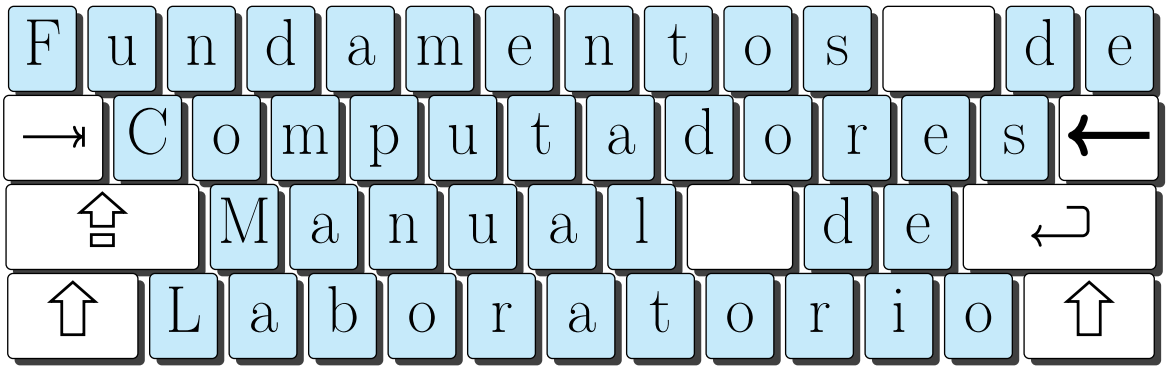


Christian Tenllado
tenllado@ucm.es

Luis Piñuel
lpinuel@ucm.es

Departmento de
Arquitectura de Computadores
y Automática

Índice general

Prólogo

Este manual se ha diseñado como soporte para el laboratorio de segundo cuatrimestre de la asignatura de Fundamentos de Computadores (FC), impartida en todas las titulaciones de la Facultad de Informática de la UCM. El manual deriva del trabajo desarrollado por los autores y colaboradores, a lo largo de los últimos 7 años, en las asignaturas relacionadas con el Laboratorio de Estructura de Computadores.

La asignatura introductoria de FC presenta al alumno una visión global e integradora del funcionamiento de un computador. En la primera mitad de la asignatura se introducen los principios básicos de la electrónica digital y el diseño lógico, que permiten construir máquinas de estado y algorítmicas sencillas.

En la segunda mitad de la asignatura se presenta el modelo de computador von Neumann y se analiza un posible diseño de procesador básico con subsistemas de memoria y entrada/salida simplificados. En esta segunda parte, el laboratorio se centra en estudiar en detalle el modelo de máquina ofrecido al programador. 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 y le resulte así más fácil corregirlo.

Para montar este laboratorio los profesores escogimos como plataforma experimental una placa de prototipado de Embest, basada en un Sistema en Chip de SAMSUNG con procesador ARM7TDMI. La selección de la familia ARM se debe principalmente a la sencillez de su repertorio de instrucciones RISC y al enorme éxito que tenía ya dicha familia en el mercado de los sistemas empotrados en el momento de la compra, y que hoy en día se ha extendidos a otros ámbitos, incluyendo los sistemas de altas prestaciones. Cuando montamos los laboratorios esta placa ofrecía una buena relación calidad/precio, sin embargo hoy en día podemos encontrar otras placas en el mercado más económicas y versátiles, con procesadores mucho mejores. Adaptar estas prácticas a esas nuevas placas es prácticamente trivial, excepto para la última práctica, donde se requiere algo más de conocimiento y el manejo de la documentación de la placa objetivo.

El manual se ha organizado en cinco prácticas que el alumno debe hacer en orden. Cada práctica presenta los objetivos que persigue e introduce algunos conceptos teóricos nuevos que el alumno debe asimilar para realizar con éxito la práctica. La última sección de cada guión explica al alumno lo que debe hacer para completar la práctica.

Todas las prácticas excepto la última pueden realizarse sobre simulador, sin necesidad por tanto de tener ningún equipamiento de laboratorio. En la última práctica sin embargo, será necesario utilizar la placa de prototipado, puesto que se realiza un programa para manejar algunos dispositivos de entrada/salida, con el fin de que el alumno comprenda los mecanismos básicos por los que el computador puede comunicarse con el resto del mundo.

Finalmente, los autores queremos agradecer la colaboración de muchos profesores del Departamento de Arquitectura de Computadores y Automática, que han contribuido con sus comentarios y revisiones a este documento, especialmente a Román Hermida Correa, Daniel A. Chaver Martínez, Segundo Esteban San Román y José Ignacio Gómez Pérez.

Práctica 1

Descubriendo el Entorno de Trabajo

1.1. Objetivos

Este laboratorio está pensado como un refuerzo de la asignatura de Fundamentos de Computadores. Permitirá al alumno afianzar sus conocimiento sobre del funcionamiento de un computador con arquitectura Von Neumann. Utilizaremos como componente principal un procesador de ARM, concretamente el ARM7TDMI. En esta primera práctica introduciremos los conceptos básicos de la programación en ensamblador y trataremos de familiarizarnos con el entorno de desarrollo y la arquitectura de la cpu utilizada. En concreto, los objetivos que perseguimos en esta práctica son:

- Adquirir práctica en el manejo del repertorio de instrucciones ARM, incluyendo saltos y accesos a memoria.
- Familiarizarse con el entorno de desarrollo y las herramientas GNU para ARM, ensamblador, enlazador y depurador esencialmente.
- Comprender la estructura de un programa en ensamblador y el proceso de generación de un binario a partir de éste.

1.2. Introducción al funcionamiento de un computador

Un computador, como máquina electrónica que es, sólo entiende señales eléctricas, lo que en electrónica digital se corresponde con apagado y encendido. Por lo tanto, el alfabeto capaz de ser comprendido por un computador se corresponde con dos dígitos: el 0 y el 1 (alfabeto binario).

Las órdenes que queramos proporcionar al computador serán un conjunto de 0s y 1s con un significado conocido de antemano, que el computador podrá decodificar para realizar su funcionalidad. El nombre para una orden individual es instrucción. Programar un computador a base de 0s y 1s (lenguaje máquina) es un trabajo muy laborioso y poco gratificante. Por lo que 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. 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. Si se puede escribir un programa que traduzca órdenes sencillas

`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

(lenguaje ensamblador) a ceros y unos (lenguaje máquina), ¿qué impide escribir un programa que traduzca de una notación de 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.

