG, pero pueden consultar los archivos README del repositorio de David Welch[9].

Este apéndice está basado en el contenido de dicho repositorio, y el código fuente del bootloader que mostramos aquí es idéntico, el cual reproducimos con permiso del autor.

C.2. Cable USB-serie desde el ordenador de desarrollo

Con esta opción hacemos todo el trabajo de ensamblado y enlazado en nuestro ordenador de desarrollo, para luego transferir el archivo Bare Metal por el puerto

serie directamente a la Raspberry. Necesitamos un adaptador USB-serie como el de la siguiente figura C.1.

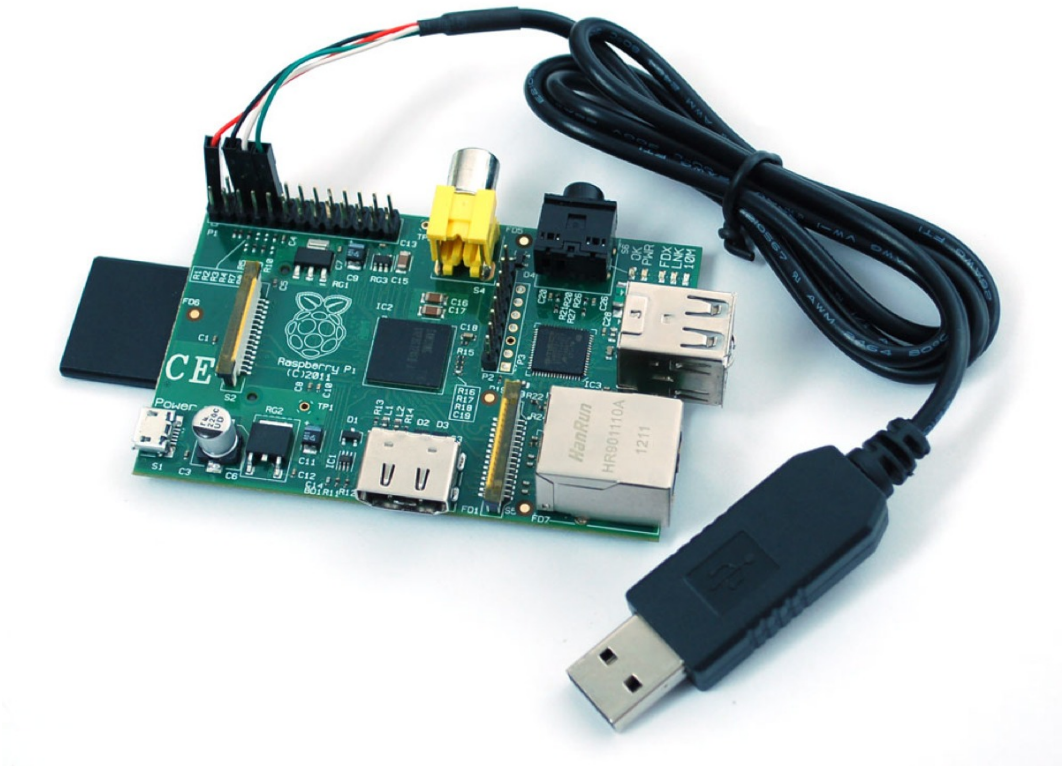

Figura C.1: Cable USB-serie

Lo primero que tenemos que hacer es cargar el bootloader https://github.com/dwelch67/raspberrypi/blob/master/bootloader05/kernel.img?raw=true en la SD y alimentar la Raspberry. Es necesario resetear la Raspberry cada vez que queramos cargar un programa Bare Metal nuevo, y esto se hace manualmente desenchufando y enchufando la alimentación, o bien con el método automático que explicamos en la última sección.

Debemos conectar los 3 cables que van del adaptador USB-serie a la Raspberry, con los pines tercero, cuarto y quinto de la fila superior del puerto GPIO. El tercer pin es la masa, en el adaptador es un cable negro o marcado con GND en la serigrafía. El cuarto pin es GPIO 14 ó TXD, que se corresponde con el pin RXD en el adaptador. Por último el quinto pin es GPIO 15 ó RXD y va conectado al pin TXD del adaptador. Nótese que los cables están cruzados, el pin que transmite desde el PC es el que recibe en la Raspberry y viceversa.

La primera vez que probemos el cable es recomendable probar con un programa

simple como un LED parpadeante, por ejemplo el último `esbn5.s` del capítulo 4. A partir del código fuente generamos el binario en Bare Metal, para lo cual necesitamos el toolchain (cadena de herramientas) ARM, de la que usaremos el ensamblador `as`, el enlazador `ld` y el copiador de secciones `objcopy`. A estas herramientas, que generan binarios o ejecutables para una plataforma diferente a la de la máquina que realiza la compilación, se las denomina *herramientas de compilación cruzada*.

En estos momentos tenemos el binario Bare Metal generado, llamémoslo `esbn5.img`. El siguiente paso es enviar este archivo por el puerto serie de forma que se ejecute en la Raspberry. Pero no lo enviamos de cualquier manera, sino que emplearemos un protocolo de transferencia que se llama XMODEM. Por suerte es uno de los protocolos mejor soportados por los emuladores de terminal.

Dependiendo de nuestra plataforma hay distintos emuladores de terminal disponibles. Para Windows tenemos `HyperTerminal` o `Tera Term`, y en Linux tenemos `minicom`, aunque para lo que queremos hacer (enviar un archivo) nos basta con el comando `sx`.

Antes de nada hay que configurar los parámetros del puerto serie en el emulador de terminal que estemos empleando. Los valores son: 8 bits de datos, sin paridad, 1 bit de parada, sin flujo de control y velocidad de transferencia de 115200 baudios. Son todos parámetros por defecto excepto la velocidad, por lo que hay que asegurarse de cambiar la velocidad antes de proceder a transferir el archivo.

Luego elegimos el protocolo, `XMODEM`, y le damos a transferir, seleccionando nuestro `esbn5.img` como archivo de origen. Si todo ha ido bien debería aparecer un mensaje indicándolo en nuestro programa terminal y observaremos el LED parpadenado en la Raspberry, prueba de que la transferencia ha sido exitosa.

