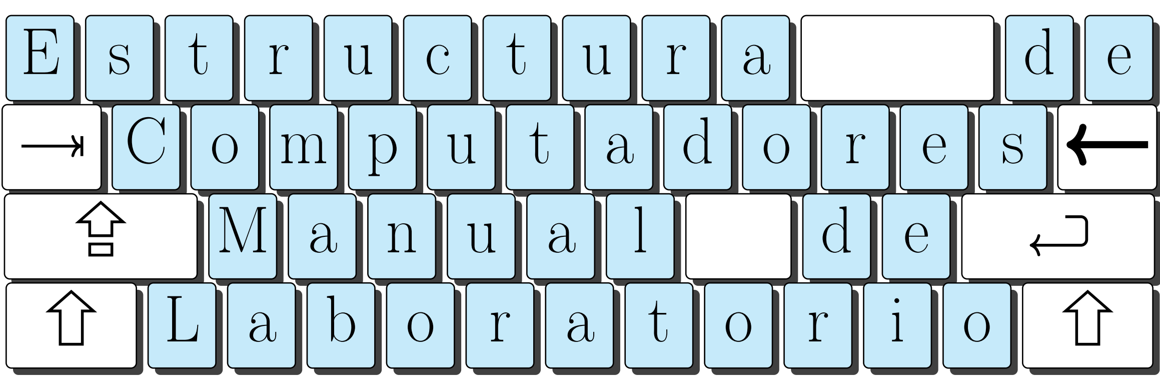

Christian Tenllado
tenllado@ucm.es

Luis Piñuel
lpinuel@ucm.es

Departmento de
Arquitectura de Computadores
y Automática

Aviso legal
Este documento está sujeto a una licencia *Reconocimiento - NoComercial - CompartirIgual 3.0 Unported* de Creative Commons
(http://creativecommons.org/licenses/by-nc-sa/3.0/deed.es_ES).
Basado en *"Prácticas de Estructura de Computadores empleando un MCU ARM"* de Luis Piñuel y Christian Tenllado publicado bajo licencia *Reconocimiento - NoComercial - CompartirIgual 3.0 Unported* de Creative Commons
(http://creativecommons.org/licenses/by-nc-sa/3.0/deed.es_ES)

Índice general

Prólogo **V**

1. Introducción al Computador **1**
1.1. Características de la arquitectura ARM v6 3
1.2. Instrucciones aritmético-lógicas. 7
1.3. Instrucciones de multiplicación 10
1.4. Instrucciones de acceso al Registro de Estado 11
1.5. Instrucciones de acceso a memoria (Load y Store) 12
1.6. Instrucciones de Load y Store múltiple 16
1.7. Instrucciones de Salto 17

2. Introducción al ensamblador **21**
2.1. Generación de un ejecutable 21
2.2. Ensamblador GNU para ARM 22
2.3. Enlazador GNU para ARM 29
2.4. Programación en Ensamblador 31

3. Eclipse **37**
3.1. Creación de un proyecto en Eclipse 37
3.2. Depuración sobre circuito 45

4. Subrutinas **55**
4.1. Implementación de subrutinas 57
4.2. Estándar de llamadas a subrutinas 59
4.3. Marco de Activación 62
4.4. Un ejemplo 64
4.5. Variables locales 66
4.6. Compiladores y optimización 68

5. Combinando C y Ensamblador **71**
5.1. Símbolos: resolución y relocalización 71
5.2. Arranque de un programa C 76
5.3. Tipos compuestos 76
5.4. Eclipse: depurando un programa C 80

6. Descubriendo el sistema de Entrada/Salida **83**
6.1. Fundamentos del sistema de E/S . . . 83
6.2. Sistema de E/S del BCM2835 . . . 85
6.3. Raspberry Pi 1 y Gertboard . . . 90
6.4. Uso de máscaras de bits . . . 94
6.5. E/S activa con espera de respuesta . . . 97

7. E/S mediante interrupciones **105**
7.1. Excepciones e Interrupciones . . . 105
7.2. Excepciones en el ARM1176JZF-S . . . 108
7.3. Controlador de Interrupciones del BCM2835 . . . 112
7.4. Uso de pines GPIO para generación de interrupciones . . . 116
7.5. Interrupciones periódicas . . . 118
7.6. Inicialización y configuración del sistema . . . 120
7.7. Ejemplo de E/S por interrupciones . . . 123

Bibliografía **126**

Prólogo

Este manual se ha diseñado como soporte para el laboratorio Estructura de Computadores (EC), impartida en la titulación de Ingeniería Electrónica de Comunicaciones de la Facultad de CC. Físicas de la UCM.

La asignatura de EC presenta al alumno una visión global del funcionamiento de un computador y su comunicación con el mundo exterior, a partir de los conocimientos adquiridos en las asignaturas previas de Circuitos Digitales e Informática. Concretamente se estudia el modelo de computador von Neumann y se analiza un posible diseño de procesador básico con subsistemas de memoria y entrada/salida simplificados.

El laboratorio asociado se centra principalmente en estudiar en detalle el modelo de máquina ofrecido al programador, es decir, la arquitectura del repertorio de instrucciones (interfaz HW/SW), y los mecanismos de entrada/salida (comunicación del computador con el mundo exterior).

El estudio del lenguaje ensamblador es por tanto un vehículo fundamental para que el alumno comprenda el funcionamiento básico de un computador y que entienda qué tipo de código máquina podrá ser generado a partir del código de alto nivel que escriba. Por ello, en este laboratorio describimos el proceso de compilación, ensamblado y enlazado para que el alumno pueda comprender el problema que resuelve cada una de estas etapas, y, en caso de error, sepa en cuál de ellas se produce.

Para montar este laboratorio se escogió una plataforma experimental económica basada en la placa Raspberry Pi 1, con un ARM1176JZF-S. La selección de la familia ARM se debe principalmente a la sencillez de su repertorio de instrucciones RISC y al enorme éxito que dicha familia tiene en el mercado de los sistemas empotrados. Cuando se montaron los laboratorios esta placa ofrecía una buena relación calidad/precio y parecía adaptarse bien al laboratorio. Sin embargo, hoy en día podemos encontrar otras placas en el mercado más adecuadas para este laboratorio, a un coste similar y con mejor documentación. Por ello, en un futuro próximo se pretende adaptar este laboratorio al uso de otra placa.

Este manual servirá tanto de apoyo para la realización de las prácticas de laboratorio como de libro de texto para la programación en ensamblador y la arquitectura del repertorio de instrucciones del ARM1176JZF-S.

Capítulo 1

Introducción al Computador

Un computador es máquina de cálculo electrónica que procesa información digitalizada atendiendo a una lista de instrucciones que almacena internamente, produciendo una información de salida digitalizada.

La información se representa con señales eléctricas que sólo toman dos valores discretos de voltaje (por ejemplo, 0V y 5V). Por lo tanto, el alfabeto capaz de ser comprendido por un computador se corresponde con dos dígitos: el 0 y el 1 (alfabeto binario).

La Figura 1.1 es un esquema de bloques con los componentes o subsistemas principales que componen un computador, que son:

- El procesador: es el componente principal, controla el funcionamiento del computador y ejecuta las operaciones indicadas por las instrucciones
- La memoria: encargado de almacenar temporalmente (mientras el computador está alimentado) los datos y las instrucciones del programa (almacenamiento interno).
- El sistema de entrada/salida: permite al computador interactuar con el exterior, permitiendo la transferencia de datos entre el computador y el entorno exterior.
- El sistema de interconexión: proporciona un medio de comunicación entre el procesador, la memoria y el subsistema de E/S.

Podemos pensar en el computador como una máquina que repite constantemente e indefinidamente el siguiente patrón:

1. El procesador lee de la memoria la *siguiente* instrucción a ejecutar.
2. El procesador decodifica (interpreta) la instrucción, identifica los operandos y los lee.
3. El procesador ejecuta la operación indicada por la instrucción sobre los operandos.
4. El procesador almacena el resultado en el lugar indicado por la instrucción.

Como vemos, el computador es una máquina cuya funcionalidad no se define cuando se construye o diseña, sino cuando se programa.

Las instrucciones que ejecuta un computador estarán almacenadas como un conjunto de 0s y 1s en la memoria (realmente representados por dos valores de tensión en algún componente electrónico, como por ejemplo un condensador). Sin embargo, programar un computador

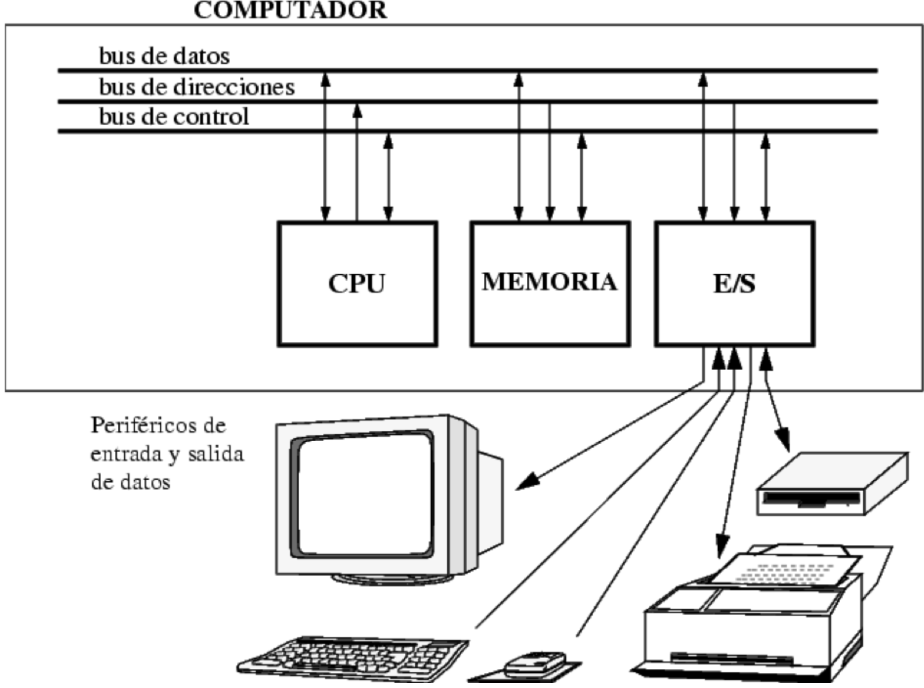

Figura 1.1: Esquema de bloques de un computador con arquitectura Von Neummann

a base de 0s y 1s (lenguaje máquina) es un trabajo muy laborioso, poco gratificante, poco eficiente y muy propenso a errores. Por ello se ha inventado un lenguaje simbólico (lenguaje ensamblador) formado por órdenes sencillas que se pueden traducir de manera directa al lenguaje de 0s y 1s que entiende el computador. Este proceso se ilustra en la Tabla 1.1. El lenguaje ensamblador requiere que el programador escriba una línea para cada instrucción que desee que la máquina ejecute, es un lenguaje que *fuerza al programador a pensar como la máquina*.

Pero, si se puede escribir un programa que traduzca órdenes sencillas (lenguaje ensamblador) a ceros y unos (lenguaje máquina), ¿qué impide escribir un programa que traduzca de una notación de más alto nivel a lenguaje ensamblador? Nada. De hecho, actualmente la mayoría de los programadores escriben sus programas en un lenguaje, que podíamos denominar *más natural* (lenguaje de alto nivel: C, pascal, FORTRAN...). El lenguaje de alto nivel es más sencillo de aprender e independiente de la arquitectura hardware sobre la que se va a terminar ejecutando. Estas dos razones hacen que desarrollar cualquier algoritmo utilizando la programación de alto nivel sea mucho más rápido que utilizando lenguaje ensamblador.

Los programadores de hoy en día deben su productividad a la existencia de un programa que traduce el lenguaje de alto nivel a lenguaje ensamblador. A ese programa se le denomina compilador.

Sin embargo, para comprender cómo funciona un computador, es imprescindible estudiar al menos para algún computador su propio lenguaje, el interfaz que el diseñador del hardware ha puesto entre el software y lo que realmente sucede dentro del procesador. Este interfaz HW/SW se conoce como la Arquitectura del Repertorio de Instrucciones (*Instruction Set Architecture* o *ISA*), o símplemente la Arquitectura, y define entre otras cosas:

- el conjunto y formato de las instrucciones que ejecuta el procesador

C = A + B;	Lenguaje de alto nivel
COMPILADOR	

`ldr R1, A`
`ldr R2, (*B*)`
`add R3, R1, R2`
`str R3, C`
	Lenguaje ensamblador

ENSAMBLADOR + ENLAZADOR	
e51f1014 e51f2014 e0813002 e50f3018	Lenguaje máquina (hexadecimal)

Tabla 1.1: Proceso de generación de órdenes procesables por un computador

- los modos de direccionamiento de los operandos
- el modelo de memoria
- los tipos de datos soportados
- los recursos visibles por el programador
- los modos de ejecución
- las excepciones e interrupciones

En el resto de este capítulo se presenta una introducción a la arquitectura del procesador que vamos a utilizar en el laboratorio. Algunas de sus características se describirán por encima y otras se omitirán con el fin de facilitar su estudio. En los capítulos siguientes iremos profundizando y ampliando progresivamente la información aquí expuesta.

1.1. Características de la arquitectura ARM v6

Los procesadores de la familia ARM tienen una arquitectura RISC de 32 bits, ideal para realizar sistemas empotrados de elevado rendimiento y reducido consumo, como teléfonos móviles, PDAs, consolas de juegos portátiles, etc. Tienen las características principales de cualquier RISC:

- Un gran banco de registros.
- Arquitectura *load/store*, es decir, las instrucciones aritméticas operan sólo sobre registros, no directamente sobre memoria.
- Modos de direccionamiento simples. Las direcciones de acceso a memoria (*load/store*) se determinan sólo en función del contenido de algún registro y el valor de algún campo de la instrucción (valor inmediato).
- Formato de instrucciones uniforme. Todas las instrucciones ocupan 32 bits con campos del mismo tamaño en instrucciones similares.

La arquitectura ARM tiene varios modos de ejecución. En este capítulo trataremos sólo el modo usuario, que se diferencia de todos los demás por no ser privilegiado, con un acceso restringido a determinados recursos.

1.1.1. Modelo de Memoria

La arquitectura ARM sigue un modelo Von Neumann con un mismo espacio de direcciones para instrucciones y datos. Esta memoria es direccionable por byte y está organizada en palabras de 4 bytes.

Aunque la versión 6 de la arquitectura ARM permite que algunas instrucciones realicen accesos no alineados a datos, por compatibilidad con versiones previas no se hará uso de accesos no alineados. Las instrucciones del subconjunto básico [1] siempre deben almacenarse en posiciones de memoria alineadas a 4 bytes (32 bits).

La Arquitectura ARM permite los dos tipos de ordenación posibles, *little-endian* y *big-endian*, aunque no simultáneamente. Por defecto usaremos *little-endian*.

1.1.2. Tipos de datos

En general los procesadores ARM pueden trabajar con datos enteros, con o sin signo, de los siguientes tamaños:

- 1 byte
- 2 bytes (*halfword*)
- 4 bytes (*word*)

Los enteros con signo se representan siempre con el convenio de complemento a 2.

En realidad la mayor parte de las operaciones se realizan sobre registros de 32 bits (4 bytes). Cuando el dato se lee de memoria y se trae a un registro del procesador, se indica en la operación de carga (`load`) el tamaño del dato, y el procesador extiende convenientemente el dato a 4 bytes teniendo en cuenta si el dato es con o sin signo. Del mismo modo, cuando queremos almacenar un dato en memoria, indicaremos su tamaño en la operación de carga (`store`) y el procesador realizará la escritura en memoria cogiendo del registro sólo los bytes indicados.

1.1.3. Registros

En modo usuario son visibles 15 registros de propósito general (R0-R14), un registro de contador de programa (PC), que en ciertas circunstancias puede emplearse como si fuese de propósito general (R15), y un registro de estado (CPSR). Todas las instrucciones pueden direccionar y escribir cada uno de los 16 registros en cada momento. El registro de estado, CPSR, debe ser manipulado por medio de instrucciones especiales.

Aunque el registro contador de programa pueda ser empleado en una instrucción como si se tratase de uno de propósito general, es preciso tener en cuenta los efectos laterales que ello pueda ocasionar. Si se escribe sobre él, se producirá un salto en el flujo de instrucciones del programa. Además, debido al diseño interno del procesador (segmentación) cuando se realiza la lectura de este registro, el valor proporcionado es el de la dirección de la instrucción que lee el PC más 8, es decir, dos instrucciones después de la actual.

[1] Para las instrucciones pertenecientes a las extensiones *Thumb* o *Jazelle* esto no se aplica.

Registro de estado

El registro de estado, CPSR, almacena información adicional necesaria para determinar el estado del programa, por ejemplo el signo del resultado de alguna operación anterior o el modo de ejecución del procesador. Es el único registro que tiene restricciones de acceso.

Está estructurado en campos con un significado bien definido: *flags (f)*, *estado (s)*, *extensión (x)* y *control (c)*, como ilustra la Figura 1.2. El campo de *flags* contiene los indicadores de condición y el campo de *control* contiene distintos bits que sirven para controlar el modo de ejecución. Los otros dos campos o no se usan (*DNM - DoNotModify*, *RAZ - ReadAsZero*) o contienen bits que se utilizan sólo para instrucciones muy específicas (ej. GE para SIMD multimedia) que no veremos de momento. La Tabla 1.2 describe el significado de los los bits más importantes de estos campos. Los bits N, Z, C y V (indicadores de condición) son modificables en modo usuario, mientras que los bits I, F y M sólo son modificables en los modos privilegiados. Del mismo modo, el bit T, que permite cambiar de subconjunto de instrucciones (*Thumb*), sólo puede ser modificado en los modos privilegiados, ignorándose su escritura cuando se está en modo usuario.

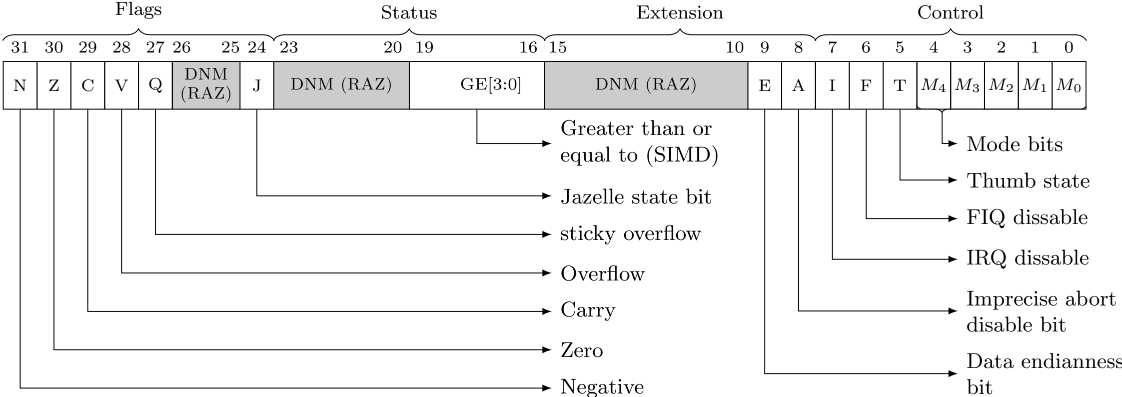

Figura 1.2: Registro de Estado del ARM v6 (CPSR).

1.1.4. Ejecución predicada

Una característica particular de la arquitectura ARM, es que todas las instrucciones, pueden ejecutarse condicionalmente. Para ello, el formato de toda instrucción contiene un campo Condición, que indica la condición que deben cumplir los bits de Flag del CPSR, tal como se indica a continuación:

```
Instrucción{Condicion}  Operandos   @ Si Condición ejecutar Instrucción
                                    @ sobre Operandos
```

Si dicha condición se cumple, la instrucción se ejecuta normalmente, si no, la instrucción no surte ningún efecto sobre el estado arquitectónico (es decir, no modifica ningún registro ni palabra de memoria). Si no aparece ninguna condición la instrucción se ejecuta siempre. La Tabla 1.3 muestra las alternativas posibles para este campo. Más adelante, veremos algunos ejemplos al tratar cada tipo de instrucción. Para no complicar la explicación del resto de las instrucciones se omitirá en cada caso que puede existir el campo condición.

Tabla 1.2: Descripción de los distintos campos del CPSR.

Campo	**Bit**	**Significado**
Bits de Flag	N	Indica si la última operación dio como resultado un valor negativo (N = 1) o positivo (N = 0).
	Z	Se activa (Z = 1) si el resultado de la última operación fue cero, de lo contrario permanece inactivo (Z = 0).
	C	Su valor depende del tipo de operación: ▪ Para una suma o una comparación (CMP), C = 1 si hubo *carry*. ▪ Para una resta o una comparación, C = 0 si hubo *underflow*. ▪ Para las operaciones de desplazamiento, toma el valor del bit saliente. ▪ El resto de operaciones no modifican su valor.
	V	En el caso de una suma o una resta, V = 1 indica que hubo un *overflow*.
	Q	*Sticky Overflow*, este flag es usado por algunas instrucciones de multiplicación y aritmética fraccional y su particularidad reside en que no se borra automáticamente.
Bit J	J	Indica el repertorio de instrucciones activo: (0) *ARM* o *Thum* dependiendo del bit T, (1) *Jazelle*.
Bits de Interrupción	I	Si $I = 1$ se deshabilitan las interrupciones externas (IRQ).
	F	Si $F = 1$ se deshabilitan las interrupciones externas rápidas (FIQ).
Bits de Modo	M[4:0]	Indican el modo de ejecución del procesador: *usr*(10000), *fiq*(10001), *irq*(10010), *svc* (10011), *abt*(10111), *und*(11011), *sys*(11111).
Bit T	T	Indica el repertorio de instrucciones activo: *ARM*(0), *Thumb*(1).

Tabla 1.3: Condiciones asociadas a las instrucciones.

Mnemotécnico	**Descripción**	**Flags**
EQ	Igual	Z=1
NE	Distinto	Z=0
CS/HS	*Carry Set*/ Mayor o igual (sin signo)	C=1
CC/LO	*Carry Clear*/Menor (sin signo)	C=0
MI	*Minus*/Negativo	N=1
PL	*Plus*/Positivo o cero	N=0
VS	*Overflow*/Desbordamiento	V=1
VC	*No Overflow*	V=0
HI	Mayor que (sin signo)	C=1 & Z=0
LS	Menor o igual que (sin signo)	C=0 or Z=1
GE	Mayor o igual que (con signo)	N=V
LT	Menor que con signo	N!=V
GT	Mayor que con signo	Z=0 & N=V
LE	Menor o igual que (con signo)	Z=1 or N!=V
AL	Siempre (incondicional)	

1.1.5. Instrucciones

El repertorio de instrucciones de ARM es relativamente grande, y a medida que han ido apareciendo nuevas versiones se han ido añadiendo algunas instrucciones nuevas sobre el repertorio ARM básico. Por motivos pedagógicos, en este laboratorio vamos a limitarnos a estudiar un subconjunto del repertorio básico de ARM, válido en la mayor parte de sus procesadores. Por ejemplo, el microcontrolador ARM1176JZF-S que usaremos en el laboratorio posee varias extensiones o repertorios adicionales – como *Thumb*, *TrustZone*, *Jazelle*, *DSP*, *VFP*) – que no serán tenidas en consideración en el presente documento.

En las siguientes secciones estudiaremos en detalle el repertorio de instrucciones de ARM, clasificando las instrucciones en seis grupos:

- Aritmético-lógicas
- Multiplicación
- Consulta/modificación del registro de estado
- Carga y almacenamiento (load/store)
- Carga y almacenamiento múltiple
- Salto

1.2. Instrucciones aritmético-lógicas.

En este apartado veremos las instrucciones que usan la unidad aritmético-lógica (ALU) del procesador. Esto no incluye a las instrucciones de multiplicación, que veremos en el siguiente apartado, ya que se ejecutan en otro módulo.

La ALU del ARM tiene una de sus entradas conectada a la salida de un desplazador de barril, tal como se ve en la figura 1.3. Esto permite que a uno de los operandos se le aplique un desplazamiento antes de la operación de la ALU. Recordemos que un desplazamiento a la izquierda equivale a multiplicar por una potencia de 2, y uno a la derecha a dividir por una potencia de 2.

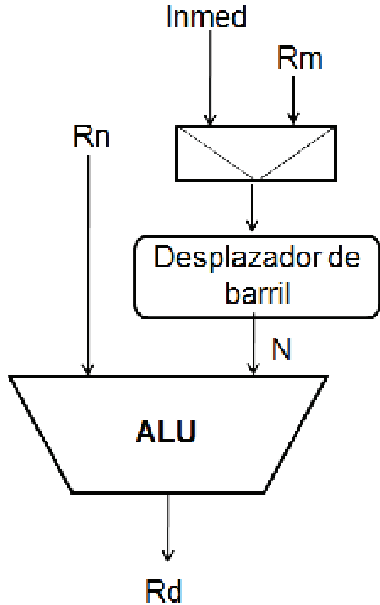

Figura 1.3: Esquema de la ALU, con un desplazador de barril conectado al segundo operando.

La sintaxis de las instrucciones aritmético-lógicas es la siguiente:

```
Instruccion{S}  Rd, Rn, N      @ Rd (←) Rn Oper N
```

Donde:

- `Instrucción`: alguno de los mnemotécnicos de la tabla 1.4
- `S`: si se incluye este campo la instrucción modifica los indicadores (*flags*) de condición de CPSR.
- `Rd`: registro destino (donde se almacena el resultado)
- `Rn`: registro fuente (primer operando). Todas las instrucciones menos `MOV`.
- `N`: segundo operando obtenido de la salida del desplazador de barril. Normalmente se denomina *shifter_ operand*.

Como podemos ver, además de escribir el resultado en el registro destino, cualquier instrucción puede modificar los flags de CPSR si se le añade como sufijo una S.

La Tabla 1.4 resume las instrucciones aritmético-lógicas más comunes. La mayor parte son instrucciones de dos operandos en registro que escriben su resultado en un tercer registro. Algunas como MOV tienen sólo un operando. Otras, como CMP, no escriben el resultado en registro, sólo modifican los bits del CPSR (en estos casos no es necesario indicarlo mediante *S*).

Tabla 1.4: Instrucciones aritmético-lógicas comunes. ShiftOp representa el *shifter_ operand*.

Mnemo	**Operación**	**Acción**
AND	AND Lógica	`Rd` ← `Rn AND ShiftOp`
ORR	OR Lógica	`Rd` ← `Rn OR ShiftOp`
EOR	OR exclusiva	`Rd` ← `Rn EOR ShiftOp`
ADD	Suma	`Rd` ← `Rn + ShiftOp`
SUB	Resta	`Rd` ← `Rn - ShiftOp`
RSB	Resta inversa	`Rd` ← `ShiftOp - Rn`
ADC	Suma con acarreo	`Rd` ← `Rn + ShiftOp + Carry Flag`
SBC	Resta con acarreo	`Rd` ← `Rn - ShiftOp - NOT(Carry Flag)`
RSC	Resta inversa con acarreo	`Rd` ← `ShiftOp - Rn - NOT(Carry Flag)`
CMP	Comparar	Hace `Rn - ShiftOp` y actualiza los flags de CPSR convenientemente
CMN	Comparar negado	Hace `Rn + ShiftOp` y actualiza los flags de CPSR convenientemente
MOV	Mover entre registros	`Rd` ← `ShiftOp`
MVN	Mover negado	`Rd` ← `NOT ShiftOp`
BIC	Borrado de bit (*Bit Clear*)	`Rd` ← `Rn AND NOT(ShiftOp)`

1.2.1. Modos de direccionamiento

Como hemos visto, las instrucciones aritmético-lógicas tienen generalmente dos operandos fuente y un operando destino. De los dos operandos fuente, uno debe ser siempre un registro (Rn). El segundo operando, que suele denominarse *shifter_ operand*, puede adoptar las siguientes formas:

a) Un Inmediato, opcionalmente desplazado. El inmediato debe poderse codificar con 8 bits y opcionalmente puede ser desplazado con una operación de rotación a la derecha (`ROR`). El desplazamiento es obligatoriamente un número par de posiciones, indicado por otro inmediato de 4 bits (posibles valores de rotación $0, 2, 4, \ldots, 30$).

b) Un registro, opcionalmente desplazado. En este caso hay cinco posibles operaciones de desplazamiento:

   - Desplazamiento lógico a la derecha (LSR)
   - Desplazamiento lógico a la izquierda (LSL)
   - Desplazamiento aritmético a la derecha (ASR), que extiende el bit de signo.
   - Rotación a la derecha (ROR).
   - Rotación a la derecha rellenando con el bit de carry del CPSR (RRX). El bit de carry se sustituye con el bit saliente. Es como una rotación de 33 bits, donde el bit más significativo es el bit de carry del CPSR.

   La cantidad a desplazar puede ser a su vez dada por un inmediato de 5 bits o por un tercer registro.

Los desplazamientos se realizan a través del desplazador de barril, como ilustra la figura 1.3. Para ampliar esta información consultar el manual de referencia del ARM [arm].

Veamos un ejemplo significativo: la instrucción `MOV R1, #4096`. En este caso el *shifter_operand* es un inmediato de más de 8 bits, que debe codificarse como un inmediato de 8 bits (`0x40`) con una rotación a la derecha de un número par posiciones (26). Como veremos en la sección 2.2, este trabajo lo hace el ensamblador por nosotros.

1.2.2. Ejemplos

```
ADD  R0, R1, #1          @ R0 = R1 + 1
ADD  R0, R1, R2          @ R0 = R1 + R2
ADD  R0, R1, R2, lsl #1  @ R0 = R1 + (R2 << 1) = R1 + 2*R2
ADD  R0, R1, R2, lsl R3  @ R0 = R1 + (R2 << R3)
BIC  R4, #0x05           @ Borra los bits 0 y 2 de R4
MOV  R11, #0             @ Escribe cero en R11
MOV  R15, R3, lsl #2     @ Copia en R15 (PC) el resultado de desplazar a la
                         @ izquierda R3 dos posiciones. ¡Provoca un salto!
SUB  R3, R2, R1          @ R3 = R2 - R1
SUBS R3, R2, R1          @ R3 = R2 - R1. Modifica los flags del registro de
                         @ estado en función del resultado
ADDEQ R7, R1, R2         @ Si el bit Z de CPSR está activo R7 = R1 + R2
```

1.3. Instrucciones de multiplicación

Hay diversas variantes de la instrucción de multiplicación debido a que al multiplicar dos datos de n bits necesitamos $2n$ bits para representar el resultado, y a que se dispone de multiplicaciones con y sin signo. Si se les pone el sufijo S modificarán además los bits Z y N del registro de estado de acuerdo con el valor y el signo del resultado. Los operandos siempre están en registros y el resultado se almacena también en uno o dos registros, en función de su tamaño (32 o 64 bits).

La sintaxis de las instrucciones de multiplicación es la siguiente:

```
Instruccion{Cond}{S} Registros
```

donde:

- `Instrucción`: una de las de la tabla 1.5.
- `Cond`: condición para ejecución condicional
- `S`: si se incluye este campo la instrucción modifica los indicadores de condición.
- `Registros`: ver su uso para cada instrucción en la tabla 1.5.

Las principales variantes se describen en la Tabla 1.5. Para completar la lista, es preciso consultar el manual de referencia de la arquitectura ARM [arm].

Tabla 1.5: Instrucciones de multiplicación.

Mnemotécnico	Operación
MUL Rd, Rm, Rs	Rd ← (Rm∗Rs)[31..0].
MLA Rd, Rm, Rs, Rn	Rd ← (Rn + Rm∗Rs)[31..0].
SMULL RdLo, RdHi, Rm, Rs	RdHi ← (Rm∗Rs)[63..32] y RdLo ← (Rm∗Rs)[31..0], donde los operandos son de 32 bits con signo.
UMULL RdLo, RdHi, Rm, Rs	RdHi ← (Rm∗Rs)[63..32] y RdLo ← (Rm∗Rs)[31..0], donde los operandos son de 32 bits sin signo.
SMLAL RdLo, RdHi, Rm, Rs	RdHi ← (RdHi:RdLo + Rm∗Rs)[63..32] y RdLo ← (RdHi:RdLo + Rm∗Rs)[31..0], donde los operandos son de 32 bits con signo.
UMLAL RdLo, RdHi, Rm, Rs	RdHi ← (RdHi:RdLo + Rm∗Rs)[63..32] y RdLo ← (RdHi:RdLo + Rm∗Rs)[31..0], donde los operandos son de 32 bits sin signo.

1.3.1. Ejemplos

```
MUL   R4, R2, R1        @ R4 = R2 x R1
MULS  R4, R2, R1        @ R4 = R2 x R1, modifica los flags del registro
                        @ de estado en función del resultado
MLA   R7, R8, R9, R3    @ R7 = R8 x R9 + R3
SMULL R4, R8, R2, R3    @ R4 = [R2 x R3]31..0
                        @ R8 = [R2 x R3]63..32
UMULL R6, R8, R0, R1    @ R8/R6 = R0 x R1
UMLAL R5, R8, R0, R1    @ R8/R5 = R0 x R1 + R8/R5
```

1.4. Instrucciones de acceso al Registro de Estado

Estas instrucciones permiten consultar o modificar el registro de estado (CPSR). Si nos hallamos en un modo privilegiado, también pueden trabajar con el registro de salvaguarda o sombra del registro de estado (SPSR), con el que trabajaremos en prácticas posteriores. La Tabla 1.6 resume su comportamiento.

Tabla 1.6: Instrucciones de manejo del Registro de Estado

Mnemotécnico	Operación
`MRS{<cond>} <Rd>, STReg`	`Rd <- STReg` donde `STReg` puede ser `CPSR` o `SPSR`.
`MSR{<cond>} STReg_campos, Op2`	donde los campos pueden ser `c,x,s,f` para los campos de control, extensión, estado y flags (ver figura 1.2), y `Op2` un registro o un inmediato. Modifica los campos indicados del registro de estado (`CPSR` o `SPSR`) con el valor de `Op2`.

1.5. Instrucciones de acceso a memoria (Load y Store)

Las instrucciones de **load** (LDR) se utilizan para cargar un dato de memoria sobre un registro. Las instrucciones de **store** (STR) realizan la operación contraria, copian en memoria el contenido de un registro. La sintaxis de estas instrucciones es:

```
LDR{Sufijo} Rd, Dir            @ Rd <- Memoria( Dir )
STR{Sufijo} Rf, Dir            @ Rf -> Memoria( Dir )
```

donde `Dir` indica la dirección de memoria desde la que leer (LDR) o sobre la que escribir (STR) y `Sufijo` se explicará más adelante.

1.5.1. Modos de direccionamiento

Para formar la dirección de memoria a la que se desea acceder, se utiliza un registro base y, opcionalmente, un desplazamiento. Esta suma se realiza con la ALU entera y el desplazamiento es el segundo operando. Esto permite expresar el desplazamiento de tantas formas como el segundo operando de las instrucciones aritmético lógicas:

- Un valor inmediato
- Un registro (índice)
- Un registro desplazado (multiplicado o dividido por una potencia de 2)

Además, hay tres formas de combinar el registro base y el desplazamiento:

- preindexado sin actualización,
- preindexado con actualización, y
- postindexado

En resumidas cuentas, el operando en memoria de las instrucciones LDR y STR admite una gran cantidad de modos de direccionamiento: indirecto de registro, indirecto de registro con desplazamiento, indirecto de registro indexado, indirecto de registro con post indexado, etc. Mostramos a continuación los casos más utilizados:

- Indirecto de registro (preindexado). Como ilustra la Figura 1.4 la dirección de memoria a la que deseamos acceder se encuentra en un registro del banco de registros. El formato en ensamblador sería:

  ```
  LDR Rd, [Rb] @ Rd ← Memoria( Rb )
  STR Rf, [Rb] @ Rf → Memoria( Rb )
  ```

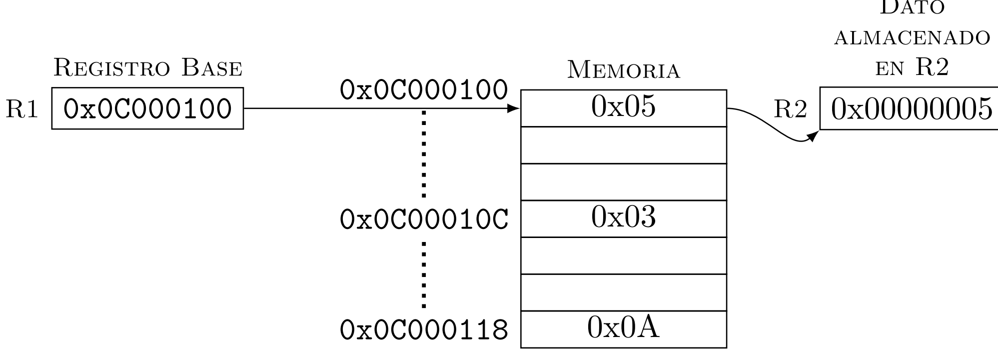

Figura 1.4: Ejemplo de ejecución de la instrucción `ldr r2,[r1]`.

- Indirecto de registro con desplazamiento inmediato (preindexado). Como ilustra la Figura 1.5, la dirección de memoria a la que deseamos acceder se calcula sumando una constante a la dirección almacenada en un registro del banco de registros, que llamamos registro base. El desplazamiento se codifica como valor inmediato (en la propia instrucción) con 12 bits y en convenio de complemento a 2. El formato en ensamblador sería:

```
LDR Rd, [Rb, #desp] @ Rd ← Memoria( Rb + desp )
STR Rf, [Rb, #desp] @ Rf → Memoria( Rb + desp )
```

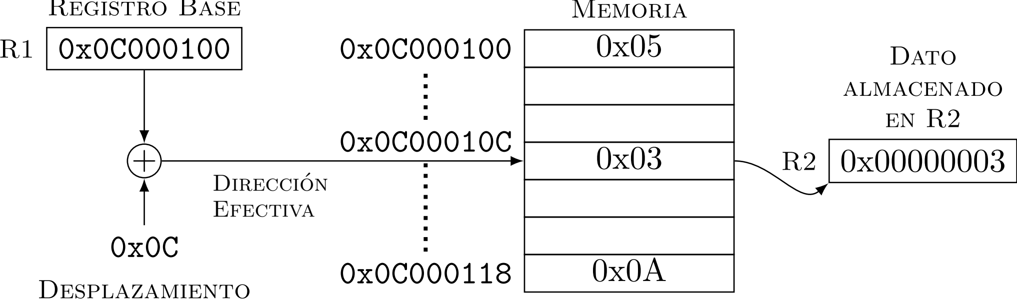

Figura 1.5: Ejemplo de ejecución de la instrucción `ldr r2,[r1,#12]`.

- Indirecto de registro con desplazamiento inmediato y actualización (preindexado). Como ilustra la Figura 1.6, el desplazamiento se suma o resta al registro base para formar la dirección de memoria y, después, el registro base se actualiza con este resultado (la diferencia en la codificación con el modo anterior está en el signo `!`). Es muy utilizado para recorridos de arrays.

```
LDR Rd, [Rb, #desp]!    @ Rd ← Memoria( Rb + desp )
                        @ Rb ← Rb + desp
STR Rf, [Rb, #desp]!    @ Rf ← Memoria( Rb + desp )
                        @ Rb ← Rb + desp
```

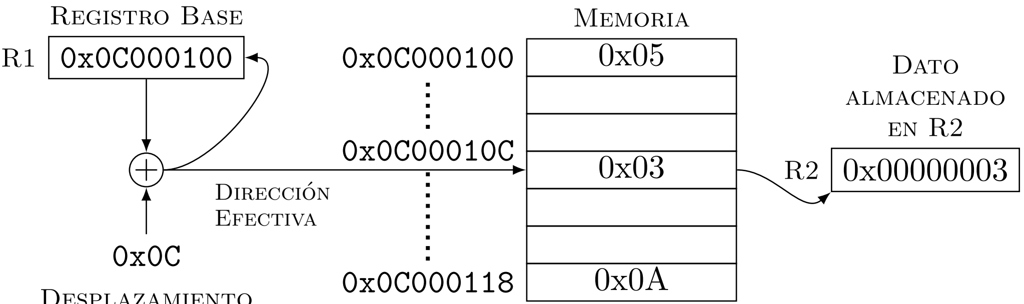

Figura 1.6: Ejemplo de ejecución de la instrucción `ldr r2, [r1,#12]!`.

- Indirecto de registro con post-indexado. Como ilustra la Figura 1.7, la dirección de memoria se calcula sólo con el registro base. Después del acceso, el registro base y el desplazamiento se suman o restan y el resultado se almacena en el registro base.

```
LDR Rd, [Rb], #desp      @ Rd ← Memoria( Rb )
                         @ Rb ← Rb + desp
STR Rf, [Rb], #desp      @ Rf → Memoria( Rb )
                         @ Rb ← Rb + desp
```

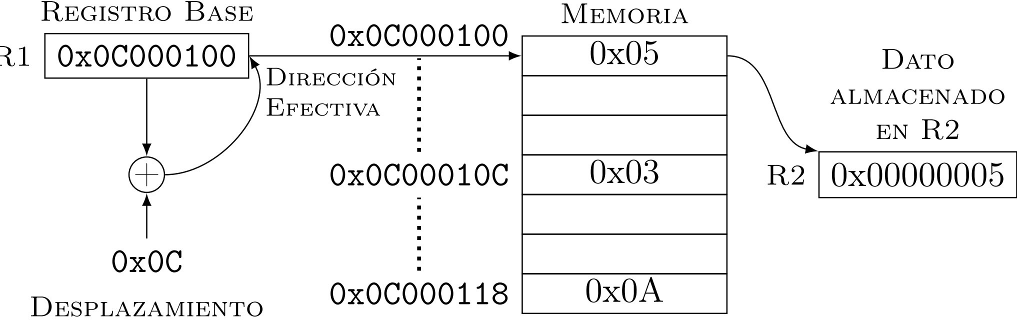

Figura 1.7: Ejemplo de ejecución de la instrucción `ldr r2, [r1],#12`.