1.3. Introducción a la Arquitectura ARM

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 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 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.

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 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.

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)*, *reservados* (no se usan en ARM v4) y *control (c)*, como ilustra la Figura 1.1. 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. La Tabla 1.2 describe el significado de cada uno de los bits de estos campos. Los que bits están reservados para uso futuro no son modificables y siempre se leen como cero. Los indicadores de condición, bits N, Z, C y V, son modificables en modo usuario, mientras que los bits I, F y M sólo son modificables en los modos privilegiados.

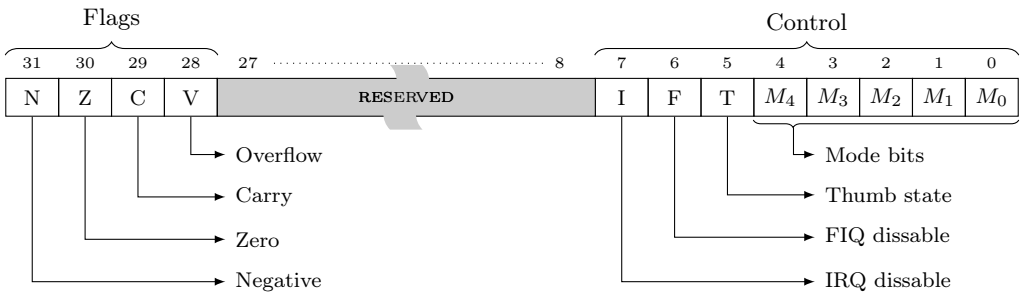


Figura 1.1: Registro de Estado del ARM v4 (CPSR).

Tabla 1.2: Descripción del campo de flags del CPSR.

Bit	**Significado**
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 las operaciones de desplazamiento, toma el valor del bit saliente.
V	En el caso de una suma o una resta, V = 1 indica que hubo un *overflow*.

1.4. Repertorio de 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.

Como es habitual, dividiremos las instrucciones en seis grupos:

- Aritmético-lógicas
- Multiplicación
- Acceso a Memoria
- Salto

1.4.1. Instrucciones aritmético-lógicas.

En este apartado veremos las instrucciones que usan la unidad aritmético-lógica (ALU) del procesador. No incluimos aquí las instrucciones de multiplicación, que veremos en el siguiente apartado, debido a que se ejecutan en otro módulo.

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 1.3
- `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, denominado *shifter_ operand*. Es muy versátil, pero de momento nos conformaremos con dos posibles modos de direccionamiento: un registro o un inmediato.

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 1.3 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 1.3: 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)

Ejemplos

```
ADD   R0, R1, #1      @ R0 = R1 + 1
ADD   R0, R1, R2      @ R0 = R1 + R2
BIC   R4, #0x05       @ Borra los bits 0 y 2 de R4
MOV   R11,#0          @ Escribe cero en R11
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
```

1.4.2. 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. Las principales variantes se describen en la Tabla 1.4. 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).

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
```

Tabla 1.4: 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.

1.4.3. 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. En ARMv4 existen varios modos de direccionamiento para las instrucciones `ldr`/`str`, sin embargo, en esta práctica nos centraremos sólo en los dos más utilizados:

- **Indirecto de registro** 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 )
  ```

  En la Figura 1.2 se ilustra el resultado de realizar la operación `ldr r2,[r1]`.

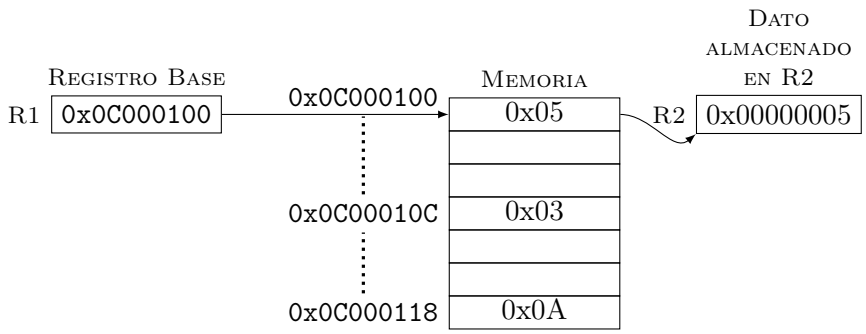


Figura 1.2: Ejemplo de ejecución de la instrucción `ldr r2,[r1]`.

- **Indirecto de registro con desplazamiento inmediato** 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 )
```

En la Figura 1.3 se ilustra el resultado de realizar la operación `ldr r2,[r1,#12]`.

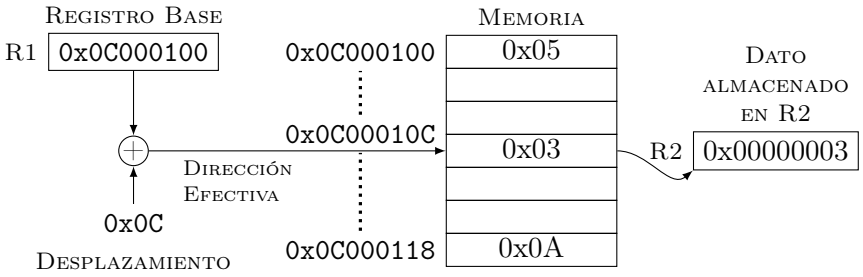


Figura 1.3: Ejemplo de ejecución de la instrucción `ldr r2,[r1,#12]`.

Ejemplos

```
LDR   R1,  [R0]          @ Carga en R1 lo que hay en Memoria(R0)
LDR   R8,  [R3, #4]      @ Carga en R8 lo que hay en Memoria(R3 + 4)
LDR   R12, [R13, #-4]    @ Carga en R12 lo que hay en Memoria(R13 - 4)
STR   R2,  [R1, #0x100]  @ Almacena en Memoria(R1 + 256) lo que hay en R2
```

1.4.4. Instrucciones de Salto

La instrucción explícita de salto es Branch (`B`). La distancia a saltar está limitada por los 24 bits con los que se puede codificar el desplazamiento dentro de la instrucción (operando 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 1.5 (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. Sin embargo, como veremos a la hora de programar en ensamblador utilizaremos etiquetas y serán el ensamblador y el enlazador los encargados de calcular el valor exacto del desplazamiento.

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 instrucciones aritmeticológicas), pero 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 escribiendo una dirección absoluta en el contador de programa, por ejemplo con la instrucción `MOV`.

Ejemplos

Tabla 1.5: Condiciones asociadas a las instrucciones.

Mnemotécnico	**Descripción**	**Flags**
EQ	Igual	Z=1
NE	Distinto	Z=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
(vacío)	Siempre (incondicional)	