En Linux es fácil automatizar este proceso con el comando `sx`, que es creando el siguiente script `enviar`.

```
stty -F /dev/ttyUSB0 115200
sx $1 < /dev/ttyUSB0 > /dev/ttyUSB0
```

Y para enviar el archivo anterior con este script escribimos bajo línea de comandos lo siguiente.

```
./enviar esbn5.img
```

C.3. Cable serie-serie que comunica dos Raspberries

Esta configuración es ideal si queremos emplear una Raspberry como ordenador de desarrollo. Aunque también está la alternativa de trabajar con un ordenador de desarrollo aparte conectado a una de las Raspberries mediante `ssh`. La ventaja de

esta última alternativa es que podemos compilar desde la Raspberry sin necesidad de tener instaladas las herramientas de compilación cruzada en tu ordenador. Y otra ventaja es que no necesitas estar físicamente cerca de la Raspberry ni tener enchufado el adaptador USB, te puedes conectar inalámbricamente a la Raspberry mediante un router Wifi (con cable Ethernet entre el router y la Raspberry).

Lo primero es diferenciar las dos Raspberries. A una la llamamos Raspberry de desarrollo, en la cual tendremos instalado Raspbian y es en la que trabajamos directamente (con teclado y pantalla) o bien nos conectamos con ella mediante ssh. A la otra la llamamos Raspberry Bare Metal, en la que sobreescribimos el kernel.img de la SD con el mismo bootloader de antes. Es en esta Raspberry donde se ejecutan los programas Bare Metal que vamos a desarrollar y por tanto donde enchufaremos nuestra placa auxiliar.

La conexión entre ambas Raspberries se hace uniendo ambas masas y cruzando los cables TXD y RXD de cada puerto serie, como viene indicado en la figura C.2.

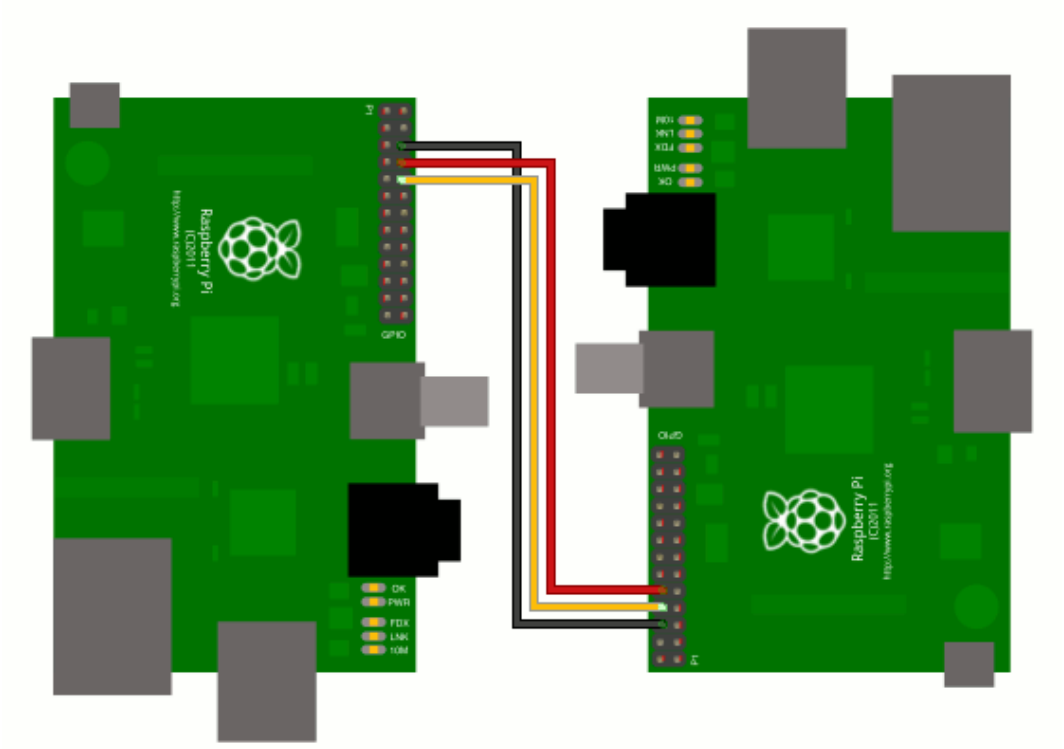

Figura C.2: Dos raspberries en serie cruzado

Por defecto el puerto serie en Raspbian viene configurado como salida de consola. Esta configuración no nos interesa, se usa para diagnosticar errores mostrando por un terminal los mensajes del arranque. Pero nosotros queremos usarlo como un

puerto serie genérico, para lo cual es necesario hacer los siguientes cambios.

En el archivo `/etc/inittab` descomentamos la línea que empieza con `T0:23...` y que hace mención a la cadena `ttyAMA0`, y guardamos el archivo.

Luego en el archivo `/boot/cmdline.txt` buscamos los dos sitios (puede haber uno sólo) donde aparece `ttyAMA0`. Borramos los textos que hay entre espacios y que incluyen el `ttyAMA0`, y después guardamos.

Para comprobar que todo ha ido bien reseteamos la Rasbperry con `sudo reboot` y tras el arranque escribimos.

```
cat /proc/cmdline
```

Comprobando que efectivamente no hay se hace ninguna referencia a `ttyAMA0`, y luego escribimos este otro comando.

```
ps aux | grep ttyAMA0
```

Para verificar que el único proceso que se lista en la salida es el del propio comando `ps` y no existen otros.

Llegados a este punto ya tenemos el puerto serie disponible para nosotros. El resto de pasos serían como en el caso anterior, pero cambiando la referencia que se hace al puerto. Donde antes aparecía `/dev/ttyUSB0` (o algo similar) lo cambiamos por `/dev/ttyAMA0`.

C.4. Reseteo automático

Resulta tedioso tener que desenchufar y enchufar la Raspberry cada vez que queremos introducir un nuevo programa Bare Metal. Una solución intermedia es soldar los dos pines de Reset que están serigrafiados como `P6` en la Raspberry 2.0 o como `RUN` en el modelo A+/B+. A éstos pines le podemos conectar un pulsador, con lo que simplificamos el reseteo, en lugar de desenchufar y enchufar pulsamos un botón.