- Indirecto de registro con desplazamiento por registro (preindexado). Como ilustra la Figura 1.8, la dirección de memoria a la que deseamos acceder se calcula sumando la dirección almacenada en un registro (base) al valor entero (offset) almacenado en otro registro, llamado registro índice. Este segundo registro puede estar opcionalmente multiplicado por una potencia de 2 (utilizando el desplazador de barril como el segundo operando de las instrucciones aritmético lógicas), haciendo este modo de direccionamiento muy útil para el acceso a arrays. Su sintaxis es:

```
LDR Rd, [Rb, Ri, LSL #N]   @ Rd ← Memoria( Rb + (Ri << N))
STR Rf, [Rb, Ri, LSL #N]   @ Rf → Memoria( Rb + (Ri << N))
```

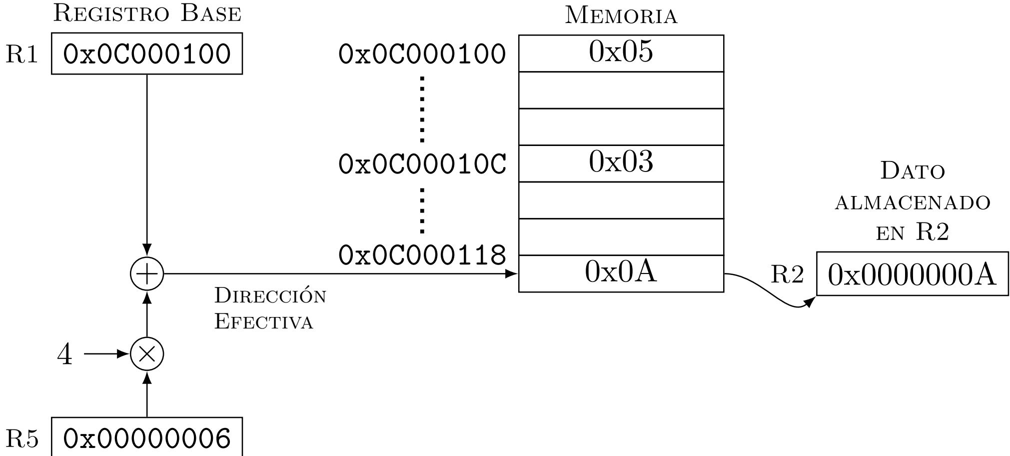

Figura 1.8: Ejemplo de ejecución de la instrucción ldr r2, [r1, r5, LSL #2].

1.5.2. Tamaño de dato

Además de operar sobre datos de tamaño palabra (32 bits), la versión del ARM que utilizamos permite operar sobre datos de tamaño media palabra y byte. Esto se indica añadiendo un sufijo `H` o `B` a las instrucciones `LDR` y `STR`.

```
LDR  Rd, [Rb + #desp]      @ Rd (←) Memoria( Rb + desp )[31..0]
LDRH Rd, [Rb + #desp]      @ Rd (←) Memoria( Rb + desp )[15..0]
LDRB Rd, [Rb + #desp]      @ Rd (←) Memoria( Rb + desp )[7..0]
```

En el caso de un load, cuando el dato cargado es menor de 32 bits puede elegirse entre extender el bit de signo o rellenar con ceros. Para extender el signo del operando se añade el sufijo `S` a la instrucción:

```
LDRB  Rd, [Rb + #desp]      @ Rd (←) Memoria( Rb + desp )[7..0], Rd[31..8] (←) 0
LDRSB Rd, [Rb + #desp]      @ Rd (←) SignExtend( Memoria( Rb + desp )[7..0] )
```

El valor inmediato (`desp`) está limitado a 12 bits para `ldr`/`str` de tamaño palabra o byte sin signo y a 8 bits para `ldr`/`str` de media palabra o byte con signo.

1.5.3. Ejemplos

```
LDR   R1,  [R0]              @ Carga en R1 lo que hay en Mem[R0]
LDR   R8,  [R3, #4]          @ Carga en R8 lo que hay en Mem[R3 + 4]
LDR   R12, [R13, #-4]        @ Carga en R12 lo que hay en Mem[R13 - 4]
STR   R2,  [R1, #0x100]      @ Almacena en Mem[R1 + 256] lo que hay en R2
STR   R4,  [PC, R1, LSL #2]  @ Almacena en Mem[PC + R1*4] lo que hay en R4
LDR   R11, [R3, R5, LSL #2]  @ Carga en R11 lo que hay en Mem[R3 + R5*4]
LDRB  R5,  [R9]              @ Carga un byte de Mem[R9] en R5, rellenando con 0
LDRSH R5,  [R9]              @ Carga en R5 la media palabra de Mem[R9] con
                             @ extensión de signo
LDR   R1,  [R0, #4]!         @ Carga Mem[R0 + 4] en R1 y actualiza R0 = R0 + 4
STRB  R7,  [R6, #-1]!        @ Almacena el contenido del último byte de R7
                             @ en Mem[R6-1] y actualiza R6 = R6 - 1,
LDR   R3,  [R9], #4          @ Carga R3 desde Mem[R9] y actualiza R9 = R9 + 4
```

```
STR   R2,  [R5], #8          @ Almacena el contenido de R2 en Mem[R5] y
                             @ actualiza R5 = R5 + 8
```

1.6. Instrucciones de Load y Store múltiple

Además de las instrucciones de load y store convencionales, la arquitectura ARM ofrece instrucciones de load y store múltiple. Este tipo de instrucciones son muy útiles para la gestión de pila, entrada y salida de subrutinas y las copias de bloques de datos en memoria, y además permiten reducir el tamaño del código.

El Load Múltiple (`LDM`) permite la carga simultánea de varios registros del conjunto `R0 -R14,PC` con datos ubicados en posiciones de memoria consecutivas, mientras que el Store Múltiple (`STM`) permite hacer la operación contraria, almacenar en posiciones de memoria consecutivas el valor de varios registros. Debemos tener en cuenta que si incluimos el PC en la lista de registros de un LDM, al realizarse la carga se producirá un salto.

La codificación de estas instrucciones en ensamblador es la siguiente:

```
LDM<Modo> Rb[!], {lista de registros}[^]
STM<Modo> Rb[!], {lista de registros}
```

Donde `Rb` es el registro base que contiene una dirección a partir de la cual se obtiene la dirección de memoria por la que comienza la operación. El signo ! tras el registro base es opcional, si se pone, el registro base quedará actualizado adecuadamente para encadenar varios accesos de este tipo. Finalmente, el circunflejo opcional de la instrucción `LDM` permite actualizar el registro de estado `CPSR` con el registro de sombra `SPSR` del modo actual, si en la lista de registros aparece el PC. Esto permite utilizar esta instrucción para realizar un retorno correcto de una rutina de tratamiento de interrupción, como veremos más adelante.

1.6.1. Modos de direccionamiento

Las instrucciones `LDM` y `STM` generan siempre un rango de direcciones de memoria consecutivas para realizar su operación correspondiente. Hay cuatro modos de direccionamiento que permiten generar este rango de cuatro formas distintas:

- *Increment After* (`IA`): La dirección de comienzo se toma del registro base `Rb`:

  ```
  dirección de comienzo = Rb
  ```

- *Increment Before* (`IB`): La dirección de comienzo se genera sumando 4 al contenido del registro base `Rb`:

  ```
  dirección de comienzo = Rb + 4
  ```

- *Decrement After* (`DA`): La dirección de comienzo se forma restando al contenido del registro `Rb` cuatro veces el número de registros de la lista más 4:

  ```
  dirección de comienzo = Rb - 4*#Registros + 4;
  ```

- *Decrement Before* (`DB`): La dirección de comienzo se forma restando al contenido del registro `Rb` cuatro veces el número de registros de la lista:

```
dirección de comienzo = Rb - 4*#Registros;
```

El funcionamiento de las instrucciones a las que dan lugar se ilustra en las Figuras 1.9 y 1.10. Empezando en la dirección de comienzo, se recorren todos los registros arquitectónicos en orden (primero R0, luego R1, ...). Si el registro está en la lista de registros de la instrucción se hace la operación correspondiente (load o store) con él y se incrementa en cuatro la dirección. Si no está se pasa al registro siguiente.

1.6.2. Ejemplos

```
STMDB SP!, {R4-R10,FP,LR}  @ Empezando en la dirección SP-4*9 copia el
                           @ contenido de los registros R4-R10, FP y LR.
                           @ SP queda con la dirección de comienzo (SP-4*9)
                           @ Se apilan los registros.
STMFD SP!, {R4-R10,FP,LR}  @ Lo mismo que la anterior
PUSH       {R4-R10,FP,LR}  @ Lo mismo que la anterior

LDMIA SP!, {R4-R10,FP,LR}  @ Empezando en la dirección SP copia el
                           @ contenido de la memoria en los registros
                           @ R4-R10, FP y LR.
                           @ SP queda con la dirección original + 4*9
                           @ Se desapilan los registros.
LDMFD SP!, {R4-R10,FP,LR}  @ Lo mismo que la anterior
POP        {R4-R10,FP,LR}  @ Lo mismo que la anterior
```

1.7. Instrucciones de Salto

Hay dos instrucciones explícitas de salto: Branch (B) y Branch and Link (BL). Las dos realizan un salto relativo al PC, hacia atrás o hacia delante. La distancia a saltar está limitada por los 24 bits con los que se puede codificar el desplazamiento dentro de la instrucción (inmediato). La sintaxis es la siguiente (los campos que aparecen entre llaves son opcionales):

```
B{Condición}    Desplazamiento    @ PC (←) PC + Desplazamiento
```

donde `Condición` indica una de las condiciones de la tabla 1.3 (EQ,NE,GE,GT,LE,LT,...) y `Desplazamiento` es un valor inmediato con signo (precedido de #) que representa un desplazamiento respecto del PC.

La dirección a la que se salta debe ser codificada como un desplazamiento relativo al PC. Debido al diseño interno del procesador (segmentación), cuando se lee el PC en realidad contiene la dirección de dos instrucciones después de la instrucción de salto (está incrementado en 8). Por ello el desplazamiento (inmediato) debe calcularse a partir de la siguiente expresión:

$$Dir_Destino = Dir_Actual + 8 + Inmediato$$

Sin embargo, a la hora de programar en ensamblador utilizaremos etiquetas (ver sección **??**) y serán el ensamblador y el enlazador los encargados de calcular el valor exacto del desplazamiento, como veremos más adelante. Si el inmediato no puede ser representado con 24 bits se producirá un error.

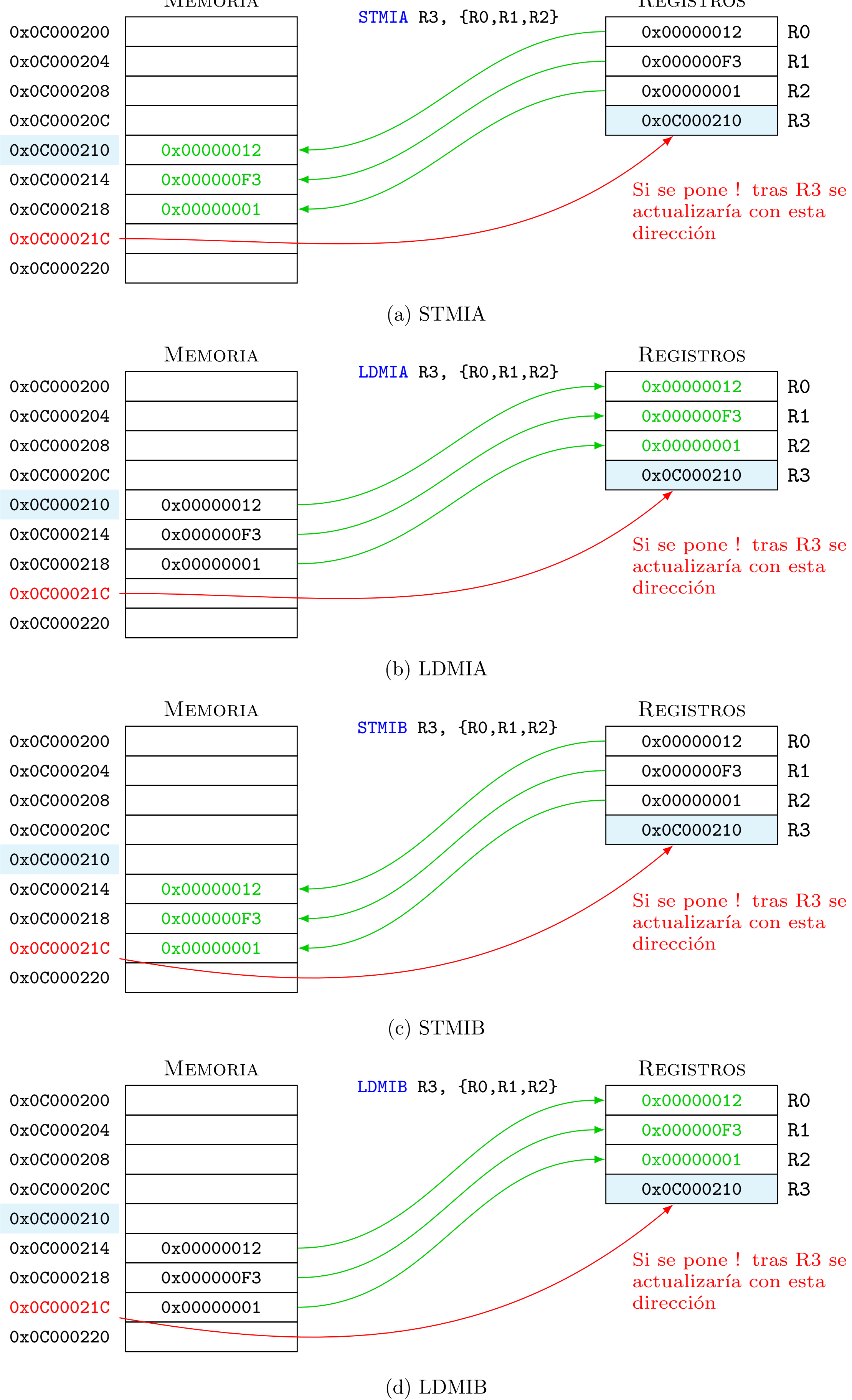

Figura 1.9: Instrucciónes de LDM/STM con modos IA/IB

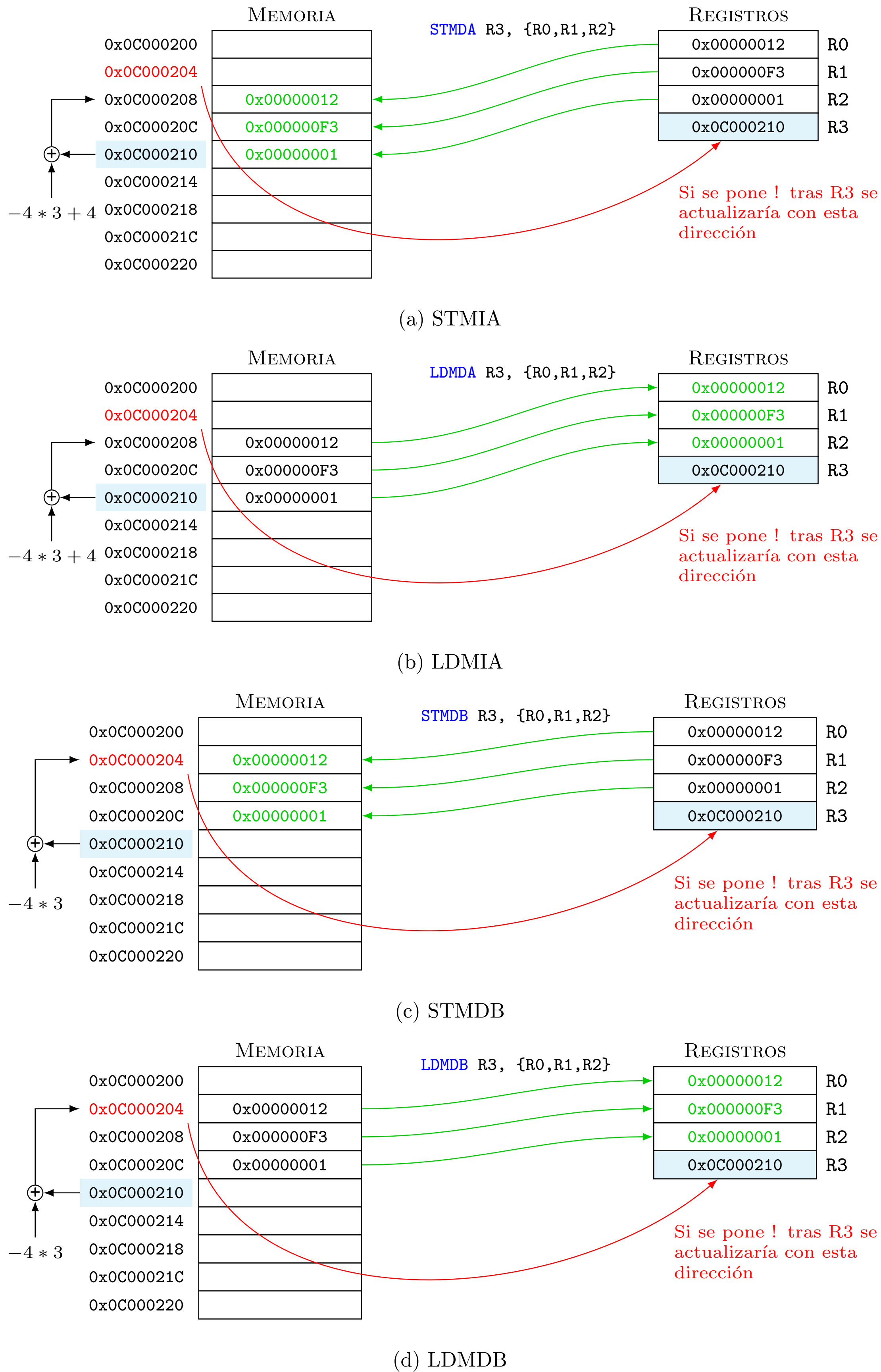

Figura 1.10: Instrucciónes de LDM/STM con modos DA/DB

La instrucción BL además de saltar almacena en el registro R14 (*Link Register* o LR) la dirección de la instrucción siguiente al salto. Se utiliza para realizar saltos a subrutina, como veremos más adelante.

También se puede provocar un salto escribiendo una dirección absoluta en el contador de programa, o bien empleando las instrucciones de salto con intercambio: Branch and Exchage (BX) o Branch with link, and exchange (BLX). Estas instrucciones son análogas a las anteriores pero modifican el PC a partir de un registros especificado como argumento, lo que permite además realizar cambios de modo (*ARM* - *Thumb*) en función de valor del bit 0 del registro.

Los saltos condicionales se ejecutan solamente si se cumple la condición del salto. Una instrucción anterior tiene que haber activado los indicadores de condición del registro de estado. Normalmente esa instrucción anterior es CMP (ver sección **??**), aunque puede ser cualquier instrucción con sufijo S. En el caso de que no se cumpla la condición, el flujo natural del programa se mantiene, es decir, se ejecuta la instrucción siguiente a la del salto.

Finalmente, debemos hacer notar que se puede provocar también un salto con cualquier instrucción que escriba en el contador de programa.

1.7.1. Ejemplos

```
B     etiqueta    @ Salta incondicional a etiqueta
BEQ   etiqueta    @ Salta a etiqueta si Z=1
BLS   etiqueta    @ Salta a etiqueta si Z=1 o N=1
BHI   etiqueta    @ Salta a etiqueta si C=1 y Z=0
BCC   etiqueta    @ Salta a etiqueta si C=0
MOV   PC, #0      @ R15 = 0, salto absoluto a la dirección 0
```

Capítulo 2

Introducción al ensamblador

En este capítulo vamos a ver cómo podemos generar un fichero ejecutable a partir de un código fuente escrito en lenguaje ensamblador. Para ello estudiaremos el proceso de compilación, ensamblado y enlazado, centrándonos en la misión que tiene cada una de estas etapas. Después analizaremos el conjunto de herramientas de gnu para realizar estas tareas: el compilador (`gcc`), el ensamblador (`as`) y el enlazador (`ld`). Finalizaremos dando las primeras pautas al alumno para que aprenda a implementar programas sencillos en ensamblador.

2.1. Generación de un ejecutable

En el *System on Chip* (SoC) que tenemos en el laboratorio, si no se hace uso de la *Memory Management Unit* (MMU) y no se emplea sistema operativo, todo programa en ejecución tendrá un espacio de direcciones físicas único para datos y código (no hay memoria virtual). Este mapa de memoria deberemos estructurarlo en regiones, en las que colocaremos las variables globales del programa, el código, la pila, etc. Esta estructuración la realizaremos mediante las herramientas de compilación, ensamblado y enlazado.

La figura 2.1 ilustra el proceso de generación de un ejecutable a partir de uno o más ficheros fuente y de algunas bibliotecas. Cada fichero en código C pasa por el preprocesador que modifica el código fuente original siguiendo directivas de preprocesado (`#define`, `#include`, etc.). El código resultante es entonces compilado, transformándolo en código ensamblador. El ensamblador genera el código objeto para la arquitectura destino (ARM en nuestro caso).

Los ficheros de código objeto son ficheros binarios con cierta estructura. En particular debemos saber que estos ficheros están organizados en secciones con nombre. Normalmente suelen definirse siempre tres secciones: `.text` para el código, `.data` para datos (variables globales) con valor inicial y `.bss` para datos no inicializados. Para definir estas secciones simplemente tenemos que poner una línea con el nombre de la sección. A partir de ese momento el ensamblador considerará que debe colocar el contenido subsiguiente en la sección con dicho nombre.

Por ejemplo el programa del cuadro 1 tiene una sección `.bss` en la que se reserva espacio para almacenar el resultado, una sección `.data` para declarar variables con valor inicial y una sección `.text` que contiene el código del programa. La directiva `.equ` permite declarar símbolos constantes (que recordamos no se almacenarán en memoria).

Debemos tener en cuenta que en el laboratorio desarrollamos sobre un PC para un entorno con procesador ARM, y por tanto necesitamos hacer *compilación cruzada*. Para distinguir

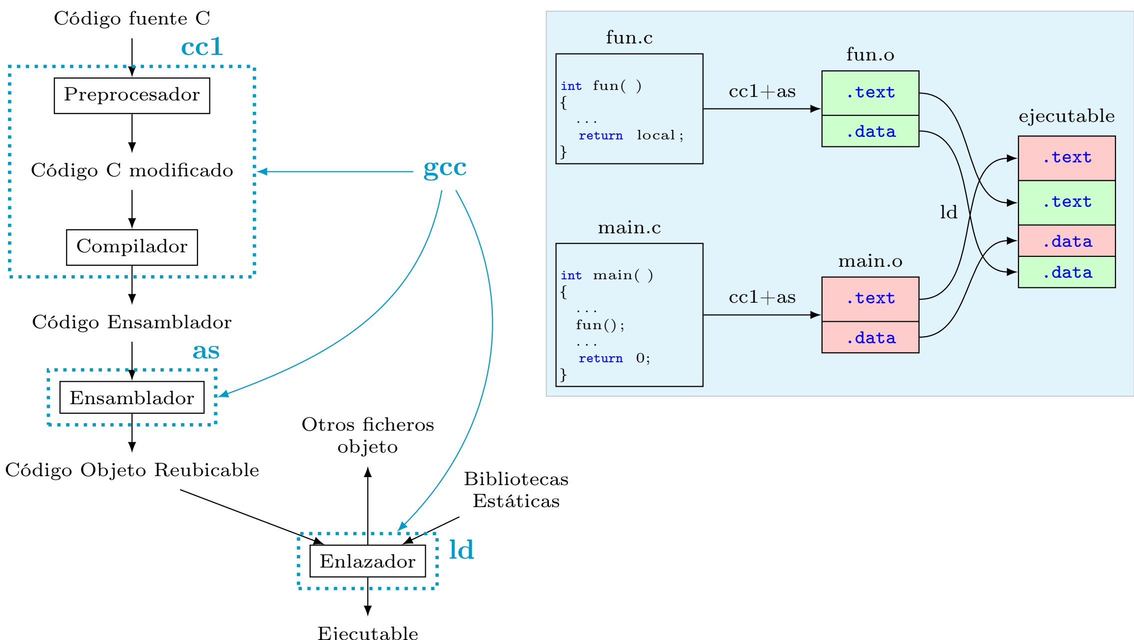

Figura 2.1: Proceso de generación de código ejecutable

las herramientas cruzadas de las nativas del PC suele añadírselas un prefijo que describe la arquitectura objetivo para la que se compila, en nuestro caso este prefijo es: `arm-none-eabi-`. Estas herramientas pueden utilizarse directamente desde un interprete de línea de comandos (terminal en Linux o Mac OS X o `cmd` en Windows). En cualquier caso, debemos siempre emplear la sintaxis y reglas de programación propias de estas herramientas.

2.2. Ensamblador GNU para ARM

Para explicar la estructura de un programa en lenguaje ensamblador y las distintas directivas del ensamblador utilizaremos como guía el sencillo programa descrito en el cuadro 1.

Lo primero que podemos ver en el listado del cuadro 1 es que un programa en lenguaje ensamblador no es más que un texto estructurado en líneas con el siguiente formato:

```
etiqueta: <instrucción o directiva>    @ comentario
```

Cada uno de estos campos es opcional, es decir, podemos tener por ejemplo líneas con instrucción pero sin etiqueta ni comentario.

El fichero que define el programa comienza con una serie de órdenes (directivas de ensamblado) dedicadas a definir dónde se van a almacenar los datos, ya sean datos de entrada o de salida. A continuación aparece el código del programa escrito con instrucciones del repertorio ARM.

Como el procesador de ARM es una máquina Von Neuman los datos y las instrucciones utilizan el mismo espacio de memoria. Como programa informático (aunque esté escrito en lenguaje ensamblador), los datos de entrada estarán ubicados en unas direcciones de memoria, y los datos de salida se escribirán en direcciones de memoria reservadas para ese fin. De esa manera si se desean cambiar los valores de entrada al programa se tendrán que

Cuadro 1 Ejemplo de programa en ensamblador de ARM.

```
        .global start

        .equ UNO, 0x01

        .data
DOS:    .word  0x02

        .bss
RES:    .space 4

        .text
start:
         MOV R0, #UNO
         LDR R1, =DOS
         LDR R2, [R1]
         ADD R3, R0, R2
         LDR R4, =RES
         STR R3, [R4]
FIN:     B .
        .end
```

cambiar a mano los valores de entrada escritos en el fichero. Para comprobar que el programa funciona correctamente se tendrá que comprobar los valores almacenados en las posiciones de memoria reservadas para la salida una vez se haya ejecutado el programa.

Los términos utilizados en la descripción de la línea son:

- `etiqueta`: es una cadena de texto que el ensamblador relacionará con la dirección de memoria correspondiente a ese punto del programa. Si en cualquier otro punto del programa se utiliza esta cadena en un lugar donde debiese ir una dirección, el ensamblador sustituirá la etiqueta por el modo de acceso correcto a la dirección que corresponde a la etiqueta. Por ejemplo, en el programa del cuadro 1 la instrucción LDR R1,=DOS carga en el registro `R1` el valor de la etiqueta `DOS`, es decir, la dirección en la que hemos almacenado nuestra variable `DOS`, actuando a partir de este momento el registro R1 como si fuera un puntero. Debemos notar aquí que esta instrucción no se corresponde con ningún formato de ldr válido. Es una pseudo-instrucción, una facilidad que nos da el ensamblador, para poder cargar en un registro un valor inmediato o el valor de un símbolo o etiqueta. El ensamblador reemplazará esta instrucción por un ldr válido que cargará de memoria el valor de la etiqueta `DOS`, aunque necesitará ayuda del enlazador para conseguirlo, como veremos más adelante.

- `instrucción`: el mnemotécnico de una instrucción de la arquitectura destino (algunas de las descritas en el Capítulo 1, ver figura 2.3). A veces puede estar modificado por el uso de etiquetas o macros del ensamblador que faciliten la codificación. Un ejemplo es el caso descrito en el punto anterior donde la dirección de un load se indica mediante una etiqueta y es el ensamblador el que codifica esta dirección como un registro base más un desplazamiento.

- `directiva`: es una orden al propio programa ensamblador. Las directivas permiten ini-

cializar posiciones de memoria con un valor determinado, definir símbolos que hagan más legible el programa, marcar el inicio y el fin del programa, etc (ver figura 2.2). Debemos tener siempre en cuenta que, aparte de escribir el código del algoritmo mediante instrucciones de ensamblador, el programador en lenguaje ensamblador debe reservar espacio en memoria para las variables, y en caso de que deban tener un valor inicial, escribir este valor en la dirección correspondiente. La Tabla 2.1 recoge las principales directivas del ensamblador GNU, de ellas las más utilizadas son:

- .global: exporta un símbolo para que pueda utilizarse desde otros ficheros, resolviéndose las direcciones en la etapa de enlazado. El comienzo del programa se indica mediante la directiva .global start, y dicha etiqueta debe aparecer otra vez justo antes de la primera instrucción del programa, para indicar dónde se encuentra la primera instrucción que el procesador debe ejecutar.
- .equ: define un símbolo con un valor. De forma sencilla podemos entender un símbolo como una cadena de caracteres que será sustituida allí donde aparezca por un valor, que nosotros definimos. Por ejemplo, .equ UNO, 0x01 define un símbolo `UNO` con valor `0x01`. Así, cuando en la línea MOV R0, #UNO se utiliza el símbolo, el ensamblador lo sustituirá por su valor.
- .word: se suele utilizar para inicializar las variables de entrada al programa. Inicializa la posición de memoria actual con el valor indicado tamaño palabra (también podría utilizarse .byte). Por ejemplo, en el programa del cuadro 1 la línea DOS: .word 0x02 inicializa la posición de memoria con el valor `0x02`, dónde `0x` indica hexadecimal.
- .space: reserva espacio en memoria tamaño byte para guardar las variables de salida, si éstas no se corresponden con las variables de entrada. Siempre es necesario indicar el espacio que se quiere reservar. Por ejemplo, en el programa del cuadro 1 la línea RES: .space 4 reserva cuatro bytes (una palabra (word)) que quedan sin inicializar. La etiqueta `RES` podrá utilizarse en el resto del programa para referirse a la dirección correspondiente a esta palabra.
- .end: Finalmente, el ensamblador dará por concluido el programa cuando encuentre la directiva `.end`. El texto situado detrás de esta directiva de ensamblado será ignorado.

- Secciones. Normalmente el programa se estructura en secciones, generalmente .text, .data y .bss. Para definir estas secciones simplemente tenemos que poner una línea con el nombre de la sección. A partir de ese momento el ensamblador considerará que debe colocar el contenido subsiguiente en la sección con dicho nombre.
  - .bss: es la sección en la que se reserva espacio para almacenar el resultado.
  - .data: es la sección que se utiliza para declarar las variables con valor inicial
  - .text: contiene el código del programa.
- `comentario`: una cadena de texto para comentar el código. Con `@` se comenta hasta el final de la línea actual. Pueden escribirse comentarios de múltiples líneas como comentarios C (entre `/*` y `*/`).

Las líneas que contienen directamente una instrucción serán codificadas correctamente como una instrucción de la arquitectura objetivo, ocupando así una palabra de memoria (4

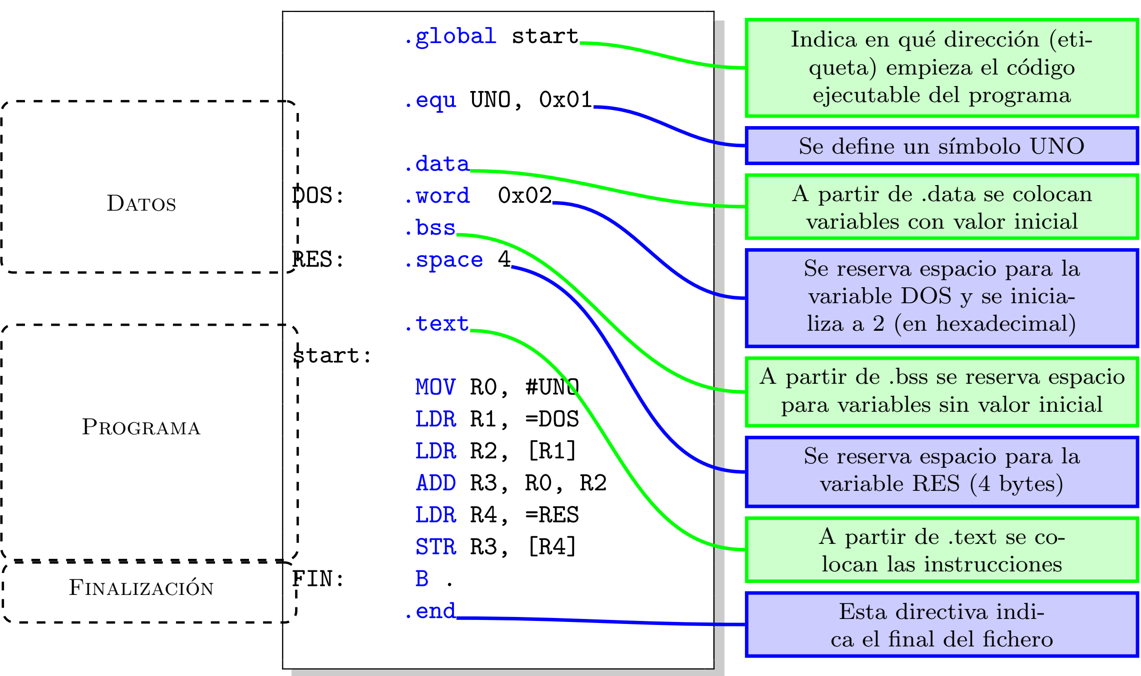

Figura 2.2: Descripción de la estructura de un programa en ensamblador: datos.

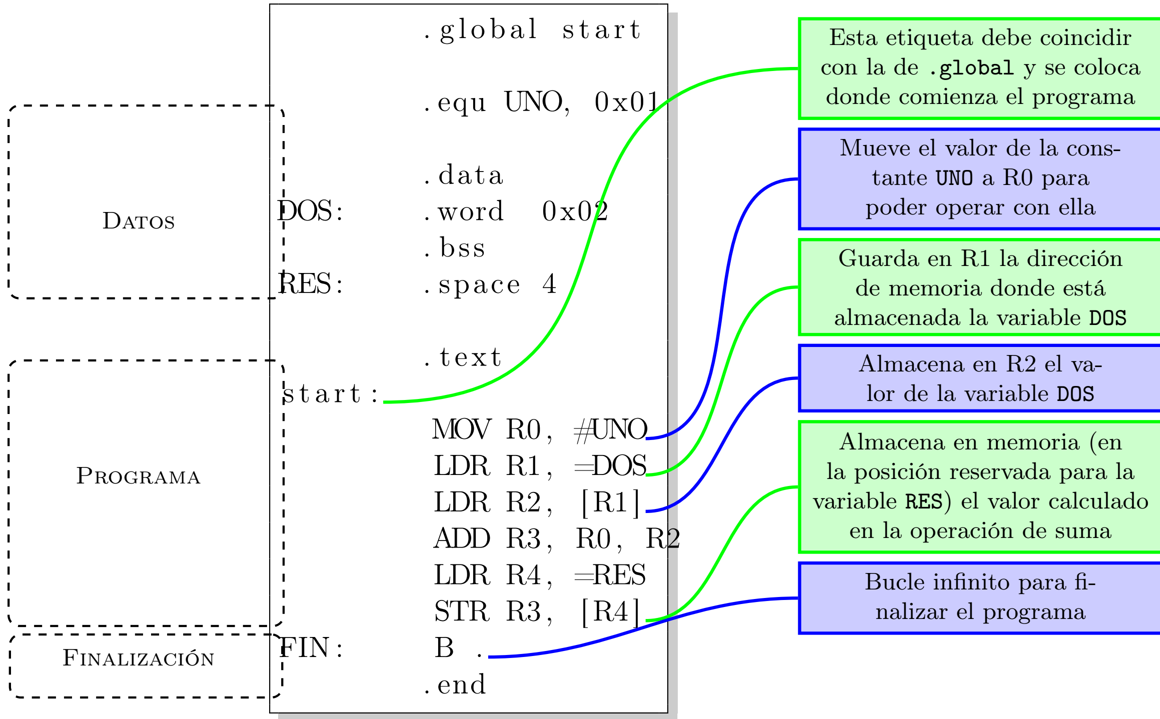

Figura 2.3: Descripción de la estructura de un programa en ensamblador: instrucciones.

Tabla 2.1: Directivas de ensamblado (GNU ARM).

Directiva	Descripción
.global <símbolo>	Exporta un símbolo para otros ficheros en la etapa de enlazado.
.extern <símbolo>	Declara un símbolo definido en un fichero externo. El símbolo no será resuelto hasta el enlazado.
.end	Marca el final del programa. El ensamblador no interpretará nada de lo que venga a continuación de .end.
.ascii "cadena"	Inserta la cadena en memoria.
.asciz "cadena"	Igual que .ascii pero inserta un byte 0 al final de la cadena.
.align	Alinea una posición a una palabra.
.byte <byte1> {,<byte2>,...}	Inserta una serie de bytes en esta posición en el código.
.word <word1> {,<word2>,...}	Inserta palabras de 32 bits en esta posición en el código.
.ltorg	Inserta en este punto la tabla (pool) de literales. Es especial para ARM.
.include "fichero"	Inserta el contenido del fichero en esta posición.
.equ <símbolo>, <valor>	Declara un símbolo y le asigna un valor. Siempre que se encuentre el símbolo por el código será sustituido por el valor asignado. Actúa como un #define en C.
.space <número de bytes> {, < valor de relleno>}	Reserva espacio para un número de bytes, opcionalmente inicializados con el valor de relleno.
.section <nombre sección> {, "<flags>"}	Inicia una nueva sección de código o de datos. Las secciones creadas por defecto por el ensamblador GNU son .text para el código, .data para datos inicializados y .bss para datos sin inicializar. Estas secciones ya tienen sus flags por defecto. Para indicar que un bloque debe insertarse en una sección ya definida podemos usar sólo el nombre de la sección, por ejemplo .text. Los flags permitidos para las secciones en el formato ELF son: Flag Significado a permiso de lectura w permiso de escritura x permiso de ejecución

bytes). Cuando se utiliza una *facilidad* del ensamblador, éste puede sustituirla por más de una instrucción, ocupando varias palabras. Si la línea es una directiva que implica reserva de memoria el ensamblador reservará esta memoria y colocará después la traducción de las lineas subsiguientes.

Los pasos seguidos por el entorno de compilación sobre el código presentado en el Cuadro 1 se resumen en la Tabla 2.2. El código de la primera columna se corresponde con el código en lenguaje ensamblador tal y como lo programamos. Utilizamos etiquetas para facilitarnos una escritura rápida del algoritmo sin tener que realizar cálculos relativos a la posición de las variables en memoria parar realizar su carga en el registro asignado a dicha variable. En la columna del centro aparece el código ensamblador generado por el programa ensamblador al procesar el código de la izquierda. Vemos que ya todas las instrucciones tienen la forma correcta del repertorio ARM. Las etiquetas utilizadas en el código anterior se han traducido por un valor relativo al contador de programa (el dato se encuentra X posiciones por encima o por debajo de la instrucción actual). Una vez obtenido este código desambiguado ya se pude realizar una traducción directa a ceros y unos, que es el único lenguaje que entiende el procesador, esa traducción está presente en la columna de la derecha, donde por ejemplo los bits más significativos se corresponden con el código de condición, seguidos del código de operación de cada una de las instrucciones.

Tabla 2.2: Traducción de código ensamblador a binario

Código Ensamblador	Código objeto desensamblado	Codificación (Hexadecimal)
start:		
MOV R0, #UNO	MOV r0, #1	e3a00001
LDR R1, =DOS	LDR r1, [pc, #16]	e59f1010
LDR R2, [R1]	LDR r2, [r1]	e5912000
ADD R3, R0, R2	ADD r3, r0, r2	e0803002
LDR R4, =RES	LDR r4, [pc, #8]	e59f4008
STR R3, [R4]	STR r3, [r4]	e5843000
FIN: B .		

2.2.1. Pseudo-instrucción LDR

En el ejemplo del Cuadro 1 hemos cargado la dirección de memoria representada por la etiqueta `DOS` en el registro `R1` utilizando una forma de la instrucción `LDR` que no se corresponde a ninguno de los modos de direccionamiento estudiados:

```
LDR R1,=DOS
```

La Figura 2.4 ilustra el funcionamiento de esta pseudo-instrucción, dónde suponemos que tras el enlazado la dirección asociada a `DOS` es `0x208`. Utilizando `LDR R1, =DOS` conseguimos cargar la dirección `0x208` en `R1`.

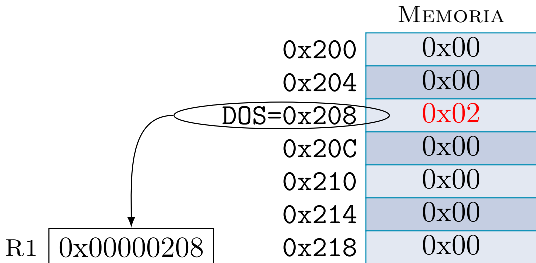

Figura 2.4: Ilustración del funcionamiento de la pseudoinstrucción LDR R1, =DOS.

Esta codificación de `LDR` no se corresponde exactamente con ninguno de los modos de direccionamiento válidos, es decir, no es una instrucción ARM válida, es una facilidad proporcionada por el ensamblador que nos permite guardar en el registro `R1` la dirección asociada a `DOS`. De hecho, parecería mucho más lógico utilizar una instrucción para escribir un valor inmediato en un registro, algo como:

```
MOV R1, #DOS
```

Sin embargo el número de bits reservados en el código de la instrucción para codificar el operando inmediato de las instrucciones aritmetico-lógicas no permite codificar un valor de 32 bits, que es lo que ocupa la dirección representada por `DOS`.