```
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
```

1.4.5. Estructuras de control de flujo en ensamblador

Con las instrucciones presentadas hasta ahora podemos implementar cualquier algoritmo. Resulta conveniente pararse un momento a pensar cómo podemos codificar las estructuras básicas de control de flujo que hemos aprendido en los cursos de programación estructurada. Si lo hacemos nos daremos cuenta de que existen varias maneras de implementar cada una de ellas. La tabla 1.6 presenta algunos ejemplos, con el fin de que sirvan de guía al alumno.

1.5. Estructura de un programa en lenguaje ensamblador

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 de del repertorio ARM. Como el 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 definidos 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 cambiar a mano los valores de entrada escritos en el fichero. Para comprobar que el programa fun-

Tabla 1.6: Ejemplos de implementaciones de algunas estructuras de control habituales.

Pseudocódigo	Ensamblador

`if (R1 > R2)`
`  R3 = R4 + R5;`
`else`
`  R3 = R4 - R5`
`sigue la ejecución normal`
	`  CMP R1, R2`
	`  BLE else`
	`  ADD R3, R4, R5`
	`  B fin_if`
	`else:  SUB R3, R4, R5`
	`fin_if: sigue la ejecución normal`

`for (i=0; i<8; i++) {`
`  R3 = R1 + R2;`
`}`
`sigue la ejecución normal`
	`  MOV R0, #0  @R0 actúa como índice i`
	`for:  CMP R0, #8`
	`  BGE fin_for`
	`  ADD R3, R1, R2`
	`  ADD R0, R0, #1`
	`  B for`
	`fin_for: sigue la ejecución normal`

`do {`
`  R3 = R1 + R2;`
`  i = i + 1;`
`} while( i != 8 )`
`sigue la ejecución normal`
	`  MOV R0, #0  @R0 actúa como índice i`
	`do:  ADD R3, R1, R2`
	`  ADD R0, R0, #1`
	`  CMP R0, #8`
	`  BNE do`
	`sigue la ejecución normal`

`while (R1 < R2)`
`{`
`  R2= R2-R3;`
`}`
`sigue la ejecución normal`
	`while:  CMP R1, R2`
	`  BGE fin_w`
	`  SUB R2, R2, R3`
	`  B while`
	`fin_w: sigue la ejecución normal`

ciona 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

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
```

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 la sección 1.4, ver figura 1.5). 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 inicializar 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 1.4). 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. Las directivas 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 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 siguiente tabla:

En la Tabla 1.7 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

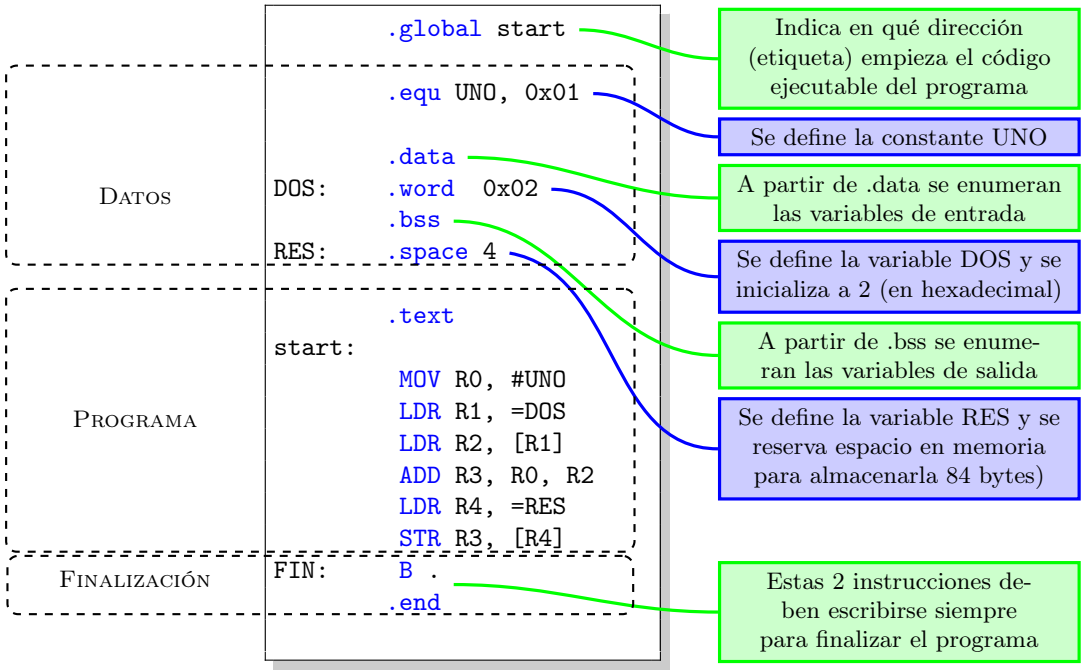


Figura 1.4: Descripción de la estructura de un programa en ensamblador: datos.