Sin embargo es muy conveniente buscar una solución totalmente automática, mediante la cual el propio script que envía el archivo envíe una señal de Reset justo antes del envío. Para evitar confusiones, en lugar de montar los dos pines del conector, montamos sólo uno, el de la señal que provoca el Reset (el otro es la masa). Viene señalada con un círculo rojo en la figura C.3.

De lo que se trata ahora es de enviar un pulso negativo a esa señal. En el caso del cable USB-Serie lo haremos por una señal programable que no se emplea en el enlace serie llamada `DTR`. En el caso de conexión serie-serie entre dos Raspberries usaremos el pin GPIO 18 (justo a la derecha de RXD). Valdría cualquier otro pin, pero por cercanía empleamos este.

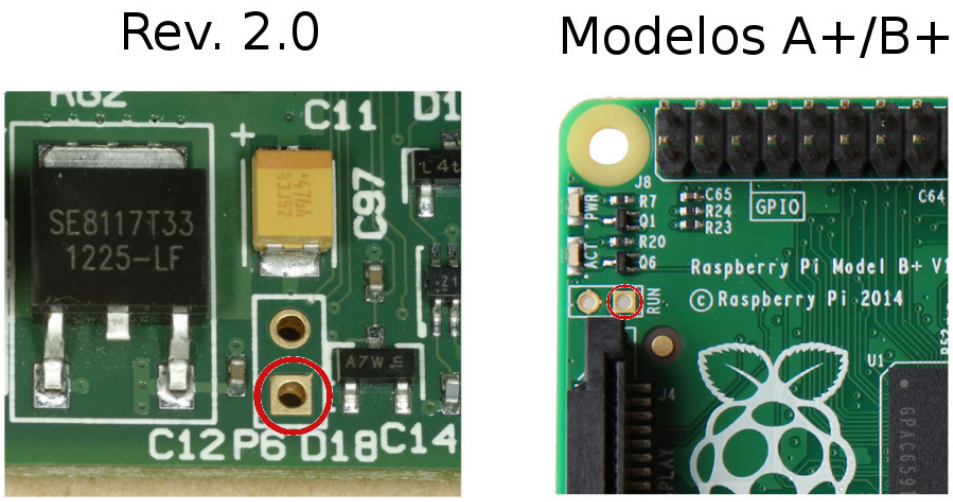

Figura C.3: Señal de Reset donde montar el pin

Desgraciadamente con el cable USB-serie no podemos utilizar el comando `sx`, ya que dichos comando sobreescribe el valor del pin `DTR`. El siguiente programa, además de enviar el pulso `DTR` para provocar el reset, trasfiere el archivo a la Raspberry. Con esto ya no necesitaríamos la herramienta `sx`.

Listado C.1: sendx.c

```
#include <termios.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/ioctl.h>

int main(int argc, char* argv[]){
  FILE *fi;
  int i, fd;
  unsigned char j, eot= 0x04, buf[132];
  struct termios attr;
  buf[0]++;
  buf[2]--;
  if( argc != 2 )
    printf( "sendx v1.00 by Antonio Villena, 21 Dec 2014\n\n"
            "  sendx <input_file>\n\n"
            "  <input_file>   Bare Metal input binary file\n\n"),
    exit(0);
```

```
fi= fopen(argv[1], "r");
if( !fi )
  printf("Couldn't open file %s\n", argv[1]),
  exit(1);
fd= open("/dev/ttyUSB0", O_RDWR | O_NOCTTY | O_NDELAY);
if( fd == -1 )
  printf("Couldn't open serial device /dev/ttyUSB0\n"),
  exit(1);
tcgetattr(fd, &attr);
attr.c_cflag= B115200 | CS8;
attr.c_oflag= attr.c_iflag= attr.c_lflag= 0;
tcsetattr(fd, TCSANOW, &attr);
i= TIOCM_DTR;
ioctl(fd, TIOCMSET, &i);
usleep( 100*1000 );
i= 0;
ioctl(fd, TIOCMSET, &i);
fcntl(fd, F_SETFL, 0);
usleep( 50*1000 );
tcflush(fd, TCIOFLUSH);
read(fd, &j, 1);
printf("Initializing file transfer...\n");
while ( fread(buf+3, 1, 128, fi)>0 ){
  buf[1]++;
  buf[2]--;
  for ( buf[131]= 0, i= 3; i < 131; i++ )
    buf[131]+= buf[i];
  if( write(fd, buf, 132) != 132 )
    printf("Error writing to serial port\n"),
    exit(-1);
  read(fd, &j, 1);
  if( j == 6 )  // ACK
    printf("."),
    fflush(stdout);
  else
    printf("Received %d, expected ACK\n", j),
    exit(-1);
}
write(fd, &eot, 1);
read(fd, &j, 1);
if( j != 6 )  // ACK
  printf("No ACK for EOT message\n"),
  exit(-1);
```

```
  printf("\nFile transfer successfully.\n");
  fclose(fi);
  close(fd);
}
```

En el caso de las dos Raspberries instalamos primero el paquete `wiringPi`.

```
git clone git://git.drogon.net/wiringPi
cd wiringPi
./build
```

E incluímos el pulso reset mediante comandos, por ejemplo nuestro script que compila y envía el archivo (todo en un paso) quedaría así.

```
gpio export 18 out
as -o tmp.o $1
gpio export 18 in
ld -e 0 -Ttext=0x8000 -o tmp.elf tmp.o
objcopy tmp.elf -O binary tmp.img
stty -F /dev/ttyAMA0 115200
sx tmp.img < /dev/ttyAMA0 > /dev/ttyAMA0
```

Observamos que el pulso de reset dura lo que tarde el programa en ensamblar, duración más que suficiente como para provocar un Reset en la Raspberry Bare Metal. Para llamar al script escribimos algo como esto.

```
./compila esbn5.s
```

C.5. Código fuente del bootloader

Mostramos la parte principal del programa bootloader, el resto de archivos están en el repositorio.