Una solución inteligente sería escribir en memoria la dirección representada por `DOS`, y luego hacer un load de la dirección donde hemos escrito `DOS`. Esto es fácil hacerlo, basta con escribir al final de la sección `.text`, después de la última instrucción de nuestro programa:

```
pDOS: .word DOS
```

y entonces cargaríamos el valor de la etiqueta `A` con:

```
ldr r1, pDOS
```

que el ensamblador transformaría en

```
LDR R1,[PC, #Desplazamiento]
```

donde el `Desplazamiento` lo calcularía el ensamblador como la distancia relativa entre la instrucción `LDR` y la etiqueta `DOS` (como están en la misma sección este desplazamiento no depende del enlazado). Pues bien, esto es justo lo que se hace con la pseudo-instrucción `LDR R1, =DOS`, que es más cómoda de utilizar.

Los ficheros objeto no son todavía ejecutables. El ensamblador ha generado el contenido de cada una de las secciones, pero falta decidir en qué direcciones de memoria se van a colocar dichas secciones y resolver los símbolos. Por ejemplo el valor de cada etiqueta no se conoce hasta que no se decide en qué dirección de memoria se va a colocar la sección en la que aparece.

Podríamos preguntarnos por qué no decide el ensamblador la dirección de cada sección y genera directamente el ejecutable final. La respuesta es que nuestro programa se implementará generalmente con más de un fichero fuente, pero las etapas de compilación y ensamblado se hacen fichero a fichero para reducir la complejidad del problema. Es más, utilizaremos frecuentemente bibliotecas de las que ni siquiera tendremos el código fuente. Por lo tanto, es necesario añadir una etapa de enlazado para componer el ejecutable final. El programa que la realiza se conoce como enlazador.

Como ilustra la Figura 2.1, el enlazador tomará los ficheros de código objeto procedentes de la compilación de nuestros ficheros fuente y de las bibliotecas externas con las que enlacemos, y reubicará sus secciones en distintas direcciones de memoria para formar las secciones del ejecutable final. En este proceso todos los símbolos habrán quedado resueltos, y como consecuencia todas nuestras etiquetas tendrán asignadas una dirección definitiva. En esta etapa el usuario puede indicar al enlazador cómo colocar cada una de las secciones utilizando un *script* de enlazado.

2.3. Enlazador GNU para ARM

Como hemos visto al comienzo de la sección, el enlazador es la última pieza de la cadena. Toma una serie de ficheros de código objeto y forma con ellos el binario ejecutable, resolviendo todos los símbolos que quedaban por resolver y conformando el mapa de memoria del ejecutable.

Por defecto, el enlazador genera una sección `.text` en el fichero de salida (ejecutable) concatenando las secciones `.text` de los ficheros objeto de entrada, en el orden que aparecen en la línea de órdenes de la invocación del enlazador. Lo mismo se hace con las secciones `.data` y `.bss`. Sin embargo este comportamiento por defecto se puede modificar utilizando un fichero de configuración (o script de enlazado) de `ld`.

La sintaxis de este fichero es sencilla, el cuadro 2 presenta un ejemplo.

Cuadro 2 Ejemplo de *script* de configuración de `ld`.

```
MEMORY
{
    ram : ORIGIN = 0x8000, LENGTH = 0x1000
}

SECTIONS
{
    .text : { *(.text*) } > ram
    .data : { *(.data*) } > ram
    .bss : { *(.bss*) } > ram
}
```

En esencia el fichero indica cómo formar las secciones de salida (las del ejecutable) a partir de las secciones de entrada (ficheros objeto a enlazar), y cómo han de colocarse las secciones de salida en memoria. Para ello el fichero tiene una entrada `SECTIONS` que incluye una lista de secciones de salida (en este caso `.text`, `.data` y `.bss`). La definición de cada sección de salida tiene el siguiente formato:

```
secname start BLOCK(align) (NOLOAD) : AT ( ldadr )
{
    contenido de la sección

} >region =fill
```

Los campos `secname` y `contenido` son obligatorios, el resto opcionales. La descripción de cada campo es la siguiente:

- `secname`: es el nombre de la sección de salida. En el ejemplo del cuadro 2 se crean tres secciones de salida, de nombre `.text`, `.data` y `.bss` respectivamente.
- `start`: indica la dirección de emplazamiento de la sección, que puede representarse con una expresión o una dirección absoluta. En el ejemplo del cuadro 2 no se utiliza.
- `contenido`: define el contenido de la sección de salida a partir de las secciones de entrada. En el ejemplo del cuadro 2 el contenido de las tres secciones se compone concatenando las secciones de entrada del mismo nombre (en `.text` todas las secciones `.text` de entrada, en `.bata` todas las `.bata` de entrada, ...), es decir, igual que en el comportamiento por defecto.
- `BLOCK(align)`: Se utiliza para emplazar un bloque con un determinado alineamiento, dado por la expresión `align`. Lo que sucede es que el puntero de posición actual `"."` avanza hasta la siguiente posición que respete el alineamiento deseado antes de colocar la sección.
- `(NOLOAD)`: Evita que la sección sea cargada en memoria en tiempo de ejecución. Se utiliza por ejemplo cuando hemos definido un contenido para una ROM y este contenido ya se cargó una vez. Al cargar el programa la sección no se cargará.
- `AT(ldadr)`: Indica la dirección de carga de la sección. Si no se especifica, la dirección de carga será la dirección de emplazamiento de la sección. Esto permite cargar una sección en un lugar (por ejemplo una ROM) pero resolviendo los símbolos como si estuviese en otro lugar (dirección de emplazamiento).
- `> region`: Indica que la sección debe ser asignada a la región o bloque de memoria, que debe haber sido definida con una entrada `MEMORY`. Si no se ha definido dirección, el enlazador tomará la primera dirección libre de dicha región. Si al emplazar la sección no hubiese espacio suficiente en la región de memoria, se producirá un error de enlazado. Esta es la forma que hemos utilizado en el ejemplo del cuadro 2.
- `=fill`: Especifica un valor de relleno que asignar a los huecos encontrados en la sección (direcciones no asignadas).

Es frecuente dar al enlazador la información sobre los rangos de memoria válidos de nuestro sistema y cuáles debe evitar usar el enlazador; y utilizar esta información para ubicar las secciones. Esto se hace, como ilustra el cuadro 2, utilizando la entrada `MEMORY`, cuya sintaxis es:

```
MEMORY
  {
    name [(attr)] : ORIGIN = origin, LENGTH = len
    ...
  }
```

El significado de los distintos campos es el siguiente:

- `name`: el nombre que se le asigna en el *script* de enlazado al bloque de memoria.
- `(attr)`: es una lista opcional de atributos que especifican si se debe utilizar este bloque de memoria para emplazar secciones de entrada que no se hayan emplazado explícitamente en ninguna sección de salida. Los atributos válidos son:

    - `R`: secciones de sólo lectura
    - `W`: secciones de lectura/escritura
    - `X`: secciones de código (ejecutables)
    - `A`: secciones ubicables
    - `I`: secciones inicializadas
    - `L`: lo mismo que `I`
    - `!`: la sección será emplazada en el bloque si no satisface ninguno de los atributos listados.

- `origin`: expresión que indica el comienzo del bloque de memoria.
- `len`: expresión que indica el tamaño del bloque.

Para más detalle consultar [ld-].

Finalmente el *script* de enlazado de `ld` permite especificar el punto de entrada a nuestro programa, es decir la primera instrucción a ejecutar, mediante la directiva `ENTRY(símbolo)`. La siguiente lista resume todas las formas en que podemos definir el punto de entrada al programa, en orden decreciente de prioridad:

1. Parámetro -e en la línea de órdenes de `ld`
2. El comando `ENTRY(símbolo)` en el fichero de configuración de ld
3. El símbolo `start` o `_start` si está definido
4. La dirección del primer byte de la sección `.text`, si existe
5. La dirección 0x0

2.4. Programación en Ensamblador

Con las instrucciones descritas en Capítulo 1 y las herramientas presentadas en las secciones anteriores podemos crear un programa que implemente cualquier algoritmo.

Sin embargo, la programación en ensamblador es más complicada que la programación en lenguajes de alto nivel. El programador tiene que descomponer una operación compleja en operaciones simples que el computador puede realizar, viéndose muy constreñido en la forma de expresar las operaciones. Además se tiene que ocupar de gestionar el almacenamiento de las variables, trabajando sólo con tipos de datos simples. También tiene que ocuparse colocar convenientemente datos y código en memoria.

La tarea de programar en lenguaje ensamblador es por tanto complicada de aprender y muy propensa a errores. Por ello aconsejamos a los alumnos a seguir el procedimiento que detallamos a continuación en todos las prácticas:

1. Obtener una descripción en alto nivel (pseudocódigo o código C/C++) del algoritmo a implementar.
2. Identificar las variables del algoritmo y decidir su emplazamiento. Para cada variable tenemos dos alternativas:

- Almacenamiento en memoria. Reservaremos espacio en la sección `.data` o `.bss` según tengan o no valor inicial, poniendo delante una etiqueta con el nombre de la variable para poder referenciarla.
- Almacenamiento en registro. Escogeremos un registro para almacenar la variable, y sólo lo usaremos para eso. Por motivos que quedarán claros más adelante, elegiremos un registro entre `R4-R11`.

Se aconseja que los arrays sean siempre almacenados en memoria. También es conveniente almacenar en memoria los datos de entrada y de salida al programa.

3. Traduciremos instrucción a instrucción el código de alto nivel a ensamblador, respetando las siguientes directrices:

   - Usaremos los registros `R0-R3` y `R12` para almacenar valores temporales (cálculos intermedios o auxiliares). Por ejemplo para almacenar temporalmente el valor de una variable que hemos decidido almacenar en memoria.
   - Cuando la línea a traducir suponga acceder a una variable almacenada en memoria, la traeremos primero a un registro temporal (load). Luego haremos la operación usando ese registro.
   - Cuando la línea a traducir suponga escribir en una variable almacenada en memoria, tendremos que calcular su valor sobre un registro temporal que luego llevaremos a memoria (store).
   - Cuando tengamos que acceder a arrays, usaremos un registro temporal para almacenar la dirección de comienzo del array. También usaremos registros temporales para calcular la dirección de un elemento en los casos que requieran un cálculo complejo, no proporcionado directamente por un modo de direccionamiento válido.
   - Usaremos un patrón conocido para implementar las estructuras de control de flujo habituales de la programación estructurada. La tabla 2.3 presenta algunos ejemplos, con el fin de que sirvan de guía al alumno.

2.4.1. Mi primer programa en ensamblador

Vamos a ilustrar con un ejemplo cómo podemos implementar en ensamblador un programa que cuente el número de enteros pares que hay en un array.

Comenzaríamos por obtener una descripción tipo C/C++ del algoritmo. En este caso la descripción podría ser la siguiente:

```
#define N, 16
int A[N] = {<16 valores separados por coma>};
int i;
int pares = 0;

for (i = 0; i < N; i++)
  if ((A[i] % 2) == 0)
      pares++;
```

Después tenemos que decidir dónde almacenar cada una de las variables. Por ejemplo, podemos optar por:

Tabla 2.3: Ejemplos de implementaciones de algunas estructuras de control habituales.

<table>
<tr><th>Pseudocódigo</th><th>Ensamblador</th></tr>
<tr><td><pre>if (R1 > R2)
     R3 = R4 + R5;
else
     R3 = R4 - R5
sigue la ejecución normal</pre></td><td><pre>   CMP R1, R2
        BLE L0
        ADD R3, R4, R5
        B L1
L0:   SUB R3, R4, R5
L1:   sigue la ejecución normal</pre></td></tr>
<tr><td><pre>for (i=0; i<8; i++) {
     R3 = R1 + R2;
}
sigue la ejecución normal</pre></td><td><pre>   MOV R0, #0 @R0 actúa como índice i
L0:   CMP R0, #8
        BGE L1
        ADD R3, R1, R2
        ADD R0, R0, #1
        B L0
L1:   sigue la ejecución normal</pre></td></tr>
<tr><td><pre>do {
     R3 = R1 + R2;
     i = i + 1;
} while(i != 8)
sigue la ejecución normal</pre></td><td><pre>   MOV R0, #0 @R0 actúa como índice i
L0:   ADD R3, R1, R2
        ADD R0, R0, #1
        CMP R0, #8
        BNE L0
sigue la ejecución normal</pre></td></tr>
<tr><td><pre>while (R1 < R2)
{
       R2= R2-R3;
}
sigue la ejecución normal</pre></td><td><pre>L0:   CMP R1, R2
        BGE L1
        SUB R2, R2, R3
        B L0
L1:   sigue la ejecución normal</pre></td></tr>
</table>

- `A` lo almacenamos en memoria, ya que es un array.
- `i` la almacenamos en registro, ya que es una variable auxiliar. Elegimos `R4`.
- `pares` la almacenamos en memoria, por ser una variable de salida.
- `N` la dejamos como símbolo (quedará almacenado en las instrucciones en las que interviene como operando inmediato).

De acuerdo con estas decisiones podemos escribir ya las secciones de datos de nuestro programa:

```
    .equ N, 16
    .data
A:      .word <lista de N valores separados por coma>
pares:  .word 0
```

Ahora podemos ir traduciendo el código. Empezamos por la primera línea, que resulta ser un bucle `for`. Usaremos el patrón descrito en la Tabla 2.3 para este bucle. Primero la línea de inicialización `i = 0`. Después un bloque en el que se realiza la comprobación de la condición y el salto fuera del bucle si no se cumple. La primera línea del bloque se identifica con una etiqueta y la primera línea fuera del bucle con otra. Luego el cuerpo del bucle, seguido de la línea de actualización `i++`, seguido de un salto incondicional a la etiqueta del bucle. Es decir:

```
    mov r4, #0
L0: cmp r4, #N
    bge L1

    <cuerpo del bucle>

    add r4, r4, #1
    b L0
L1:
```

Para la siguiente línea nos fijaremos en el patrón para el `if` en la Tabla 2.3. Es un bloque de comparación, y un salto fuera del `if` si no se cumple la condición. Para evaluar la condición debemos leer un elemento del array `A` que está en memoria. Por tanto, cargaremos la dirección de `A` en un registro temporal, `R0` por ejemplo, y luego cargaremos el elemento `A[i]` en el mismo registro, aprovechando el modo de direccionamiento indirecto con desplazamiento por registro escalado.

Para ver si el número es par, el código C examina el resto de la división por 2. Si es cero, el número `A[i]` es par. La división por 2 supone desplazar un bit a la derecha, el resto de la división es el bit desplazado. Por lo tanto basta con evaluar el bit menos significativo de `A[i]`. Esto lo podemos hacer con una operación `and` entre `A[i]` y `1`.

El resultado podría:

```
    ldr r0, =A
    ldr r0, [r0, r4, lsl #2]
    and r0, r0, #1
    cmp r0, #0
    bne L2

    <cuerpo del if>
```

```
L2:
```

Finalmente, en el cuerpo del `if` hay que sumar 1 a `pares`. Como pares está en memoria primero la traeremos a un regitro temporal, por ejemplo a `R0`. Pero antes tenemos que cargar la dirección de pares en un registro, usaremos `R1` para ello para poder reutilizarlo para almacenar el resultado en pares. El resultado es:

```
    ldr r1, =pares
    ldr r0, [r1]
    add r0, r0, #1
    str r0, [r1]
```

Para componer el programa completo sólo tenemos que unir los fragmentos anteriores correctamente, y poner las directivas y etiquetas que faltan (también añadimos un bucle infinito al final para que el procesador no continúe ejecutando instrucciones que no hemos escrito), resultando:

```
    .global start

    .equ N, 16
    .data
A:      .word <lista de N valores separados por coma>
pares:  .word 0

    .text
start:
    mov r4, #0
L0: cmp r4, #N
    bge L1

    ldr r0, =A
    ldr r0, [r0, r4, lsl #2]
    and r0, r0, #1
    cmp r0, #0
    bne L2

    ldr r1, =pares
    ldr r0, [r1]
    add r0, r0, #1
    str r0, [r1]

L2:
    add r4, r4, #1
    b L0
L1:
    b .
    .end
```

Como vemos, el procedimiento descrito permite al programador iniciado centrarse en la traducción de cada instrucción, sin tener en cuenta lo que se ha hecho con las anteriores ni lo que se hará con las siguientes. Esto simplifica enormemente el problema, reduce la posibilidad de errores y hace que el código sea más fácil de modificar y corregir sin introducir nuevos errores.

Este tipo de código es similar al que los compiladores generan para la depuración, cuando no hacen ninguna optimización. Se aconseja al alumno que utilice esta estrategia en sus prácticas para facilitar su aprendizaje, aún cuando el número de líneas de código que se generan así sea mayor que para un código optimizado.

Capítulo 3

Eclipse

En este laboratorio vamos a utilizar Eclipse como Entorno de Desarrollo Integrado (IDE). Es una aplicación con interfaz gráfica de usuario que nos permitirá desarrollar tanto en lenguaje ensamblador como en C, y depurar el código desarrollado mientras se ejecuta en el hardware destino (depuración en circuito).

Eclipse nos ofrece principalmente la interfaz gráfica y la gestión de los proyectos. Para realizar el resto de tareas hace uso de las herramientas de GNU descritas en el Capítulo 2.

En el resto del capítulo veremos cómo crear un proyecto en eclipse, como compilar, ensamblar y enlazar dicho proyecto y como depurarlo mientras ejecuta sobre la placa que utilizamos en el laboratorio.

3.1. Creación de un proyecto en Eclipse

Para todas nuestras prácticas deberemos crear un proyecto Eclipse para compilación y depuración cruzadas utilizando el plugin *GNU ARM*. Vamos ilustrar el proceso creando un proyecto para un programa que obtiene el mayor de dos números que se pasan como entrada. Usaremos este mismo ejemplo en la sección 3.2 para ilustrar el proceso de depuración.

Para crear el proyecto de Eclipse los pasos a seguir son:

1. Abrir Eclipse. Al iniciarse la aplicación nos mostrará una ventana de selección de `workspace`, como la que la Figura 3.1. Debemos seleccionar un `workspace` propio, que no es más que un directorio en el que Eclipse guardará información sobre los proyectos de todas nuestras prácticas. Para que el ordenador del laboratorio vaya lo más ágil posible, es conveniente que pongamos nuestro `workspace` en una carpeta local del equipo (no en un pen-drive).

2. Una vez seleccionado se abrirá la ventana principal de Eclipse. Si acabamos de crear el `workspace` entonces el aspecto será como el que se muestra en la Figura 3.2.

3. Debemos cerrar la pestaña `Wellcome` haciendo click en la cruz de la pestaña. Entonces la ventana quedará como la que muestra la Figura 3.3, es la perspectiva C/C++ de Eclipse

   Esta perspectiva está organizada de la siguiente manera:

   - Panel Izquierdo: el explorador de proyectos. Aparecerán todos los proyectos que tengamos en el `workspace`, cuando los tengamos.

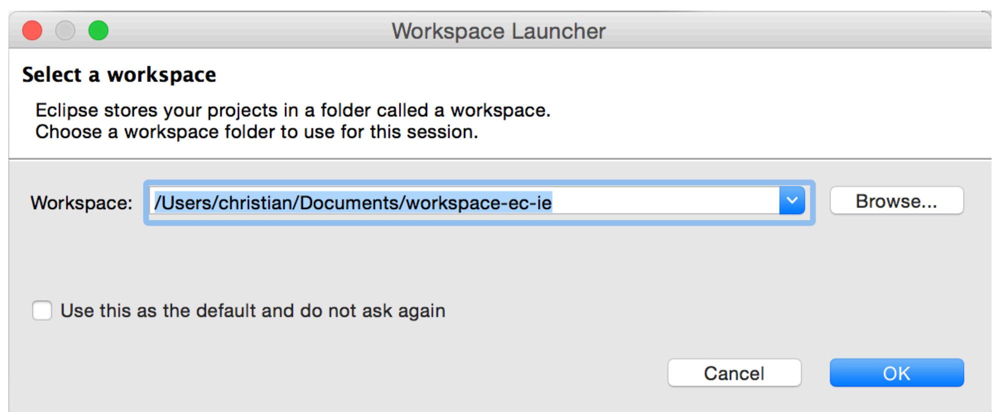

Figura 3.1: Ventana de selección de `workspace`.

Figura 3.2: Ventana de eclipse al abrir un `workspace` vacío.

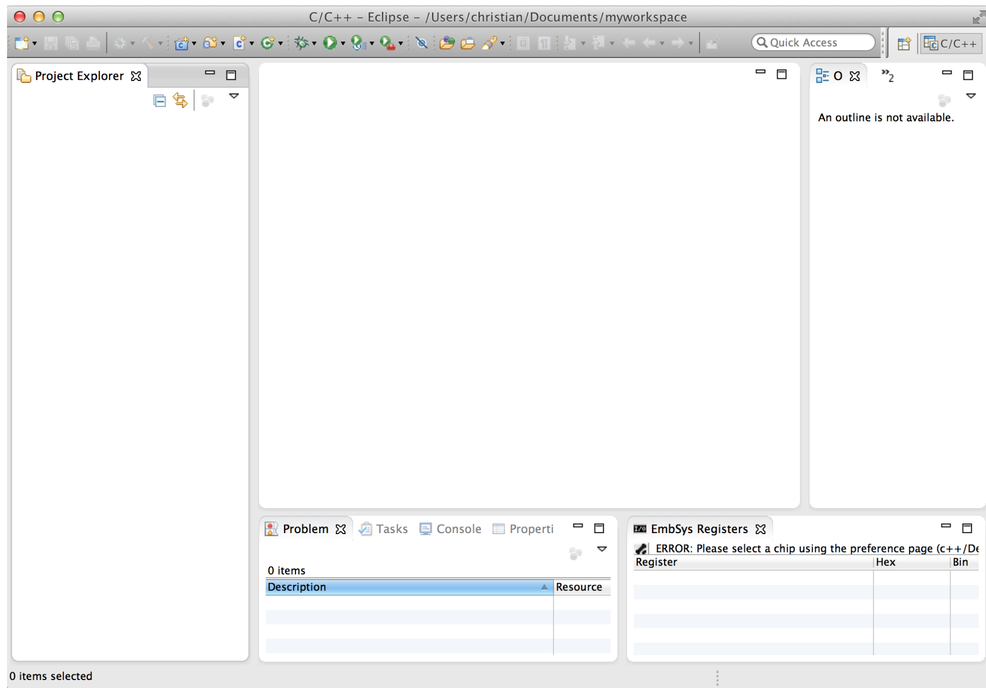

Figura 3.3: Ventana de eclipse con la perspectiva C/C++ sin proyectos.

- Panel central: el editor . Nos permitirá editar los ficheros que contendrán el código fuente de nuestros programas.
- Panel derecho. Nos permite explorar los símbolos del proyecto activo (funciones, variables, etc).
- Panel inferior. Tiene varias pestañas, entre las que destacan la pestaña de errores de compilación y la consola. Desde la primera podemos ver los errores y saltar a la línea de código fuente que los causó haciendo click sobre el error. En la segunda podemos ver los comandos que ejecuta Eclipse para hacer la compilación.

4. Para crear el proyecto seleccionamos `File`→`New`→`C Project`, con lo que se abrirá una ventana como la de la Figura 3.4. Como indica la figura, seleccionamos las opciones `Executable`, `Empty Project` y Cross ARM GCC. Elegimos el nombre del proyecto y pulsamos dos veces `Next`. Aparecerá una ventana similar a la de la Figura 3.5, y seleccionamos la ruta donde está instalado el toolchain `arm-none-eabi-gcc` en nuestro equipo. Finalmente pulsamos `Finish` y tendremos el proyecto vacío visible en el explorador de proyectos.

5. Ahora vamos a añadirle un fichero con el código fuente de nuestro primer programa en ensamblador. Para ello seleccionamos `File`→`New`→`Source File`, con lo que se abrirá una ventana como la de la Figura 3.6. Para que Eclipse tome el fichero como un fichero fuente con código ensamblador le ponemos extensión `.asm` o `.S`. Una vez creado, haciendo doble click sobre el fichero en el panel izquierdo se abrirá una pestaña en editor, en la que copiamos el código del cuadro 3.

6. Antes de configurar la compilación de nuestro proyecto vamos a añadir a él un fichero más, que servirá para indicar al enlazador cómo debe construir el mapa de memoria del

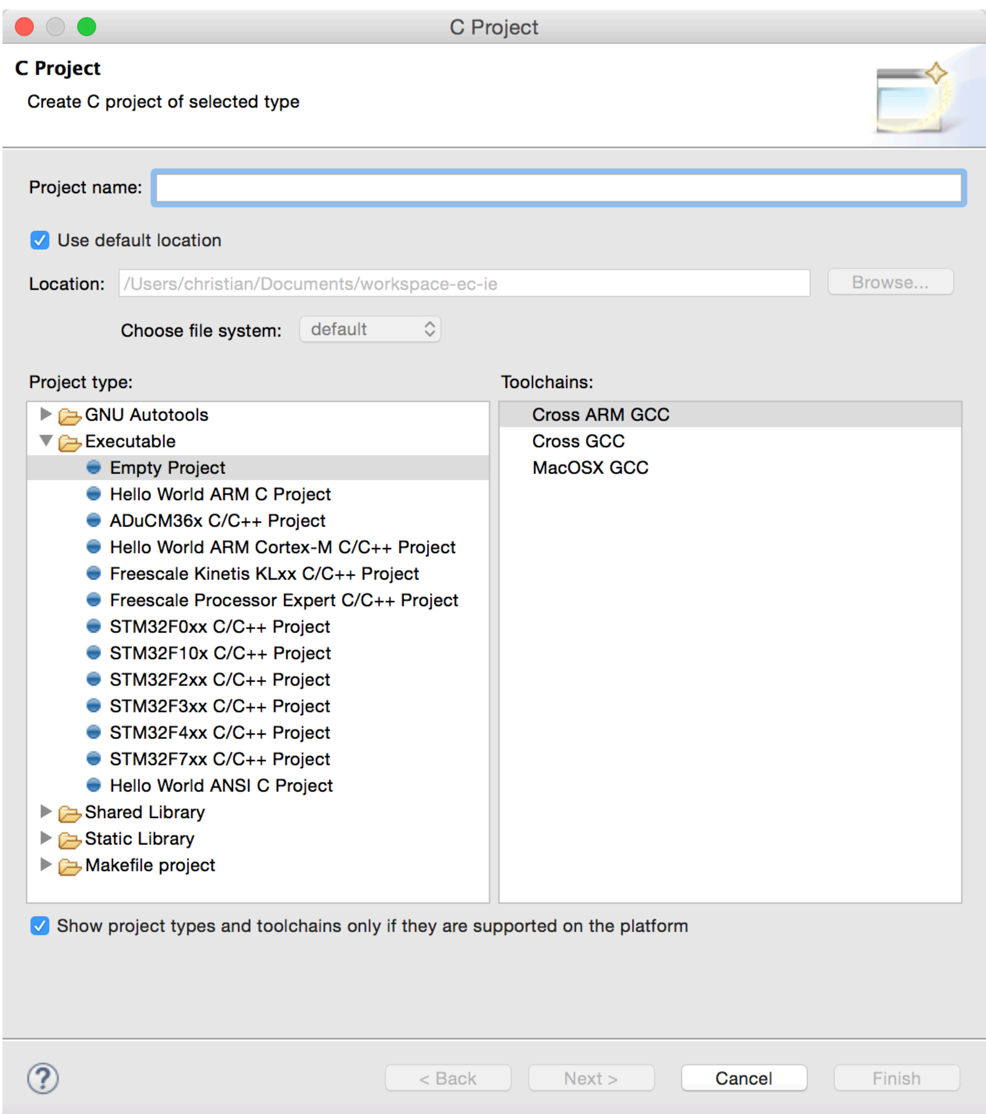

Figura 3.4: Ventana de creación de C project de tipo GNU ARM.

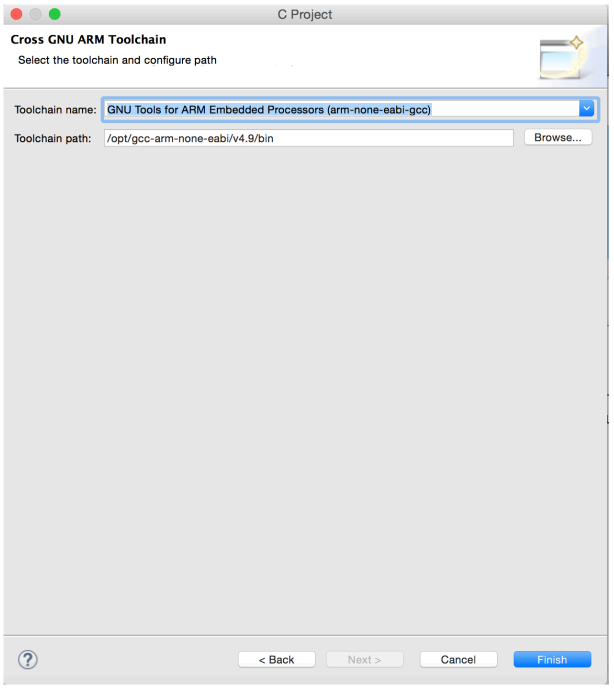

Figura 3.5: Ventana de selección del toolchain a utilizar.

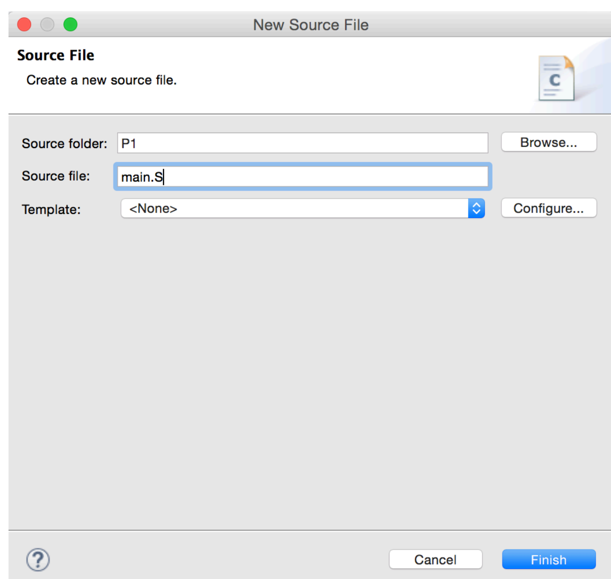

Figura 3.6: Ventana de creación de nuevo fichero fuente.

Cuadro 3 Programa en lenguaje ensamblador que compara dos números y se queda con el mayor.

```
.global start
.data
X:      .word 0x03
Y:      .word 0x0A

.bss
Mayor:  .space 4

.text
start:
        LDR R4, =X
        LDR R3, =Y
        LDR R5, =Mayor
        LDR R1, [R4]
        LDR R2, [R3]
        CMP R1, R2
        BLE L0
        STR R1, [R5]
        B L1
L0:
        STR R2, [R5]
L1:
        B .
        .end
```

ejecutable final. Aprovechamos para ver otra forma de añadir un fichero al proyecto. En el explorador del proyecto seleccionamos el proyecto y pulsamos el botón derecho del ratón, y seleccionamos `New→File`. Se abrirá una ventana como la de la Figura 3.7, seleccionamos el proyecto y ponemos `memmap` como nombre del fichero. Una vez creado, lo abrimos en el editor y copiamos el contenido del cuadro 2.

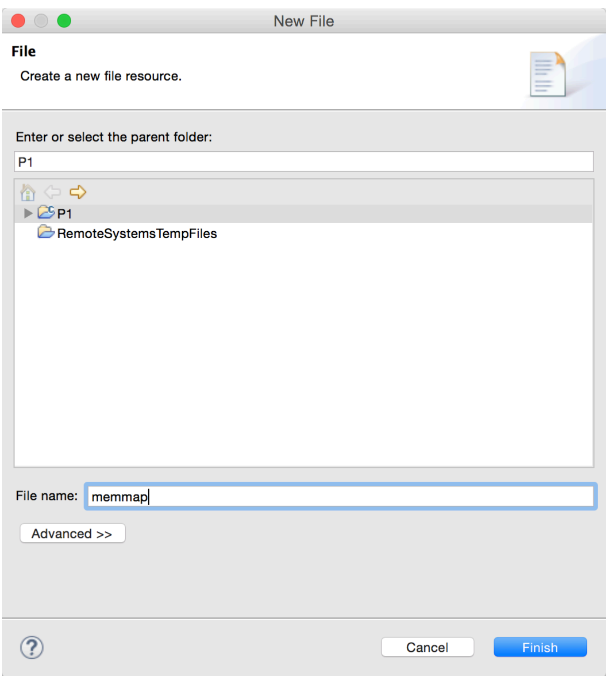

Figura 3.7: Ventana para añadir un nuevo fichero al proyecto.

7. Por último debemos configurar el proyecto para que la compilación se realice correctamente. Para ello seleccionamos el proyecto en el panel izquierdo, pulsamos el botón derecho del ratón y seleccionamos la entrada `Properties` en la parte inferior del desplegable. Con ello se abrirá una ventana como la que muestra la Figura 3.8. En esta ventana seleccionamos `C/C++ Build→Settings` y

    - En `Target Processor` seleccionamos
        - Como procesador `arm1176jzf-s`
        - ARM como `Instruction Set`
        - `Little Endian` en `Endianness`.
        - En `Float ABI` seleccionamos `Library (soft)`.
        - En `Unaligned Access` seleccionamos `Disabled`.
    - En `cross ARM C Linker`, seleccionamos `General`, y en el panel `Script file (-T)` debemos añadir una entrada (botón add, que tiene una cruz verde) y escribir la ruta al fichero `memmap` de nuestro proyecto. Podemos seleccionarlo gráficamente pulsando el botón `Workspace`. También debemos marcar la casilla `Do not use start files (-nostartfiles)` y desmarcar la casilla `Remove unused sections (-Xlinker -gc-sections)`. La Figura 3.9 muestra el resultado.

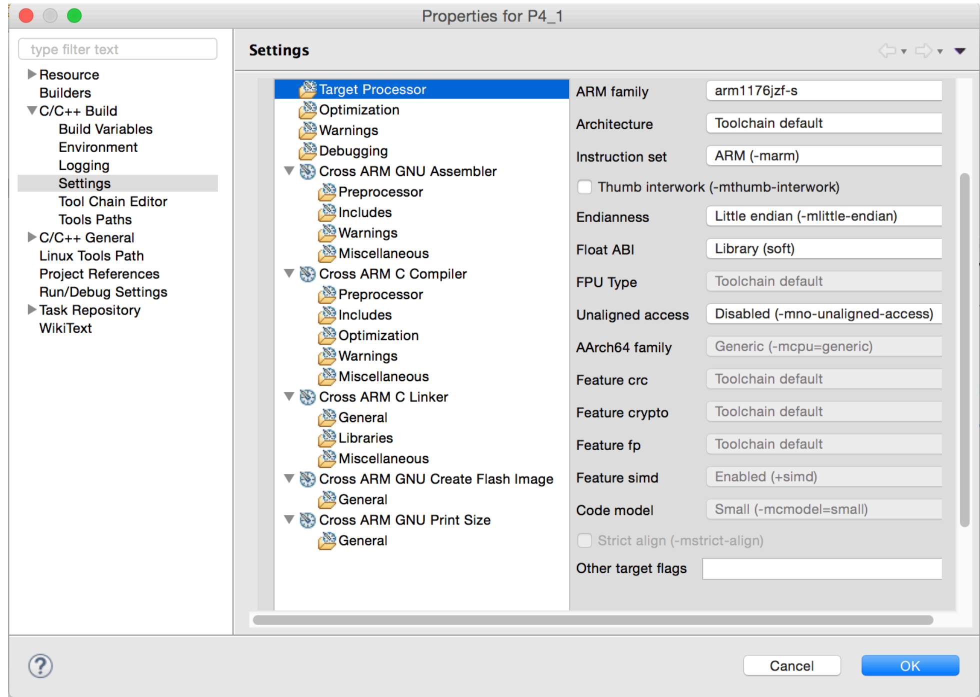

Figura 3.8: Ventana de propiedades (properties) del proyecto.

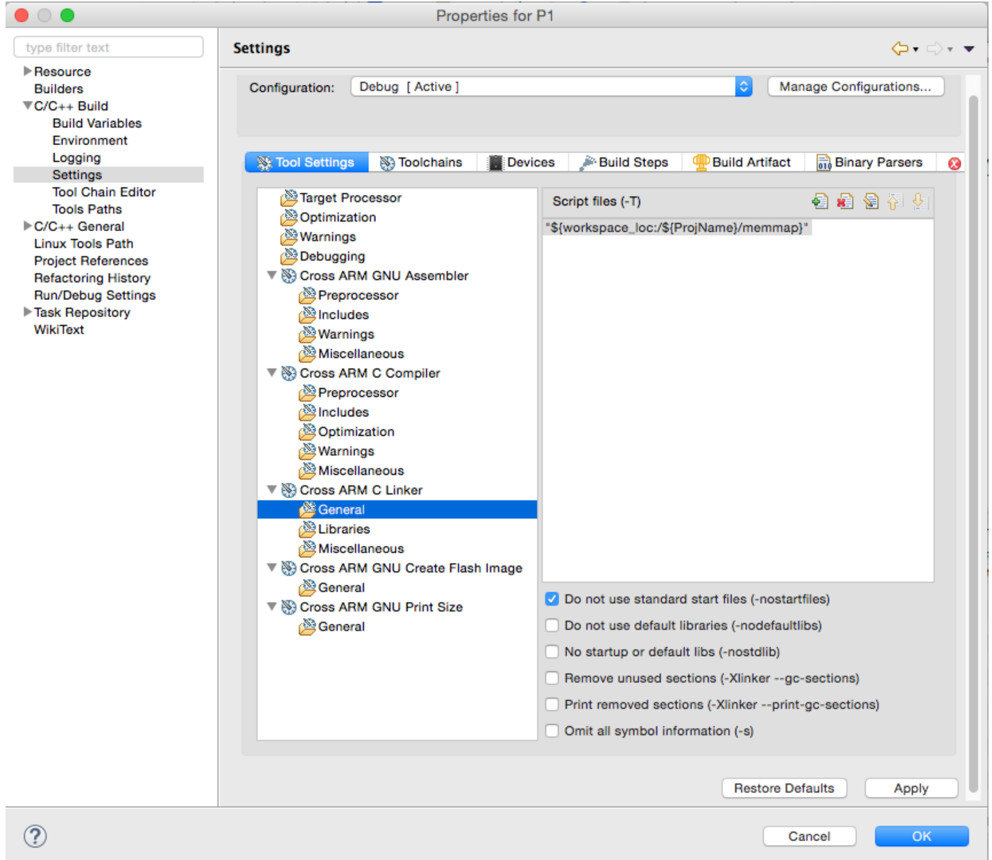

Figura 3.9: Ventana de `Cross ARM C Linker`→`General`.

8. Finalmente podemos compilar el proyecto. Para ello seleccionamos `Project`→`Build Project`. Acabado este paso habremos obtenido el ejecutable final, con extensión `.elf` (*Executable Linked Format*, que se encontrará en el subdirectorio `Debug`, dentro del directorio de proyecto de nuestro `workspace`.

3.2. Depuración sobre circuito

Tras generar el ejecutable de nuestro proyecto, nos toca comprobar que funciona correctamente. Para ello usaremos el depurador `arm-none-eabi-gdb` utilizando Eclipse como interfaz. El depurador nos permitirá ejecutar el programa instrucción a instrucción, poner puntos de parada (`breakpoints`), examinar registros, memoria, etc.

En estas primeras prácticas tendríamos dos posibilidades: simular el códgio o ejecutarlo directamente sobre la placa. En cuanto comencemos a trabajar la E/S sólo la segunda opción será válida, ya que el simulador no permite simular la E/S. Por este motivo vamos a centrarnos en la depuración en circuito, haciendo de vez en cuando alguna mención a los cambios que habría que hacer para utilizar el simulador integrado en gdb [1]. Para hacer la depuración en circuito resulta conveniente tener instalado el plugin de Eclipse `GDB Hardware Debugging`.

Los pasos a seguir son los siguientes:

1. Conectamos la Raspberry Pi al PC mediante el conector USB del dongle de Olimex.

2. Abrimos la perspectiva `Debug`. Para ello seleccionamos `Window`→`Open Perspective`→`Debug`. La apariencia de la ventana de Eclipse cambiará a la mostrada en la Figura 3.10. Como podemos ver, está dividida en varias regiones, cada una de ellas con pestañas:

   - Superior Izquierda: información sobre el proceso de depuración lanzado.
   - Superior Derecha: pestañas donde podemos visualizar los registros, las variables, los breakpoints, ...
   - Central Izquierda: código fuente en depuración. Al principio sólo aparecen los ficheros que tenemos abiertos en la perspectiva C/C++. Se irán abriendo automáticamente nuevas pestañas si el programa en ejecución salta a alguna función definida en otro fichero compilado con símbolos de depuración.
   - Central Derecha: en principio sólo nos muestra una lista de los símbolos definidos por el programa. Es interesante añadir a este panel una pestaña con el código desensamblado. Para ello seleccionamos `Window`→`Show View`→`Disassembly`.
   - Inferior: múltiples pestañas con distinto propósito. Por ejemplo aquí podemos poner el visor de memoria, seleccionando `Window`→`Show View`→`Memory`.