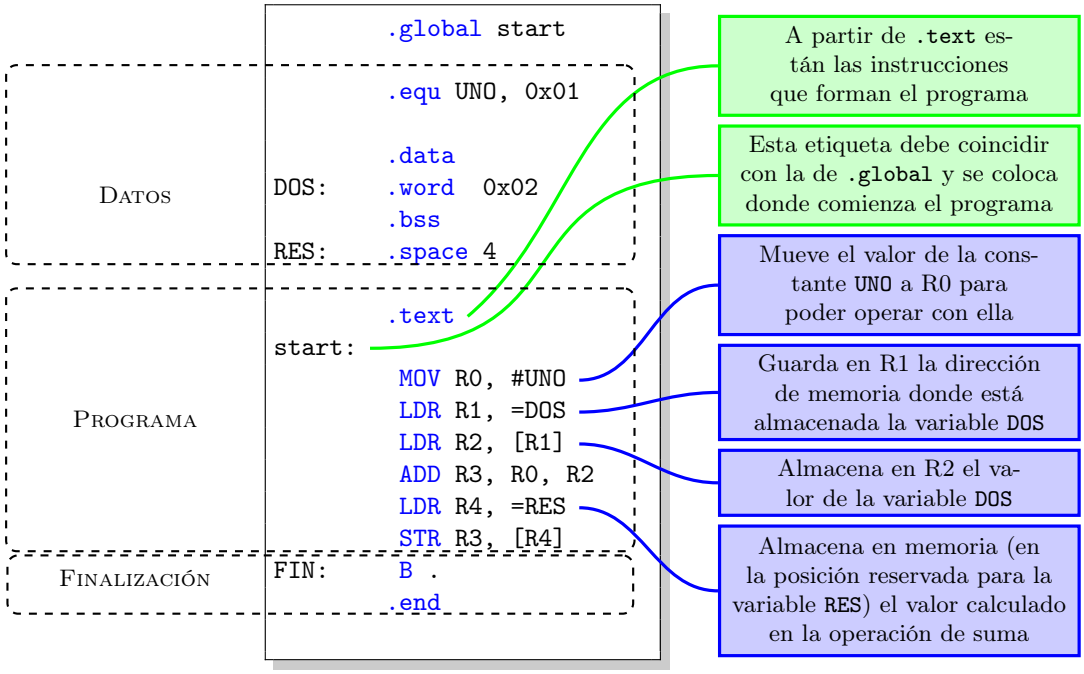


Figura 1.5: Descripción de la estructura de un programa en ensamblador: instrucciones.

Tabla 1.7: 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 .`		

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.

1.6. Introducción al entorno de desarrollo

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 sobre un simulador de la arquitectura ARM o depurar en circuito sobre una placa con un procesador de ARM.

Eclipse nos ofrece principalmente la interfaz gráfica y la gestión de los proyectos. Para realizar el resto de tareas hace uso de otras herramientas de GNU: el ensamblador (`as`), el compilador (`gcc`), el enlazador (`ld`) y el depurador (`gdb`). Además, debemos tener en cuenta que desarrollamos sobre un PC para un entorno con procesador ARM, y por tanto necesitamos hacer *compilación cruzada*. Para distinguir 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 también 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.

1.6.1. 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*. Para ello seguiremos los siguientes pasos:

1. Abrimos Eclipse haciendo doble click sobre el icono del escritorio.

2. Al iniciarse la aplicación nos mostrará una ventana de selección de `workspace`, como la que la Figura 1.6. 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 la carpeta `C:\\hlocal`. Es importante también que siempre guardemos una copia de nuestro `workspace` en un lugar seguro, porque el directorio `hlocal` es local al puesto y común para todos los usuarios.

3. 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 1.7.

4. Debemos cerrar la pestaña `Wellcome` haciendo click en la cruz de la pestaña. Entonces la ventana quedará como la que muestra la Figura 1.8, es la perspectiva C/C++ de Eclipse, que está organizada de la siguiente manera:

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

5. Para crear el proyecto seleccionamos `File`→`New`→`C Project`, con lo que se abrirá una ventana como la de la Figura 1.9. Como indica la figura, seleccionamos las opciones `ARM Cross Target Application`, `Empty Project` y ARM GCC (Sourcery G++ Lite). Elegimos el nombre del proyecto y pulsamos `Finish`. Tendremos el proyecto vacío visible en el explorador de proyectos.

6. 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 1.10. 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 2.

7. 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 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 1.11, seleccionamos el proyecto y ponemos `ld_script.ld` como nombre del fichero. Una vez creado, lo abrimos en el editor y copiamos el contenido del cuadro 3.

8. Finalmente 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 1.12. En esta ventana seleccionamos `C/C++ Build`→`Settings` y

- Debemos comprobar que en `Target Processor` está seleccionado `arm7tdmi` como procesador, no están marcadas ninguna de las casillas `Thumb*` y está seleccionada la opción `Little Endian`.

- Debemos seleccionar `ARM Sourcery GCC C Linker`→`General`, y en la casilla `Script file (-T)` debemos escribir la ruta al fichero `ld_script.ld` que hemos añadido al fichero. Podemos seleccionarlo gráficamente pulsando el botón `Browse`.

9. Compilamos 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`.

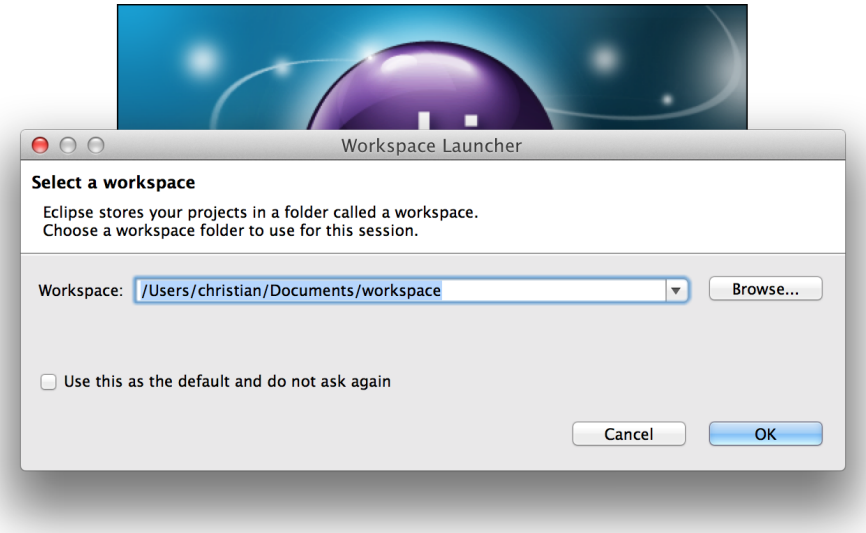


Figura 1.6: Ventana de selección de `workspace`.

Figura 1.7: Ventana de eclipse al abrir un `workspace` vacío.