Listado C.2: bootloader05.c

```
//------------------------------------
unsigned char xstring[256];
//------------------------------------
int notmain ( void )
{
    unsigned int ra;
    unsigned int rx;
    unsigned int addr;
    unsigned int block;
    unsigned int state;
```

```
    unsigned int crc;

    uart_init();
    hexstring(0x12345678);
    hexstring(GETPC());
    hexstring(ARMBASE);
    timer_init();

//SOH 0x01
//ACK 0x06
//NAK 0x15
//EOT 0x04

//block numbers start with 1

//132 byte packet
//starts with SOH
//block number byte
//255-block number
//128 bytes of data
//checksum byte (whole packet)
//a single EOT instead of SOH when done, send an ACK on it too

    block=1;
    addr=ARMBASE;
    state=0;
    crc=0;
    rx=timer_tick();
    while(1)
    {
        ra=timer_tick();
        if((ra-rx)>=4000000)
        {
            uart_send(0x15);
            rx+=4000000;
        }
        if((uart_lcr()&0x01)==0) continue;
        xstring[state]=uart_recv();
        rx=timer_tick();
        if(state==0)
        {
            if(xstring[state]==0x04)
```

```
        {
            uart_send(0x06);
            for(ra=0;ra<30;ra++) hexstring(ra);
            hexstring(0x11111111);
            hexstring(0x22222222);
            hexstring(0x33333333);
            uart_flush();
            BRANCHTO(ARMBASE);
            break;
        }
    }
    switch(state)
    {
        case 0:
        {
            if(xstring[state]==0x01)
            {
                crc=xstring[state];
                state++;
            }
            else
            {
                //state=0;
                uart_send(0x15);
            }
            break;
        }
        case 1:
        {
            if(xstring[state]==block)
            {
                crc+=xstring[state];
                state++;
            }
            else
            {
                state=0;
                uart_send(0x15);
            }
            break;
        }
        case 2:
        {
```

```
                if(xstring[state]==(0xFF-xstring[state-1]))
                {
                    crc+=xstring[state];
                    state++;
                }
                else
                {
                    uart_send(0x15);
                    state=0;
                }
                break;
            }
            case 131:
            {
                crc&=0xFF;
                if(xstring[state]==crc)
                {
                    for(ra=0;ra<128;ra++)
                    {
                        PUT8(addr++,xstring[ra+3]);
                    }
                    uart_send(0x06);
                    block=(block+1)&0xFF;
                }
                else
                {
                    uart_send(0x15);
                }
                state=0;
                break;
            }
            default:
            {
                crc+=xstring[state];
                state++;
                break;
            }
        }
    }
    return(0);
}
```

Al comienzo se envían tres cadenas en hexadecimal por el puerto (0x12345678, GETPC() y ARMBASE) para indicar que el bootloader está listo para recibir. Esto lo podemos ver si empleamos un programa terminal como `minicom` para leer del puerto.

Después se inicializan algunas variables y nos metemos en el bucle principal. Se supone que el primer byte que tenemos que recibir desde el host es `SOH`, en hexadecimal es `0x01`. Si pasado un tiempo no recibimos nada, enviamos un `NAK` (`0x15`) para indicarle al host que estamos vivos. En realidad este comando sirve para decirle al host que el paquete recibido es erróneo, que nos lo envíe nuevamente.

El host enviará a la Raspberry el archivo en trozos de 128 bytes cada uno (rellenando el último trozo con ceros hasta que ocupe 128 bytes) con este formato.

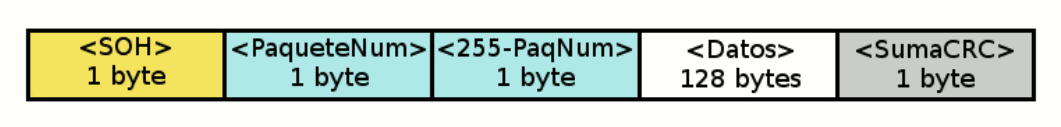

Figura C.4: Formato de paquete XMODEM

Se trata del byte `SOH` seguido del número de bloque, luego tenemos otra vez el número de bloque pero complementado, a continuación los 128 bytes de datos para acabar con un último byte de suma de comprobación. Este último byte es la suma de todos los anteriores, quedándonos con los 8 bits menos significativos del resultado.

Entonces la Raspberry lleva la cuenta del byte por el que vamos dentro de dicho paquete a partir de `switch(state)`. De tal forma que si `state` vale 0, lo que esperamos es `SOH` o `EOT`, cualquier otro valor indica que algo va mal por tanto enviamos un `NAK` al host y ponemos `state` a cero.

Para los estados 1 y 2 simplemente comprobamos que el byte recibido coincide con el número de bloque, y reportamos error en caso contrario de la misma forma que antes (enviando `NAK` y `state=0`).

Luego tenemos los estados que van entre 3 y 131, en los que vamos escribiendo el fichero en memoria e incrementando el puntero, a la vez que vamos calculando el byte de suma para la comprobación.

Por último tenemos el estado 131, en el cual ya hemos recibido los bytes de datos y lo que leemos ahora es el byte de suma de comprobación. Comparamos que coincide con el valor esperado, respondiendo con `ACK`, o notificamos del error como siempre (con `NAK` y `state=0`).

En cuanto el host recibe el `ACK` del último paquete enviado, éste en lugar de enviar de nuevo un paquete completo, envía un sólo byte, `EOT`, para indicar a la Raspberry que ya no quedan más paquetes por enviar y se acaba la transmisión.

Esta situación la comprueba la Raspberry al principio de cada paquete, de tal forma que si recimibos un `EOT` del host damos por acabada la transmisión y ejecu-

tamos el archivo Bare Metal leído con `BRANCHTO`, que en bajo nivel se corresponde con saltar a `0x8000`.

En la figura C.5 tenemos un ejemplo completo de transmisión. En él se envían 4 paquetes, con errores y reenvíos en los paquetes 2 y 3. Podría tratarse de un archivo que ocupase 500 bytes, y que la utilidad `sx` haya rellenado en el último paquete 12 bytes con ceros, para que de esta forma todos los paquetes ocupen 128 bytes (la parte útil, contando cabeceras y demás cada paquete ocupa 132 bytes).