3. Para la conexión a la placa y el posterior volcado de código, usaremos una herramienta externa llamada `OpenOCD`. Para ello seleccionamos `Run`→`External Tools`→`External Tools Configurations...` (también puede llegarse al mismo sitio pinchando en la flecha del botón). Se abrirá una ventana en la que podemos configurar el uso de una

[1] Para simular debemos disponer de una versión del toolchain en la que gdb haya sido compilado con la opción de simulador activa, que no es lo común hoy en día

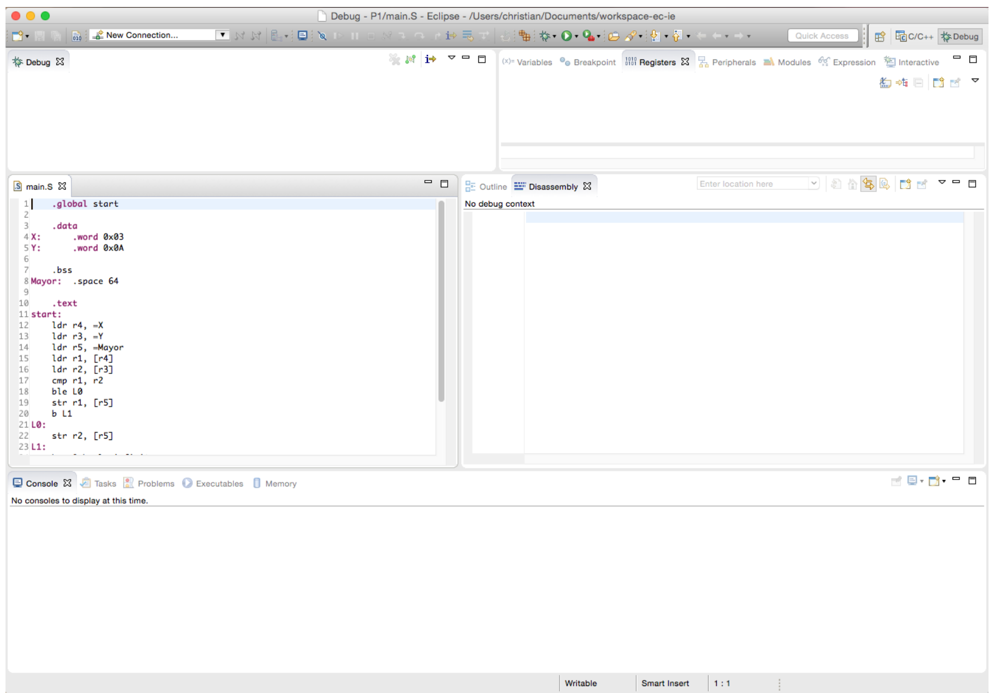

Figura 3.10: Perspectiva de depuración.

nueva herramienta externa. Para ello debemos pinchar en `Program`, con lo que se habilitarán unos botones y pinchamos en el primero de ellos ( ). La Figura 3.11 muestra cómo debemos rellenar el resto de la ventana. Para rellenar la primera entrada podemos pinchar en el botón `Browse File System...` y buscamos el ejecutable de `OpenOCD` en el sistema (téngase en cuenta que la ruta mostrada en la figura puede ser distinta a la ruta que tengamos que poner en el laboratorio). Del mismo modo, para seleccionar el directorio de trabajo podemos pinchar en el botón `Browse Workspace...`. Finalmente debemos rellenar los argumentos al programa (`-f test/raspi-ec-ie.cfg`) tal y como se indica en la Figura 3.11. Terminamos pinchando en `Apply` y `Close`. Esta herramienta externa la tendremos que ejecutar antes de comenzar la depuración, con la placa conectada a un puerto USB del equipo de laboratorio, tal y como indicamos en la siguiente sección.

4. Creamos una configuración de depuración para el proyecto usando el plugin `GDB Hardware Debugging`. Para ello seleccionamos `Run`→`Debug configurations...`, y se abrirá una ventana como la mostrada en la Figura 3.12. En el panel izquierdo seleccionamos `GDB Hardware Debugging` y pulsamos el botón que está en la parte superior izquierda del panel ( ) para crear la configuración. Deberíamos tener una ventana como la mostrada en la Figura 3.13.

   Ahora debemos rellenar correctamente las siguientes pestañas de la configuración:

   - `Main`: en esta pestaña debemos seleccionar el proyecto y el ejecutable que queremos depurar. Ambos pueden ser escogidos gráficamente pulsando en los botones `Browse` y `Search Project` respectivamente.
   - `Debugger`: en esta pestaña indicamos el depurador a utilizar. En el laboratorio la ruta al toolchain cruzado se ha añadido al path del usuario, por tanto basta con

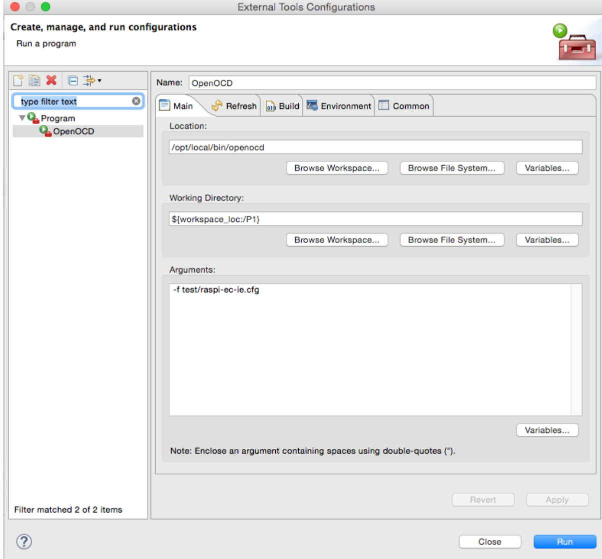

Figura 3.11: Ventana de configuración de OpenOCD como herramienta externa.

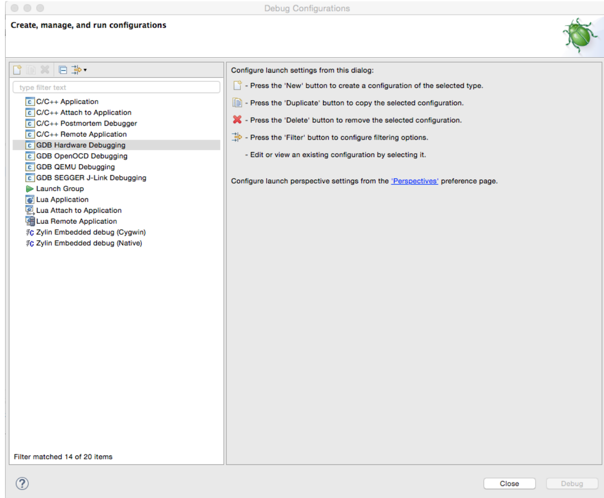

Figura 3.12: Ventana de configuraciones de depuración.

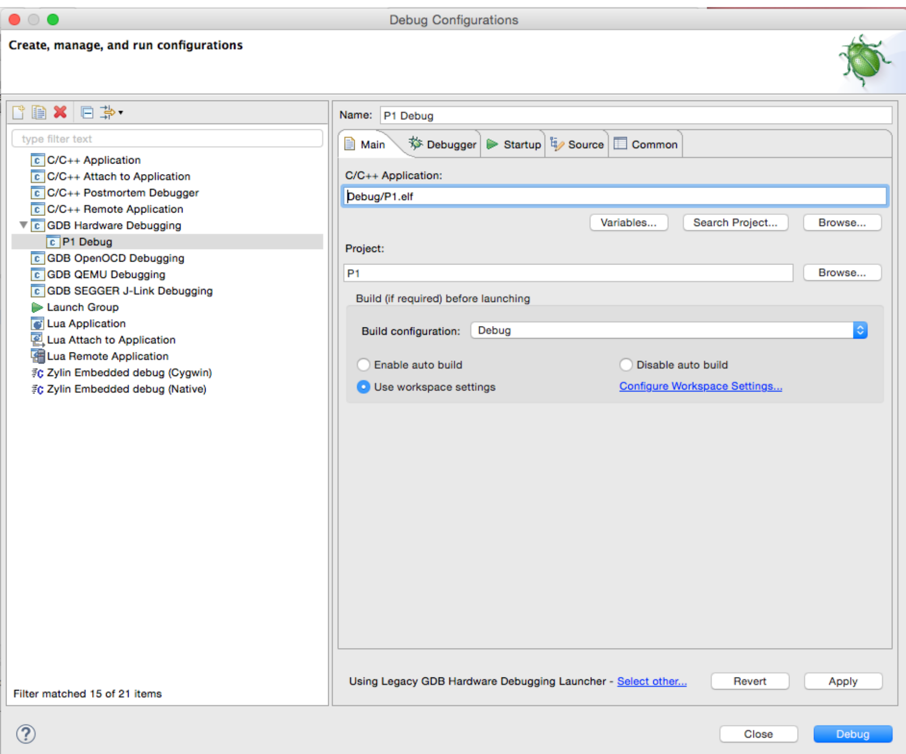

Figura 3.13: Ventana de creación de una nueva configuración GDB Hardware Debugging para el proyecto.

poner el nombre del depurador en la entrada `GDB debugger`: `arm-none-eabi-gdb`. Después básicamente seleccionamos el puerto al que se debe conectar para realizar una depuración remota (3333), tal y como indica la Figura 3.14.

- `Startup`: En esta pestaña configuramos las acciones que deben realizarse cuando se realice la conexión con el servidor de depuración habilitado por OpenOCD. La Figura 3.15 ilustra cómo debemos configurar esta pestaña. Como podemos ver en la parte inferior, en el campo *Set breakpoint at*, se habilita un *breakpoint* inicial en el símbolo `start`, que permite que la ejecución quede detenida al comienzo de nuestro programa tras haber cargado éste en memoria.
- `Common`: Por último, y por comodidad, en esta pestaña marcamos *Debug*. Esto hará que esta configuración de depuración salga en favoritos, y sea más cómodo utilizarla.

5. Una vez completados los pasos anteriores ya estamos en disposición de depurar nuestro programa. Para ello debemos ejecutar primero OpenOCD, pulsando en la flecha del botón de herramientas externas ( ) y seleccionando nuestra herramienta externa OpenOCD. Este paso no tenemos por qué repetirlo siempre si dejamos OpenOCD lanzado.

6. Finalmente podemos comenzar la depuración pinchando en la flecha del icono de depuración ( ) y seleccionando nuestra configuración (como la hayamos llamado antes).

La Figura 3.17 nos muestra el estado inicial que debería tener la ventana en depuración si hemos seguido todos los pasos:

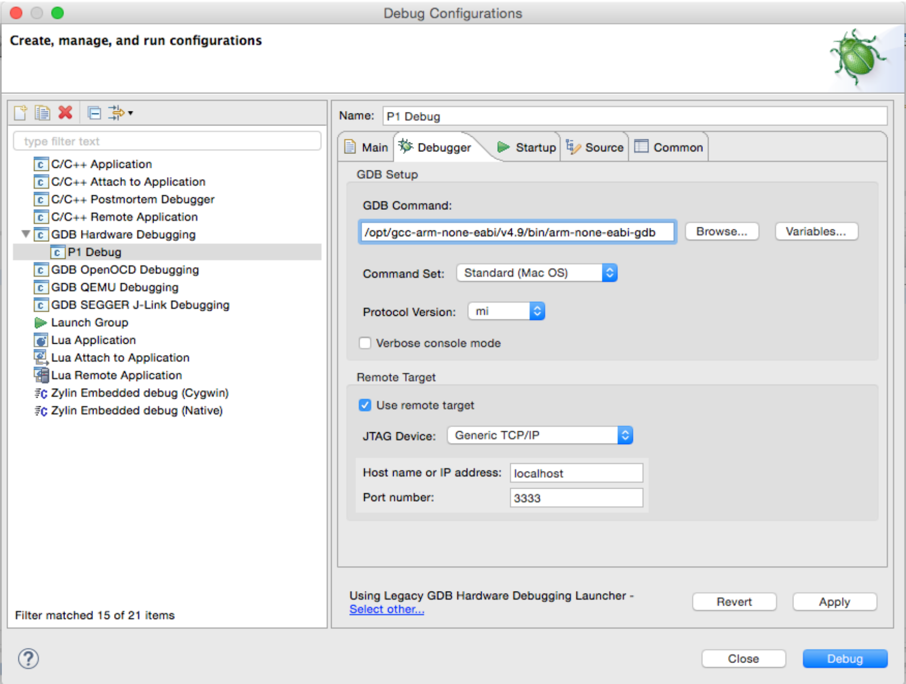

Figura 3.14: Pestaña Debugger.

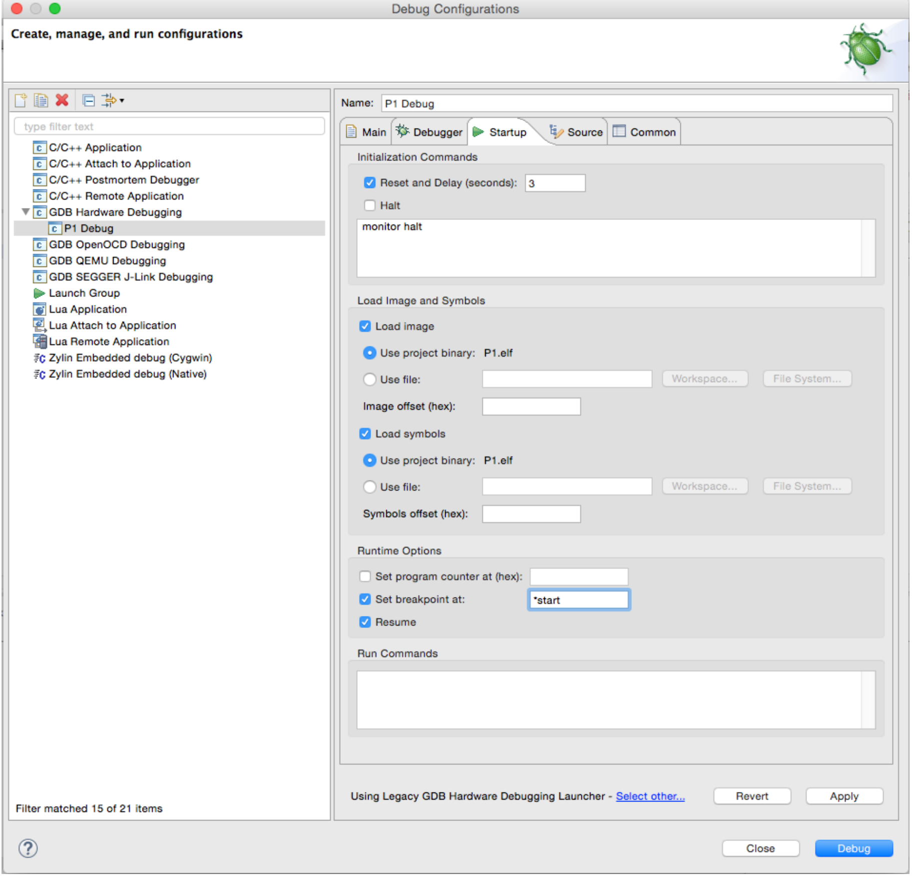

Figura 3.15: Pestaña Startup.

- En la parte izquierda del panel central debemos encontrar el código de nuestro programa (el del cuadro 3), con la primera línea de código tras la etiqueta `start` marcada en verde, con una flecha en el marco izquierdo. Esto quiere decir que el programa a comenzado correctamente su ejecución y que se ha detenido en el breakpoint que hemos puesto en la dirección de `start`.

- En la parte derecha del panel central debemos encontrar el código desensamblado. Es el contenido de la memoria en el entorno de la dirección de la instrucción actual, reinterpretada como instrucciones.

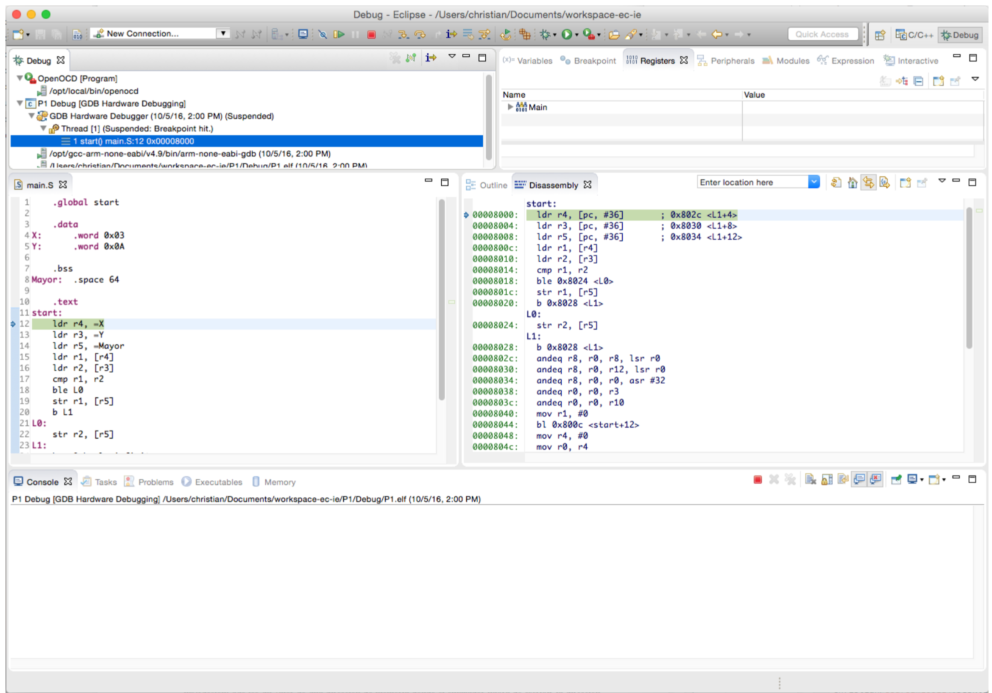

Figura 3.16: Comienzo de la sesión de depuración.

Notemos la diferencia entre los dos paneles. En el izquierdo tenemos el código fuente, tal y como lo programamos, haciendo uso de las facilidades proporcionadas por el ensamblador. En el lado derecho tenemos las instrucciones tal y como las ha generado el ensamblador. Fijémonos por ejemplo en la instrucción `LDR R4,=X`, que aprovecha una facilidad del ensamblador para escribir en el registro `R4` la dirección correspondiente a la etiqueta `X`. Como podemos ver en el panel derecho, se ha traducido por `LDR R4,[PC,#36]`, que utiliza un modo de direccionamiento correcto para las instrucciones `LDR` en ARM [2]. También podemos observar que la dirección equivalente a `PC+36` aparece como un comentario tras el signo `;`. En cualquiera de los dos paneles podemos poner breakpoints. Si marcamos el botón en el panel de desensamblado, las instrucciones fuente se mezclarán con las instrucciones máquina correspondientes. Esto facilita el seguimiento del código máquina desensamblado, sobre todo cuando el código fuente es de un lenguaje de alto nivel como C, como veremos en las siguientes prácticas.

[2]El proceso de esta traducción implica tanto al ensamblador como al enlazador. El ensamblador pone esta instrucción, que lee un valor de una dirección de memoria donde el enlazador habrá de escribir la dirección correspondiente a la etiqueta `X`. Estudiaremos este proceso en más detalle en la práctica 4.

Antes de comenzar con la depuración de nuestro código vamos a abrir una visor de memoria para poder evaluar el valor de nuestras variables. Para ello, lo primero que tenemos que hacer es abrir la pestaña `Memory` en el panel inferior, si no la tenemos abierta ya (`Window`→`Show View`→`Memory`). Seleccionamos esta pestaña y añadimos un nuevo monitor pinchando en el icono ✚. Se abrirá una ventana como la de la Figura ?? en la que debemos escribir la dirección a partir de la que queremos monitorizar la memoria. Si conocemos la dirección podemos ponerla, pero en este caso hemos colocado los datos tras el código y no sabemos exactamente cuánto ocupaba la sección `.text`. En este caso lo más cómodo es indicarle que use la dirección del símbolo `X`, escribiendo en el recuadro `&X`, como podemos ver en la Figura ??.

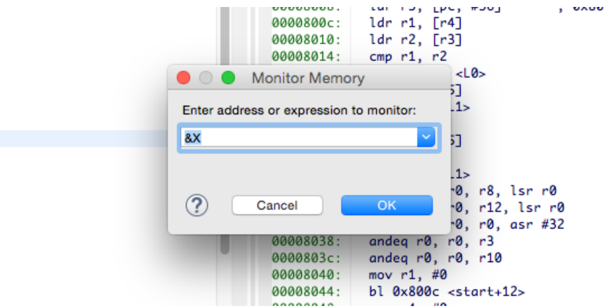

Figura 3.17: Comienzo de la sesión de depuración.

Ahora, para depurar nuestro código se pude pulsar sobre el icono de `Resume` ▶ o F8 y después pulsar sobre el icono de `Suspend` ⏸.

- ¿Cómo sabemos si el código se ha ejecutado correctamente? El dato mayor se ha escrito en la posición de memoria reservada y etiquetada como `Mayor`. Se puede comprobar su valor en el visor de memoria, que habrá quedado marcado en rojo como ilustra la Figura 3.18.

Sin embargo, para entender mejor el funcionamiento de cada una de las instrucciones conviene realizar una ejecución paso a paso. Además, si el resultado no es correcto tendremos que depurar el código para encontrar cuál es la instrucción incorrecta, para lo que una ejecución paso a paso nos será muy útil. Para ello:

- Paramos la depuración pulsando el botón `Terminate`, ■, luego seleccionamos la entrada del arm-none-eabi-gdb de la pestaña `Debug` y volvemos a pulsar el botón `Terminate`. Finalmente seleccionamos la entrada `Debu [GDB Hardware Debugging]`, pulsamos el botón derecho del ratón y seleccionamos la opción `Terminate and Remove` en el menú desplegable.

- Si necesitamos modificar el código, pasamos a la perspectiva `C/C++`, editamos los ficheros, los guardamos y recompilamos el proyecto. Luego volvemos a la perspectiva `Debug`.

- Podemos iniciar una sesión de depuración con la última configuración de depuración utilizada pulsando el botón ✲▾.

- Para ejecutar paso a paso tenemos las siguientes opciones:

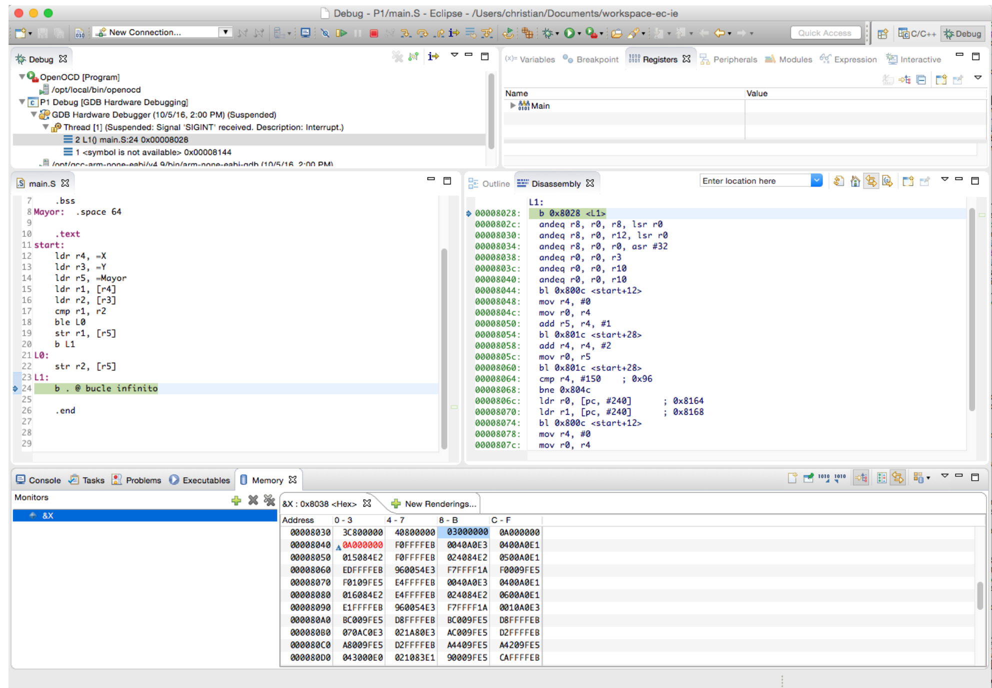

Figura 3.18: Resultado de la ejecución completa del código en depuración.

- Step Over ( ): ejecuta hasta la siguiente instrucción. Si es una llamada a subrutina para después de la ejecución completa de la subrutina.
- Step Into ( ): ejecuta hasta la siguiente instrucción. Si es una llamada a subrutina se detiene en la primera instrucción de la subrutina.
- Poner un `Breakpoint` (punto de parada) y reanudar la ejecución pulsando el botón `Resume` . Para poner un `Breakpoint` nos ponemos en el margen izquierdo de alguna instrucción (tanto en el panel de código fuente como en el panel de desensamblado) y hacemos un doble click. Aparecerá la marca . Podemos poner varios `Breakpoints`, la ejecución se detendrá en aquel que se alcance primero.

Además, nos interesará observar los cambios que va realizando nuestro programa, para ello:

- Para ver cómo va cambiando el valor de los registros a medida que ejecutamos las instrucciones usaremos el visor de registros, que se encuentra en el panel superior derecho. Conviene seleccionar para este visor, con el botón , el layout horizontal.
- En el visor de memoria veremos los cambios que nuestro programa realice sobre la memoria.

La Figura 3.19 muestra un ejemplo de una sesión de depuración, donde hemos ido ejecutando paso a paso hasta la instrucción siguiente a `BLE L0`. Podemos ver que la condición no se cumple y que se pasa a ejecutar el bloque que comienza en la etiqueta `L0`. Podemos ver el estado de los registros en este momento, y como el `PC` contine la dirección `0x00008024` correspondiente a la siguiente instrucción a ejecutar. También podemos ver cómo el panel de desensamblado nos marca en verde las últimas instrucciones ejecutadas, quedando sin marcar las instrucciones en la rama del then.

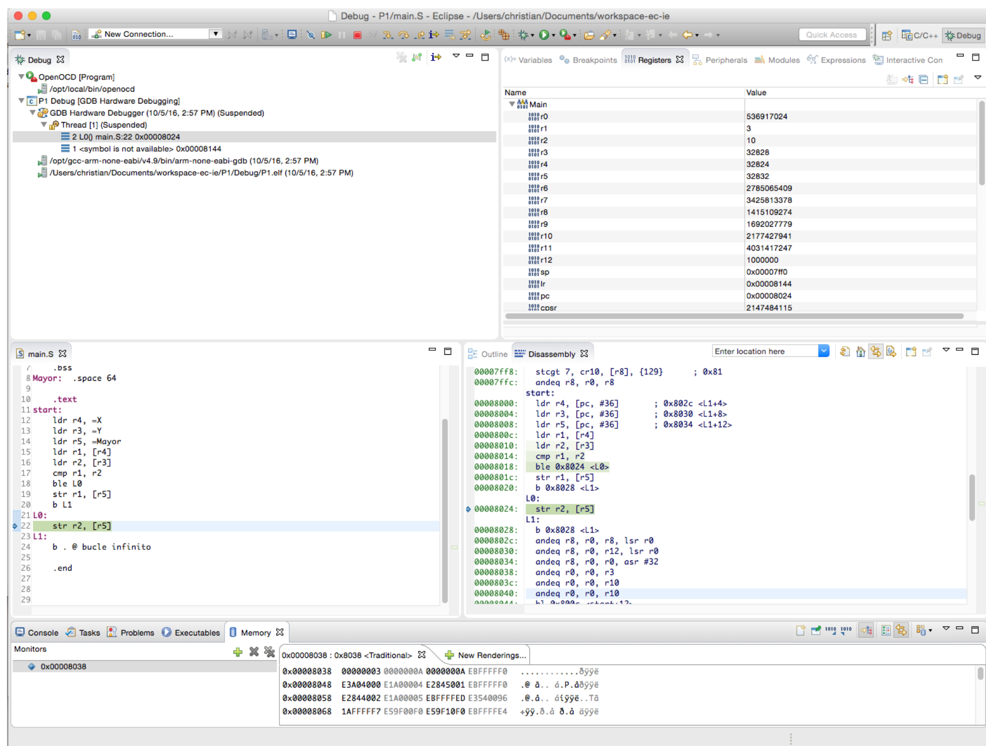

Figura 3.19: Sesión de depuración activa sobre el código de ejemplo.

Capítulo 4

Subrutinas

Con lo que hemos visto hasta ahora podemos implementar sencillos programas escritos en lenguaje ensamblador. Sin embargo, en cuando la funcionalidad del programa se complique un poco vamos a estar muy limitados.

Para poder afrontar problemas más complejos es conveniente utilizar el concepto de subrutina, un fragmento de código que podemos invocar desde cualquier punto del programa (incluyendo otra o la propia subrutina), retomándose la ejecución del programa en la instrucción siguiente a la invocación cuando la subrutina haya terminado.

La Figura 4.1 presenta un ejemplo, con un programa principal que invoca la subrutina `SubRut1`. Las flechas discontinuas indican el cambio en el flujo de ejecución, tanto en la invocación como en el retorno.

Las subrutinas se usan para simplificar el código y poder reusarlo, y son el mecanismo con el que los lenguajes de alto nivel implementan procedimientos, funciones y métodos.

Una subrutina puede estar parametrizada, esto es, el algoritmo que implementa puede especificarse en función de unos parámetros, a los que podemos dar valor en la invocación. Por ejemplo, la subrutina podría tener un bucle desde 1 hasta `N`, siendo `N` un parámetro. Cuando se invoca la subrutina se asigna un valor a `N` específico para esa invocación. Se dice que se pasan los parámetros en la llamada o invocación. Una subrutina puede además retornar un valor.

Como ilustra la Figura 4.1 sólo existe una copia del código de una subrutina, que debe colocarse fuera de la región del programa principal.

Las subrutinas deben además preservar el estado arquitectónico (el valor de los registros visibles al programador) ya que de lo contrario no podríamos invocarlas desde cualquier punto del programa sin afectar al resto del programa.

Finalmente, el uso de subrutinas permite repartir el trabajo, las subrutinas implementadas por unos pueden ser utilizadas por otros, sin necesidad de que estos conozcan los detalles de implementación de las mismas. Para ello es imprescindible ponerse de acuerdo en cómo implementarlas. Por eso en todas las arquitecturas existe un estándar de llamadas a subrutina, que todos deben respetar si quieren que su código pueda interactuar con el de los demás.

Si queremos utilizar subrutinas debemos pues plantearnos las siguientes cuestiones:

- ¿Cómo se hace la invocación y el retorno de una subrutina?
- Si la subrutina debe operaciones sobre registros, ¿cómo puede preservar el estado arquitectónico?

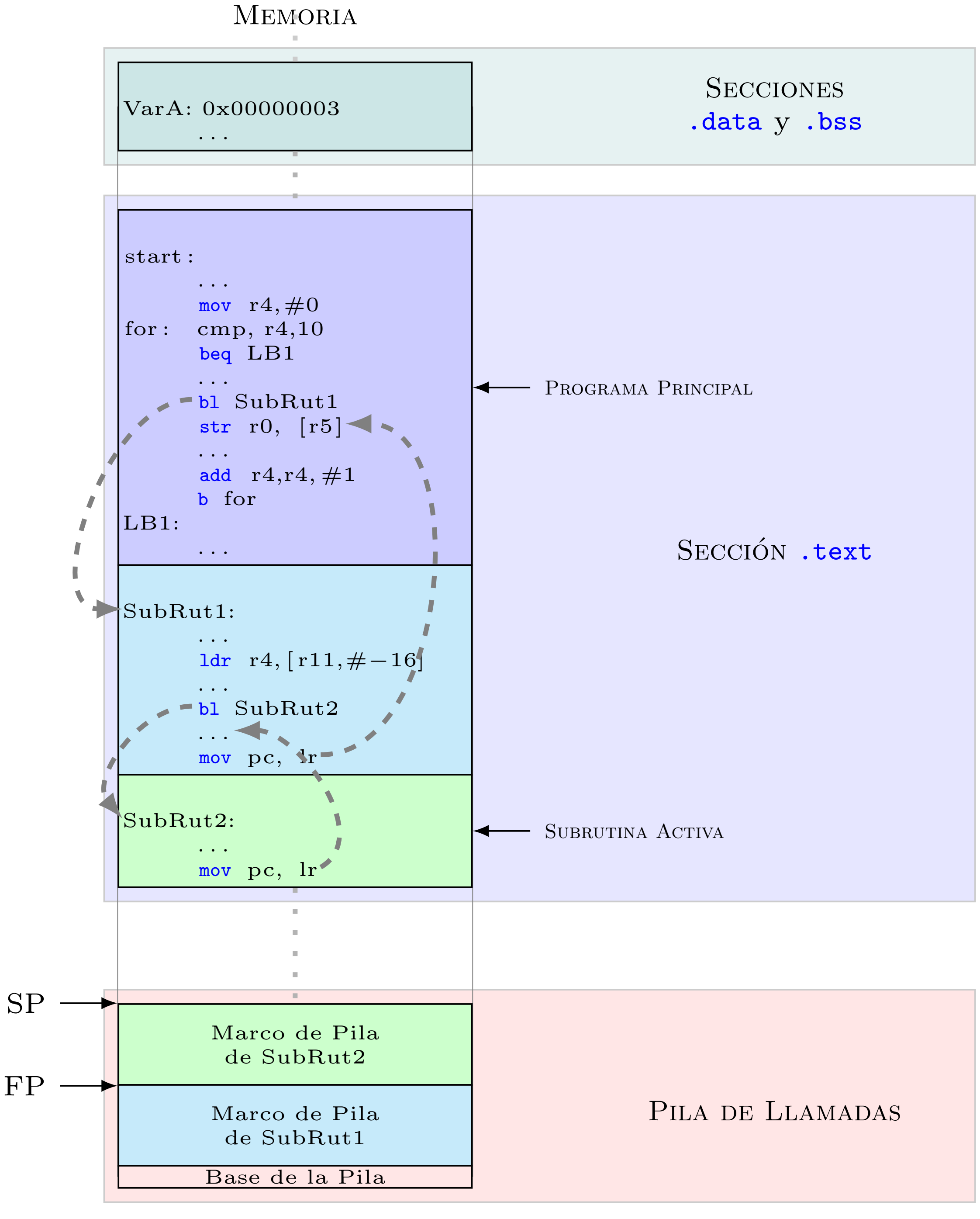

Figura 4.1: Subrutinas y Pila de Llamadas. El programa principal invoca la subrutina `SubRut1`, que a su vez invoca a `SubRut2`, que pasa a ser la subrutina activa. Los punteros `SP` y `FP` delimitan el marco de la subrutina activa.

- ¿Cómo se pasan los parámetros a una subrutina y cómo devuelve esta el resultado?

Las siguientes secciones responden a estas cuestiones, y nos presentan el estándar de llamadas a subrutina para la arquitectura ARM. Estudiaremos también el concepto de variable local y cómo podemos utilizar este valioso recurso.

4.1. Implementación de subrutinas

Comenzaremos abordando el problema de la llamada subrutina. Comenzaremos por identificar la primera instrucción de una subrutina con una etiqueta. Entonces para invocarla basta con hacer dos cosas:

- almacenar en algún lugar conocido (acuerdo tácito con el que implementa la subrutina) la dirección de la instrucción por la que debe continuar la ejecución cuando termine la subrutina, y
- hacer un salto a la etiqueta que indica el comienzo de la subrutina.

Esto puede hacerse de varias maneras, pero generalmente todos los repertorios de instrucciones proporcionan una instrucción especial para la invocación de subrutinas. En el caso de ARM esta instrucción es el *Branch and Link*, cuya sintaxis en ensamblador es:

```
BL Etiqueta
```

Esta instrucción hace dos cosas: guarda en el registro `R14` la dirección siguiente a la instrucción `BL` (la dirección de retorno) y salta a la dirección indicada por `Etiqueta`. Cuando se hace este uso del registro `R14` se le denomina registro de enlace (*link register*), y por eso el ensamblador permite referirse a `R14` como `LR`.

Para retornar de la subrutina basta con escribir la dirección de retorno sobre `PC`. En el caso de ARM esta dirección estará almacenada en `LR` (`R14`), y para escribirla en `PC` bastaría con usar la instrucción `MOV` (como en la Figura 4.1):

```
MOV PC, LR
```

Alternativamente podríamos usar la instrucción `BX` (*brach and exchange*):

```
BX LR
```

La segunda cuestión es referente a la preservación del estado arquitectónico. La Figura 4.1 ilustra este problema. El programa principal recorre un bucle `for` usando `r4` para almacenar el iterador. En el cuerpo del bucle se llama a la subrutina `SubRut1` que contiene la instrucción:

```
ldr   r4, [r11, #-16]
```

El registro `r4` es machacado, con el indeseado efecto lateral de modificar el iterador del bucle. Obviamente esto no pasaría si el programador no hubiese escogido `r4` para almacenar el contador del bucle. Sin embargo nos interesa que quién utilice la subrutina no tenga que conocer los detalles de implementación de la misma para usarla.

La solución a este problema es sencilla, toda subrutina debe copiar temporalmente en memoria los registros que vaya a modificar, y restaurarlos justo antes de hacer el retorno, una vez que haya completado su trabajo.

Por tanto una subrutina necesitará temporalmente para su ejecución un poco de memoria donde poder guardar los registros que va a modificar. Fijémonos que esta reserva de memoria no puede ser estática, ya que no sabemos cuántas invocaciones de una subrutina puede haber

en curso. Pensemos por ejemplo en una implementación recursiva del factorial de un número natural n. Tendremos al final n invocaciones en vuelo y por tanto necesitaremos memoria para salvar n veces el contexto, pero n es un dato de entrada. La mejor solución es que cada subrutina reserve la memoria que necesite para guardar el contexto al comenzar su ejecución (cada vez que se invoca) y la libere al terminar.

Para esto se utiliza una región de memoria conocida como la pila de llamadas (*call stack*). Es una región continua de memoria cuyos accesos siguen una política LIFO (*Last-In-First-Out*), que sirve para almacenar información relativa a las subrutinas activas del programa. La pila suele ubicarse al final de la memoria disponible, en las direcciones más altas, y suele crecer hacia las direcciones bajas.

La región de la pila utilizada por una rutina en su ejecución se conoce como el Marco de Activación o Marco de Pila de la subrutina (*Activation Record* o *Stack Frame* en inglés). El ejemplo de la Figura 4.1 muestra el estado de la pila cuando la subrutina `SubRut1` ha invocado a la subrutina `SubRut2` y esta última está en ejecución. La pila contiene entonces los marcos de activación de las subrutinas activas.

Para simplificar la generación del código de las subrutinas, suelen estructurarse en tres partes:

```
Código de entrada (prólogo)
Cuerpo de la subrutina
Código de salida (epílogo)
```

El prólogo es el código encargado de construir el marco de activación de la subrutina en la cima de pila. Asimismo, el epílogo es el código encargado de extraer el marco de activación de la pila, dejándola tal y como estaba antes de la ejecución del prólogo.

Habitualmente la gestión de la pila se lleva a cabo mediante un puntero al último elemento (cima de la pila) que recibe el nombre de *stack pointer* (`SP`), que normalmente es almacenado en un registro arquitectónico inicializado a la base de la región de pila cuando comienza el programa (pila vacía). También es frecuente el uso de un segundo registro que denominado *frame pointer* (`FP`) que se deja apuntando a la base del marco de activación.

Las subrutinas usan su marco de activación como una zona de memoria privada, sólo accesible desde el contexto de la propia subrutina. Una parte del marco la utilizará para guardar una copia de los registros que vaya a modificar durante su ejecución, para restaurarlos en la ejecución del epílogo. Veremos más adelante como también podemos usar el marco de activación para almacenar en memoria variables locales a la subrutina, que se crean con el marco y dejan de ser accesibles cuando la subrutina finaliza.

Finalmente, queda por resolver el problema del paso de parámetros y el retorno del resultado. Para pasar los parámetros tenemos dos alternativas:

1. Pasarlos por registro (como hemos hecho con la dirección de retorno).

2. Pasarlos por memoria. El lugar ideal sería la pila de llamadas. El programa los copiaría en la cima de la pila, la subrutina insertaría su marco de activación justo por encima de los parámetros, pudiendo acceder a ellos con desplazamientos relativos a `SP` o `FP`.

Para devolver el resultado tenemos las mismas opciones, devolverlo en algún registro arquitectónico o copiarlo en la pila.

Es evidente que debe haber un acuerdo entre el programador que escribe la subrutina y el programador que escribe el programa que la invoca, deben estar de acuerdo en cómo

se pasarán los parámetros y qué registros y/o direcciones de memoria se usarán para cada parámetro. Debemos pensar que lo habitual es utilizar código escrito por otras personas o código generado por un compilador a partir de un programa que escribamos en algún lenguaje de alto nivel, y por lo tanto necesitamos una serie de normas generales para la invocación y la escritura de subrutinas. Estas normas las especifica el fabricante, ARM en nuestro caso, en el estándar de llamadas a subrutinas, que forma parte de lo que se denomina el *Application Binary Interface* (ABI).

4.2. Estándar de llamadas a subrutinas

El *ARM Architecture Procedure Call Standard* (AAPCS) es el estándar vigente que regula las llamadas a subrutinas en la arquitectura ARM [aap] (sustituye a los estándares anteriores APCS y ATPCS). Especifica una serie de normas para que las rutinas puedan ser compiladas o ensambladas por separado, y que a pesar de ello puedan interactuar. En definitiva, supone un contrato entre el código que invoca y la subrutina invocada. Los siguientes apartados presentan de forma resumida los aspectos más relevantes que cubre el estándar. No obstante debemos notar que dicho estándar está en constante revisión y evolución, por ello es importante en la práctica comprobar la versión del ABI con la que se vaya a trabajar y estudiar el estándar correspondiente.

4.2.1. Uso de registros arquitectónicos

El estándar determina qué registros arquitectónicos debe preservar una subrutina y cuales no es necesario que preserve. Como mencionamos arriba, si la subrutina necesita utilizar alguno de los registros que debe preservar tendrá que copiarlo antes en su marco de activación y restaurar su valor antes del retorno. Por otro lado, el código que invoca a una subrutina debe ser muy consciente de que la subrutina no tiene la obligación de preservar algunos registros, y debe considerar siempre que el valor que tengan estos registros se perderá a través de la llamada.