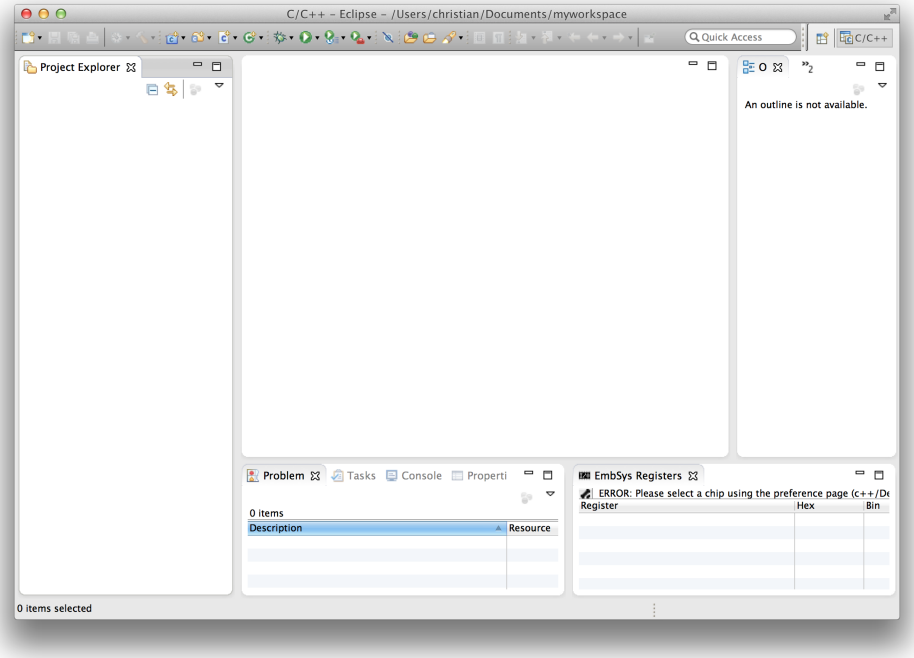


Figura 1.8: Ventana de eclipse con la perspectiva C/C++ sin proyectos.

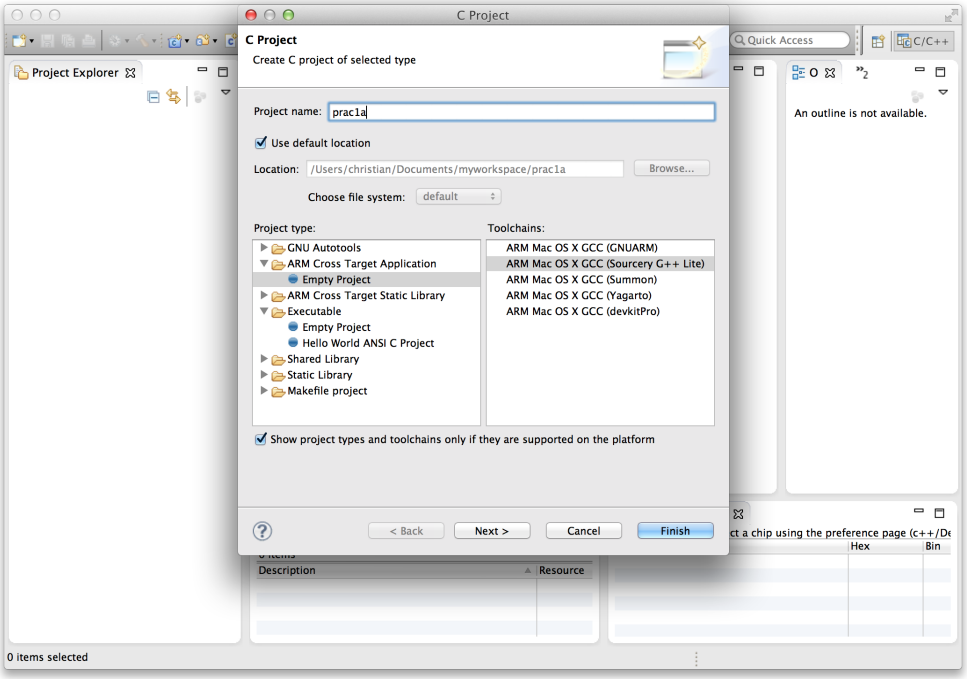


Figura 1.9: Ventana de creación de C project de tipo GNU ARM.

Cuadro 2 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 else
        STR R1, [R5]
        B FIN
else:   STR R2, [R5]
FIN:    B .
        .end
```

Cuadro 3 Script de enlazado.