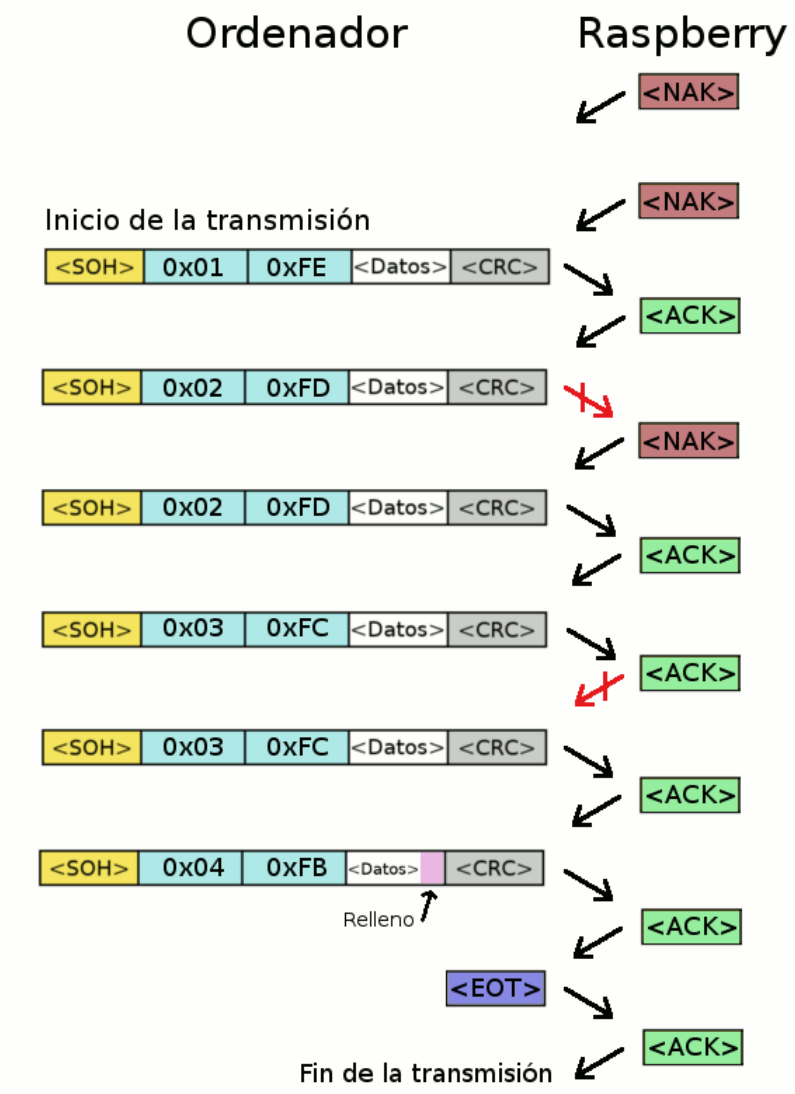

Figura C.5: Ejemplo de transmisión

Apéndice D

Resistencias programables de pull-up y pull-down

Contenido

D.1. Introducción

En general las resistencias de pull-up y pull-down son resistencias que se ponen en las entradas para fijar la tensión que de otra forma quedaría indeterminada, al estar en situación de circuito abierto o alta impedancia. El ejemplo típico donde se usa es en un pulsador. Eléctricamente un pulsador no es más que un interruptor que deja pasar la corriente cuando está pulsado y se queda en circuito abierto en su posición de reposo (sin pulsar). De los dos contactos que tiene, uno se conecta a masa y el otro al pin de entrada de la Raspberry. Así que cuando lo pulsamos hacemos un corto que llevaría los cero voltios de la masa al pin de entrada (enviamos un cero lógico), pero cuando está sin pulsar no enviamos nada al pin, éste se queda en lo que se denomina alta impedancia.

Todos los pines del GPIO en la Raspberry se pueden configurar por software para que se comporten como queramos: o bien sin resistencia, o con una resistencia

a Vcc (pull-up) o con una resistencia a masa (pull-down). Este tipo de resistencias son débiles (weak) debido a que están dentro de la pastilla (SoC) y se implementan con transistores. Se puede anular el efecto de estas resistencias poniendo resistencias externas.

D.2. Pulsadores en la placa auxiliar

En la placa auxiliar tenemos dos pulsadores conectados a GPIO 2 y GPIO 3. No es casualidad que estén conectados concretamente a esos dos pines. Son los únicos pines que tienen resistencias externas de pull-up, concretamente de 1K8, que anulan cualquier configuración interna que pongamos. La razón es porque las resistencias internas son débiles, tienen un valor aproximado de unos 50K. Cuando hay dos resistencias en paralelo como es el caso, la de menor valor anula el efecto de la de mayor valor.

Por tanto si configuramos GPIO 2 ó GPIO 3 como entradas, independientemente del valor que configuremos por software, se comportarán siempre como si sólo tuviesen una resistencia de pull-up.

El propósito de este apéndice es aprender a cambiar la configuración de los pull-ups/pull-downs en caso de usar otras placas auxiliares distintas. En la nuestra las únicas entradas (pulsadores) que hay están resueltas con las resistencias antes comentadas que tiene la Raspberry sólo en esos dos pines.

D.3. Ejemplo de aplicación

El montaje que proponemos es con uno de los pines de la fila superior, en concreto el pin GPIO 18 que hay a la derecha de los pines del puerto serie. En estos ejemplos no vamos a requerir la placa auxiliar, de esta forma dejamos libres los pines Vcc (3.3V), ya que necesitaremos uno para el último ejemplo.

D.3.1. Pulsador a masa sin cambiar configuración

En este primer ejemplo vamos a tratar de encender el LED interno de la Raspberry llamado ACT (OK) mediante un pulsador externo. El primer montaje sería el de la figura. El esquema sería el de la figura D.1.

Ahora escribimos el código. Como el pin GPIO que controla dicho LED es distinto en los modelos normales que en los A+/B+, enviamos la señal a ambos pines. En el modelo normal sería GPIO 16 y en el plus, el GPIO 47.