La tabla 4.1 describe los usos que el AAPCS da a cada uno de los registros. Como vemos los registros reciben un nombre alternativo (alias), que nos recuerda el uso que les asigna el AAPCS. Observemos que aunque no sea necesario preservar el valor de `LR`, este registro es el que contiene la dirección de retorno. Si la subrutina hace llamadas a otras subrutinas (o la misma si es recursiva) tendrá que utilizar `LR` para escribir una nueva dirección de retorno (recordemos que esto lo hace `BL` automáticamente). Esto sucede en el ejemplo de la Figura 4.1 con `SubRut1`, que invoca a `SubRut2`. Entonces `SubRut1` deberá copiar también `LR` en su marco de activación para no perder la dirección de retorno al programa principal. Esto no es necesario si la subrutina no invoca otras subrutinas (`SubRut2`). Esas subrutinas se dice que son subrutinas hoja.

Tabla 4.1: Uso de los registros arquitectónicos según el AAPCS

REGISTROS	ALIAS	DEBEN SER PRESERVADOS	USO SEGÚN AAPCS
r0-r3	a1-a4	NO	Se utilizan para pasar parámetros a la rutina y obtener el valor de retorno.
r4-r10	v1-v7	SÍ	Se utilizan normalmente para almacenar variables debido a que deben ser preservados en llamadas a subrutinas.
r11	fp, v8	SÍ	Es el registro que debe utilizarse como *frame pointer*.
r12	ip	NO	Puede usarse como registro auxiliar en prólogos.
r13	sp	SÍ	Es el registro que debe utilizarse como *stack pointer*.
r14	lr	NO	Es el registro que debe utilizarse como *link register*, es decir, el utilizado para pasar la dirección de retorno en una llamada a subrutina.

4.2.2. Modelo de pila

La figura 4.2 ilustra el modelo *full-descending* impuesto por el AAPCS, que se caracteriza por:

- SP se inicializa con una dirección de memoria *alta* y crece hacia direcciones más bajas.
- SP apunta siempre a la última posición **ocupada**.

Además, se impone como restricción que el SP debe tener siempre una dirección múltiplo de 4.

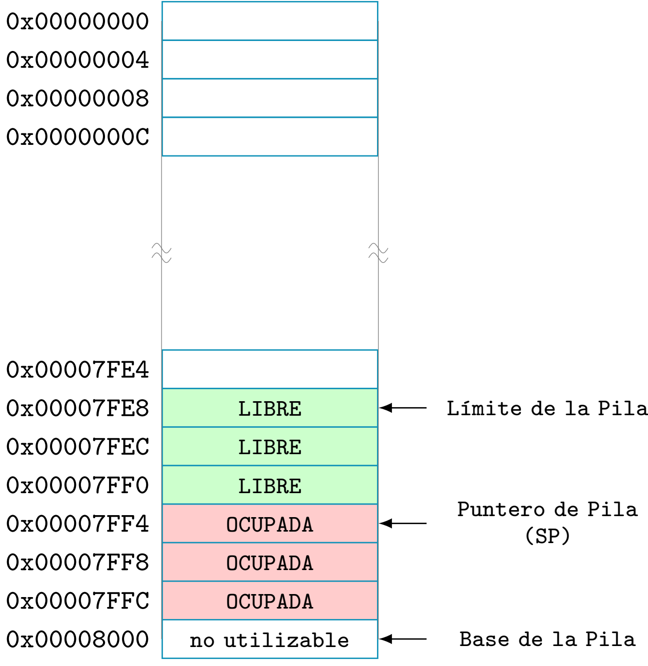

Figura 4.2: Ilustración de una pila *Full Descending*.

4.2.3. Paso de parámetros

Según el estándar, los cuatro primeros parámetros de una subrutina se deben pasar por registro, utilizando en orden los registros R0-R3. A partir del cuarto parámetro hay que pasarlos por memoria, escribiéndolos en orden en la cima de la pila:

- el primero de los parámetros restantes en [SP],
- el siguiente en [SP+4],
- el siguiente en [SP+8],
- ...

De esta forma la subrutina siempre puede acceder a los parámetros que se le pasan por memoria con desplazamientos positivos a FP o SP.

Observemos que esto supone colocarlos en la parte superior del marco de activación de la rutina invocante. Esto hay que tenerlo en cuenta en el prólogo para calcular correctamente el espacio total de marco que necesita la subrutina. La figura 4.3 ilustra un ejemplo en el que una subrutina SubA debe invocar otra subrutina SubB, pasándole 7 parámetros que llamamos Param1-7. Como podemos ver, los cuatro primeros parámetros se pasan por registro (R0-R3), mientras que los últimos tres se pasan por pila, en la cima del marco de SubA.

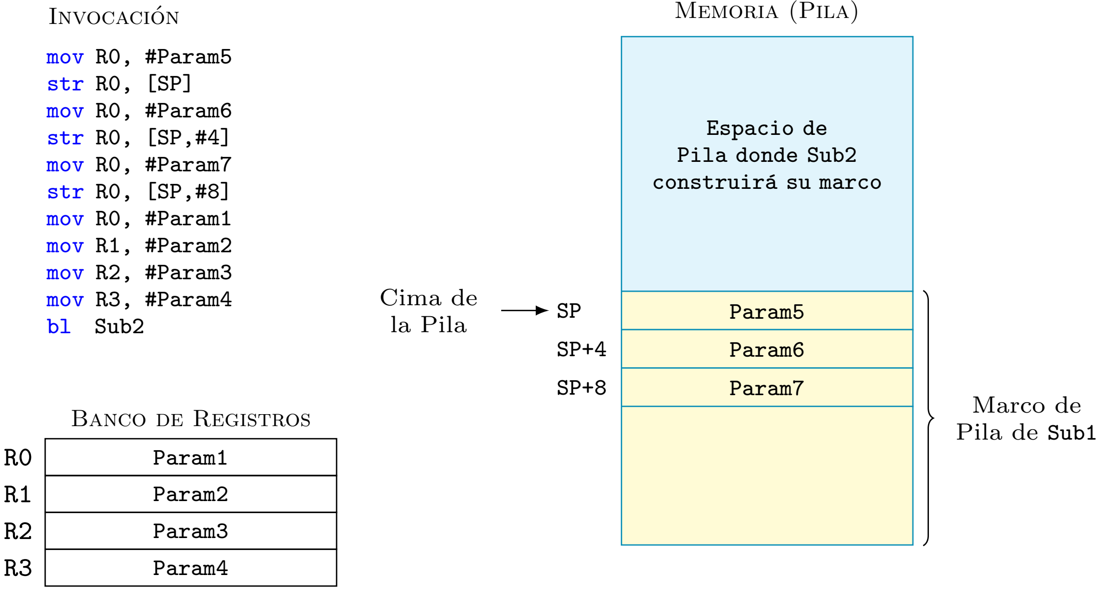

Figura 4.3: Paso de los parámetros Param1-7 desde SubA a SubB. El código de invocación asume que los parámetros son constantes (valores inmediatos), en caso de ser variables habría que cargarlos con instrucciones ldr en lugar de mov.

4.2.4. Valor de retorno

Para aquellas subrutinas que devuelven un valor (funciones en lenguajes de alto nivel), el AAPCS especifica que deben devolverlo en R0 [1].

4.3. Marco de Activación

Los estándares antiguos (APCS y ATPCS) daban algunas posibles estructuras para el marco de activación de una rutina. Sin embargo, con el AAPCS desaparecen estas restricciones. Cualquier rutina que cumpla con las condiciones presentadas en la Sección 4.2 estaría conforme al estándar. No obstante, en este laboratorio vamos a seguir siempre las siguientes pautas:

- Si nuestra subrutina invoca a otras subrutinas, tendremos que almacenar `LR` en la pila, ya que contiene la dirección de retorno, que sobrescribiremos al pasar a su vez la dirección de retorno a las subrutinas invocadas.
- En el caso de utilizar frame pointer (opcional y obsoleto) usaremos el registro `R11` que admite el alias `FP`.
- Aunque `SP` es un registro que hay que preservar, no será necesario incluirlo en el marco, ya que en el epílogo podremos calcular el valor anterior de `SP` revirtiendo las operaciones hechas en el prólogo.
- De los registros `R4-R11` insertaremos sólo aquellos que modifiquemos en el cuerpo de la subrutina.

La Figura 4.4 ilustra el marco de activación resultante de aplicar estas directrices a una subrutina que no es hoja, que es el que genera el compilador gcc-4.7 con depuración activada y sin optimizaciones[2].

Una de las ventajas de utilizar `FP` es que se forma en la pila una lista enlazada de marcos de activación. Al salvar `FP` en el marco de una subrutina, estamos almacenando la dirección de la base del marco de activación de la subrutina que la invocó. Podemos así recorrer el árbol de llamadas a subrutina hasta llegar a la primera. Para identificar esta primera subrutina el programa principal pone `FP` a 0 antes de invocarla.

En el siguiente apartado estudiaremos cómo construir este marco de activación en el prólogo de la subrutina y cómo restaurar el contexto en su epílogo.

4.3.1. Prólogo y Epílogo

Para insertar el marco de activación en la pila (*push*) suelen utilizarse en ARM las instrucciones de store múltiple. Del mismo modo, para extraer el marco de la pila (*pop*) una vez terminada la subrutina suelen utilizarse instrucciones de load múltiple.

El cuadro 4 describe un posible prólogo para la construcción del marco de activación representado en la Figura 4.4, en el que no utilizamos *frame pointer*. La primera instrucción apila los registros a preservar sobre el marco de la rutina invocante, dejando `SP` apuntando a la nueva cima. Debemos incidir en que no es necesario apilar siempre todos los registros

[1] Esta es una simplificación válida para valores de tamaño palabra o menor, que será suficiente en todos los casos en este laboratorio. Para más información debe consultarse el documento oficial del AAPCS [aap]

[2] Con los flags -g -O0

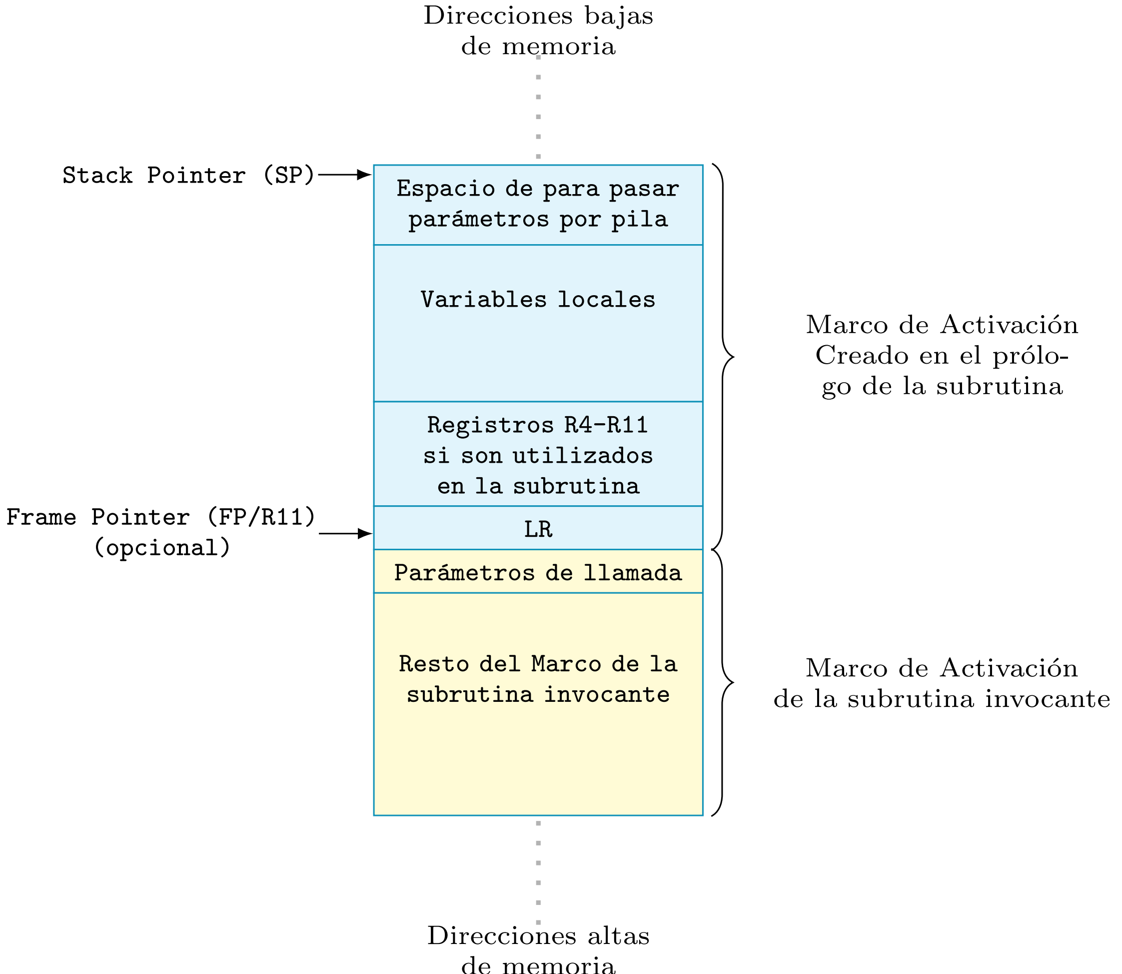

Figura 4.4: Estructura de Marco de Activación recomendada. En el caso de subrutinas hoja podremos omitir la copia de `LR`.

`R4-R11`, sólo aquellos que se vayan a modificar en la rutina. La siguiente instrucción reserva nuevo espacio en el marco, el necesario para almacenar las variables locales y los parámetros que deba pasar por pila a las subrutinas que invoque.

Cuadro 4 Prólogo para insertar en la pila el Marco de Activación de la Figura 4.4.

```
PUSH {R4-R11,LR}                @ Copiar registros en la pila
SUB  SP, SP, #4*NumPalabrasExtra   @ Espacio extra necesario
```

Existen algunos casos especiales en los que podemos simplificar este prólogo:

- Si no pasamos más de cuatro parámetros a ninguna subrutina y no vamos a utilizar variables locales, no tendremos que reservar espacio extra. Podemos entonces eliminar la segunda instrucción.
- Si no apilamos ningún registro (sucedería en una subrutina hoja que no modifique ninguno de los registros `r4-r10`), en este caso podemos eliminar la primera instrucción.

- Una combinación de los dos casos, podríamos entonces eliminar el prólogo

El cuadro 5 presenta el correspondiente epílogo, que restaura la pila dejándola tal y como estaba antes de la ejecución del prólogo. Lo primero que hace es asignar a SP el valor que tenía tras la ejecución de la primera instrucción del prólogo. Esto lo consigue realizando la operación contraria a la segunda instrucción del prólogo, asumiendo que no hemos alterado el valor de SP en el cuerpo de la subrutina. Con la segunda instrucción del epílogo se desapilan los registros apilados en el prólogo, dejando así la pila en el mismo estado que estaba antes de la ejecución del prólogo. Lo único que queda es hacer el retorno de subrutina, que es lo que hace la última instrucción del epílogo. En R4-R11 y LR se copian los valores que tenían al entrar en la subrutina, por lo que al terminar la ejecución del epílogo se restaura por completo el estado arquitectónico.

Cuadro 5 Epílogo para extraer de la pila el Marco de Activación de la Figura 4.4.

```
ADD  SP, SP, #4NumPalabrasExtra @ SP \textcolor{ForestGreen}{\leftarrow} dirección
    del \textcolor{ForestGreen}{1^{er}} registro@ apiladoPOP R4-R11,LR @ Desapila
    restaurando registrosBX LR @ Retorno de subrutina
```

De nuevo, hay algunas situaciones en las que podemos simplificar este epílogo:

- Si no hemos reservado espacio extra (NumPalabrasExtra=0) y no hemos alterado el valor de SP en el cuerpo de la subrutina, podemos eliminar la primera instrucción del epílogo, ya que SP tendrá el valor que le asignó la primera instrucción del prólogo.

- Si no apilamos ningún registro, podemos eliminar la segunda instrucción.

4.4. Un ejemplo

El código de la Figura 4.5 ilustra un ejemplo de un programa que utiliza la subrutina Recorre para recorrer un array A. En cada posición j selecciona el mayor entre los elementos A[j] y A[j+1] utilizando la subrutina Mayor. El mayor de los dos se copia en la posición j de otro array B. A la izquierda podemos ver la implementación C y a la derecha la implementación en ensamblador. Observemos que al comienzo del programa se inicializa SP con el símbolo _stack, definido con una directiva .equ. También se inicializa FP a cero, por si al llamar a la primera subrutina se mete el valor 0 en la base de su marco cuando salve FP, dejando una marca al depurador que indica que es la primera subrutina del árbol de llamadas.

A la hora de implementar las subrutinas anteriores hemos optado por almacenar todas sus variables en registros, usando como siempre los registros R4-R11 que son los que las subrutinas deben preservar. Sin embargo, rápidamente encontraremos casos en los que esto no sea suficiente, porque tengamos más variables que registros o porque tengamos que almacenar algún array. En estos casos es necesario utilizar variables locales, privadas para una determinada invocación de una subrutina, emplazadas en memoria. Estas se conocen como variables locales.

Código C	Ensamblador

```c
#define N 4
int A[N]={7,3,25,4};
int B[N];

void Recorre();
int Mayor();

void main(){
  Recorre (A, B, N);
}

void Recorre (int A[], int B[], int M){
  for(int j=0; j<M-1; j++){
    B[j] = Mayor(A[j],A[j+1]);
  }
}

int Mayor(int X, int Y){
  if(X>Y)
    return X;
  else
    return Y;
}
```

```
    .equ _stack, 0x8000
    .global start
    .equ N, 4
    .data
A:  .word 7,3,25,4
    .bss
(*\tt B*):  .space N*4

    .text
start:
    ldr sp, =_stack
    mov fp, #0
    ldr r0, =A
    ldr r1, =(*\tt B*)
    mov r2, #N
    bl Recorre
    b .

Recorre:
    push {r4-r8,lr}
    mov r4, r0       @ R4, A
    mov r5, r1       @ R5, B
    sub r6, r2, #1   @ R6, M-1
    mov r7, #0       @ R7, j
L0:
    cmp r7, r6
    bge L1
    ldr r0, [r4, r7, lsl #2]
    add r8, r7, #1
    ldr r1, [r4, r8, lsl #2]
    bl Mayor
    str r0, [r5, r7, lsl #2]
    add r7, r7, #1
    b L0
L1:
    pop {r4-r8,lr}
    mov pc, lr

Mayor:
    cmp r0, r1
    bgt L2
    mov r0, r1
L2:
    mov pc, lr
    .end
```

Figura 4.5: Ejemplo con subrutinas

4.5. Variables locales

Las variables almacenadas en las secciones *.data* o *.bss* que hemos usado hasta ahora se conocen como variables globales. Existen desde que se inicia el programa, y persisten en memoria hasta que el programa finaliza. Este almacenamiento no puede ser utilizado para las variables locales, que deben crearse en el momento de la llamada y su espacio debe quedar libre tras la llamada a la subrutina.

Consideremos como ejemplo la siguiente declaración de una función C:

```
int funA( int a1, int a2)
{
  int v1, v2;
  int v3 = 0;
  ...
```

Esta declaración nos dice que la función tendrá

- 2 variables locales `a1-a2`, que deben ser inicializadas con los 2 parámetros que deben ser pasados en la llamada. Por ejemplo, una posible llamada en C podría ser `a = funA(1,b);`, donde pasamos el valor 1 con los que se deberá inicializar la variable local `a1`, mientras que a `a2` le asignaremos como valor inicial lo que contenga la variable `b` en el momento de la llamada.
- 2 variables locales `v1-v2` sin valor inicial.
- 1 variable local `v3` con valor inicial 0.

Tendremos una versión privada de las variables locales para cada invocación de la función. Por ejemplo, supongamos que `funA` es una función recursiva, es decir, que se invoca a sí misma. Así, tendremos varias *instancias activas* de `funA` en un momento dado. Cada una de estas instancias debe tener su propia variable `a1` (y del resto), accesible sólo desde esa instancia.

Necesitamos por tanto un espacio privado para cada instancia o invocación de la función, donde podamos alojar estas variables, que debe quedar libre al finalizar su ejecución. Como es habitual tenemos dos opciones:

1. Almacenar las variables locales en los registros `R4-R11`. Dado que estos registros deben ser preservados, si la subrutina invoca a otra subrutina que quiera usar ese mismo registro para alguna de sus variables, no habrá problema. La subrutina invocada hará una copia de este registro en la pila que restaurará al finalizar. Por este motivo el estándar admite para estos registros los alias `v1-v8`.
2. Almacenar las variables locales en memoria, en el marco de activación de la subrutina. Así, cada instancia de una función recursiva tendrá su propia copia de la variable en su propio marco de activación. Al reservar espacio en el marco de activación de la subrutina deben satisfacerse las restricciones del estándar, siendo necesario que este espacio tenga un tamaño múltiplo de 4 bytes. Esta ubicación es la que facilita más la depuración.

Si optamos por emplazar las variables locales en memoria, la posición dentro del marco de activación de cada variable la decide el programador. En el cuerpo de la subrutina las

variables se direccionarán con desplazamientos fijos relativos a SP (o FP si lo usamos).Por lo tanto, a la hora de codificar una subrutina, lo primero que tendremos que hacer es decidir dónde ubicar cada una de sus variables:

- Si la variable decidimos ubicarla en un registro, debemos elegir un registro entre R4-R11.
- Si la variable decidimos ubicarla en memoria, debemos elegir un offset desde la cima del marco. Contaremos cuantas variables en memoria tenemos para codificar correctamente el prólogo.

Hay una excepción habitual a este emplazamiento de variables locales en memoria. Supongamos que una subrutina SubA invoca a una subrutina SubB que recibe 5 parámetros. Como sabemos, SubA debe pasar el quinto parámetro por memoria y debe reservar espacio en la cima de su propio marco de activación para almacenar este parámetro. En este caso, reservar otra vez espacio en el marco de SubB para almacenar la variable local correspondiente resulta un desperdicio de memoria. Por este motivo es habitual que SubB utilice el espacio reservado por SubA para almacenar su variable local.

Tras el prólogo, lo primero que debe hacer la subrutina es inicializar las variables locales que tengan valor inicial. Entre estas estarán las variable locales declaradas en el prototipo de la función, que tendrán que ser inicializadas con los valores pasados como parámetros a la subrutina.

El siguiente apartado nos muestra un ejemplo de programación de una subrutina.

4.5.1. Ejemplo

Tomemos por ejemplo la siguiente función C, que devuelve el mayor de dos números enteros:

```c
int Mayor(int X, int Y){
  int m;
  if(X>Y)
    m = X;
  else
    m = Y;
  return m;
}
```

La función tiene tres variable locales. Podemos decidir su almacenamiento de la siguiente manera:

- X,m a memoria, en las posiciones [sp] y [sp+4] respectivamente.
- Y a registro, en R4.

En este caso necesitaremos reservar 8 bytes adicionales en el marco para almacenar dos variables. El código ensamblador resultante sería:

```
1 Mayor:
2     push {r4}
3     sub sp, sp, #8
```

```
4       str r0, [sp]      @ Inicialización de X con el primer parámetro
5       mov r4, r1        @ Inicialización de Y con el segundo parámetro
6
7       @ if( X > Y )
8       ldr r0, [sp]      @ r0 ← X
9       cmp r0, r4
10      ble L0
11      @ then
12      ldr r0, [sp]      @ m = X;
13      str r0, [sp,#4]
14      b L1
15      @ else
16 L0:  str r4, [sp, #4]  @ m = Y;
17
18 L1:  @ return m
19      ldr r0, [sp, #4]  @ valor de retorno
20      add sp, sp, #8
21      pop {r4}
22      mov pc, lr
```

Como vemos hemos generado un código que es una traducción línea a línea del código C, siguiendo las directrices expuestas en la Sección 2.4. Este código visto en ensamblador parece redundante y poco eficiente. Por ejemplo la línea 8 carga en `r0` la variable local `X`, cuando acabamos de inicializar dicha variable con el valor de `r0` y no hemos modificado el registro entre medias. Sin embargo, como se discute en la siguiente sección, este tipo de codificación es el más sencillo de seguir, entender y depurar. Por ello se recomienda al alumno el uso de este estilo de programación en ensamblador mientras esté aprendiendo.

4.6. Compiladores y optimización

Los compiladores se estructuran generalmente en tres partes: *front end* o *parser*, *middle end*, y *back end*. La primera parte se encarga de comprobar que el código escrito es correcto sintácticamente y semánticamente y de traducirlo a una representación intermedia independiente del lenguaje. El *middle end* se encarga de analizar el código en su representación intermedia y realizar sobre él algunas optimizaciones, que pueden tener distintos objetivos, por ejemplo, reducir el tiempo de ejecución del código final, reducir la cantidad de memoria que utiliza o reducir el consumo energético en su ejecución. Finalmente el *back end* se encarga de generar el código máquina para la arquitectura destino (generalmente dan código ensamblador como salida y es el ensamblador el que produce el código máquina final).

Cuando no se realiza ninguna optimización del código se obtiene un código ensamblador que podíamos decir que es una traducción literal del código C: cuando se genera el código para una instrucción C no se tiene en cuenta lo que se ha hecho antes con ninguna de las instrucciones, ignorando así cualquier posible optimización. La Figura 4.6 muestra un ejemplo de una traducción de la función `main` de un sencillo programa C, sin optimizaciones (`-O0`) y con nivel 2 de optimización `-O2`. Como podemos ver, en la versión sin optimizar podemos identificar claramente los bloques de instrucciones ensamblador por las que se ha traducido cada sentencia C. Para cada una se sigue sistemáticamente el mismo proceso: se cargan en registros las variables que están a la derecha del igual (loads), se hace la operación correspondiente sobre registros, y se guarda el resultado en memoria (store) en la variable que aparece a la izquierda del igual.

Este tipo de código es necesario para poder hacer una depuración correcta a nivel de C. Si estamos depurando puede interesarnos parar al llegar a una sentencia C (en la primera instrucción ensamblador generada por su traducción), modificar las variables con el depurador, y ejecutar la siguiente instrucción C. Si el compilador no ha optimizado, los cambios que hayamos hecho tendrán efecto en la ejecución de la siguiente instrucción C. Sin embargo, si ha optimizado puede que no lea las variables porque asuma que su valor no ha cambiado desde que ejecutó algún bloque anterior. Por ejemplo esto pasa en el código optimizado de la Figura 4.6, donde la variable `i` dentro del bucle no se accede, sino que se usa un registro como contador. La variable `i` sólo se actualiza al final, con el valor de salida del bucle. Es decir, que si estamos depurando y modificamos `i` dentro del bucle, esta modificación no tendrá efecto, que no es lo que esperaríamos depurando a nivel de código C.

Otro problema típico, producido por la reorganización de instrucciones derivada de la optimización, es que podemos llegar a ver saltos extraños en el depurador. Por ejemplo, si el compilador ha desplazado hacia arriba la primera instrucción ensamblador generada por la traducción de una instrucción C, entonces cuando estemos en la instrucción C anterior y demos la orden al depurador de pasar a la siguiente instrucción C, nos parecerá que el flujo de ejecución vuelve hacia atrás, sin que haya ningún motivo aparente para ello. Esto no quiere decir que el programa esté mal, sino que estamos viendo un código C que ya no tiene una relación tan directa o lineal con el código máquina que realmente se ejecuta, y frecuentemente tendremos que depurar este tipo de códigos directamente en ensamblador.

Por motivos didácticos y prácticos (menos probabilidad de error, más fácil de modificar y depurar), es preferible que el alumno se acostumbre a escribir código ensamblador similar al mostrado en la parte izquierda de la Figura 4.6, a menos que se indique expresamente lo contrario. Este código se obtiene siguiendo las pautas presentadas en la Sección 2.4.

```c
#define N 10
int A[N];
int B[N];
int C,D,i;

int main(void) {
  for (i=0;i<N-1;i+=2) {
     A[i]   = B[i] + C;
     A[i+1] = B[i] - D;
  }
  return 0;
}
```

Compilación sin optimizaciones. gcc -O0

```
main:
    push {fp}
    add fp, sp, #0
    ldr r3, =i
    mov r2, #0
    str r2, [r3]
    b COND
LOOP:
    ldr r3, =i                    A[i]= B[i] + C
    ldr r2, [r3]
    ldr r3, =i
    ldr r1, [r3]
    ldr r3, =B
    ldr r1, [r3, r1, lsl #2]
    ldr r3, =C
    ldr r3, [r3]
    add r1, r1, r3
    ldr r3, =A
    str r1, [r3, r2, lsl #2]
    ldr r3, =i                    A[i+1]= B[i] - D
    ldr r3, [r3]
    add r2, r3, #1
    ldr r3, =i
    ldr r1, [r3]
    ldr r3, =B
    ldr r1, [r3, r1, lsl #2]
    ldr r3, =D
    ldr r3, [r3]
    sub r1, r1, r3
    ldr r3, =A
    str r1, [r3, r2, lsl #2]
    ldr r3, =i
    ldr r3, [r3]
    add r2, r3, #2
    ldr r3, =i
    str r2, [r3]
COND:
    ldr r3, =i
    ldr r3, [r3]
    cmp r3, #8
    ble LOOP
    mov r3, #0
    mov r0, r3
    add sp, fp, #0
    pop {fp}
    bx  lr
```

Compilación con optimizaciones. gcc -O2

```
main:
    ldr r3, =C
    push {r4, r5, r6}
    ldr ip, =A        @ ip <- A
    ldr r6, [r3]      @ r6 <- C
    ldr r3, =D
    ldr r4, =B
    ldr r5, [r3]      @ r5 <- D
    mov r2, ip        @ r2 <- &A[0]
    mov r3, #0        @ r3 es i*4
LOOP:
    ldr r1, [r4, r3] @ r1 <- B[i]
    add r0, r6, r1
    str r0, [ip, r3] @ A[i] <- r0
    add r3, r3, #8   @ 4*i <- 4*(i + 2)
    sub r1, r1, r5
    cmp r3, #40
    str r1, [r2, #4] @ *(r2+1) = r1
    add r2, r2, #8   @ salta 2 enteros
    bne LOOP
    ldr r3, =i
    mov r2, #10
    str r2, [r3]
    mov r0, #0
    pop {r4, r5, r6}
    bx  lr
```

Figura 4.6: C a ensamblador con gcc-4.7: optimizando (-O2) y sin optimizar (-O0)

Capítulo 5

Combinando C y Ensamblador

Hasta ahora los proyectos que hemos realizado tenían únicamente un fichero fuente. Sin embargo esto no es lo normal, sino que la mayor parte de los proyectos reales se componen de varios ficheros fuente. Es frecuente además que la mayor parte del proyecto se programe en un lenguaje de alto nivel como C/C++, y solamente se programen en ensamblador aquellas partes donde sea estrictamente necesario, bien por requisitos de eficiencia o bien porque necesitemos utilizar directamente algunas instrucciones especiales de la arquitectura. Para combinar código C con ensamblador tendremos que dividir necesariamente el programa en varios ficheros fuente, separando el código C del código ensamblador[1].

Cuando tenemos un proyecto con varios ficheros fuente, utilizaremos en alguno de ellos variables o subrutinas definidas en otro. Recordemos que la etapa de compilación se hace de forma independiente sobre cada fichero y es una etapa final de enlazado la que combina los ficheros objeto formando el ejecutable. Es en esta ésta última etapa se deben resolver todas las *referencias cruzadas* entre los ficheros objeto.

El objetivo de este capítulo es estudiar este mecanismo y su influencia en el código generado. Además presentaremos los principales tipos de datos compuestos utilizados por los lenguajes de alto nivel, como C, y veremos qué nuevas facilidades nos ofrece el entorno de desarrollo usado en el laboratorio para la depuración de códigos escritos en lenguaje C.

5.1. Símbolos: resolución y relocalización

El contenido de un fichero objeto es independiente del lenguaje en que fue escrito el fichero fuente. Es un fichero binario estructurado que contiene una lista de secciones con su contenido, y una serie de estructuras adicionales. Una de ellas es la *Tabla de Símbolos*, que, como su nombre indica, contiene información sobre los símbolos utilizados en el fichero fuente. Las *Tablas de Símbolos* de los ficheros objeto se utilizan durante el enlazado para *resolver* todas las referencias pendientes.

La tabla de símbolos de un fichero objeto en formato `elf` puede consultarse con el programa `nm`. Veamos lo que nos dice `nm` para un fichero objeto creado para este ejemplo:

```
> arm-none-eabi-nm  -SP -f sysv ejemplo.o
Symbols from ejemplo.o:
```

[1]Una alternativa es usar inlining de ensamblador para incrustar código ensamblador en un fichero C.

```
Name              Value    Class      Type      Size     Line   Section

globalA        |00000000|   D  |      OBJECT|00000002|      |.data
globalB        |        |   U  |      NOTYPE|        |      |*UND*
main           |00000000|   T  |        FUNC|00000054|      |.text
printf         |        |   U  |      NOTYPE|        |      |*UND*
```

Sin conocer el código fuente la información anterior nos dice que:

- Hay un símbolo `globalA`, que comienza en la entrada 0x0 de la sección de datos (`.data`) y de tamaño 0x2 bytes. Es decir, será una variable de tamaño media palabra.
- Hay otro símbolo `globalB` que no está definido (debemos importarlo). No sabemos para qué se va a usar en `ejemplo.o`, pero debe estar definido en otro fichero.
- Hay otro símbolo `main`, que comienza en la entrada 0x0 de la sección de código (`.text`) y ocupa 0x48 bytes. Es la función de entrada del programa C.
- Hay otro símbolo `printf`, que no está definido (debemos importarlo de la biblioteca estándar de C).

Todas las direcciones son desplazamientos desde el comienzo de la respectiva sección.

En el enlazado, cuando hay un símbolo no definido en uno de los ficheros a enlazar, se busca en las tablas de símbolos de los ficheros ya procesados para ver si alguno lo exporta. El enlazador calcula entonces la dirección de dicho símbolo y modifica el código generado por el ensamblador para que utilice la dirección correcta de dicho símbolo. Esto es lo que se conoce como resolución y relocalización de símbolos.

En los siguientes apartados vamos a ver cómo podemos especificar qué símbolos de nuestro programa serán globales, y por tanto irán en la tabla de símbolos para que puedan ser referenciados desde otro fichero de nuestro proyecto, y cómo podemos indicar en ese otro fichero que queremos utilizar un símbolo externo. Finalmente veremos cómo se produce la relocalización del código.

5.1.1. Exportación e importación de símbolos

El cuadro 6 presenta un ejemplo con dos ficheros C. El código de cada uno hace referencias a símbolos globales definidos en el otro. Los ficheros objeto correspondientes se enlazarán para formar un único ejecutable.

En C todas las variables globales y todas las funciones son por defecto símbolos globales exportados. Para utilizar una función definida en otro fichero debemos poner una declaración adelantada de la función. Por ejemplo, si queremos utilizar una función `FOO` que no recibe ni devuelve ningún parámetro, definida en otro fichero, debemos poner la siguiente declaración adelantada antes de su uso:

```
extern void FOO( void );
```

donde el modificador `extern` es opcional.

Con las variables globales sucede algo parecido, para utilizar una variable global definida en otro fichero tenemos que poner una especie de declaración adelantada, que indica su tipo. Por ejemplo, si queremos utilizar la variable global entera `aux` definida en otro fichero (o en el mismo pero más adelante) debemos poner la siguiente declaración antes de su uso:

Cuadro 6 Ejemplo de exportación de símbolos.

```
//(fichero fun.c)

extern int (var1);
static int (var2);
void (one)(void);

static void (two)(void)
{
  ...
  (var1++);
  ...
}

void (fun)(void)
{
  ...
  (var1) += 5;
  (var2) = (var1) + 1;
  ...
  (one)();
  (two)();
  ...
}
```

```
//(fichero main.c)

extern int (var1);
static int (var2);

void (one)(void);
void (fun)(void);

int main(void)
{
  ...
  (var1) = 1;
  ...
  (one)();
  (fun)();
  ...
}

int (var1);

void (one)(void)
{
  ...
  (var1)++;
  (var2) = (var1) - 1;
  ...
}
```

```
extern int aux;
```

Si no se pone el modificador `extern`, se trata como un símbolo `COMMON`. El enlazador resuelve todos los símbolos `COMMON` del mismo nombre por la misma dirección, reservando la memoria necesaria para el mayor de ellos. Por ejemplo, si tenemos dos declaraciones de una variable global `Nombre`, una como `char Nombre[10]` y otra como `char Nombre[20]`, el enlazador tratará ambas definiciones como el mismo símbolo `COMMON`, y utilizará los 20 bytes que necesita la segunda definición.

En C, si se quiere restringir la visibilidad de una función o variable global al fichero donde ha sido declarada es necesario utilizar el modificador `static` en su declaración. Esto hace que el compilador no la incluya en la tabla de símbolos del fichero objeto. De esta forma podremos tener dos o más variables globales con el mismo nombre, cada una restringida a un fichero distinto.

Por otro lado, en ensamblador los símbolos son por defecto locales, no visibles desde otro fichero. Si queremos hacerlos globales debemos exportarlos con la directiva `.global`. El símbolo `start` es especial e indica el punto (dirección) de entrada al programa, debe ser declarado siempre como global. De este modo además, se evita que pueda haber más de un símbolo `start` en varios ficheros fuente.

El caso contrario es cuando queramos hacer referencia a un símbolo definido en otro fichero. En ensamblador debemos declarar el símbolo mediante la directiva `.extern`. Por ejemplo:

```
    .extern FOO     @hacemos visible un símbolo externo
    .global start   @exportamos un símbolo local

start:
    bl FOO
    ...
```

Por lo tanto, para utilizar un símbolo exportado globalmente desde un fichero ensamblador dentro de un código C, debemos hacer una declaración adelantada del símbolo. Debemos tener en cuenta que el símbolo se asocia con la dirección del identificador. En el caso de que el identificador sea el nombre de una rutina esto corresponde a la dirección de comienzo de la misma. En C deberemos declarar una función con el mismo nombre que el símbolo externo. Además, deberemos declarar el tipo de todos los parámetros que recibirá la función y el valor que devuelve la misma, ya que esta información la necesita el compilador para generar correctamente el código de llamada a la función.

Por ejemplo, si queremos usar una rutina `FOO` que no devuelva nada y que tome dos parámetros de entrada enteros, deberemos emplear la siguiente declaración adelantada:

```
extern void FOO(int, int);
```

Si se trata de una variable, el símbolo corresponderá a una etiqueta que se habrá colocado justo delante de donde se haya ubicado la variable, por tanto es la dirección de la variable. Este símbolo puede importarse desde un fichero C declarando la variable como `extern`. De nuevo seremos responsables de indicar el tipo de la variable, ya que el compilador lo necesita para generar correctamente el código de acceso a la misma.

Por ejemplo, si el símbolo `var1` corresponde a una variable entera de tamaño media palabra, tendremos que poner en C la siguiente declaración adelantada:

```
extern short int var1;
```

Otro ejemplo sería el de un código ensamblador que reserve espacio en alguna sección memoria para almacenar una tabla y queramos acceder a la tabla desde un código C, ya sea para escribir o para leer. La tabla se marca en este caso con una etiqueta y se exporta la etiqueta con la directiva `.global`, por tanto el símbolo es la dirección del primer byte de la tabla. Para utilizar la tabla en C lo más conveniente es declarar un array con el mismo nombre y con el modificador `extern`.

5.1.2. Resolución de símbolos y relocalización

La Figura 5.1 ilustra el proceso de resolución de símbolos y relocalización con dos ficheros fuente: `init.asm`, codificado en ensamblador, y `main.c`, codificado en C. El primero declara un símbolo global `MIVAR`, que corresponde a una etiqueta de la sección `.data` en la que se ha reservado un espacio tamaño palabra y se ha inicializado con el valor `0x2`. En `main.c` se hace referencia a una variable externa con nombre `MIVAR`, declarada en C como entera (`int`).

Estos ficheros fuente son primero compilados, generando respectivamente los ficheros objeto `init.o` y `main.o`. La figura muestra en la parte superior el desensamblado de `main.o`. Para obtenerlo hemos seguido la siguiente secuencia de pasos:

```
> arm-none-eabi-gcc -O0 -c -o main.o main.c
> arm-none-eabi-objdump -D main.o
```

Se han marcado en rojo las instrucciones ensamblador generadas para la traducción de la instrucción C que accede a la variable `MIVAR`, marcada también en rojo en `main.c`. Como vemos es una traducción compleja. Lo primero que hay que observar es que la operación C es equivalente a:

```
MIVAR = MIVAR + 1:
```

Entonces, para poder realizar esta operación en ensamblador tendríamos primero que cargar el valor de `MIVAR` en un registro. El problema es que el compilador no sabe cuál es la dirección de `MIVAR`, puesto que es un símbolo no definido que será resuelto en tiempo de enlazado. ¿Cómo puede entonces gcc generar un código para cargar `MIVAR` en un registro si no conoce su dirección?

La solución a este problema es la siguiente. El compilador reserva espacio al final de la sección de código (`.text`) para una *tabla de literales*. Entonces genera código como si la dirección de `MIVAR` estuviese en una posición reservada de esa tabla. En el ejemplo, la entrada de la *tabla de literales* reservada para `MIVAR` está en la posición `0x44` de la sección de código de `main.o`, marcada también en rojo. Como la distancia relativa a esta posición desde cualquier otro punto de la sección `.text` no depende del enlazado, el compilador puede cargar el contenido de la tabla de literales con un `ldr` usando `PC` como registro base.

Como vemos, la primera instrucción sombreada en rojo carga la entrada `0x44` de la sección de texto en `r3`, es decir, tendremos en `r3` la dirección de `MIVAR`. La segunda instrucción carga en `r3` el valor de la variable y la siguiente instrucción le suma 1. Finalmente se vuelve a cargar en `r2` la dirección de `MIVAR` y la última instrucción guarda el resultado de la suma en `MIVAR`.