```
SECTIONS
{
    . = 0x0C000000;
    .data : {
      *(.data)
      *(.rodata)
    }
    .bss : {
      *(.bss)
      *(COMMON)
    }
    .text : {
      *(.text)
    }
    PROVIDE(end = .);
    PROVIDE(_stack = 0x0C7FF000 );
}
```

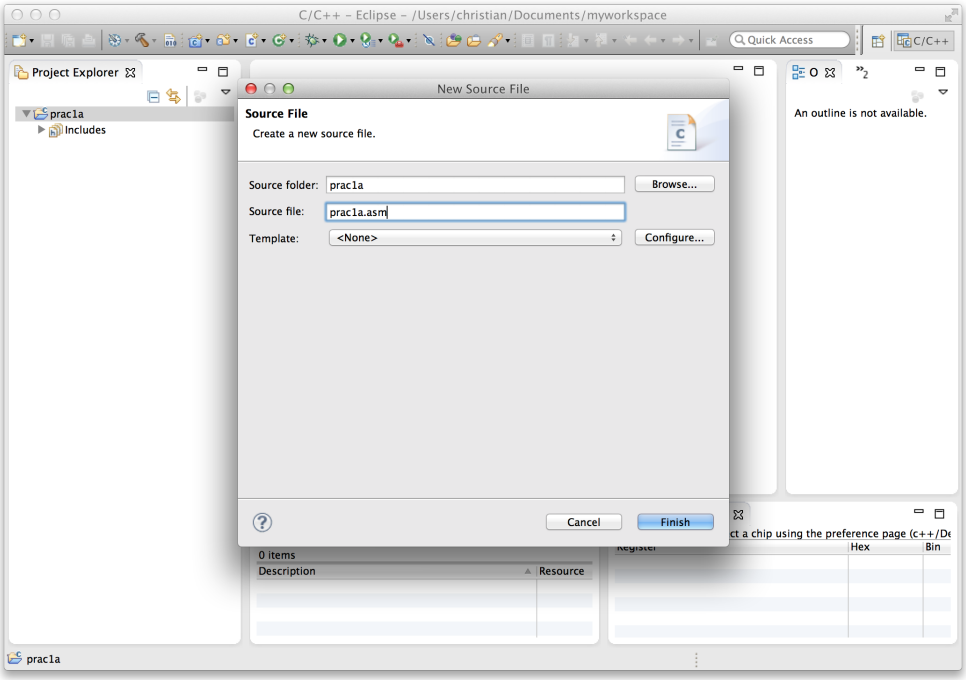


Figura 1.10: Ventana de creación de nuevo fichero fuente.

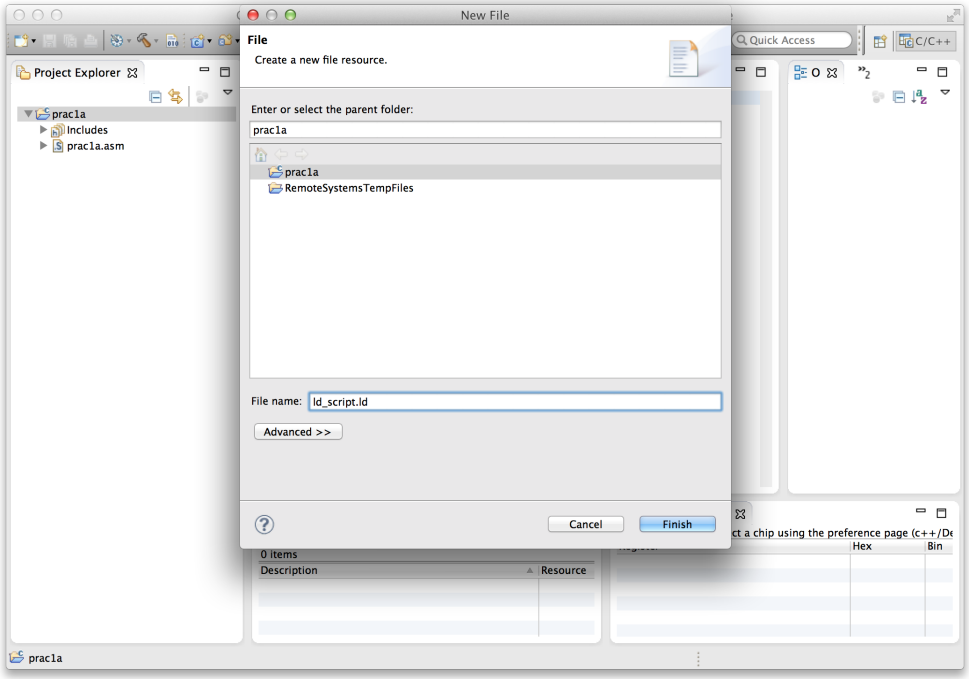


Figura 1.11: Ventana para añadir un nuevo fichero al proyecto.

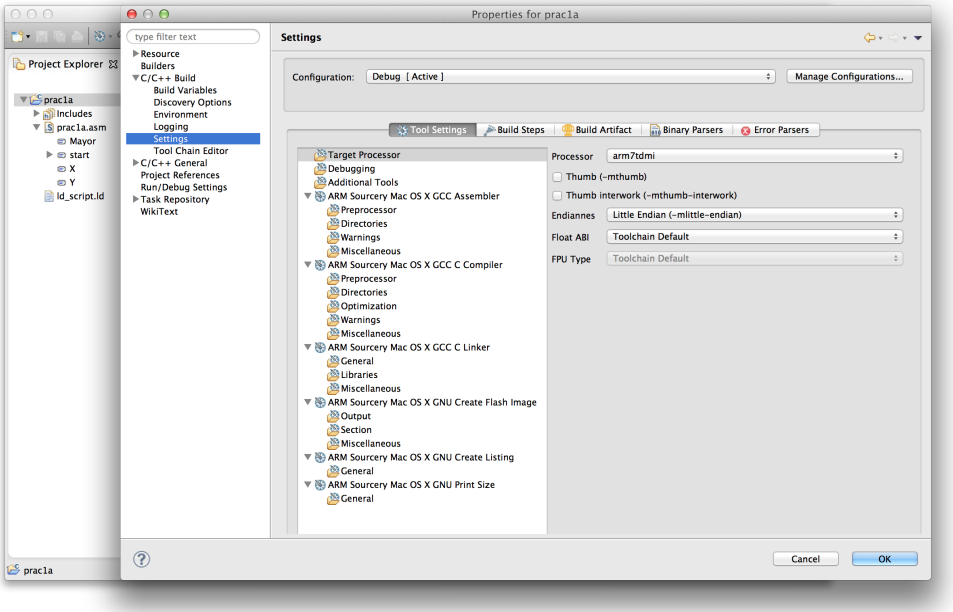


Figura 1.12: Ventana de propiedades (properties) del proyecto.

1.6.2. Depuración sobre simulador

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 no vamos a utilizar un procesador ARM real, configuraremos el depurador para que utilice un simulador.

Hay dos plugins de Eclipse que nos permiten hacer la depuración con GDB sobre el ARM: `GDB Hardware Debugging` y `Zylin Embedded debug`. El primero resulta más sencillo para la depuración en circuito, mientras que el segunda es más sencillo para utilizar con el simulador. Por ello en estas primeras prácticas usaremos `Zylin`.

Para depurar nuestro proyecto seguimos los siguientes pasos:

1. 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 1.13. 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`.

2. Creamos una configuración de depuración para el proyecto usando el plugin `Zylin`. Para ello seleccionamos `Run`→`Debug configurations...`, y se abrirá una ventana como la mostrada en la Figura 1.14. En el panel izquierdo seleccionamos `Zylin Embedded debug (Native)` 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 1.15. Ahora debemos rellenar correctamente las siguientes pestañas de la configuración:

   - 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 poner el nombre del depurador en la entrada `GDB debugger`: `arm-none-eabi-gdb`. Además de esto, en la parte superior podemos colocar un breakpoint temporal en donde nosotros queramos. Vamos a colocarlo en la dirección del símbolo `start`, para ello escribimos `*start` en el cuadro `Stop on startup at:`. El resultado se muestra en la Figura 5.9.

- Commands: en esta pestaña damos los comandos que gdb debe ejecutar en la inicialización al iniciar la depuración. En el cuadro `Initialize commands` escribimos `target sim`. Esto indica que el objetivo de depuración es el simulador. En el cuadro `Run commands` escribimos:

  ```
  load
  run
  ```

  Esto indica al depurador que cargue el fichero seleccionado el la pestaña principal, leyendo de éste los símbolos de depuración. La ventana resultante se muestra en la Figura 1.17.

3. Ya estamos listos para depurar, para ello hacemos click en el botón `Debug`. Esto guardará la configuración de depuración con el nombre seleccionado en la primera pestaña e iniciará una sesión de depuración con esta configuración. Si más adelante queremos volver a depurar este proyecto no será necesario crear una configuración de depuración, bastará con seleccionar la que acabamos de crear. La Figura 1.18 nos muestra el estado inicial que debería tener la ventana en depuración si hemos seguido todos los pasos:

   - En la parte izquierda del panel central debemos encontrar el código del cuadro 2, 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 simulada 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.

   Notemos la diferencia con el anterior. En el panel 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 de load en ARM [1]. 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 últimas prácticas.

4. Antes de simular 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 1.19 en la que debemos

[1] 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.

escribir la dirección a partir de la que queremos monitorizar la memoria. Si ponemos la dirección de descarga, `0x0C000000`, en la que hemos colocado la sección `.data`, seguida en orden por las secciones `.bss` y `.text`, el aspecto de la ventana resultante debe ser similar al mostrado en la Figura 1.20.