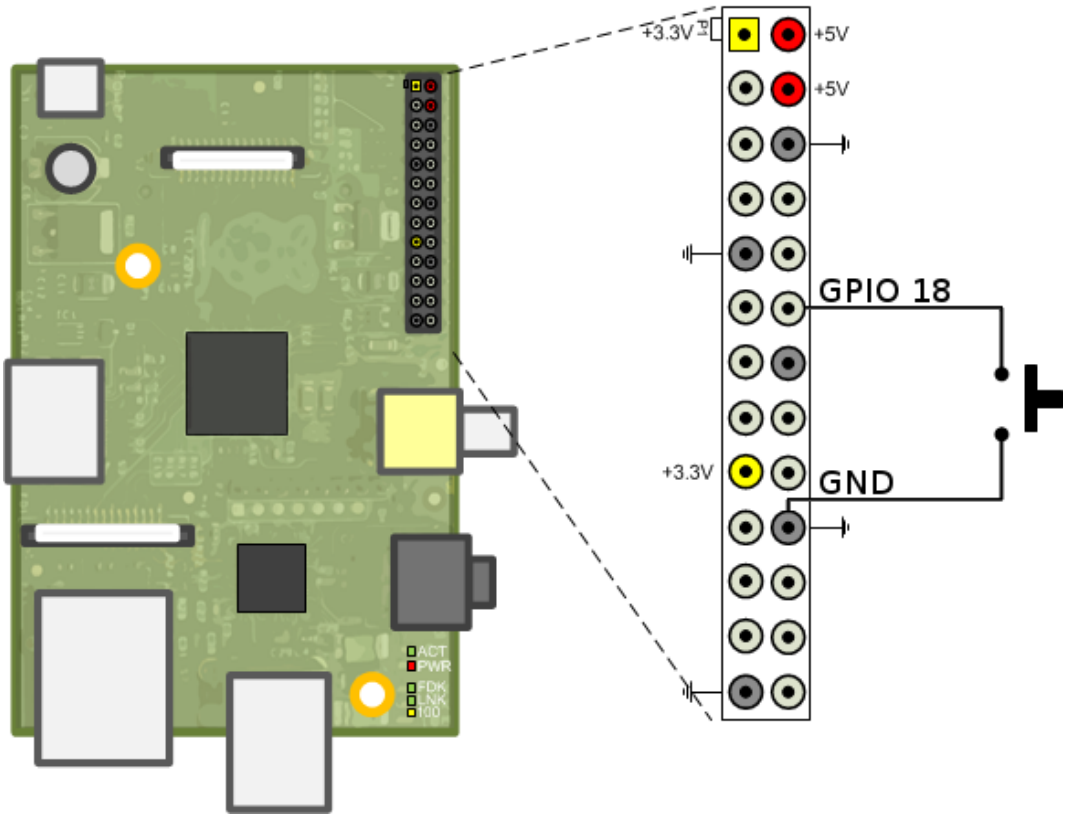

Figura D.1: Pulsador a masa

Listado D.1: apend1.s

```
        .include  "inter.inc"
.text
        ldr     r0, =GPBASE
/* guia bits           xx99988777666555444333222111000*/
        mov     r1, #0b00000000000001000000000000000000
        str     r1, [r0, #GPFSEL1]
/* guia bits           xx99988777666555444333222111000*/
        mov     r1, #0b00000000001000000000000000000000
        str     r1, [r0, #GPFSEL4]
/* guia bits           10987654321098765432109876543210*/
        mov     r2, #0b00000000000000001000000000000000
/* guia bits           32109876543210987654321098765432*/
        mov     r3, #0b00000000000000001000000000000000
bucle:  str     r2, [r0, #GPCLR0]    @ apago GPIO 16
        str     r3, [r0, #GPCLR1]    @ apago GPIO 47
```

```
        ldr     r1, [r0, #GPLEV0]
/* guia bits            10987654321098765432109876543210*/
        tst     r1, #0b00000000000001000000000000000000
        streq   r2, [r0, #GPSET0]    @ enciendo GPIO 16
        streq   r3, [r0, #GPSET1]    @ enciendo GPIO 47
        b       bucle
```

Probamos el código y comprobamos que al pulsar el botón izquierdo no pasa nada, el LED está siempre encendido.

Esto se debe a que por defecto el pin GPIO 18 está configurando con una resistencia de pull-down y nosotros necesitamos una de pull-up, de lo contrario siempre leeremos un cero por dicho pin. Los valores por defecto (tras el reset) se pueden consultar en la página 103 del datasheet, aunque la mayor parte de los pines disponibles por el puerto estan a pull-down.

Para solventar ésto hay tres opciones: o conectar el otro terminal del interruptor a Vcc en lugar de a GND, o configurar el pin para cambiarlo de pull-down a pull-up, o conectar una resistencia externa a Vcc que anule el pull-down interno. Nosotros vamos a explorar las dos primeras opciones.

D.3.2. Pulsador a masa cambiando configuración

En este ejemplo vamos a configurar GPIO 18 a pull-up de acuerdo a la siguiente figura D.2.

Para configurar un pin determinado en pull-up/pull-down/desconectado seguimos los siguientes pasos.

1. Escribir en `GPPUD` el tipo de resistencia que queremos. Un 0 sería si no queremos resistencia, un 1 si es de pull-down ó un 2 si lo que queremos es un pull-ups.

2. Esperar 150 ciclos. Esto provee el tiempo requerido de `set-up` para controlar la señal.

3. Escribir en `GPPUDCLK0/1` un 1 en la posición de los pines que queramos modificar, mientras que los que estén a 0 mantendrán su antiguo estado.