Pero, si nos fijamos bien en el cuadro, veremos que la entrada 0x44 de la sección `.text` está a 0, es decir, no contiene la dirección de `MIVAR`. ¿Era esto esperable? Por supuesto que sí, ya habíamos dicho que el compilador no conoce su dirección. El compilador pone esa entrada a 0 y añade al fichero objeto una entrada para `MIVAR` en la tabla de relocalización (*realloc*), como podemos ver con la herramienta objdump:

```
> arm-none-eabi-objdump -r main.o

main.o:     file format elf32-littlearm

RELOCATION RECORDS FOR [.text]:
OFFSET   TYPE              VALUE
00000040 R_ARM_V4BX        *ABS*
00000044 R_ARM_ABS32       MIVAR
```

Como podemos ver, hay una entrada de relocalización que indica al enlazador que debe escribir en la entrada `0x44` un entero sin signo de 32 bits con el valor absoluto del símbolo `MIVAR`.

La parte inferior de la Figura 5.1 muestra el desensamblado del ejecutable tras el enlazado. Como vemos, se ha ubicado la sección `.text` de main a partir de la dirección de memoria `0x00008010`. La entrada `0x44` corresponde ahora a la dirección `0x00008054` y contiene el valor `0x00008058`. Podemos ver también que esa es justo la dirección que corresponde al símbolo `MIVAR`, y contiene el valor `0x2` con el que se inicializó en `init.s`. Es decir, el enlazador ha resuelto el símbolo y lo ha escrito en la posición que le indicó el compilador, de modo que el código generado por este funcionará correctamente.

5.2. Arranque de un programa C

Un programa en lenguaje C está constituido por una o más funciones. Hay una función principal que constituye el punto de entrada al programa, llamada `main`. Esta función no es diferente de ninguna otra función, en el sentido de que construirá en la pila su marco de activación y al terminar lo deshará y retornará, copiando en PC lo que tuviese el registro LR al comienzo de la propia función.

Sin embargo para que la función main pueda comenzar, el sistema debe haber inicializado el registro de pila (`SP`) con la dirección base de la pila. Por este motivo (y otros que quedan fuera del alcance de este documento) el programa no comienza realmente en la función `main` sino con un código de arranque, que en el caso del toolchain de GNU se denomina *C Real Time 0* o *crt0*. Este código está definido en el contexto de un sistema operativo. Sin embargo, en nuestro caso no disponemos de sistema operativo y deberemos proporcionar nosotros el código de arranque, declarando el símbolo `start` en su primera instrucción. El cuadro 7 presenta el código de arranque que utilizaremos en esta práctica.

5.3. Tipos compuestos

Los lenguajes de alto nivel como C ofrecen generalmente tipos de datos compuestos, como arrays, estructuras y uniones. Vamos a ver cómo es la implementación de bajo nivel de estos tipos de datos, de forma que podamos acceder a ellos en lenguaje ensamblador.

5.3.1. Arrays

Un array es una estructura homogénea, compuesta por varios elementos, todos del mismo tipo y almacenados consecutivamente en memoria. En C por ejemplo, cada elemento puede ser accedido por el nombre de la variable del array seguido de uno o más subíndices encerrados entre corchetes.

```
Disassembly of section .text:

00000000 <main>:
   0:   e52db004        push    {fp}            ; (str fp, [sp, #-4]!)
   4:   e28db000        add     fp, sp, #0
   8:   e24dd00c        sub     sp, sp, #12
   c:   e59f3030        ldr     r3, [pc, #48]   ; 44 <main+0x44>
  10:   e5933000        ldr     r3, [r3]
  14:   e2833001        add     r3, r3, #1
  18:   e59f2024        ldr     r2, [pc, #36]   ; 44 <main+0x44>
  1c:   e5823000        str     r3, [r2]
  20:   e59f301c        ldr     r3, [pc, #28]   ; 44 <main+0x44>
  24:   e5933000        ldr     r3, [r3]
  28:   e2833001        add     r3, r3, #1
  2c:   e50b3008        str     r3, [fp, #-8]
  30:   e51b3008        ldr     r3, [fp, #-8]
  34:   e1a00003        mov     r0, r3
  38:   e24bd000        sub     sp, fp, #0
  3c:   e49db004        pop     {fp}            ; (ldr fp, [sp], #4)
  40:   e12fff1e        bx      lr
  44:   00000000        andeq   r0, r0, r0
```


main.c: importa `MIVAR`

```
extern int MIVAR;

int main(void)
{
    int i;

    MIVAR++;
    i = 1 + MIVAR;

    return i;
}
```

init.s: exporta `MIVAR`

```
.data
.global MIVAR
MIVAR:  .word 0x02

.text
...    @ Omitido
```

```
Disassembly of section .text:

00008000 <_start>:
    8000:       e3a0d902        mov     sp, #32768      ; 0x8000
    8004:       e3a0b000        mov     fp, #0
    8008:       eb000000        bl      8010 <main>

0000800c <End>:
    800c:       eafffffe        b       800c <End>

00008010 <main>:
    8010:       e52db004        push    {fp}            ; (str fp, [sp, #-4]!)
    8014:       e28db000        add     fp, sp, #0
    8018:       e24dd00c        sub     sp, sp, #12
    801c:       e59f3030        ldr     r3, [pc, #48]   ; 8054 <main+0x44>
    8020:       e5933000        ldr     r3, [r3]
    8024:       e2833001        add     r3, r3, #1
    8028:       e59f2024        ldr     r2, [pc, #36]   ; 8054 <main+0x44>
    802c:       e5823000        str     r3, [r2]
    8030:       e59f301c        ldr     r3, [pc, #28]   ; 8054 <main+0x44>
    8034:       e5933000        ldr     r3, [r3]
    8038:       e2833001        add     r3, r3, #1
    803c:       e50b3008        str     r3, [fp, #-8]
    8040:       e51b3008        ldr     r3, [fp, #-8]
    8044:       e1a00003        mov     r0, r3
    8048:       e24bd000        sub     sp, fp, #0
    804c:       e49db004        pop     {fp}            ; (ldr fp, [sp], #4)
    8050:       e12fff1e        bx      lr
    8054:       00008058        andeq   r8, r0, r8, asr r0

Disassembly of section .data:

00008058 <MIVAR>:
    8058:       00000002        andeq   r0, r0, r2
```

Figura 5.1: Ejemplo de resolución de símbolos.

Cuadro 7 Ejemplo de rutina de inicialización.

```
        .global start
        .equ STACK, 0x8000 @ valor inicial del puntero de pila
        .extern main

start:
        ldr sp,  =STACK  @ inicializaión de la pila
        mov fp, #0       @ inicialización del frame pointer (opcional)

        bl main
End:    b .
        .end
```

Si declaramos una variable global cadena como:

```c
char cadena[] = "hola mundo\n";
```

el compilador reservará doce bytes consecutivos en la sección .data, asignándoles los valores como indica la figura 5.2. Como vemos las cadenas en C se terminan con un byte a 0 y por eso la cadena ocupa 12 bytes. En este código el nombre del array, cadena, hace referencia a la dirección donde se almacena el primer elemento, en este caso la dirección del byte/caracter `h` (i.e. `0x0c0002B8`).

Memoria

	Dirección				
	0x0C0002B4				
cadena:	0x0C0002B8	h	o	l	a
	0x0C0002BC		m	u	n
	0x0C0002C0	d	o	\n	0
	0x0C0002C4				
	0x0C0002C8				

Figura 5.2: Almacenamiento de un array de caracteres en memoria.

La dirección de comienzo de un array debe ser una dirección que satisfaga las restricciones de alineamiento para el tipo de datos almacenados en el array.

5.3.2. Estructuras

Una estructura es una agrupación de variables de cualquier tipo a la que se le da un nombre. Por ejemplo el siguiente fragmento de código C:

```c
struct mistruct {
    char primero;
    short int segundo;
    int tercero;
};
```

```
struct mistruct rec;
```

define un tipo de estructura de nombre **struct** mistruct y una variable rec de este tipo. La estructura tiene tres campos, de nombres: primero, segundo y tercero cada uno de un tipo distinto.

Al igual que sucedía en el caso del array, el compilador debe colocar los campos en posiciones de memoria adecuadas, de forma que los accesos a cada uno de los campos no violen las restricciones de alineamiento. Es muy probable que el compilador se vea en la necesidad de *dejar huecos* entre unos campos y otros con el fin de respetar estas restricciones. Por ejemplo, el emplazamiento en memoria de la estructura de nuestro ejemplo sería similar al ilustrado en la figura 5.3.

Hueco dejado por el compilador
para alinear el segundo dato

Memoria

	Dirección				
	0x0C0002DC				
rec:	0x0C0002E0	61		1C	00
	0x0C0002E4	0A	00	00	00
	0x0C0002E8				
	0x0C0002EC				

Figura 5.3: Almacenamiento de la estructura *rec*, con los valores de los campos primero, segundo y tercero a `0x61`, `0x1C` y `0x0A` respectivamente.

5.3.3. Uniones

Una unión es un tipo de dato que, como la estructura, tiene varios campos, potencialmente de distinto tipo, pero que sin embargo utiliza una región de memoria común para almacenarlos. Dicho de otro modo, la unión hace referencia a una región de memoria que puede ser accedida de distinta manera en función de los campos que tenga. La cantidad de memoria necesaria para almacenar la unión coincide con la del campo de mayor tamaño y se emplaza en una dirección que satisfaga las restricciones de alineamiento de todos ellos. Por ejemplo el siguiente fragmento de código C:

```
union miunion {
    char primero;
    short int segundo;
    int tercero;
};

union miunion un;
```

declara una variable `un` de tipo union miunion con los mismos campos que la estructura de la sección 5.3.2. Sin embargo su huella de memoria (*memory footprint*) es muy distinta, como ilustra la figura 5.4.

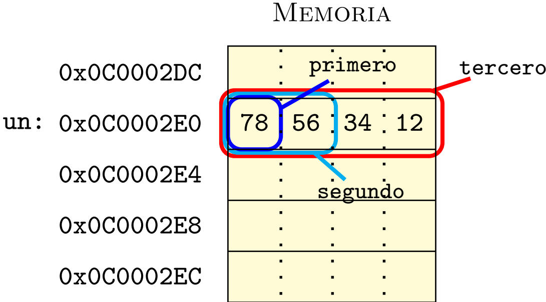

Figura 5.4: Almacenamiento de la union `un`, con los campos primero, segundo y tercero. Un acceso al campo primero da como resultado el byte `0x78`, es decir el carácter `x`. Un acceso al campo segundo da como resultado la media palabra `0x5678` (asumiendo configuración *little endian*), es decir el entero `22136`. Finalmente un acceso al campo tercero nos da como resultado la palabra `0x12345678` (de nuevo *little endian*), es decir el entero `305419896`.

Cuando se accede al campo `primero` de la unión, se accede al byte en la dirección `0x0C0002E0`, mientras que si se accede al campo `segundo` se realiza un acceso de tamaño media palabra a partir de la dirección `0x0C0002E0` y finalmente, un acceso al campo tercero implica un acceso de tamaño palabra a partir de la dirección `0x0C0002E0`.

5.4. Eclipse: depurando un programa C

Eclipse ofrece ciertas facilidades para la depuración de un programa C que conviene conocer. Como ejemplo, la Figura 5.5 ilustra una sesión de depuración de un programa C. En la parte superior derecha podemos ver el visor de variables. Para abrirlo basta con seleccionar `Widow`→`Show View`→`Variables`. Al principio sólo aparecerán las variables locales de la función que está ejecutándose en ese momento (mejor dicho, del marco de activación que tengamos seleccionado en la parte superior izquierda, donde aparece el árbol de llamadas actual). Podemos añadir las variables locales pinchando en el visor con el botón derecho del ratón y seleccionando `Add Global Variables...` del menú desplegado.

A veces resulta conveniente ver las variables en memoria, sobre todo en el caso de los arrays porque veremos varias posiciones del array cómodamente. Para esto debemos abrir el visor `Memory Monitor`, ilustrado en la parte inferior de la figura. En el ejemplo se ha creado un monitor en la dirección `0x2456C`, y se han mostrado dos representaciones del mismo, una de bytes en representación hexadecimal y otra de bytes representados en decimal (enteros). El ancho byte se ha seleccionado en la opción `Format...` del menú desplegable obtenido al pulsar con el botón derecho del ratón sobre el monitor de memoria. Para ver las dos representaciones simultaneamente se ha pulsado el botón . La pestaña `New Renderings` permite seleccionar distintos formatos de representación, de este modo seleccionamos la opción `Unsigned Integer`.

Finalmente, en ocasiones nos resultará conveniente el visor `Expressions`. Este visor, que se abrirá en la parte superior derecha junto al de variables, nos permite introducir cualquier expresión C válida y nos dará su valor. Por ejemplo, si introducimos la expresión `&num`, siendo `num` una variable de nuestro programa, veremos en el visor la dirección de memoria de `num`.

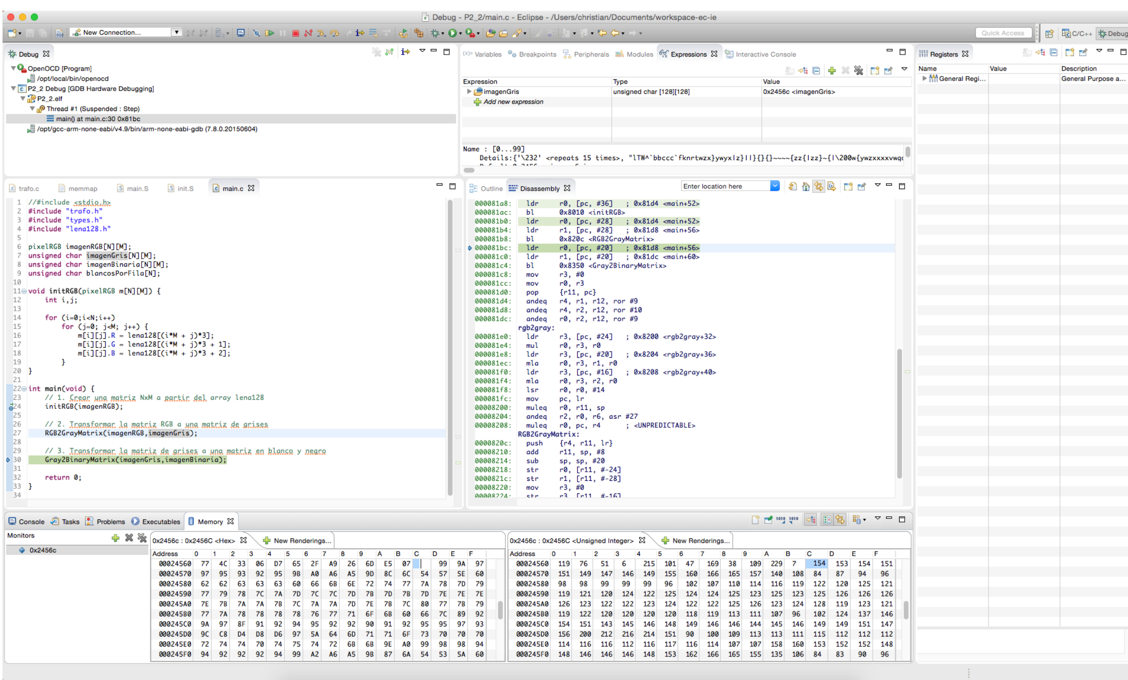

Figura 5.5: Depurando un programa C: variables y visor de memoria.

Capítulo 6

Descubriendo el sistema de Entrada/Salida

En el Capítulo 1 describimos el computador como un sistema compuesto de varios subsistemas interconectados: la cpu, la memoria y el sistema de entrada salida. Nos centramos entonces en el funcionamiento básico de la CPU. En este capítulo estudiaremos los principios y los mecanismos básicos del sistema de entrada salida.

Entender el sistema de entrada salida es complejo, ya que intervienen varios componentes en el mismo, que en el caso del sistema utilizado en el laboratorio son:

- El procesador ARM1176JZF-S
- El System on Chip BCM2835 (SoC) en el que está integrado
- La placa sobre la que está soldado el chip, y los componentes que quedan conectados a los pines del chip, es decir, la Raspberry Pi 1.
- La placa de expansión conectada a la placa principal, la Gertboard.

En las siguientes secciones introduciremos los principios generales del sistema de entrada salida, y analizaremos sobre el sistema de laboratorio la influencia de cada uno de sus componentes. Asimismo, nos familiarizaremos con los mecanismos software básicos para que nuestro programa pueda interaccionar con algunos dispositivos externos sencillos.

6.1. Fundamentos del sistema de E/S

Por lo que hemos visto hasta ahora, el computador lo único que sabe hacer es leer instrucciones de la memoria y ejecutarlas en la cpu, instrucciones que:

- Realizan operaciones aritméticas básicas sobre registros o
- Cargan datos de memoria sobre registros o
- Llevan datos de registros a memoria

Entonces, por ejemplo, ¿cómo puede encender o apagar un led un computador? La solución se ilustra en la Figura 6.1, interponemos un HW (controlador) entre la cpu y el dispositivo a controlar (led), encargado de *interpretar las operaciones* que la cpu quiera realizar sobre el dispositivo y actuar correctamente sobre el.

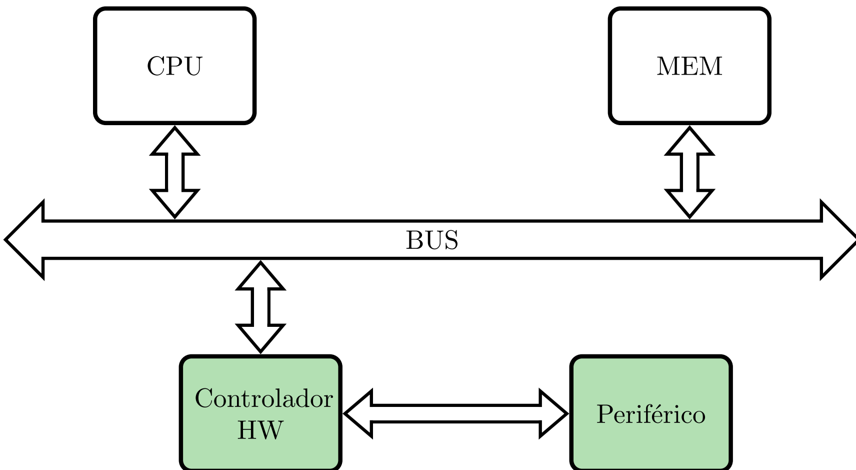

Figura 6.1: Controlador HW, interfaz entre la cpu y el dispositivo.

En el caso del led, por ejemplo, el circuito controlador podría ser similar al mostrado en la Figura 6.2. Para encender el led, el computador debe escribir un 1 en el biestable, lo que hará que el led se polarice en directa y deje pasar la corriente a su través (la resistencia limita la cantidad de corriente que pasa). Si el computador escribe un 0 en el biestable, la corriente dejará de pasar por el led que dejará de lucir.

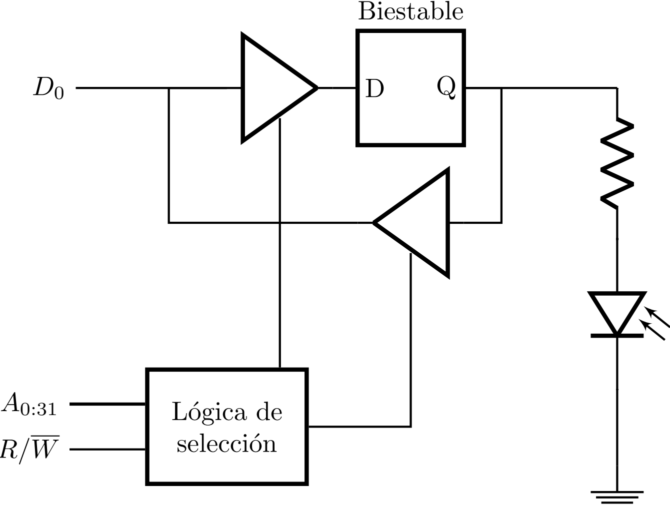

Figura 6.2: Controlador HW para encender y apagar un led.

¿Pero cómo escribe el computador en el biestable? La Figura 6.2 ilustra un posible mecanismo. El módulo de lógica de selección detecta, a partir de los bits de dirección del bus ($A_{0:31}$) que el computador quiere acceder a la dirección asignada a este biestable, y activa la puerta de paso de lectura o escritura según lo que indique la señal $R/\overline{W}$ del bus. Así el programa podrá escribir en el biestable con una instrucción store corriente sobre la dirección asignada al dispositivo. Asimismo, leería del biestable con una operación de load corriente. Este sistema se conoce como entrada/salida mapeada o ubicada en memoria.

Alternativamente podríamos añadir al bus una señal que permita distinguir si es una

operación con memoria o una operación con algún registro de entrada salida, con lo que tendríamos separado el espacio de direcciones de la memoria y de la entrada/salida. En ese caso tendríamos además que añadir instrucciones especiales al repertorio que realicen operaciones de lectura y/o escritura sobre el espacio de direcciones de dispositivos. Esta otra alternativa se conoce como entrada/salida aislada.

En definitiva, el controlador HW es un circuito que tiene un interfaz digital con el bus del procesador, que permite al procesador leer y escribir en registros propios del controlador. El contenido de cada uno de esos registros tiene un significado especial, específico para ese controlador. Además, dispone de un circuito analógico y/o digital que actual sobre el dispositivo siguiendo las instrucciones dadas por el SW a través de los registros del controlador. Es por tanto un interfaz HW/SW entre el dispositivo y el software que interacciona sobre el dispositivo.

En el contexto de un sistema operativo el acceso al HW está restringido al sistema. El código que interacciona con el dispositivo es parte del sistema y se conoce como *driver*. En el sistema de laboratorio no tenemos sistema operativo, y será nuestro programa el que tenga que interaccionar directamente con los controladores de los dispositivos. El significado o función asociada a cada registro suele venir documentado por el fabricante. Tendremos por tanto que familiarizarnos con dicha documentación a la hora de trabajar con cualquier dispositivo.

6.2. Sistema de E/S del BCM2835

La arquitectura ARM usa E/S mapeada en memoria, por lo que acceder a los dispositivos consiste en ejecutar instrucciones de lectura/escritura (`ldr`/`str`) a las direcciones adecuadas.

El ARM1176JZF-S tiene soporte para memoria virtual. Si lo activamos las direcciones virtuales generadas por el procesador son traducidas a direcciones físicas por la *Memory Management Unit* (MMU) interna del ARM. Además, el BCM2835 añade otra MMU para traducir estas direcciones físicas a un nuevo mapa de memoria de *direcciones de bus*. Sin embargo, en nuestro laboratorio no trabajaremos con la memoria virtual activada, por lo que no habrá traducción en la MMU interna del procesador.

La Figura 6.3 muestra una simplificación del mapa de memoria del BCM2835 cuando no se usa memoria virtual. Como en el laboratorio usamos la revisión 2.0 del Modelo B de la Raspberry Pi 1, disponemos de 512MB de memoria RAM. Por lo tanto podemos configurar nuestro código para que resida en cualquier dirección del rango 0x00000000 - 0x20000000.

A partir la dirección 0x20000000 comienza el rango de direcciones asignadas a los periféricos, que se extiende hasta la dirección 0x20FFFFFF. Todos los controladores HW de los periféricos de la Raspberry Pi (temporizadores, controladores I2C, GPIO, SPI, UART, PWM...) tienen sus registros ubicados en ese rango de direcciones.

Sin embargo, las direcciones físicas generadas por la cpu serán traducidas por la MMU del BCM2835, como ilustra la parte izquierda de la Figura 6.3. Esto dificulta ligeramente la programación de dispositivos de forma directa, pues la documentación de Broadcom ([bcm]) usa las *direcciones de bus* para documentar los diferentes dispositivos.

Como vemos en la Figura 6.3, la MMU del BCM2835 traduce las direcciones del siguiente modo:

- El rango físico de 0x00000000-0x20000000 se mapea al rango 0xC0000000-0xE0000000 en *direcciones de bus*

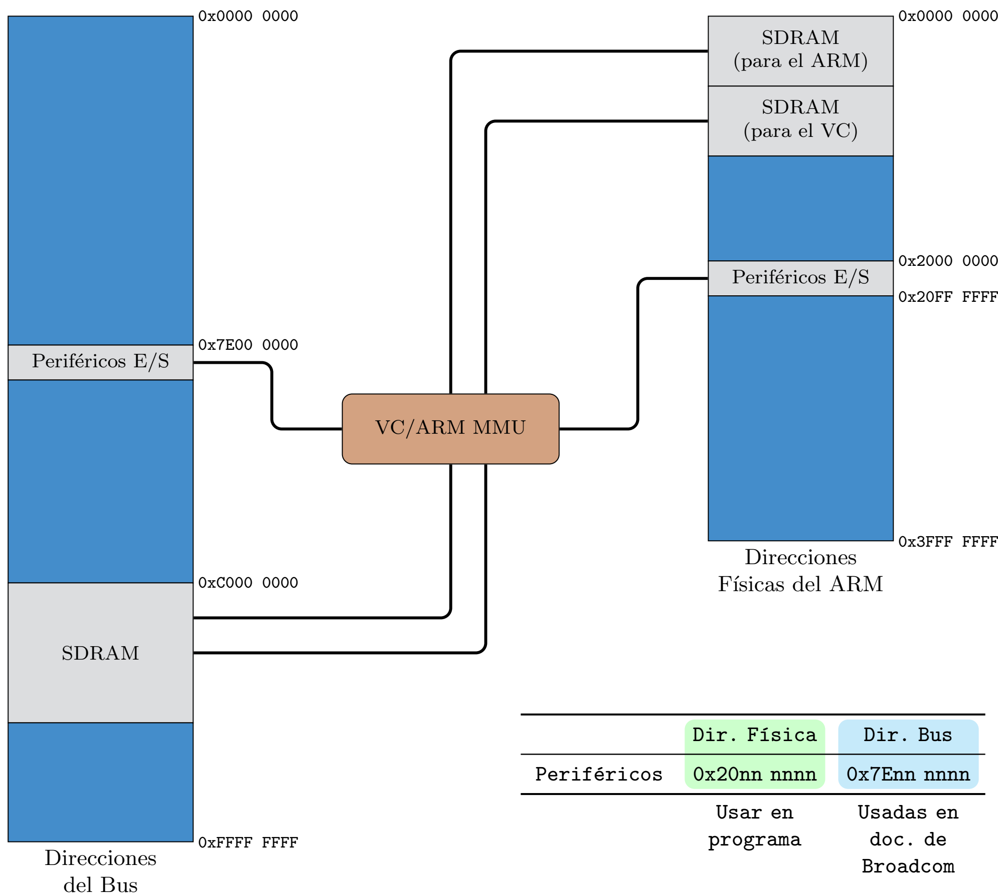

Figura 6.3: Mapa de memoria simplificado del SoC BCM2835 (consultar [bcm].

- El rango físico correspondiente a los controladores de los dispositivos de E/S (0x20000000 en adelante) se mapea al rango 0x7E000000 en adelante en *direcciones de bus*

A efectos prácticos, esto significa que una dirección de E/S documentada por Broadcom como 0x7Ennnnnn habrá que sustituirla por la 0x20nnnnnn cuando diseñemos el programa que interactúe con el dispositivo en cuestión. Hay una única excepción: si se utiliza el controlador de DMA, las direcciones que especifiquemos al controlador sí deberán ser direcciones de bus y no físicas de ARM.

El siguiente apartado describe el controlador de pines de E/S del BCM2835, que nos permitirá controlar dispositivos externos sencillos, como pulsadores y leds.

6.2.1. Controlador de pines de E/S

El SoC BCM2835 dispone de 54 pines de uso general (*General Purpose Input/Output*). Para cada uno de ellos el controlador dispone de una electrónica, ilustrada en la Figura 6.4, que permite utilizar el pin para distintas funciones.

La función de cada pin se configura a través de los registros señalados en la Tabla 6.1, que formarían parte del bloque *Funtion Registers* de la Figura 6.4. Cada uno de esos registros almacena la configuración de 10 pines GPIO. Cada pin se configura con un campo de tres bits del registro correspondiente de la siguiente forma:

- `000` El pin se configura como entrada. En este caso podremos ver si la tensión en el pin corresponde a un 1 o a un 0 lógico.
- `001` El pin se configura como salida. En este caso podemos escribir un valor 1 ó 0 y esto pondrá el pin a Vdd o Gnd respectivamente.
- Otra configuración: función alternativa. Generalmente esto implica que alguna señal interna del chip se hace accesible al exterior a través del pin (cosultar [bcm]).

Registro	Dirección (ARM)	Pines GPIO
GPFSEL0	0x20200000	GPIO0 - GPIO9
GPFSEL1	0x20200004	GPIO10 - GPIO19
GPFSEL2	0x20200008	GPIO20 - GPIO29
GPFSEL3	0x2020000C	GPIO30 - GPIO39
GPFSEL4	0x20200010	GPIO40 - GPIO49
GPFSEL5	0x20200014	GPIO50 - GPIO53

Tabla 6.1: Configuración de pines de GPIO a través de registros GPFSELn (capítulo 6 de [bcm]).

Por ejemplo, para configurar el pin GPIO0 como entrada debemos escribir `000` en los bits 2-0 del registro`GPFSEL0`. Asimismo, para configurar el pin `GPIO17` como salida debemos escribir `001` en los bits 23-21 del registro `GPFSEL1`.

Si queremos escribir en un pin configurado como salida, deberemos escribir un 1 en la posición adecuada de uno de los registros indicados en la Tabla 6.2. Los registros `GPSETn` se utilizan para escribir un 1 y los registros `GPCLRn` para escribir un 0. La posición se escoge

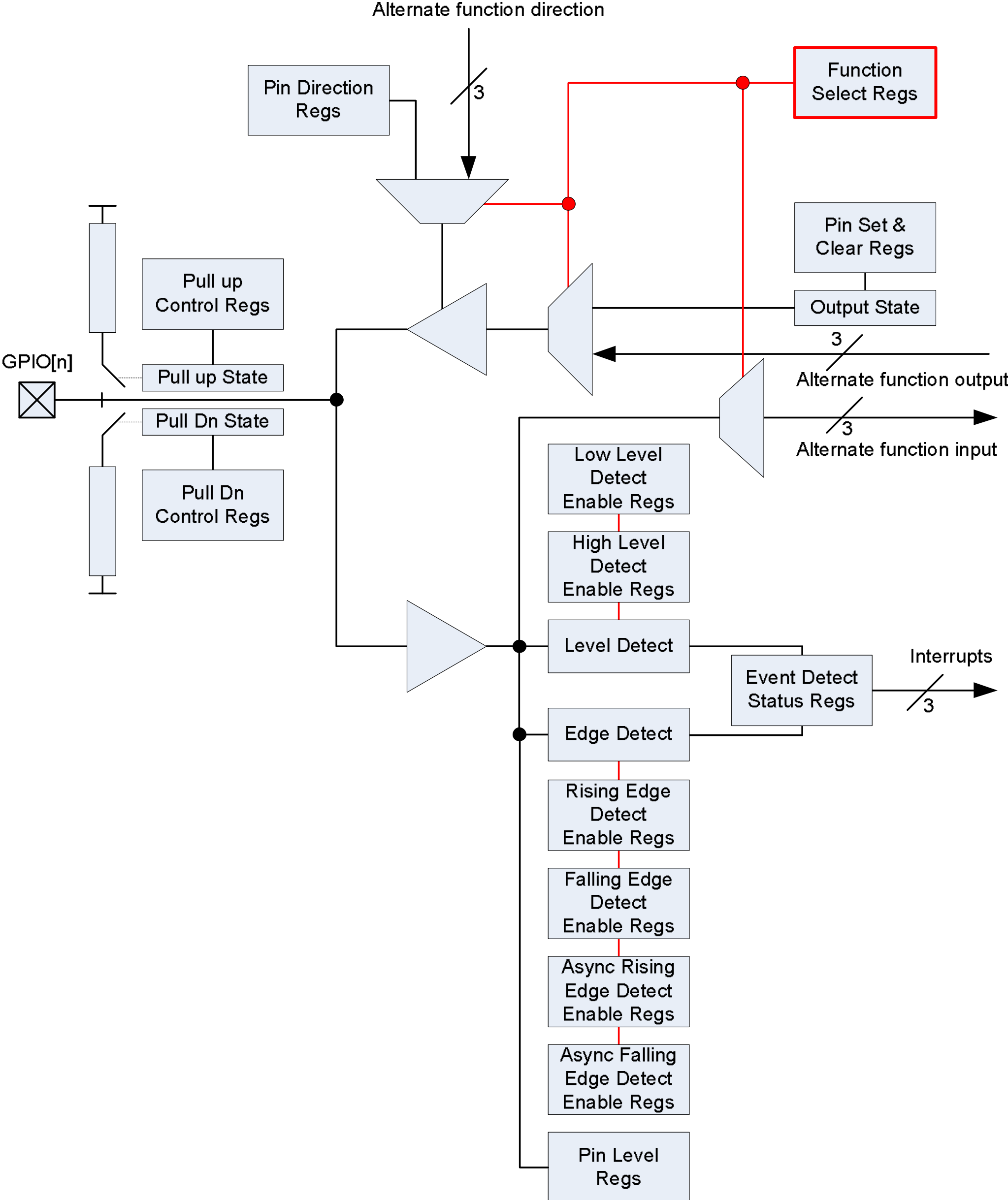

Figura 6.4: Esquema de bloques del controlador HW para los pines de E/S

en función del pin que queramos poner a 1 ó 0. Por ejemplo, para poner a 1 el pin `GPIO17` debemos escribir un 1 en el bit 17 del registro `GPSET0`. Asimismo, para poner a 0 el pin `GPIO40` debemos escribir un 1 en el bit 8 del registro `GPCLR1`.

Valor	Pin	Registro	Dirección (ARM)
1	0-31	GPFSET0	0x2020001C
1	32-53	GPFSET1	0x20200020
0	0-31	GPFCLR0	0x20200028
0	32-53	GPFCLR1	0x2020002C

Tabla 6.2: Relación de registros a utilizar para escribir un valor 1 ó 0 en un pin GPIO configurado como pin de salida

Si por el contrario queremos conocer el valor lógico de un pin configurado como entrada, basta con realizar una lectura del bit correspondiente en el registro indicado en la Tabla 6.3. Por ejemplo, si queremos conocer el valor del GPIO23, tendríamos que consultar el bit 23 del registro `GPFLEV0`. En cambio si queremos conocer el valor del GPIO40, tendríamos que consultar el bit 8 del registro `GPFLEV1`.

Pin	Registro	Dirección (ARM)
0-31	GPFLEV0	0x20200034
32-53	GPFLEV1	0x20200038

Tabla 6.3: Relación de registros a utilizar para leer el valor de un pin GPIO configurado como entrada

Finalmente, el controlador GPIO permite conectar a cada pin un circuito de *pull-up* o *pull-down*. Esto es interesante cuando el pin se configura como pin de entrada para leer el estado de alguna electrónica conectada a dicho pin, que sólo puede forzar el pin a uno de los dos valores lógicos. El ejemplo típico es una línea en colector o drenador abierto, como en el caso del pulsador de la Figura 6.5. El pulsador permite conectar el pin a tierra cuando se pulsa. Sin embargo, el pin queda *al aire* cuando el pulsador no está pulsado, con lo que en el pin del GPIO leeríamos un valor indefinido. En este caso activar el circuito de *pull-up* (representado en la figura como un interruptor) haría que el pin se polarizase a Vdd cuando el pulsador no estuviese pulsado.

Para activar el circuito de *pull-up* o *pull-down* de un pin del GPIO usaremos los registros de la tabla Tabla 6.4 según indicamos a continuación:

1. Escribir en el registro `GPPUD`:
   - `01` para activar el circuito de `pull-down`
   - `10` para activar el circuito de `pull-up`
2. Esperar al menos 150 ciclos
3. Escribir en los registros *GPPUDCLK0/1* para seleccionar los pines a los que queramos activar el circuito seleccionado. Por ejemplo, si escribimos el valor 0x00000101 en el

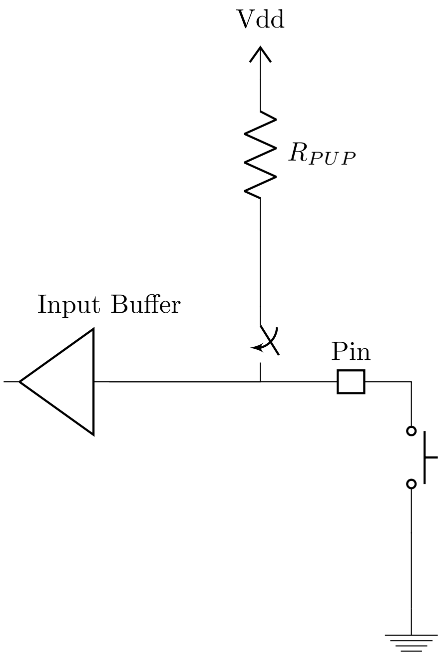

Figura 6.5: Ejemplo de uso de pin de entrada con circuito de *pull-up* activado.

registro *GPPUDCLK0* configuraremos los pines GPIO0 y GPIO8 a *pull-up* o *pull-down* según lo indicado en el paso 1.

4. Esperar al menos 150 ciclos

5. Escribir `00` en `GPPUD`

6. Escribir 0x0 en *GPPUDCLK0/1.*

Registro	Dirección (ARM)
GPPUD	0x20200094
GPPUDCLK0	0x20200098
GPPUDCLK1	0x2020009C

Tabla 6.4: Registros a utilizar para activar los circuitos de *pull-up* y *pull-down* de los pines del GPIO.

6.3. Raspberry Pi 1 y Gertboard

La Figura 6.6 representa la placa Raspberry Pi 1, revisión B/B+. Como vemos, incluye principalmente un regulador, un controlador Ethernet y varios conectores, todos ellos conectados con algunos pines del BCM2835, pieza central de la placa. Además incorpora un array de pines (conector superior) que expone 26 de los pines del GPIO (GPIO2-GPIO27) para poder conectar al chip cualquier otro dispositivo externo.

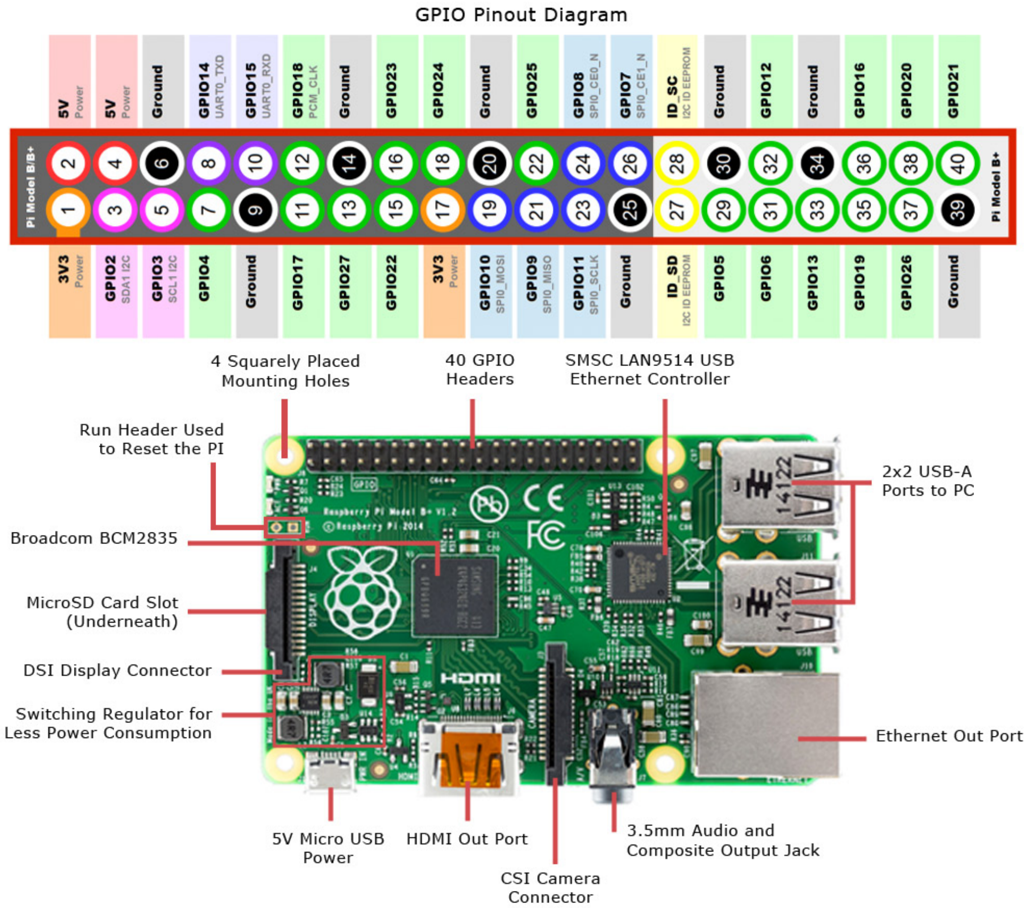

Figura 6.6: Pines expuestos en el conector de expansión de la Raspberry Pi 1.