4. Esperar otros 150 ciclos. Con esto le damos tiempo de `hold` suficiente a la señal.

5. Poner GPPUD en su estado de reposo, que sería a valor 0 (desactivado).

6. Escribir un 0 en `GPPUDCLK0/1`.

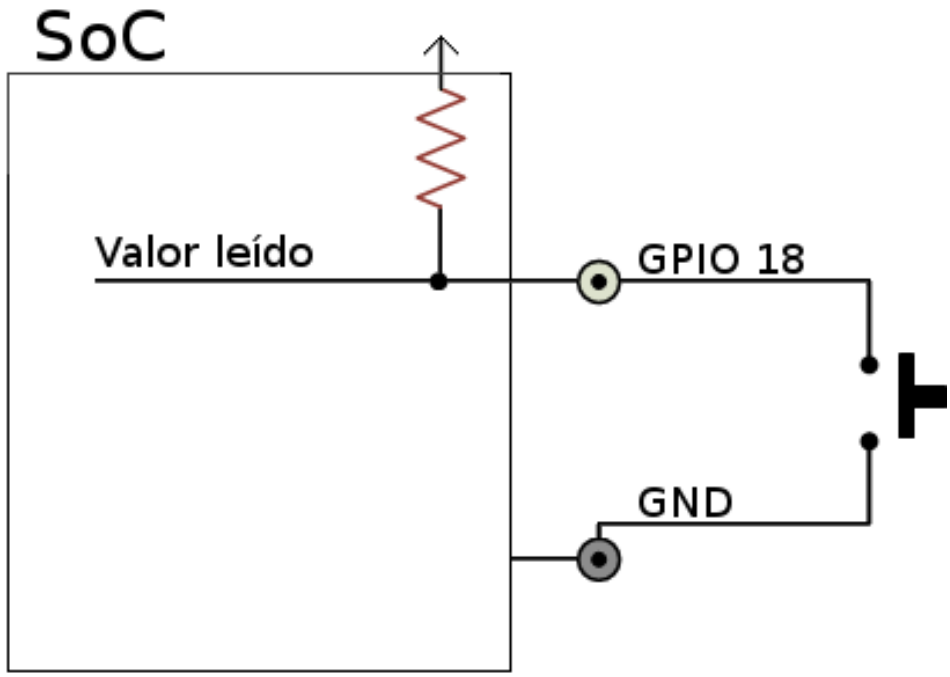

Figura D.2: Resistencia interna de pull-up

Una de las cosas que tenemos que hacer es esperar 150 ciclos (como mínimo). Como sabemos que un salto condicional tarda al menos dos ciclos en ejecutarse, nuestra rutina de retardo sería la siguiente.

```
wait:   mov     r1, #50
wait1:  subs    r1, #1
        bne     wait1
        bx      lr
```

Y el código que hace todo lo anterior, para poner a pull-up el GPIO 18 (donde hemos puesto el pulsador) es el siguiente.

```
        str     r1, [r0, #GPPUD]
        bl      wait
/* guia bits            10987654321098765432109876543210*/
        mov     r1, #0b00000000000001000000000000000000
        str     r1, [r0, #GPPUDCLK0]
        bl      wait
        mov     r1, #0
        str     r1, [r0, #GPPUD]
        str     r1, [r0, #GPPUDCLK0]
```

El ejemplo completo quedaría así.

Listado D.2: apend2.s

```
        .include  "inter.inc"
.text
        ldr     r0, =GPBASE
/* guia bits           xx99988777666555444333222111000*/
        mov     r1, #0b00000000000001000000000000000000
        str     r1, [r0, #GPFSEL1]
/* guia bits           xx99988777666555444333222111000*/
        mov     r1, #0b00000000001000000000000000000000
        str     r1, [r0, #GPFSEL4]
        mov     r1, #2
        str     r1, [r0, #GPPUD]
        bl      wait
/* guia bits           10987654321098765432109876543210*/
        mov     r1, #0b00000000000001000000000000000000
        str     r1, [r0, #GPPUDCLK0]
        bl      wait
        mov     r1, #0
        str     r1, [r0, #GPPUD]
        str     r1, [r0, #GPPUDCLK0]
/* guia bits           10987654321098765432109876543210*/
        mov     r2, #0b00000000000000001000000000000000
/* guia bits           32109876543210987654321098765432*/
        mov     r3, #0b00000000000000001000000000000000
bucle:  str     r2, [r0, #GPCLR0]   @ apago GPIO 16
        str     r3, [r0, #GPCLR1]   @ apago GPIO 47
        ldr     r1, [r0, #GPLEV0]
/* guia bits           10987654321098765432109876543210*/
        tst     r1, #0b00000000000001000000000000000000
        streq   r2, [r0, #GPSET0]   @ enciendo GPIO 16
        streq   r3, [r0, #GPSET1]   @ enciendo GPIO 47
        b       bucle

wait:   mov     r1, #50
wait1:  subs    r1, #1
        bne     wait1
        bx      lr
```

Comprobamos cómo ahora sí funciona, y mientras tenemos el botón presionado, el LED se enciende, apagándose en cuanto lo soltamos.

D.3.3. Pulsador a Vcc sin cambiar configuración

Cambiamos el montaje, y en lugar de a GND conectamos el pulsador a Vcc según la figura D.3.

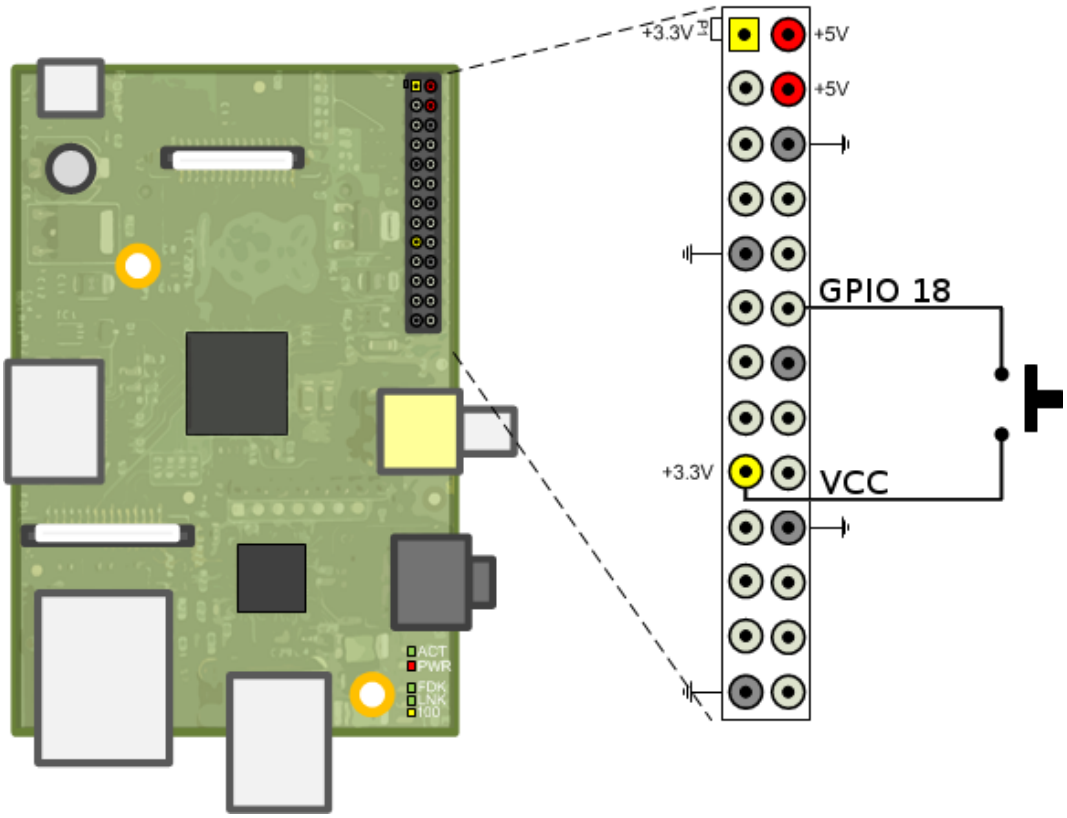

Figura D.3: Pulsador a Vcc

De esta forma aprovechamos que ese pin en concreto está conectado a pull-down tras el reset, por lo que no habría que cambiar la configuración del pin para obtener lo que vemos en la figura D.4.

Prácticamente tendríamos el mismo código que en `apend1.s` no nos funcionaba, la única diferencia es que cambiamos los `streq` por `strne`.

Listado D.3: apend3.s

```
        .include  "inter.inc"
.text
        ldr     r0, =GPBASE
/* guia bits            xx99988877766655544433322211000*/
        mov     r1, #0b00000000000001000000000000000000
```

```
        str     r1, [r0, #GPFSEL1]
/* guia bits          xx99988777666555444333222111000*/
        mov     r1, #0b00000000001000000000000000000000
        str     r1, [r0, #GPFSEL4]
/* guia bits          10987654321098765432109876543210*/
        mov     r2, #0b00000000000000001000000000000000
/* guia bits          32109876543210987654321098765432*/
        mov     r3, #0b00000000000000001000000000000000
bucle:  str     r2, [r0, #GPCLR0]   @ apago GPIO 16
        str     r3, [r0, #GPCLR1]   @ apago GPIO 47
        ldr     r1, [r0, #GPLEV0]
/* guia bits          10987654321098765432109876543210*/
        tst     r1, #0b00000000000001000000000000000000
        strne   r2, [r0, #GPSET0]   @ enciendo GPIO 16
        strne   r3, [r0, #GPSET1]   @ enciendo GPIO 47
        b       bucle
```

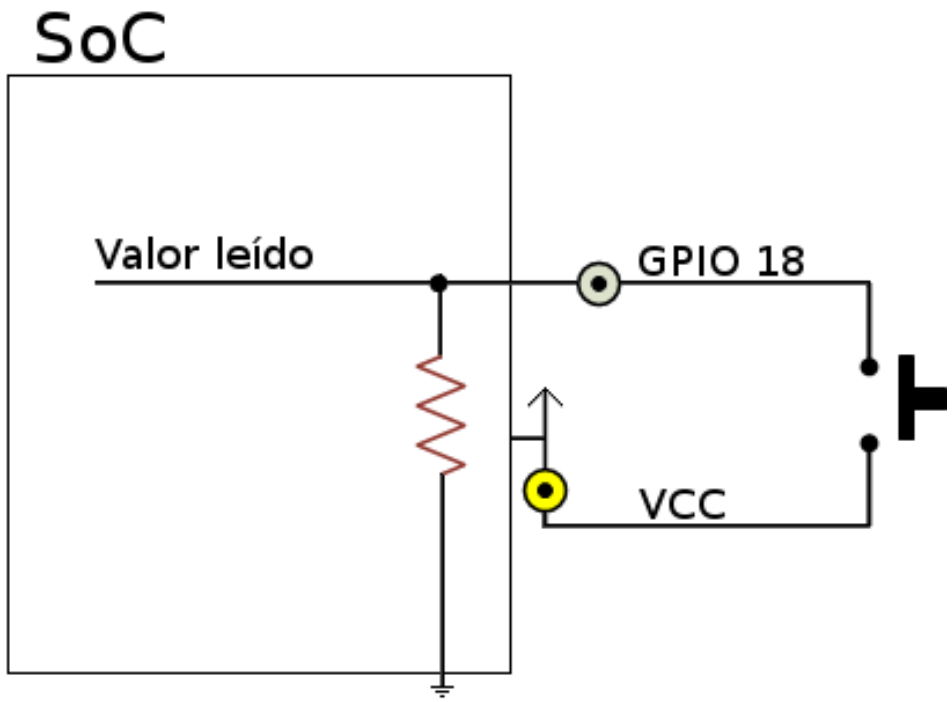

Figura D.4: Resistencia interna de pull-down

Bibliografía

[1] David Thomas. Introducción al arm. http://thinkingeek.com/2013/01/09/arm-assembler-raspberry-pi-chapter-1/, 2012.

[2] rferrer de THINK IN GEEK. Tutorial de asm para raspberry pi. http://thinkingeek.com/2013/01/09/arm-assembler-raspberry-pi-chapter-1/, 2013-2014.

[3] Alex Chadwick. Baking pi - desarrollo de sistemas operativos. http://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/index.html, 2013.

[4] Broadcom Corporation. Bcm2835 arm peripherals. http://www.raspberrypi.org/wp-content/uploads/2012/02/BCM2835-ARM-Peripherals.pdf, 2012.

[5] Qemu - emulating raspberry pi the easy way (linux or windows!). http://xecdesign.com/qemu-emulating-raspberry-pi-the-easy-way, 2012.

[6] ARM Limited. Arm1176jzf technical reference manual. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0301h/DDI0301H_arm1176jzfs_r0p7_trm.pdf, 2004-2009.

[7] Documentación gdb. https://sourceware.org/gdb/current/onlinedocs/gdb/, 1998-2014.

[8] Wikipedia. Formato elf. http://es.wikipedia.org/wiki/Executable_and_Linkable_Format, 2013.

[9] David Welch. Repositorio de david welch. https://github.com/dwelch67/raspberrypi, 2012-2014.

[10] Gerardo Bandera Burgueño, Maria Ángeles González Navarro, Eladio D. Gutiérrez Carrasco, Julián Ramos Cózar, Sergio Romero Montiel, Maria Antonia Trenas Castro, and Julio Villalba Moreno. *Prácticas de Estructura de Computadores*. Universidad de Málaga, 2002.

[11] Embedded Linux Wiki. Gpio y otros periféricos a bajo nivel. http://elinux.org/RPi_Low-level_peripherals, 2012-2014.

[12] Ignacio Moreno Doblas. Plantilla de pfc/tfg/tfm en latex, 2014.