En nuestro laboratorio usamos este array de pines para conectar el BCM2835 a una placa de expansión diseñada por Gert van Loo llamada Gertboard[ger]. La figura 6.7 muestra un diagrama de bloques de dicha placa y da una idea de su conexión con la Raspberry. La placa incorpora algunos dispositivos sencillos que podemos conectar, por medio de cables y a través de un array de pines incorporado en la propia placa, a los pines de E/S del GPIO. Los componentes de la Gertboard son:

- 12 drivers (circuitos) de E/S, cada uno conectado a un LED que estará encendido si la entrada está a '1' lógico y apagado en caso contrario.
- 3 pulsadores.
- 6 puertos conectados a un array de transistores *Darlington* en colector abierto, que toleran hasta 50V y 0.5A.
- Un controlador de motor (48V, 4A)
- Un conector de 28 pines para incorporar un microcontrolador ATmega
- Un conversor Digital-Analógico (8, 19 o 12 bits).
- Un conversor Analógioc-Digital de 10 bits

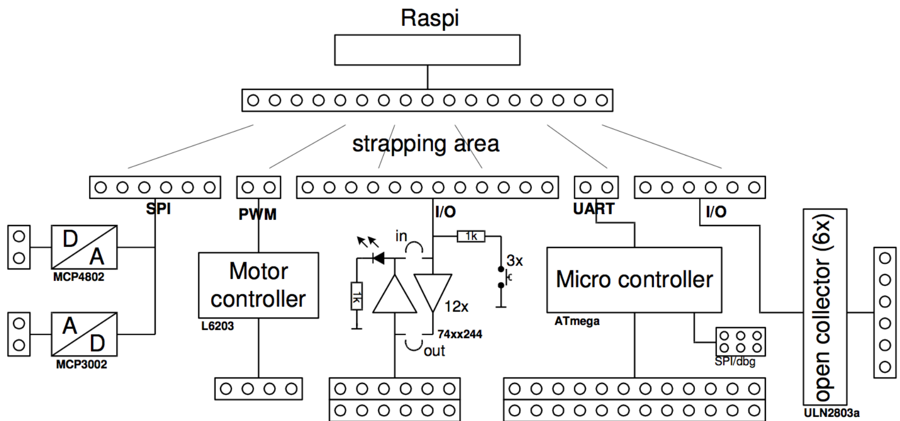

Figura 6.7: Diagrama de bloques de la placa Gertboard

.

El voltaje de referencia (el que se considera un 1 lógico) en la Gertboard es 3.3V, si bien es posible conseguir 5V a partir de la Raspberry Pi. Para enviar la alimentación correcta (3.3V) a todos los componentes de la placa es **imprescindible instalar un *jumper* sobre los pines marcados como J7.** Si no se coloca este *jumper*, tal y como se indica en la Figura 6.8, ningún componente de la placa funcionará adecuadamente.

La conexión entre la Gertboard y la Raspberry Pi se hace a través del conector J1, de 26 pines. El conector J2 expone estos 26 pines para su conexión a los componentes de la Gertboard, estando identificados en la serigrafía de la placa cada uno de los pines. La numeración indicada (GP25, GP24...) coincide, salvo alguna excepción[1], con los nombres de

[1]Los pines GP0 y GP1 se corresponden con GPIO2 y GPIO3 respectivamente y el pin GP21 que se corresponde con el GPIO27

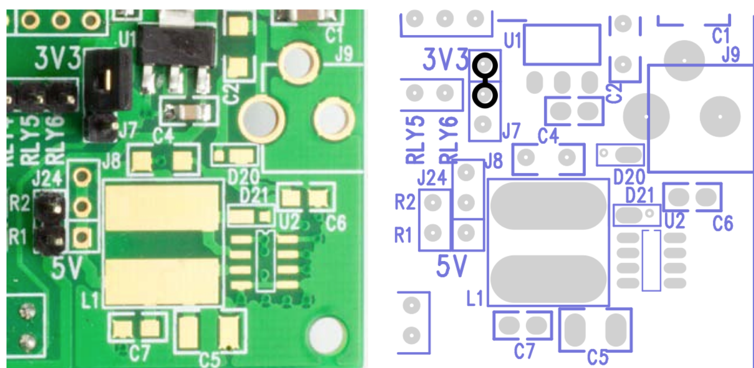

Figura 6.8: Instalación de jumper en pines J7

.

señales usados en el SoC BCM2835: la señal GPIOn de la documentación del SoC BCM2835 se corresponde con el pin etiquetado GPn en el conector J2 de la Gertboard.

En el siguiente apartado vamos a ver cómo debemos realizar las conexión del jumper J2 con los drivers de E/S de la Gertboard, de modo que podamos encender y apagar los leds y detectar pulsaciones de los pulsadores desde los pines de la Raspberry Pi.

6.3.1. Uso de LEDs y pulsadores

Como se ha mencionado previamente la placa Gertboard dispone de 12 puertos/drivers de E/S. La Figura 6.9 muestra el diagrama del circuito de los puertos 4 a 12. Para utilizar uno de estos puertos como salida es preciso conectar el *jumper* del buffer de salida del puerto (observar la serigrafía *output* en la placa). Para usarlo como entrada es preciso conectar el *jumper* del buffer de entrada (notar la serigrafía *input* en la placa).

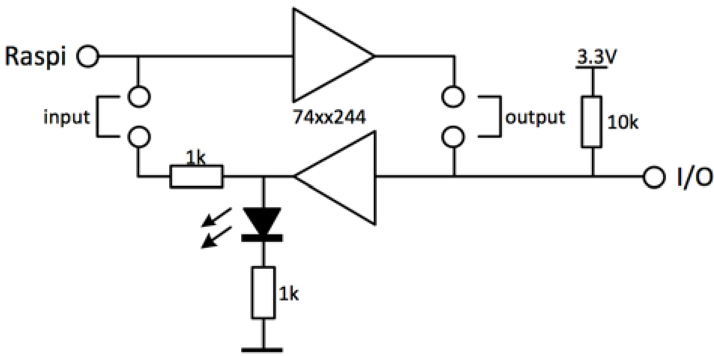

Figura 6.9: Circuito de los puertos de E/S 4-12 de la Gertboard .

Los puertos 1, 2 y 3 tienen un circuito ligeramente distinto que incorpora un pulsador conectado entre la entrada y la tierra, como vemos en el esquema de la Figura 6.10. Estos pulsadores están identificados en la serigrafía de la placa como S1, S2 y S3. Para poder leer uno de estos pulsadores desde la Raspberry Pi, deberemos conectar la entrada del circuito a uno de los pines del GPIO, usando un cable que una la entrada con el pin correspondiente del conector J2. Además, tendremos que asegurarnos de que el *jumper* del buffer de entrada no esté conectado. De lo contrario el buffer forzaría el pin a un 1 lógico aunque esté pulsado

el pulsador, debido a que su entrada está conectada a 3.3V a través de la resistencia de 10k en la salida del circuito. Se puede no obstante conectar el *jumper* del buffer de salida para visualizar el estado del pulsador en el LED.

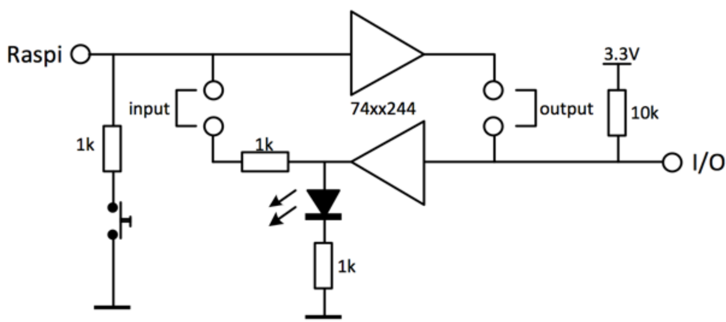

Figura 6.10: Circuito de los puertos de E/S 1-3 de la Gertboard .

6.4. Uso de máscaras de bits

A la hora de escribir/leer los registros de control/datos de un dispositivo, a menudo querremos consultar el valor de un bit concreto. Sin embargo, como ya hemos visto en numerosas ocasiones, los registros del ARM son de 32 bits. ¿Cómo podemos consultar el valor de un bit cualquiera de entre esos 32? ¿Cómo podemos ponerlo a 0/1 sin modificar el resto? Lo mismo sucede en C si queremos modificar o consultar el valor de un sólo bit en una variable entera, que puede representar el valor que queremos escribir en un puerto o que hemos leído de él. Para conseguirlo, deberemos utilizar operaciones AND, OR, NOT y XOR a nivel de bit, bien sea en ensamblador o en C, utilizando un operando constante que denominaremos *máscara*.

6.4.1. Operaciones con máscaras en Ensamblador

Las operaciones que solemos considerar son:

- Consulta de un bit en un registro, generalmente porque queremos hacer unas acciones u otras en función de su valor (saltos condicionales). Para ello hacemos una AND del registro con una máscara que tenga todos los bits a 0 excepto el bit que queremos consultar (que pondremos a 1). Si el resultado es distinto de cero es que el bit consultado estaba a 1 en el registro. Si el resultado es 0 es que estaba a 0. Por ejemplo, si queremos consultar el valor del bit 6 (empezando a numerar en 0) del registro r1, podríamos hacer lo siguiente:

  ```
  and r2, r1, #0x00000040
  cmp r2, #0
  beq CODIGO_PARA_BIT_A_CERO
  @ CODIGO PARA BIT A 1
  ```

  Entonces r2 tomaría el valor 0x00000040 si el bit 6 estaba a 1 y 0x00000000 en caso contrario. Por lo tanto, si el resultado es distinto de cero el bit estaba a 1. Si por el contrario el resultado es 0 significa que el bit estaba a 0.

- Poner a 0 algunos bits de un registro sin modificar el resto. En este caso hacemos una AND del registro con una máscara que tenga a 0 los bits que queremos poner a 0 y a 1 los bits que no queramos modificar. Por ejemplo, si queremos escribir un 0 únicamente en los bits 3 y 5 del registro r3, dejando el resto inalterados, podríamos hacer lo siguiente:

  ```
  and r3, r3, #0xFFFFFFD7
  ```

  Muchas veces resulta más sencillo construir la máscara negando (complemento a 1) la máscara contraria. Hay varias formas de hacerlo, por ejemplo con mvn:

  ```
  mvn r2, #0x00000028
  and r3, r3, r2
  ```

- Poner a 1 algunos bits de un registro sin modificar el resto. En este caso hacemos una OR del registro con una máscara que tenga a 1 los bits que queremos poner a 1 y a 0 los bits que no queramos modificar. Por ejemplo, si queremos escribir un 1 únicamente en los bits 3 y 5 del registro r3, dejando el resto inalterados, podríamos hacer lo siguiente:

  ```
  orr r3, r3, #0x00000028
  ```

- Invertir el valor de algunos bits, dejando el resto inalterados. En este caso hacemos una XOR del registro con una máscara que tenga a 1 los bits que queremos invertir y a 0 los bits que no queremos modificar. Por ejemplo, si queremos invertir únicamente en los bits 3 y 5 del registro r3, dejando el resto inalterados, podríamos hacer lo siguiente:

  ```
  eor r3, r3, #0x00000028
  ```

Finalmente conviene que nuestro código sea lo más sencillo posible de seguir. El uso de máscaras dificulta la legibilidad del código y suele inducir a errores que son difíciles de encontrar, ya que al ver la máscara nos cuesta distinguir cuales son los bits que estamos modificando y cuales no. Por este motivo se recomienda definir símbolos para las máscaras y construir las máscaras con operaciones lógicas de máscaras sencillas, obtenidas con desplazamientos a la izquierda de 0x1. Por ejemplo, si queremos dos máscaras, la primera con los bits 1 y 2 a uno y el resto a cero, y la segunda con los bits 3 y 7 a 0 y el resto a uno, podemos escribir:

```
@definimos simbolos para las máscaras
.equ MASK1,  ((0x1 << 1) | (0x1 << 2))
.equ MASK2, ~((0x1 << 3) | (0x1 << 7))

...

@usamos las máscaras
mov R1, #MASK1
mov R2, #MASK2
```

El ensamblador traduce estas instrucciones por:

```
mov r1, #6
mvn r2, #136
```

6.4.2. Operaciones con máscaras en C

En C podemos hacer las mismas operaciones utilizando los operadores a nivel de bit (*bitwise* operators), que son:

- `AND`: operador `&` (no confundir con el operador lógico `&&` que se utiliza para evaluaciones condición, tomando 0 como falso y distinto de cero como verdadero).
- `OR`: operador `|` (no confundir con el operador lógico `||`).
- `NOT`: operador `~` (no confundir con el operador lógico `!`).
- `XOR`: operador `^`.
- Desplazamiento a la izquierda: operador `<<`.
- Desplazamiento a la derecha: operador `>>`.

Así, las operaciones habituales serían:

- Consulta de un bit en una variable. Para ello hacemos una AND de la variable con una máscara que tenga todos los bits a 0 excepto el bit que queremos consultar (que pondremos a 1). Observemos que si el bit estaba a 1 el resultado será distinto de cero, que es considerado como verdadero en C en cualquier evaluación lógica, y si el bit estaba a 0 el resultado será 0, que es considerado falso en cualquier evaluación lógica. Por ejemplo, si queremos ejecutar un código sólo si el 6 de una variable A está a 1, podríamos hacer lo siguiente:

```
if (A & 0x40) {
  // Código a ejecutar si el
  // bit 6 de A está a 1
}
```

- Poner a 0 algunos bits de una variable sin modificar el resto. Como en ensamblador, hacemos una AND de la variable con una máscara que tenga a 0 los bits que queremos poner a 0 y a 1 los bits que no queramos modificar. Por ejemplo, si queremos escribir un 0 únicamente en los bits 3 y 5 de `A`, dejando el resto inalterados, podríamos hacer lo siguiente:

```
A = A & ~(0x28);
```

  que puede escribirse también como:

```
A &= ~(0x28);
```

- Poner a 1 algunos bits de una variable sin modificar el resto. Como en ensamblador, hacemos una OR de la variable con una máscara que tenga a 1 los bits que queremos poner a 1 y a 0 los bits que no queramos modificar. Por ejemplo, si queremos escribir un 1 únicamente en los bits 3 y 5 de `A`, dejando el resto inalterados, podríamos hacer lo siguiente:

```
A = A | 0x28;
```

  que puede escribirse también como:

```
A |= 0x28;
```

- Invertir el valor de algunos bits, dejando el resto inalterados. En este caso hacemos una XOR de la variable con una máscara que tenga a 1 los bits que queremos invertir y a 0 los bits que no queremos modificar. Por ejemplo, si queremos invertir únicamente en los bits 3 y 5 de `A`, dejando el resto inalterados, podríamos hacer lo siguiente:

  ```
  A = A ^ 0x28;
  ```

  que puede escribirse también como:

  ```
  A ^= 0x28;
  ```

Por supuesto, igual que comentamos para el caso del lenguaje Ensamblador, conviene que nuestro código sea lo más legible posible. En C suele ser habitual definir macros del preprocesador para las máscaras, construyéndolas con operaciones lógicas sobre máscaras sencillas obtenidas de desplazamientos a la izquierda de 0x1. Por ejemplo, si queremos dos máscaras, la primera con los bits 1 y 2 a uno y el resto a cero, y la segunda con los bits 3 y 7 a 0 y el resto a uno, podemos escribir:

```
// definimos simbolos para las máscaras
#define MASK1  ((0x1 << 1) | (0x1 << 2))
#define MASK2 ~((0x1 << 3) | (0x1 << 7))

...

// Si los bits 1 y 2 están a 1
if ((A & MASK1) == MASK1) {
  //poner los bits 3 y 7 a 0
  A = A & MASK2;
}
```

6.5. E/S activa con espera de respuesta

En las secciones anteriores hemos presentado el HW del sistema de E/S y hemos estudiado con detalle la estructura del controlador de pines de E/S. Colocando adecuadamente un cable podríamos, por ejemplo, conectar el pin del GPIO23 a la entrada del puerto de E/S 1 de la Gertboard. Configurando dicho pin como entrada y activando su *pull-up* podríamos leer el estado del pulsador.

También podríamos, por ejemplo, usar otro cable para conectar el GPIO17 a la entrada del puerto 4 de la Gertboard. Configurando dicho puerto como salida podríamos encender o apagar el led asociado a ese puerto escribiendo en el pin GPIO17.

Entonces, podríamos hacer un sencillo programa que constantemente leyese el estado del pulsador, y si detecta que se ha pulsado cambie el estado del led. Pero, ¿cómo sería el código para hacer eso? ¿Como leemos y/o escribimos en un registro de E/S en lenguaje C? ¿Cómo debemos estructurar nuestro programa? Los apartados siguientes revisan algunos conceptos del lenguaje C que iremos usando para responder a estas preguntas y finalmente presentan un programa con el comportamiento buscado.

6.5.1. Punteros

Un puntero no es más que una variable que almacena una dirección de memoria. Podríamos pensar que esto no es nada nuevo, ya que cualquier entero de 32 bits puede ser utilizado para almacenar una dirección de 32 bits ¿verdad? Lo que pasa es que C proporciona además dos operadores unarios nuevos asociados a los punteros, que son:

- El operando de desreferenciación `*`, que debe ponerse delante de la variable puntero (`*variable_puntero`). La desreferencición de un puntero implica un acceso a la dirección de memoria almacenada en el puntero (no hay que confundirla con la dirección en la que se almacena el propio puntero). Se suele decir que el puntero *apunta* a una dirección de memoria, y el operando `*` permite acceder a la dirección apuntada.

- El operando dirección-de `&`, que se puede poner delante de cualquier variable (`&variable_cualquiera`) para obtener la dirección en la que se almacena dicha variable. Este operando suele utilizarse para asignar a un puntero la dirección de una variable. Por ejemplo, esto es lo que se utiliza en C para pasar una variable por referencia a una función, en realidad se pasa por valor la dirección de la variable, que se asigna a una variable local de la función tipo puntero. Desreferenciando dicho puntero accedemos a la variable, y podremos modificar su valor dentro de la función.

Como vemos, se utilizan los mismos caracteres que para los operandos de multiplicación y AND bit a bit. Cuando un `*` aparece delante de una variable tipo puntero el compilador lo interpreta como una desreferenciación del puntero, y no como un operando de multiplicación. Si el `*` aparece delante de una variable entera, el compilador interpretará que es una operación de multiplicación, y no una desreferencición. Por este motivo necesitamos el tipo puntero. Si el resultado del operando dirección-de se asigna a una variable que no es de tipo puntero se producirá un error de compilación.

En C los punteros son tipados, es decir, debemos identificar el tipo de la variable a la que apuntará el puntero. Para declarar un puntero a un tipo ponemos:

```
tipo * nombre_del_puntero;
```

Como cualquier otra variable, podemos declararla con valor inicial o sin él. Por ejemplo para declarar un puntero `mipuntero` que sirva para apuntar a enteros escribiríamos:

```
int *mipuntero;
```

¿Y qué quiere decir que al desreferenciar un puntero se accede a la dirección apuntada por dicho puntero? La respuesta exacta a esta pregunta depende de la arquitectura para la que se genere el código. En el caso del ARM (y cualquier otro procesador RISC) el acceso implica:

- Un load a la dirección almacenada en el puntero (la dirección donde apunta) si la desreferenciación implica una lectura. Por ejemplo si aparece a la derecha de un igual, en un paso como parámetro a una función, en la condición de un `if`, etc.

- Un store a la dirección almacenada en el puntero si la desreferenciación implica una escritura, es decir, a la izquierda de un igual en una asignación.

Como los punteros son tipados, la operación load/store será la adecuada para el tamaño del dato apuntado. Por ejemplo, si es un puntero `char*` se utilizarían las instrucciones ARM `ldrb`/`strb`.

Entonces, para poder realizar la E/S en lenguaje C no tenemos más que asignarle a un puntero la dirección del registro de E/S que queramos acceder y desreferenciar el puntero. Por ejemplo, para poner a 1 el bit GPIO17 y encender el led del puerto de la gertboard al que lo hayamos conectado podríamos hacer lo siguiente:

```
unsigned int* pGPCLR0 = (unsigned int*) 0x20200028;
*pGPCLR0 = (0x1 << 17);
```

Además de la desreferenciación, C permite sumar o restar un valor entero a un puntero. Ojo, no permite sumar dos punteros, sino sumar un valor entero a un puntero. Como programadores de ensamblador debemos estar acostumbrados a esta operación, el puntero representa la dirección base y el entero el deplazamiento, como en el caso de las instrucciones `ldr` y `str` del repertorio ARM. Sin embargo hay un matiz importante. Gracias a que los punteros son tipados, el compilador sabe cuántos bytes ocupa el tipo de dato apuntado por el puntero. Cuando sumamos un entero `i` a un puntero de un tipo que ocupa `n` bytes, el código generado por el compilador suma `i*n` a la dirección almacenada en el puntero. Esto permite utilizar cómodamente un puntero para recorrer un array. Si inicializamos el puntero con la dirección del primer elemento del array y le sumamos `i` al puntero, obtendremos la dirección del elemento con índice `i`. Por ejmplo, en el siguiente código utilizamos un puntero para escribir el valor 10 en la posición 5 del array `A` (es decir en A[5]):

```
unsigned int A[100] ={0};
unsigned int *p = A;
*(p + 5) = 10;
```

Finalmente, podemos también indexar un puntero como si fuese un array. Es decir, si p es un puntero, podemos poner p[i] para acceder a la dirección almacenada en `p` más `i` veces el número de bytes ocupados por el tipo apuntado, es decir, es equivalente a poner $*(p+i)$.

6.5.2. Direcciones volátiles

Prácticamente sabemos ya todo lo que hay que saber para poder manejar los puertos de E/S en lenguaje C, tan sólo falta darnos cuenta de una importante sutilidad: el contenido de los registros de E/S puede cambiar sin que lo cambie el programa, por ejemplo, porque pulsemos un pulsador. Esto no lo sabe el compilador si no se lo indicamos.

Para ilustrar el problema supongamos que tenemos un bucle que está examinando el estado del pulsador del puerto 1 de la gertboard, que a su vez está conectado con un cable al GPIO23, esperando sin hacer nada hasta que el pulsador sea pulsado (bit a 0). El código podría ser parecido al siguiente:

```
unsigned int* pGPLEV0 = (unsigned *)0x20200034;
...
while ((*pGPLEV0 & (0x1 << 23)) == 0);
```

y esperaríamos que el compilador tradujese este código por uno similar a este:

```
    .equ rGPLEV0, 0x20200034
    .text
    ...
    ldr r0, =rGPLEV0
L0:
    ldr r1,[r0]
    and r1, r1, #(0x1 << 23)
```

```
    cmp r1, #0
    beq L0
L1:
```

Fijémonos que dentro del bucle (entre `L0:` y `L1:`) no se modifica lo que hay en la dirección `0x20200034`, sólo se lee. El compilador podría entonces *optimizar* el código sacando la instrucción `ldr r1, [r0]` fuera del bucle, con lo que quedaría:

```
    .equ rGPLEV0, 0x20200034
    .text
    ...
    ldr r0, =rGPLEV0
    ldr r1,[r0]
L0:
    and r1, r1, #(0x1 << 23)
    cmp r1, #0
    beq L0
L1:
```

Entonces si el pulsador no estaba pulsado cuando se entra en el bucle, el bucle será infinito. ¿Qué ha pasado? El compilador asume que si el código que él genera no cambia el valor almacenado en `0x20200034`, le basta con leerlo una vez al principio (¡porque no va a cambiar!). Lo que hay que hacer es indicarle al compilador que esa dirección de memoria es *volátil*, que su valor puede cambiar por mecanismos ajenos al programa, y que por tanto debe volver a leer el valor cada vez que quiera consultarlo. Esto se consigue utilizando el modificador `volatile` en la declaración del puntero. Así, el código correcto para escanear el pulsador sería:

```
volatile unsigned int* pGPLEV0 = (unsigned *)0x20200034;
...
while ((*pGPLEV0 & (0x1 << 23)) == 0);
```

6.5.3. Ejemplo de programa de E/S por espera activa

Ya estamos preparados para describir el programa que permita conmutar un led cuando pulsemos un pulsador en la gertboard. Vamos a ver paso a paso las partes de nuestro programa.

Comenzaremos declarando unas macros del preprocesador para el acceso a los registros de las Tablas 6.1-6.4 que necesitamos acceder. Las formaremos desreferenciando un puntero construido directamente a partir de una constante entera con la dirección del registro de E/S, reinterpretada como puntero por medio de un *casting*:

```
#define GPFSEL1    (*(volatile unsigned *)0x20200004)
#define GPFSEL2    (*(volatile unsigned *)0x20200008)
#define GPSET0     (*(volatile unsigned *)0x2020001C)
#define GPCLR0     (*(volatile unsigned *)0x20200028)
#define GPLEV0     (*(volatile unsigned *)0x20200034)
#define GPPUD      (*(volatile unsigned *)0x20200094)
#define GPPUDCLK0  (*(volatile unsigned *)0x20200098)
```

Asimismo, crearemos algunas funciones auxiliares simples. Por ejemplo, funciones para encender, apagar o conmutar el led, una función para leer el estado del pulsador y una función de espera:

```
#define LED_OFF 0
#define LED_ON  1
#define BUTTON_PUSHED   0
#define BUTTON_RELEASED 1

int led; //estado del led

void led_on(void)
{
    GPSET0 = 0x1 << 17;
    led = LED_ON;
}

void led_off(void)
{
    GPCLR0 = 0x1 << 17;
    led    = LED_OFF;
}

void led_switch(void)
{
    if (led == LED_OFF)
        led_on();
    else
        led_off();
}

int button_read(void)
{
    int button = BUTTON_RELEASED;

    if ((GPLEV0 & (0x1 << 23)) == 0)
        button = BUTTON_PUSHED;

    return button;
}

void short_wait(void)
{
  int w;
  for (w=0; w<100; w++) {
    w++;
    w--;
  }
}
```

Agruparemos la configuración del sistema en una función, poniendo el GPIO17 como salida y el GPIO23 como entrada, activando también su circuito de *pull-up*:

```
void setup_gpio()
{
    // Configurar GPIO 23 como INPUT (000) --> Boton
    GPFSEL2 = GPFSEL2 & ~(0x7 << 3*3);

    // Configurar GPIO 17 como OUTPUT (001) --> LED
    GPFSEL1 = (GPFSEL1 & ~(0x7 << 7*3)) | (0x1 << 7*3);

    // Activar el pull-up del GPIO 23
    GPPUD     = 0x2;
    short_wait();
    GPPUDCLK0 = (0x1 << 23);
    short_wait();
    GPPUD     = 0;
    GPPUDCLK0 = 0;
}
```

Finalmente el programa principal se encargará de configurar los pines, de apagar inicialmente el led y después repetiría indefinidamente el siguiente proceso:

1. Esperar hasta que se pulse el pulsador
2. Conmutar el estado del led
3. Esperar unos milisegundos para eliminar posibles rebotes en la línea del pin debidos a la pulsación

El código resultante sería:

```
int main (void)
{
    setup_gpio();
    led_off();
    while (1) {
        // Bucle de espera activa
        while (button_read() == BUTTON_RELEASED);

        led_switch();
        // Eliminamos rebotes
        short_wait();
    }
    return 0;
}
```

Esta forma de interactuar con los dispositivos se denomina E/S activa con espera de respuesta, y se caracteriza porque el procesador se mantiene ocupado ejecutando instrucciones que no hacen un trabajo útil a la espera de que el dispositivo responda. En este caso, el programa espera en un bucle que no hace nada hasta que detecta que se ha pulsado el pulsador. Este mecanismo es muy poco eficiente si tratamos con dispositivos lentos con bajo ancho de banda.

En el siguiente capítulo estudiaremos la E/S por interrupciones, que nos permitirá que un dispositivo *avise* al computador de cuándo está listo (o cuando ha sucedido un evento que requiere su atención) y el programador podrá registrar un código que sería ejecutado al detectarse este *aviso* del dispositivo. En el ejemplo descrito arriba, la rutina registrada se encargaría de conmutar el led. Mientras el dispositivo no avise el computador puede realizar cualquier otra tarea útil. Este nuevo mecanismo es mucho más eficiente para tratar dispositivos lentos que no tengan un ancho de banda excesivamente grande.

Capítulo 7

E/S mediante interrupciones

La idea fundamental de la E/S por interrupciones se ilustra en la Figura 7.1. El dispositivo puede indicar al procesador que ha sucedido algún evento que requiere su atención activando una señal de interrupción. Si esta señal no está bloqueada, hará que el procesador *interrumpa* lo que está haciendo de forma controlada y pase a ejecutar una nueva tarea encargada de atender al dispositivo. Al finalizar dicha tarea el procesador retomará su trabajo por donde lo había dejado, como si la interrupción no se hubiese producido.

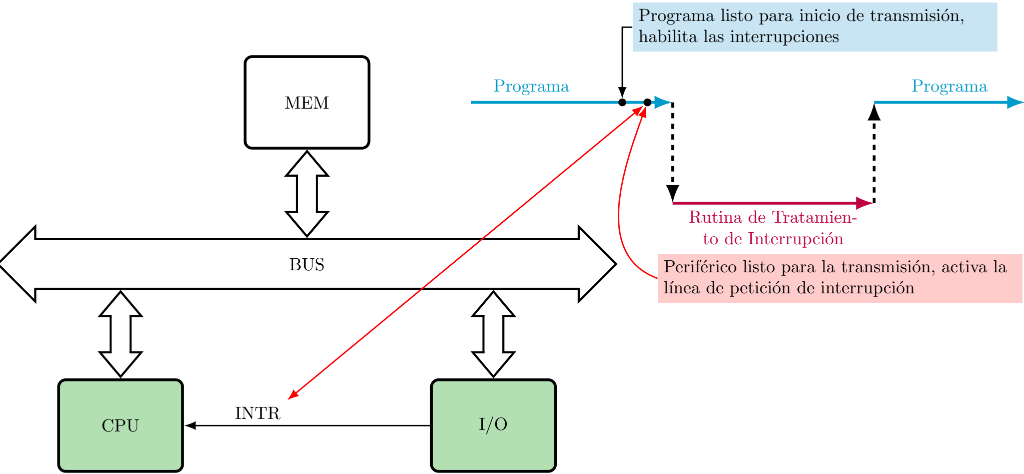

Figura 7.1: Concepto de Interrupción

En este capítulo analizaremos el funcionamiento del sistema de interrupciones de la Raspberry Pi 1. De nuevo hay varios componentes que intervienen en el sistema, el procesador, el controlador de interrupciones incluido en el BCM2835 y los dispositivos capaces de funcionar por interrupciones. Finalmente veremos cómo estructurar un código que gestione algún dispositivo por interrupciones.

7.1. Excepciones e Interrupciones

Una excepción es un mecanismo que permite atender eventos inesperados, con origen interno (ej: intento de ejecutar una instrucción no definida) o externo (ej: solicitud de interrupción externa por parte de un dispositivo). Normalmente cuando el origen es externo se

utiliza el nombre de interrupción.

La idea es sencilla, cuando se produce una excepción el procesador interrumpe de forma controlada su ejecución y pasa a ejecutar una rutina específica (habitualmente denominada *Interrupt Service Routine* o simplemente *ISR*) que tratará esa excepción.

El tratamiento de excepciones es un problema complejo que complica el diseño interno del procesador. Sin embargo, a alto nivel todos los procesadores con un modelo de excepciones precisas realizan una secuencia de pasos similar a la siguiente:

1. Se interrumpe la ejecución de instrucciones, completando el procesamiento de la instrucción actual[1].

2. Se almacena en un lugar seguro una dirección de retorno. En muchos procesadores en realidad la dirección almacenada no es exactamente la dirección de retorno sino que ésta se calcula a partir de la dirección guardada, posiblemente de forma distinta para cada excepción.

3. Se inhabilitan las interrupciones, para que durante el tratamiento de la interrupción no se produzca otra vez la interrupción. Esto es muy importante ya que el dispositivo no retirará su petición hasta que la *ISR* le notifique que se está tratando. Si las interrupciones no fuesen inhabilitadas se entraría en un bucle infinito.

4. Se almacena todo o parte del estado arquitectónico en un lugar seguro (registros o pila).

5. Se cambia el modo de ejecución del procesador a un modo privilegiado.

6. Se copia en el PC una dirección prefijada

7. Se retoma la ejecución de instrucciones

Esta secuencia lleva de forma efectiva al procesador a saltar a esa dirección prefijada, en un modo de ejecución privilegiado y con parte o todo el estado arquitectónico copiado en un lugar seguro. En dicha dirección normalmente hay un salto a la *ISR* o a algún código intermedio encargado de saltar finalmente a la *ISR*.

La *ISR* no puede ser diseñada como una subrutina corriente, siguiendo el estándar de llamadas a procedimientos de la arquitectura, ya que no es invocada por el programador sino que se llega a ella de forma asíncrona sin control por su parte. Por tanto, la *ISR* debe preservar el estado del procesador completo, para que al finalizar pueda restaurarlo y retomar la ejecución del programa en el punto en que se dejó. Además, el retorno de la *ISR* también difiere del retorno normal de una subrutina, puesto que hay que restaurar todo el estado arquitectónico, volver al modo de ejecución original y posiblemente cambie la forma de obtener la dirección de retorno.

Por otro lado, cuando se trata de una interrupción solicitada por algún dispositivo externo, una de las misiones fundamentales de la *ISR* es notificar al dispositivo que se está atendiendo su petición para que éste la desactive. De lo contrario se volvería a producir una

[1]Esta es una visión simplificada, que no tiene en cuenta el diseño interno del procesador. En un procesador segmentado se completaría la ejecución de la instrucción que ha provocado la excepción o, para una interrupción externa, se completaría la interrupción de la instrucción que está en ese momento en la etapa de WB

interrupción en cuanto se retorne de la *ISR*. A este proceso se le llama reconocimiento de interrupción[2].

Se dice que el tratamiento de excepciones o interrupciones es autovectorizado, cuando el procesador salta a una dirección prefijada distinta para cada excepción o interrupción. Se dice que el sistema de interrupciones es vectorizado si la dirección la proporciona el dispositivo. En ambos casos, en el paso 6 diríamos que el procesador escribe el *vector de la excepción* en el PC[3]. Generalmente el tratamiento de excepciones es autovectorizado, y las interrupciones suelen vectorizarse por medio de un controlador de interrupciones, como veremos un poco más adelante.

Habitualmente los procesadores tienen pocas líneas de interrupción, y éstas deben ser compartidas por varios dispositivos. En este caso todos los dispositivos compartirán el mismo vector de interrupción, se dice entonces que las interrupciones son no vectorizadas. En este caso la *ISR* debe encargarse de identificar por software qué dispositivo a solicitado la interrupción. A este proceso se le suele llamar *encuesta* ya que la *ISR* va consultando uno a uno los registros de estado de los controladores de los dispositivos conectados a la línea para ver si tienen activa la petición de interrupción.

Sin embargo los computadores extienden frecuentemente el sistema de interrupciones original del procesador por medio de un controlador de interrupciones externo (IC o *Interrupt Controller*). La Figura 7.2 ilustra un posible mecanismo para vectorizar una línea de interrupción cuando el procesador no tiene soporte específico para ello. En este caso los dispositivos se conectan al IC (entradas $INT_{1...n}$), en lugar de conectarse directamente a la línea de interrupción. El IC solicitará al procesador una interrupción cuando alguno de los dispositivos conectados a sus líneas de petición solicite una interrupción.

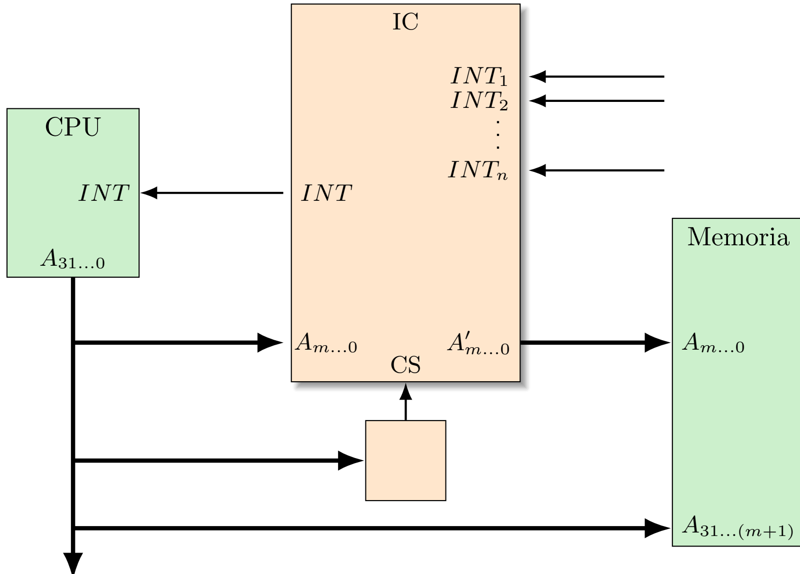

Figura 7.2: Concepto de controlador de interrupciones

[2] En algunos sistemas antiguos el reconocimiento se hacía por hardware a través de una señal de reconocimiento de interrupción ($INTA$ de Interrupt Acknowledge)

[3] En realidad en muchos procesadores la dirección escrita en el PC se obtiene sumando el vector a una dirección base prefijada, que se conoce como base de la tabla de vectores. En muchos casos esta dirección base es configurable

Los controladores más básicos no vectorizarán las líneas de interrupción. Aún así, la presencia del IC facilita enormemente la encuesta, ya que bastará con leer uno de los registros del IC para identificar al dispositivo que se debe atender, en lugar de tener que ir leyendo uno tras otro los registros de estado de los controladores de los dispositivos.

Los controladores de interrupciones más sofisticados vectorizan las líneas de interrupción. Como vemos en la Figura 7.2, el IC se interpone entre la CPU y el controlador de memoria en la conexión al bus de direcciones. Cuando se produce una interrupción el procesador trata de acceder a la dirección del vector asociado a la línea de interrupción del procesador. El IC intercepta las líneas de dirección del bus ($A_{m...0}$) y coloca en las líneas de salida hacia la memoria ($A'_{m...0}$) una dirección diferente, que depende de la línea de interrupción activa ($INT_{1...n}$). Esto permite al programador registrar una *ISR* distinta para cada una de ellas, acelerando y simplificando enormemente el proceso de identificación.

Finalmente, debemos tener en cuenta que pueden producirse varias excepciones o interrupciones simultáneamente, por lo que deberán establecerse prioridades a la hora de atenderlas. Cuando las interrupciones no son vectorizadas es la *ISR* la que dictaminará por software la prioridad. Normalmente la encuesta la realiza empezando por el dispositivo más prioritario, avanzado hacia el menos prioritario hasta que encuentra uno que ha solicitado la interrupción. Cuando las líneas están vectorizadas es el hardware el que selecciona el vector más prioritario. Generalmente los IC que vectorizan las líneas permiten configurar las prioridades de las mismas.

7.2. Excepciones en el ARM1176JZF-S

El procesador ARM1176JZF-S (arquitectura ARM V6, [arm]) distingue 9 excepciones autovectorizadas, descritas en la Tabla 7.1, y dispone de 8 modos de ejecución diferentes, que se describen a su vez en la Tabla 7.2, destinados en su mayoría a la gestión de las excepciones. Por último, la Tabla 7.3 ordenada las excepciones de mayor a menor prioridad e indica el vector y el modo de ejecución asociado a cada una[4].

El procesador estará en el modo de ejecución cuyo código (Tabla 7.2) aparezca en los cinco bits menos significativos del registro de estado (*Mode bits* en la Figura 1.2). Todos los modos excepto *usr* son privilegiados, y por tanto no imponen restricciones de acceso a los recursos del procesador. En el modo *usr* en cambio, no podemos escribir en los bits de modo del CPSR, con lo que sólo se cambiará de modo cuando se produzca alguna excepción[5]. Observemos que cuando se inicializa el sistema (Reset) lo hace en modo supervisor (SVC), que es uno de los modos privilegiados. Los modos destinados al tratamiento de las excepciones son: FIQ (*fiq*), IRQ (*irq*), Supervisor (*svc*), Abort (*abt*) y Undef (*und*); todos ellos privilegiados. Por último, el modo System (*sys*) es el único modo privilegiado al que no se llega a través de una excepción, sólo se puede cambiar a él desde otro modo privilegiado, usando la instrucción `msr` para escribir su código en los cinco bits menos significativos del CPSR. Este modo lo emplea el sistema operativo cuando necesita acceder a ciertos recursos del sistema desde fuera de un modo de excepción. Finalmente, el modo Secure Monitor Mode (*mon*) está relacionado con

[4]En al ARM1176JZF-S la dirección de salto en el tratamiento de la excepción se compone sumando el vector a una dirección base de la tabla de vectores que se puede seleccionar mediante un registro de control especial. De momento lo ignoraremos y tomaremos su valor por defecto (0).

[5]Se puede cambiar por software a modo supervisor ejecutando una instrucción `svc` o `swi`, denominada interrupción software. Esta instrucción provoca una excepción y hace que el procesador pase a modo SVC. Este es el mecanismo utilizado por el sistema operativo para ofrecer sus servicios.

Tabla 7.1: Excepciones del ARM1176JZF-S

Excepción	Descripción
Reset	Se produce cuando se activa la señal externa de reset del sistema.
Undef	Se produce cuando se intenta ejecutar una instrucción no definida. Si la condición de la instrucción no se cumple (recordemos que todas las instrucciones son condicionales) entonces la excepción no se produce.
SVC	Se produce cuando se ejecuta la instrucción `svc` (interrupción software o llamada al sistema).
IRQ	Se produce cuando se activa la línea de interrupciones externas IRQ.
FIQ	Se produce cuando se activa la línea de interrupciones externas rápidas FIQ.
Prefetch Abort	Cuando se realiza la búsqueda (fetch) de una instrucción en una dirección no válida. El controlador de memoria es el responsable de generar la interrupción.
Data Abort	Cuando se intenta acceder a memoria en una posición no válida, para lectura o escritura de datos. Es el controlador de memoria el responsable de generar la interrupción.
BKPT	Se produce cuando se ejecuta una instrucción de breakpoint software (de momento la ignoraremos).
SMC	Se produce cuando se ejecuta una instruccion de llamada al modo Secure Monitor (*SecureMonitorCall*).

Tabla 7.2: Modos del procesador

Modo	**Código**	**Uso**
usr	10000	Ejecución de código de usuario
fiq	10001	Servicio de int. rápidas
irq	10010	Servicio de int. lentas
svc	10011	Modo protegido para sistema operativo (int. sw)
mon	10110	Modo seguro empleado con extensiones *TrustZone*
abt	10111	Procesado de fallos de acceso a mem
und	11011	Manejo de instrucc. indefinidas
sys	11111	Ejecución de tareas del SO

Tabla 7.3: Correspondencia entre excepciones, modos y vectores.

Prioridad	**Excepción**	**Modo**	**Vector**
1	Reset	SVC	`0x00`
2	Data Abort	Abort	`0x10`
3	FIQ	FIQ	`0x1C`
4	IRQ	IRQ	`0x18`
6	Prefetch Abort	Abort	`0x0C`

7	Undef
	SVC
	BKPT
	SMC
		Undef
		SVC
		Abort
		Mon
			`0x04`
			`0x08`
			`0x0C`
			`0x08`

las *TrustZone Security Extensions* y de momento lo obviaremos.

Cuando describimos la arquitectura en el Capítulo 1 dijimos que el programador tenía acceso a 15 registros de propósito general, el PC y el CPSR. Sin embargo, la arquitectura dispone en realidad de 40 registros de 32 bits, incluyendo el contador de programa. Estos registros se organizan en bancos parcialmente solapados, y cada modo tiene acceso a uno de los bancos, como ilustra la figura 7.3. Como vemos, en todos los bancos los registros de la parte superior solapan con los del primer banco, y por tanto son los mismos que los registros del modo usuario. Sin embargo, los modos FIQ, IRQ, Supervisor, Abort, Undefined y Secure Monitor tienen algunos registros propios, no solapados con los del modo usuario. Por ejemplo, cada uno tiene su propio puntero de pila SP (R13), su propio registro LR (R14) y su propio registro de sombra SPSR (con la excepción en este último caso del modo usuario). Además, en el modo FIQ los registros R8-R12 son distintos de los del modo usuario. Finalmente, en el modo System se usan los mismos registros que el modo usuario.

Entonces, cuando se produce una excepción el procesador realiza automáticamente las acciones descritas en el 8. Básicamente el procesador guarda el registro de estado (CPSR) en el registro de sombra (SPSR) y la dirección de retorno en el registro r14 (lr) del modo asociado, cambia a dicho modo con las interrupciones deshabilitadas y ejecuta la instrucción que está almacenada en memoria en la dirección del vector de la excepción (ver Tabla 7.3). Esta instrucción debe ser un salto a la *ISR* encargada de tratar la excepción o, en su defecto, a un código auxiliar que termine saltando a la *ISR*. Debido al cambio de modo, la *ISR* usará los registros r13 (sp) y r14 (lr) propios dicho modo, quedando los originales intactos, y será responsabilidad de la propia *ISR* preservar el valor del resto de los registros arquitectónicos (`r0-r12`) copiándolos en la base de su marco de activación. Como cada modo tiene su propio registro de pila, es habitual inicializar estos registros con un valor distinto para que cada modo use una región de pila diferente.

Una vez completado el tratamiento de la interrupción la *ISR* debe restaurar el estado arquitectónico y retomar la ejecución en la dirección correcta, en el modo de ejecución original. Para ello deberá restaurar la copia de los registros que ha metido en la pila y hacer simultáneamente dos cosas:

- Restaurar el valor del `CPSR` a partir del valor guardado en el `SPSR` (esto hará que se regrese al modo de ejecución original).

Cuadro 8 Acciones realizadas por el procesador cuando se produce una excepción[arm]

1. Almacena la dirección de retorno en el registro r14 propio del modo de ejecución asociado a la excepción. En realidad el valor almacenado depende del tipo de excepción[6] (consultar la sección 2.12 de [arm]) lo que hace que el retorno de cada *ISR* sea distinto[7], como veremos más adelante.

```
R14_<modo_de_excepcion>  = direccion de retorno
```

2. Copia el registro de estado (CPSR) en el registro `SPSR` del modo de ejecución correspondiente a la excepción.

```
SPSR_<modo_de_excepcion> = CPSR
```

3. Pone el código del modo de ejecución correspondiente a la excepción en los bits `M[4:0]` del registro de estado.

```
CPSR[4:0] = código del modo de excepción
```

4. Cambia al estado ARM, si no lo estuviese ya[8].

```
CPSR[5] = 0 /* Cambiar a estado ARM */
```

5. Si el modo para el tratamiento de la excepción es Reset o FIQ, el procesador deshabilita las interrupciones rápidas.

```
if <modo_de_excepcion> == Reset or FIQ then
     CPSR[6] = 1 /* Deshabilitar interrupciones rápidas */
/* else CPSR[6] no se cambia */
```

6. Deshabilita las interrupciones normales.

```
CPSR[7] = 1 /* Deshabilitar interrupciones normales */
```

7. Copia en el PC el vector correspondiente a la interrupción.

```
PC = dirección del vector de excepción
```

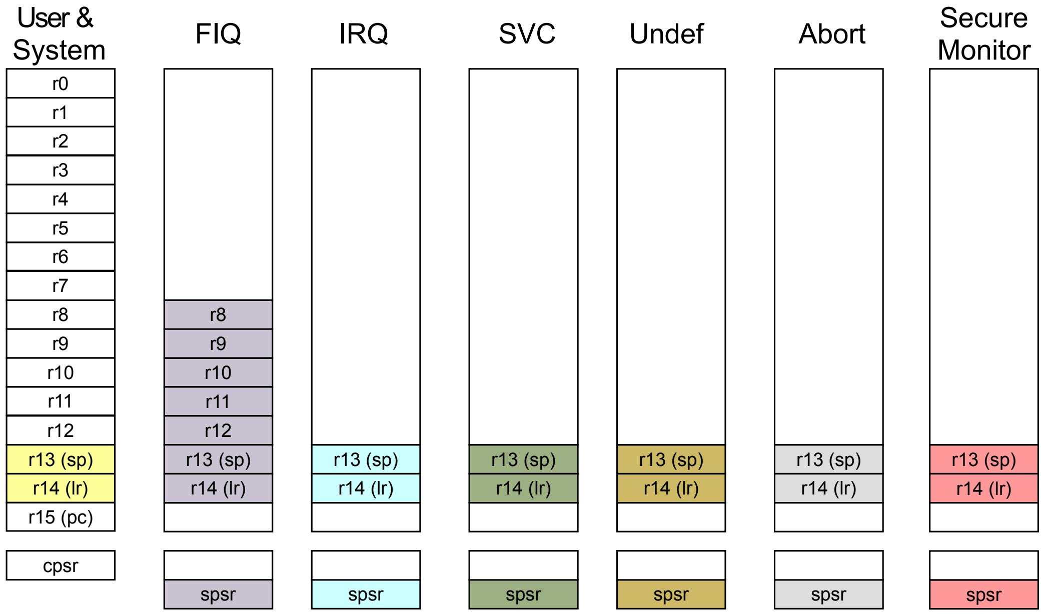

Figura 7.3: Registros visibles en cada modo de ejecución (los registros coloreados son privados a cada modo).

- Escribir en PC la dirección de retorno, que podemos calcular a partir del valor almacenado en LR siguiendo las indicaciones de la Tabla 7.4. Observemos que el cálculo concreto depende de la excepción.

Para realizar estas dos acciones simultáneamente debemos usar una instrucción que tenga el flag S activado (modificación del CPSR) y que escriba en el PC. Si se cumplen esas dos condiciones, el hardware automáticamente restaura el valor del CPSR a partir del del SPSR. Los cuadros 9 y 10 muestran dos alternativas para la implementación del prólogo y el epílogo de una *ISR*, la primera utiliza una instrucción aritmético lógica y la segunda una instrucción de load múltiple[9]. Sin embargo, no es necesario implementar las *ISR*s en ensamblador, podemos hacerlo en C. Todo lo que debemos hacer es indicar al compilador que no se trata de una función C estándar, sino de una función que debe servir como *ISR* para una excepción concreta del procesador. En gcc esto se consigue añadiendo a la declaración de la función una directiva `__attribute__` del siguiente modo:

```
void HandlerName(void) __attribute__((interrupt ( TYPE )));
```

donde `TYPE` puede ser `IRQ`, `FIQ`, `ABORT`, `UNDEF` o `SVC`. El compilador creará así un rutina con una estructura similar a las descritas por los Cuadros 9-10, en lugar de utilizar el prólogo y el epílogo de una función C estándar.

7.3. Controlador de Interrupciones del BCM2835

El BCM2835 incorpora un IC muy básico, que no hace uso del soporte específico del procesador para la gestión vectorizada de interrupciones. Básicamente demultiplexa las líneas

[9]Para las instrucciones de load múltiple la activación del flag S se lleva a cabo poniendo un acento circunflejo al final de la instrucción.

Tabla 7.4: Instrucción de retorno de excepción usual (consultar [arm]).

Excepción	**Inst. Retorno**
Reset	NA
Data Abort	`SUBS PC, R14_abt, #8`
FIQ	`SUBS PC, R14_fiq, #4`
IRQ	`SUBS PC, R14_irq, #4`
Prefetch Abort	`SUBS PC, R14_abt, #4`
BKPT	`SUBS PC, R14_abt, #4`
Undef	`MOVS PC, R14_und`
SVC	`MOVS PC, R14_svc`
SMC	`MOVS PC, R14_mon`

Cuadro 9 Rutina de tratamiento de excepciones para retorno con instrucción aritmética con bit `S` activo.

```
/* prólogo */
str      ip, [sp, -#4]!           @ salvamos ip en la pila
mov      ip, sp
stmdb    sp!, {r0-r10,fp,ip,lr,pc}  @ salvamos el resto del contexto
sub      fp, ip, #4

/* cuerpo de la rutina */

/* epílogo */
ldmdb    fp, {r0-r10, fp, sp, lr}   @ restauramos contexto'
ldmia    sp!, {ip}                 @ restauramos ip'
subs     pc, lr, #4                @ retorno copiando SPSR en CPSR
                                   @ debe sustituirse por la instrucción
                                   @ correspondiente
```

Cuadro 10 Rutina de tratamiento de excepciones para retorno con `LDM` con bit `S` activo.

```
/* prólogo */
sub      lr, lr, #4               @ modificamos lr para el retorno
                                  @ debe sustituirse por la instrucción
                                  @ correspondiente
stmdb    sp!, {sp, lr, pc}        @ lo almacenamos en la pila
stmdb    sp!, {r0-r10, fp, ip}    @ salvamos el resto del contexto en la pila
add      fp, sp, #60              @ fijamos fp a partir de sp, si no
                                  @ se guardan todos los registros
                                  @ hay que reajustar esta suma

/* cuerpo de la rutina */

/* epílogo */
ldmdb    fp, {r0-r10,fp,ip,sp,pc}^ @ restauramos contexto y retornamos
```

Tabla 7.5: Registros del IC

Offset	Acrónimo	Nombre
0x200	IRQ_BASIC	IRQ Basic Pending
0x204	IRQ_PEND_1	IRQ Pending 1
0x208	IRQ_PEND_2	IRQ Pending 2
0x20C	FIQ_CONTROL	FIQ control
0x210	IRQ_ENABLE_IRQS_1	Enable IRQs 1
0x214	IRQ_ENABLE_IRQS_2	Enable IRQs 2
0x218	IRQ_ENABLE_BASIC	Enable Basic IRQs
0x21C	IRQ_DISABLE_IRQS_1	Disable IRQs 1
0x220	IRQ_DISABLE_IRQS_2	Disable IRQs 2
0x224	IRQ_DISABLE_BASIC	Disable Basic IRQs

I y F en varias líneas de interrupción diferentes, que pueden ser configuradas para activar una de las líneas de interrupción del procesador y pueden ser enmascaradas selectivamente. Sólo algunas de las líneas puede estar configuradas para activar la línea F del procesador, y no puede haber más de una simultáneamente. El IC maneja interrupciones de 3 fuentes distintas [bcm]: periféricos específicos del ARM (también llamados básicos), generales (o *GPU peripherals*) y especiales (no documentados).

La Tabla 7.5 recoge una relación de los registros del IC, todos ellos emplazados en el mapa del bus a partir de la dirección `0x7E00B000`, que se corresponde con la dirección física `0x2000B000` (ver la Sección 6.2). Para habilitar la interrupción de uno de ellos debemos escribir un 1 en la posición correspondiente de su registro de habilitación (`IRQ_ENABLE_BASIC` o `IRQ_ENABLE_IRQS_n`). El manual de Broadcom para el BCM2835 [bcm] sólo documenta unas pocas de estas fuentes de interrupción, recogidas en las Tablas 7.6 y 7.7. Hay 64 líneas para periféricos generales (0-63), las 32 primeras se habilitan en el registro `IRQ_ENABLE_IRQS_1` y las 32 últimas en el `IRQ_ENABLE_IRQS_2`. Además, debemos habilitar globalmente la línea de interrupción del procesador, poniendo a 0 el bit de máscara correspondiente en el registro de estado (por ejemplo para la línea I sería el bit 7 del CPSR).

Respectivamente, para deshabilitar una fuente de interrupción debemos escribir un 1 en la misma posición pero en el correspondiente registro de deshabilitación.

Como el IC no vectoriza las interrupciones, sólo podremos registrar una *ISR* para tratar a todos los periféricos. Por tanto, lo primero que deberá hacer la *ISR* es identificar al periférico que solicitó la interrupción. Para simplificar esta tarea el IC dispone de tres registros de interrupciones pendientes:

- `IRQ_PEND0/1`: flags de interrupción pendiente de los dispositivos de E/S. Los flags documentados por Broadcom [bcm][10] se recogen en la Tabla 7.6.

- `IRQ_BASIC`: flags de interrupción de los dispositivos básicos, la documentación ofrecida por Broadcom se recoge en la Tabla 7.8.

La Figura 7.4 muestra un esquema de la interconexión de estos registros. Como vemos, los bits 8 y 9 del registro `IRQ_BASIC` nos permiten ver rápidamente si hay alguna petición

[10] La documentación del fabricante es francamente mejorable, y por ejemplo en este caso no documenta todos los bits de estos registros.

Tabla 7.6: Tabla de interrupciones de periféricos generales (*GPU peripherals* en [bcm]).

#	**IRQ 0-15**	#	**IRQ 16-31**	#	**IRQ 32-47**	#	**IRQ 48-63**
0		16		32		48	smi
1		17		33		49	gpio_int[0]
2		18		34		50	gpio_int[1]
3		19		35		51	gpio_int[2]
4		20		36		52	gpio_int[3]
5		21		37		53	i2c_int
6		22		38		54	spi_int
7		23		39		55	pcm_int
8		24		40		56	
9		25		41		57	uart_int
10		26		42		58	
11		27		43	i2c_spi_slv_int	59	
12		28		44		60	
13		29	Aux int	45	pwa0	61	
14		30		46	pwa1	62	
15		31		47		63	

Tabla 7.7: Tabla de interrupciones específicas de ARM.

#	**IRQ**
0	ARM Timer
1	ARM Mailbox
2	ARM Doorbell 0
3	ARM Doorbell 1
4	GPU0 Halted (or GPU1 halted if bit 10 of control register 1 is set)
5	GPU1 Hlated
6	Illegal access type 1
7	Illegal access type 0

Tabla 7.8: Registro `IRQ_BASIC`.

#	**IRQ**
0	ARM Timer IRQ pending
1	ARM Mailbox IRQ pending
2	ARM Doorbell 0 IRQ pending
3	ARM Doorbell 1 IRQ pending
4	GPU0 Halted IRQ pending (or GPU1 halted if bit 10 of control register 1 is set)
5	GPU1 Hlated IRQ pending
6	Illegal access type 1 IRQ pending
7	Illegal access type 0 IRQ pending
8	One or more bits set in pending register 1
9	One or more bits set in pending register 2

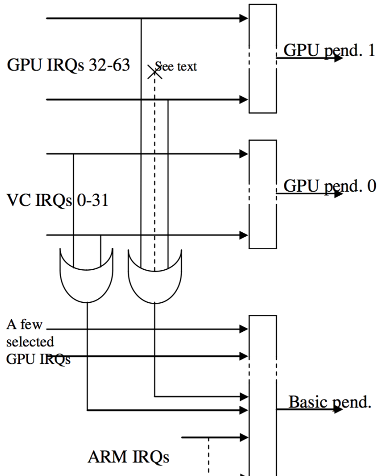

Figura 7.4: Registros de interrupciones pendientes.

pendiente en los registros `IRQ_PEND1` y `IRQ_PEND2`, lo que puede ser útil en el proceso de encuesta.

Como vemos en la Tabla 7.6, hay cuatro flags de interrupción de E/S (`gpio_int[0:3]`) para interrupciones externas generadas por los pines. El manual de Broadcom [bcm] no documenta el significado de estos flags, sin embargo puede encontrarse alguna iformación adicional en otras fuentes, como *elinux* [rpib]. Pinchando en el enlace se accede a una tabla que documenta cada uno de los pines del GPIO, y se puede comprobar que estos están organizados en tres bancos:

- Banco 0: GPIOS del 0-27, activan el flag `gpio_int[0]`
- Banco 1: GPIOS del 28-45, activan el flag `gpio_int[1]`
- Banco 2: GPIOS del 46-53, activan el flag `gpio_int[2]`

El cuarto flag (`gpio_int[3]`) se activa siempre que alguno de los otros 3 se activa (un flag para controlarlos a todos[11]).

El único dispositivo de los llamados básicos que vamos a utilizar es el *timer*. En su caso la identificación de la interrupción es sencilla, pues basta con consultar el bit 0 del registro *IRQ_BASIC*.

En cambio para los GPIOS consultar cada uno de los flags `gpio_int` del registro `IRQ_PENDING2` sólo nos diece si se ha solicitado interrupción por alguno de los pines del banco correspondiente, pero no por cuál de los GPIOs. Para distinguir el pin (o los pines) que hay que consultar además un registro del controlador GPIO, como se describe en la siguiente sección.

7.4. Uso de pines GPIO para generación de interrupciones

El controlador GPIO del BCM2835 dispone de varios detectores de eventos para cada pin configurado como entrada (ver la Figura 6.4):

[11] En realidad no es como el anillo mágico, sólo sirve para saber si alguno de los otros está en uso

- Detector de nivel bajo (0 lógico)
- Detector de nivel alto (1 lógico)
- Detector asíncrono de flanco de subida
- Detector asíncrono de flanco de bajada
- Detector síncrono de flanco de subida
- Detector síncrono de flanco de bajada

La detección síncrona de flanco permite eliminar rebotes por hardware. En este caso el circuito detector del flanco muestrea el valor del pin a intervalos constantes y sólo reconoce el flanco cuando se mantiene durante dos ciclos seguidos. Por ejemplo, el patrón `011` supondría la detección de un flanco de subida síncrono, mientras que el patrón `010` se ignoraría (se considera un flanco espúreo). Por el contrario, la detección asíncrona no utiliza un reloj para muestrear la entrada y por tanto no se filtra ningún flanco.

La Tabla 7.9 recoge los registros del controlador GPIO para la gestión de la detección de eventos. Se utiliza un registro diferente para habilitar la detección de cada evento, de modo que podemos seleccionar la detección de varios eventos en un mismo pin. Para habilitar la detección de un evento en un pin determinado, debemos poner a 1 el bit correspondiente en el registro de habilitación del evento, usando el registro 0 para los pines del 0 al 31 y el 1 para los pines del 32 al 53. Por ejemplo, si escribimos un 1 en el bit 18 del registro `GPREN0` se habilita la detección de flanco de subida síncrono en el pin GPIO18.

Tabla 7.9: Registros para detección de eventos del GPIO

	Dir. Físicas	**Descripción**
GPREN0/1	`0x2020004C/0x20200050`	Detección de Flanco de subida síncrono
GPFEN0/1	`0x20200058/0x2020005C`	Detección de Flanco de bajada síncrono
GPHEN0/1	`0x20200064/0x20200068`	Detección de Nivel alto
GPLEN0/1	`0x20200070/0x20200074`	Detección de Nivel bajo
GPAREN0/1	`0x2020007C/0x20200080`	Detección de Flanco de subida asíncrono
GPAFEN0/1	`0x20200088/0x2020008C`	Detección de Flanco de bajada asíncrono
GPEDS0/1	`0x20200040/0x20200044`	Identificación y reconocimiento de evento

El controlador activará la línea de interrupción `gpio_int[0-2]` correspondiente, según el banco en el que se encuentre el pin [rpib]. Asimismo, la línea `gpio_int[3]` se activará en cuanto se active alguna de las otras tres, como mencionamos en la sección anterior. Además, el controlador GPIO anotará la detección del evento en el registro de identificación correspondiente `GPEDS0/1`. Estos registros deben ser consultados en la rutina de tratamiento de interrupción para identificar la fuente de la interrupción, es decir, identificar el pin que ha solicitado la interrupción al procesador. Al finalizar, la rutina deberá reconocer la interrupción escribiendo un 1 en la misma posición para que el controlador retire la petición. Aunque pueda resultar contra-intuitivo, la escritura de un 1 en este registro pone a 0 el bit correspondiente, y la escritura de un 0 no modifica el valor de ese bit en el registro. Esto se hace así para permitir al software hacer modificaciones atómicas usando sólo una instrucción de store (no tiene que leer primero, modificar y luego escribir).

Por ejemplo, si hemos configurado el GPIO18 para la detección de algunos eventos, el bit 18 del `GPEDS0` se pondrá a 1 cuando se detecte alguno de estos eventos por dicho pin. Para borrar la petición de interrupción la *ISR* deberá escribir un 1 en la posición 18 del `GPEDS0`.

7.5. Interrupciones periódicas

Todos los sistemas cuentan al menos con un temporizador (*timer*) que permite generar interrupciones periódicas, fundamentales para muchas aplicaciones.

La Figura 7.5 representa un diagrama de bloques genérico de un timer común. Consiste principalmente en un contador descendente y un registro de carga, que se cargan inicialmente con un valor N. El contador decrementa este valor en uno cada vez que llega un nuevo pulso por su señal de reloj, y cuando la cuenta llega a 0 se recarga el contador con el valor almacenado en el registro de carga (N otra vez si no se ha modificado desde que se inicializó el contador) y se activa una señal de interrupción.

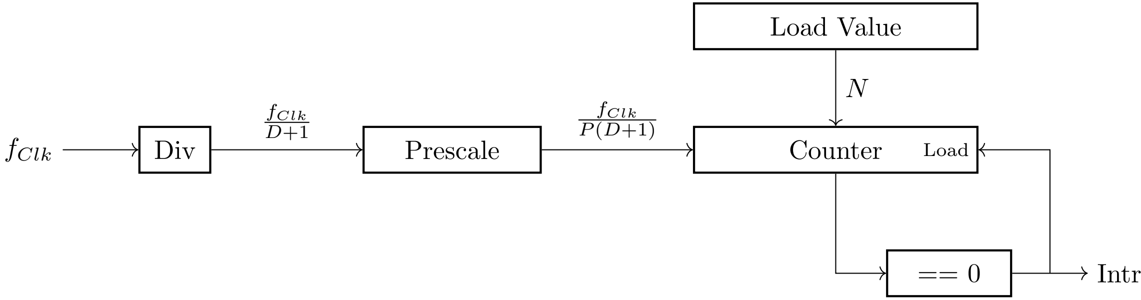

Figura 7.5: Diagrama de bloques típico de un *timer*.

El tiempo que pasa entre dos interrupciones consecutivas del timer depende del valor con el que se inicia (N) y de la señal de reloj con la que se excita, que suele construirse dividiendo una o más veces la señal de reloj del bus al que se conecta el timer. En la Figura 7.5 se utilizan dos módulos que dividen la señal de reloj conectados en serie: un módulo de división, que divide la frecuencia de su señal de entrada entre $D + 1$, y un módulo de pre-escalado que divide la frecuencia de su señal de entrada entre P. Por tanto, el tiempo entre interrupciones resulta:

$$T = \frac{N \cdot (D + 1) \cdot P}{f_{clk}} \tag{7.1}$$

Normalmente queremos generar estas interrupciones con la mayor precisión posible, por lo que buscaremos aproximarnos lo más posible al valor de T buscado. Por ejemplo, supongamos que en la Raspberry Pi queremos generar interrupciones periódicas cada 2s, para hacer parpadear un led. El BCM2835 dispone de un periférico timer basado en el SP804 [sp8], que incluye dos temporizadores:

- El timer principal cuyo funcionamiento es muy similar al descrito arriba, y permite generar interrupciones periódicas.

- El que llaman en la documentación del BCM2835 *Free Running Timer*, que es un contador que no para nunca y que no sirve para generar interrupciones periódicas.

El timer principal puede configurarse como un timer de 16 o de 32 bits [12] (N). La frecuencia de reloj de entrada es de 250MHz (no confundir con el reloj del procesador, que funciona a 700MHz), y cuenta con un módulo de pre-escalado configurable de 1, 16 o 256 (P) y un modulo divisor de 10 bits (D). Entonces, para conseguir un intervalo de 2s tendríamos:

$$2s \cdot 250MHz = N \cdot (D+1) \cdot P \quad \Longrightarrow \quad 5 \cdot 10^8 = N \cdot P \cdot (D+1)$$

Podemos empezar probando con $P = 1$ para maximizar la resolución, y buscar para $D + 1$ el mayor divisor de $5 \cdot 10^8$ tal que D sea representable con 10 bits, con el fin de obtener un valor de N no muy grande, que podamos representar con 16 o 32 bits.

Por ejemplo, si cogemos $D+1 = 10^3$, $D = 999 < 1024$. En este caso quedaría $N = 5 \cdot 10^5$, que necesita 19 bits para ser representado. Tenemos que configurar el timer para que use el contador de 32 bits.

La Tabla 7.10 describe los registros del controlador hardware del módulo timer. El registro de control permite configurar varios parámetros importantes del timer, el significado de cada uno de sus bits se describe en la Tabla 7.11.

Tabla 7.10: Registros del controlador del *timer* ARM

Registro	**Dir. Física**	**Función**
Load	`0x2000B400`	Valor de carga del timer principal
Value	`0x2000B404`	Valor actual del timer principal
Control	`0x2000B408`	Control de los dos timers
IRQ Clear/ACK	`0x2000B40C`	Borrado de interrupción
Raw IRQ	`0x2000B410`	Estado del bit de petición de interrupción
Masked IRQ	`0x2000B414`	Estado de la petición de interrupción
Reload	`0x2000B418`	Valor de recarga del timer principal
Pre-divider	`0x2000B41C`	Valor del divisor de la señal de reloj para el timer principal
Free-running counter	`0x2000B420`	Registro de cuenta del segundo contador

Cuando se escribe en el registro de carga (Load) el valor escrito se escribe también en el contador. Cuando la cuenta actual llegue a 0 el contador volverá a cargarse con el valor del registro de carga.

El timer dispone también de un registro de recarga (Reload), escribir en él es como escribir en el registro de carga pero sin actualizar directamente el valor del registro de cuenta.

El registro de valor (Value) es de sólo lectura, y permite leer el valor actual del contador.

Finalmente, el registro IRQ Clear Ack se utiliza para el reconocimiento de la interrupción. Al escribir cualquier valor en este registro, el bit de interrupción pendiente se borra.

Recordar que para que las interrupciones generadas por el *timer* lleguen al procesador, es imprescindible configurar correctamente el controlador de interrupciones escribiendo un 1 en el bit 0 del registro `IRQ_ENABLE_BASIC`.

[12] La documentación del BCM2835 dice que es de 16 ó 23 bits, sin embargo es una errata confirimada [rpia].

Tabla 7.11: Registro de control del *timer* ARM

Bits	Función
`bit 1`	Selección de contador de 16 o 32 bits
`bits 3:2`	Configuración del factor de pre-escalado (P). `00`: 1 `01`: 16 `10`: 256 `11`: 1
`bit 5`	Habilitar/deshabilitar la generación de interrupciones por el *timer*.
`bit 7`	Habilitar/deshabilitar el *timer*.
`bit 9`	Habilitar/deshabilitar el *free-running* timer.
`bits 23:16`	Factor de pre-escalado para el *free-running* timer.

7.6. Inicialización y configuración del sistema

Con lo que hemos visto hasta ahora tenemos la información fundamental para poder manejar los dispositivos más simples de la Raspberry Pi por interrupciones. Sin embargo, para hacerlo funcionar debemos configurar correctamente el sistema, que en nuestro caso consiste únicamente en lo siguiente:

- Debemos rellenar la tabla de vectores de forma adecuada. Lo que queremos es que cuando se produzca una interrupción por la línea IRQ, se termine ejecutando nuestra rutina de tratamiento de interrupción.

- Debemos configurar las pilas del modo SVC (modo en el que arranca el chip) y el modo IRQ (modo al que saltará si hay una interrupción por la línea I).

- Debemos pasar el control a la función `main`.

Aunque en nuestro laboratorio descargamos los códigos utilizando el hardware de depuración por medio de su interfaz jtag, vamos a desarrollar unos códigos que puedan arrancar automáticamente al alimentar la placa si los copiamos en la tarjeta SD de la Raspberry Pi[13]. Para ello necesitamos conocer el proceso de arranque del BCM2835. Este proceso es complejo, sobre todo porque los diseñadores del chip delegaron esta tarea en otro dispositivo integrado en el BCM2835, la unidad de procesamiento gráfico (GPU). Este proceso ha ido además evolucionando, y en su versión actual es como sigue:

1. Primera fase del `bootloader`, almacenada en una ROM interna. Carga en la cache L2 el código de la segunda fase del `bootloader`, que espera encontrar en la tarjeta SD en un fichero llamado `bootcode.bin`.

2. Segunda fase del bootloader, que realiza las siguientes acciones:

   - Habilita la SDRAM

[13] Reemplazaríamos el fichero `kernel.img` con una copia en formato `binary` de nuestro programa, renombrado como `kernel.img`

- Carga el firmware de la GPU desde la SD (fichero `start.elf`).
- Carga desde la SD el fichero `kernel.img` en la dirección `0x8000`
- Escribe en el vector de reset (0x0) una instrucción de salto a la dirección `0x8000`.
- Libera el reset sobre el ARM.

Resumiendo, en lo que a nosotros respecta, la GPU pone el valor 0x8000 en el PC, carga el fichero `kernel.img` en dicha dirección y pasa el control al ARM1176JZF-S. Por este motivo enlazaremos nuestros programas a partir de la dirección 0x8000 utilizando el siguiente script de enlazado:

```
MEMORY
{
    ram : ORIGIN = 0x8000, LENGTH = 0x1000
}

SECTIONS
{
    .text : { *(.text*) } > ram
    .data : { *(.data*) } > ram
    .bss : { *(.bss*) } > ram
}
```

que ubica las secciones de código a partir de la dirección 0x8000, y los datos debajo del código.

El Cuadro 11 muestra el código de inicialización que utilizaremos en el laboratorio. El programa comienza con una copia de la tabla de vectores, que contiene instrucciones que cargan en el PC la dirección de una ISR, extraída de una tabla que se coloca justo debajo de la tabla de vectores. Enlazaremos el código colocando esta sección `.text` la primera, con lo que la cpu ejecutará el vector de reset como primera instrucción cuando la GPU le ceda el control (o cuando descarguemos el programa a través del depurador), como si se hubiese producido un reset.

La rutina de inicio, que comienza con la etiqueta `reset`, copia estas dos tablas (vectores y direcciones de ISRs) desde la dirección 0x8000 (la dirección de `start`) a la dirección 0x0, que es la dirección base por defecto de la tabla de vectores[14]). A continuación inicializa las pilas de los modos IRQ y SVC y finalmente invoca la función `main`. Cuando se retorna de esta última función ya no hay nada que hacer, y el control de flujo se queda en un bucle infinito identificado con la etiqueta `hang`.

Todas las entradas de la tabla de ISRs se han inicializado a `hang` excepto las de `reset` e `IRQ`. Así, si se produce una excepción que no esperamos el programa se quedará de forma indefinida *colgado* en el bucle de la etiqueta `hang`. Si por el contrario se produce una interrupción por la línea IRQ se cargará en el PC la dirección de la rutina cuyo nombre sea `isr_irq`. Esta rutina debe ser definida en algún otro fichero del proyecto como símbolo global, lo que generalmente haremos declarando una función C con este nombre, declarada con el atributo `interrupt` como veremos en la siguiente sección.

[14]Hay que tener en cuenta que las instrucciones de `ldr` se van a ensamblar con direccionamientos relativos a PC. Por ejemplo, la instrucción `ldr pc,reset_handler` se ensamblará como `ldr pc, [pc, #24]`. Por tanto deben copiarse tanto la tabla de vectores como la tabla de direcciones de ISRs.

Cuadro 11 Código de inicialización del sistema

```
             .global start
             .equ SVCStack,   0x8000
             .equ IRQStack,   0x4000

             .text
start:      @ Tabla de vectores
             ldr pc,reset_handler
             ldr pc,undefined_handler
             ldr pc,svc_handler
             ldr pc,prefetch_handler
             ldr pc,data_handler
             ldr pc,unused_handler
             ldr pc,irq_handler
             ldr pc,fiq_handler

             @ Tabla de direcciones de ISRs
reset_handler:       .word reset
undefined_handler:   .word hang
svc_handler:         .word hang
prefetch_handler:    .word hang
data_handler:        .word hang
unused_handler:      .word hang
irq_handler:         .word isr_irq
fiq_handler:         .word hang

reset:       @ Rutina de inicialización (no retorna)
             ldr r0,=_start
             mov r1,#0x0000
             ldr r2,=reset     @ reset marca el fin de la copia
             @ Bucle de copia
L0:          cmp r0, r2
             beq InitStacks
             ldr r3, [r0], #4
             str r3, [r1], #4
             b L0

InitStacks: @ Inicialización de los Stacks
             @ Modificar estado (IRQ_MODE|FIQ_DIS|IRQ_DIS)
             mov r0,#0xD2
             msr cpsr_c,r0
             ldr sp, =IRQStack

             mov r0,#0xD3
             msr cpsr_c,r0
             ldr sp, =SVCStack

             mov fp,#0
             bl main
hang:        b hang
             .end
```

7.7. Ejemplo de E/S por interrupciones

Para ilustrar la idea de la gestión de la E/S por interrupciones vamos a ver cómo podríamos solucionar el mismo problema que presentamos en la Sección 6.5 pero con interrupciones en lugar de con espera activa. Es decir, queremos preparar el sistema para que podamos dejar encendido o dejar apagado el led del puerto 4 de la Gertboard, usando el pulsador del puerto 1 de la Gertboard para cambiar el estado del led, pero sin tener que muestrear el estado del pulsador. Queremos que al pulsar el pulsador se produzca una interrupción, y haremos que la *ISR* de la línea IRQ cambie el estado del led.

Comenzamos añadiendo dos macros adicionales para los registros `GPFEN0` y `GPEDS0` y otras dos para los registros del controlador de interrupciones `IRQ_PEND2` y `IRQ_ENABLE_IRQS_2`:

```
#define GPFEN0     (*(volatile unsigned *)0x20200058)
#define GPEDS0     (*(volatile unsigned *)0x20200040)

// Registros para gestión interrupciones (pp. 112 BCM2835 ARM Peripherals)
#define IRQ_PEND2           (*(volatile unsigned *)0x2000B208)
#define IRQ_ENABLE_IRQS_2   (*(volatile unsigned *)0x2000B214)
```

También tenemos que añadir la declaración y la implementación de la *ISR*, que se encargará de hacer la encuesta (en este caso sólo hay un dispositivo, con lo que simplemente comprueba que la interrupción que se ha producido es la que estamos esperando) y luego conmutará el estado del led y esperará para filtrar los posibles rebotes. Finalmente reconocerá la interrupción para que el dispositivo retire la petición:

```
// Declaración de la ISR de la línea I (IRQ)
void isr_irq(void) __attribute__ ((interrupt ("IRQ")));

// Implementación de la ISR de la línea I
void isr_irq (void)
{
    // Interrupción gpio_int[3] pendiente
    if (IRQ_PEND2 & (0x8 << 17)) {
        //Interrupción del GPIO23
        if (GPEDS0 & (0x1 << 23)) {
            //Tratamos la interrupción del botón
            led_switch();
            short_wait();
            //Reconocemos la interrupción
            GPEDS0 = 0x1 << 23;
        }
    }
}
```

Modificaremos también la rutina de `setup` para que habilite la detección de flanco de bajada en el pin GPIO23, habilite las interrupciones por las líneas `gpio_int[0-3]` y ponga a 0 el bit I del CPSR (esto lo hacemos por medio de la función `enable_irq` implementada en ensamblador). Finalmente, simplificaríamos la función `main` eliminando la espera sobre el pulsador:

```
void setup_gpio()
{
    // Configurar GPIO 23 como INPUT (000) --> Boton
    GPFSEL2 = GPFSEL2 & ~(0x7 << 3*3);

    // Configurar GPIO 17 como OUTPUT (001) --> LED
    GPFSEL1 = (GPFSEL1 & ~(0x7 << 7*3)) | (0x1 << 7*3);

    // Activar detección de flanco de bajada para GPIO 23
    GPFEN0 = (0x1 << 23);

    // Activar el pull-up del GPIO 23
    GPPUD     = 0x2;
    short_wait();
    GPPUDCLK0 = (0x1 << 23);
    short_wait();
    GPPUD     = 0;
    GPPUDCLK0 = 0;

    // Habilitar las interrupciones gpio_int[0-3]
    IRQ_ENABLE_IRQS_2 = (0xF << 17);

    // Habilitar las irq en el cpsr (codigo ensamblador)
    enable_irq();
}

int main (void)
{
    setup_gpio();

    led_off();

    while (1);

    return 0;
}
```

Este programa haría lo mismo que el programa visto en la Sección 6.5 pero sin tener ocupado al procesador esperando por pulsaciones. En este programa esto queda un poco deslucido porque el programa no tiene nada más que hacer, pero en un programa más complejo esto no será así. En este caso además, la *ISR* ha resultado muy simple puesto que sólo había un dispositivo que atender. En el siguiente apartado vamos a ver qué modificaciones debemos hacer a nuestro programa para añadir un nuevo dispositivo tratado también por interrupciones, tomando como ejemplo el timer.

7.7.1. Añadiendo interrupciones periódicas

Supongamos que queremos añadir como funcionalidad a nuestro programa que haga que el led del puerto 5 de la raspberry parpadee con un intervalo de 2s. Podemos generar interrupciones periódicas utilizando el timer, con los valores que mencionamos en la Sección 7.5. Los cambios que debemos hacer sobre nuestro programa son los siguientes:

- Añadir macros para los nuevos registros (timer y controlador de interrupciones) y otras macros auxiliares.

```
// Registros ARM Timer (pp. 196 BCM2835 ARM Peripherals y Manual SP804)
#define ARM_TIMER_LOD   (*(volatile unsigned *)0x2000B400)
#define ARM_TIMER_CTL   (*(volatile unsigned *)0x2000B408)
#define ARM_TIMER_CLI   (*(volatile unsigned *)0x2000B40C)
#define ARM_TIMER_DIV   (*(volatile unsigned *)0x2000B41C)

// Macros ARM Timer (pp. 196 BCM2835 ARM Peripherals y Manual SP804)
#define ARM_TIMER_ENABLE        (1<<7)
#define ARM_TIMER_IRQ_ENABLE    (1<<5)
#define ARM_TIMER_32b           (1<<1)
```

- En la función de setup, configurar otro GPIO como salida para poder controlar otro led. Por ejemplo podemos configurar el GPIO15 como salida. Además deberemos conectar este pin en la Gertboard a la entrada del circuito del puerto 5:

```
    // Configurar GPIO 15 como OUTPUT (001) --> LED
    GPFSEL1 = (GPFSEL1 & ~(0x7 << 5*3)) | (0x1 << 5*3);
```

  y configurar también el timer para que genere interrupciones periódicas cada 2s:

```
    // Setup del timer
    ARM_TIMER_LOD    = 500000; // N
    ARM_TIMER_DIV    = 999;    // D
    ARM_TIMER_CLI    = 0;
    // Pre-escalado 1, contador de 32 bits, habilitamos timer e interrupciones
    ARM_TIMER_CTL    = ARM_TIMER_ENABLE | ARM_TIMER_IRQ_ENABLE | ARM_TIMER_32b;
    // Habilitar las interrupciones del timer
    IRQ_ENABLE_BASIC = 0x1;
```

- Adaptar la *ISR* del timer para que haga una encuesta comprobando si la interrupción es del timer o del pulsador, atienda al dispositivo correcto y borre su flag de interrupción:

```
// Implementación de la ISR de la línea I
void isr_irq (void)
{
    // Encuesta, damos prioridad al timer
    if (IRQ_BASIC & 0x1){
        //Interrupción del timer
        led2_switch();
        //Reconocemos la interrupción
        ARM_TIMER_CLI = 0x1;
```

```
    } else if (IRQ_PEND2 & (0x8 << 17)) {
        if (GPEDS0 & (0x1 << 23)) {
            //Interrupción del GPIO23
            led1_switch();
            short_wait();
            //Reconocemos la interrupción
            GPEDS0 = 0x1 << 23;
        }
    }
}
```

- Preparar funciones de abstracción para manejar un segundo led por este nuevo puerto:

```
int led2; //estado del led

void led2_on(void)
{
    GPSET0 = 0x1 << 15;
    led2 = LED_ON;
}

void led2_off(void)
{
    GPCLR0 = 0x1 << 15;
    led2    = LED_OFF;
}

void led2_switch(void)
{
    if (led2 == LED_OFF)
        led2_on();
    else
        led2_off();
}
```

Bibliografía

[aap] The arm architecture procedure call standard. Disponible en http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0042d/index.html.Hayunacopiaenelcampusvirtual.

[arm] Arm-v6m architecture reference manual. Disponible en http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.subset.architecture.reference/index.html.

[bcm] Bcm2835 arm peripherals. Accesible en Campus Virtual.

[ger] Gertboard user manuel (rev 2.0). Disponible en http://www.element14.com/community/servlet/JiveServlet/downloadBody/48860-102-3-256002/Gertboard_User_Manual_Rev_1%200_F.pdf.

[ld-] Ld gnu linker manual. Disponible en http://sourceware.org/binutils/docs-2.23.1/ld/index.html.

[rpia] Elinux, bcm2835 datasheet errata. Se puede consultar en https://elinux.org/BCM2835_datasheet_errata.

[rpib] Elinux documentation for the raspberry pi gpios. Disponible en https://elinux.org/RPi_BCM2835_GPIOs.

[sp8] Arm dual-timer module (sp804). reference manual. Disponible en http://infocenter.arm.com/help/topic/com.arm.doc.ddi0271d/DDI0271.pdf.