5. Ahora, para simular todo el 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 1.21.

6. 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 simulación pulsando el botón `Terminate`, . Se habilitará el botón `Remove all terminated launches` que nos permitirá limpiar el panel `Debug` (a veces no se habilita este botón, entonces deberemos pinchar con el botón derecho del ratón en la sesión de depuración y seleccionar `Terminate and Remove`).
   - 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/simular paso a paso tenemos las siguientes opciones:
     - 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 1.22 muestra un ejemplo de una sesión de depuración, donde hemos ido ejecutando paso a paso hasta la instrucción siguiente a `BLE else`. Podemos ver que la condición no se cumple y que se pasa a ejecutar el bloque `else`. Podemos ver el estado de los registros en este momento, y como el `PC` contine la dirección `0x0C000030` 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.

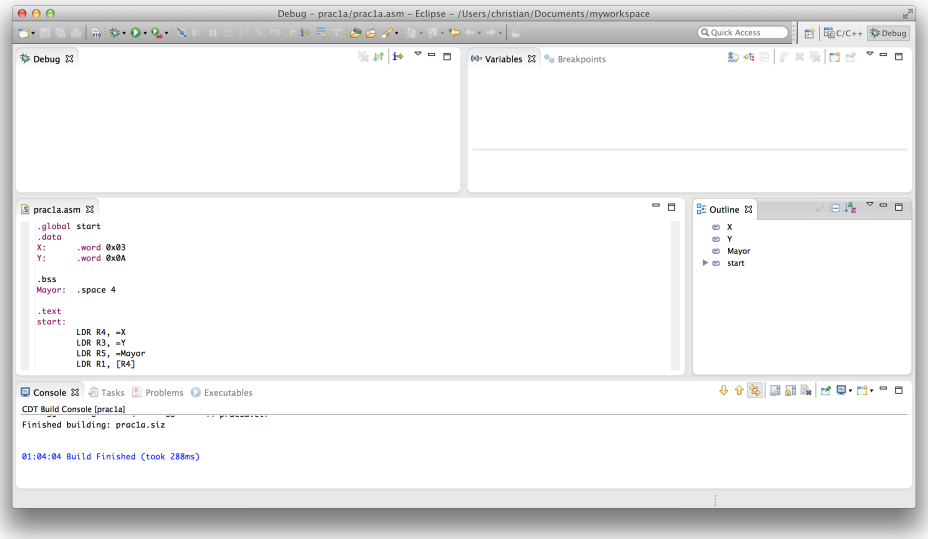


Figura 1.13: Perspectiva de depuración.

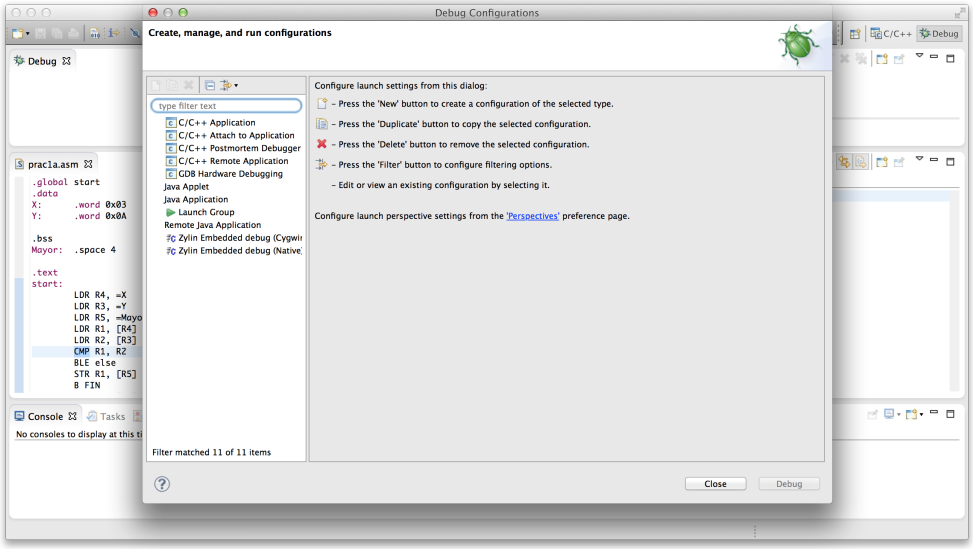


Figura 1.14: Ventana de configuraciones de depuración.

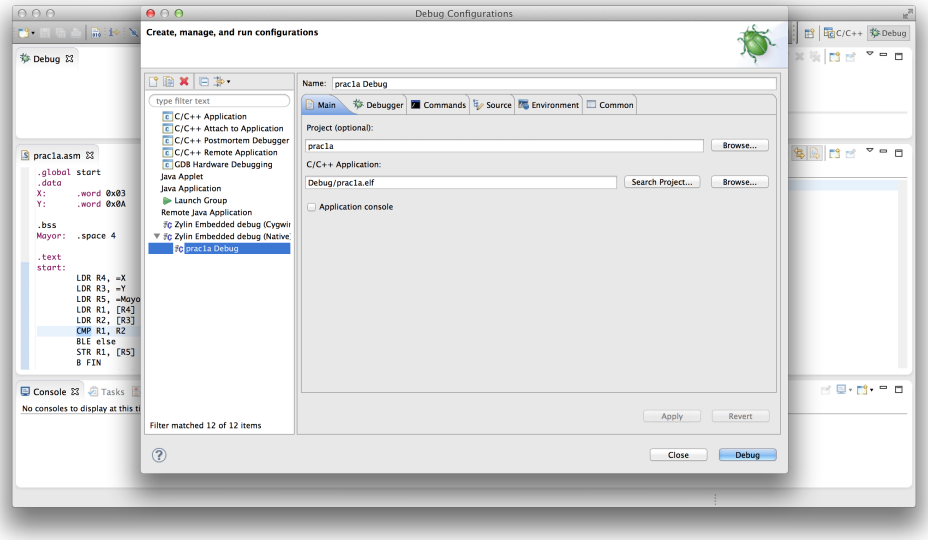


Figura 1.15: Ventana de creación de una nueva configuración Zylin nativa para el proyecto.

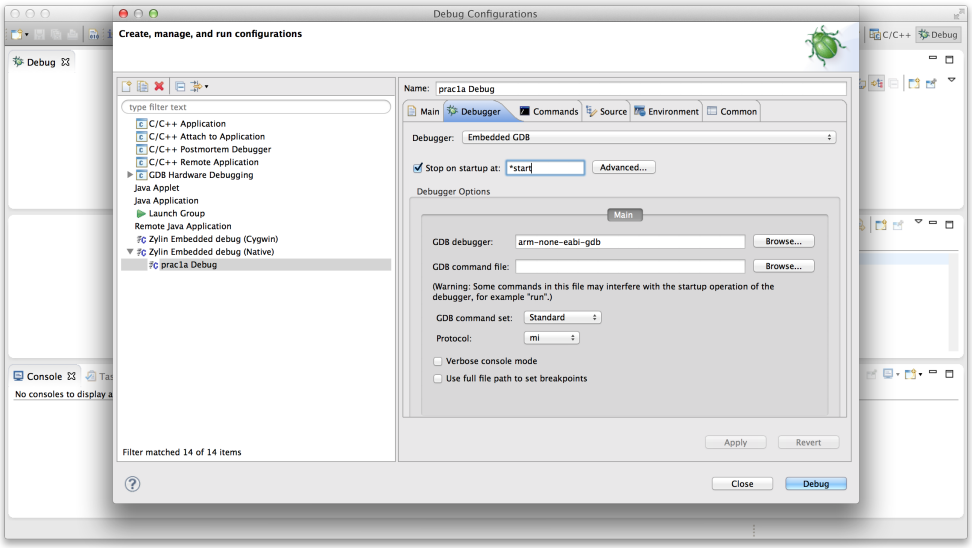


Figura 1.16: Pestaña `debugger` de la configuración de depuración Zylin.

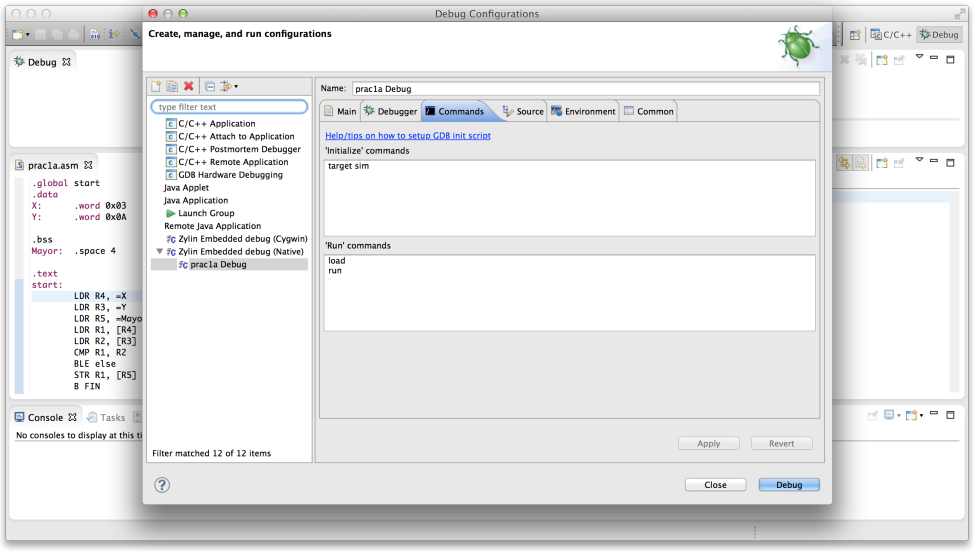


Figura 1.17: Pestaña `commands` de la configuración de depuración Zylin.

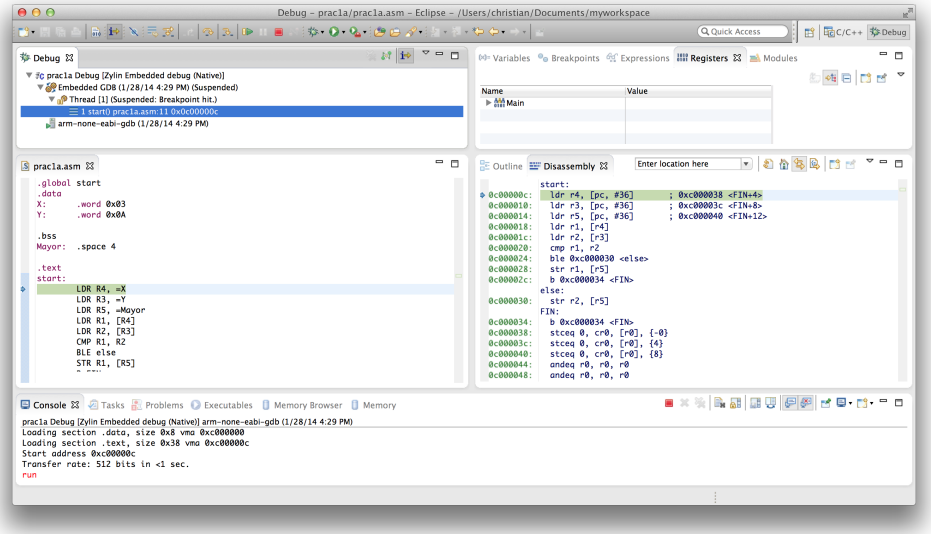


Figura 1.18: Aspecto inicial de la ventana de depuración.

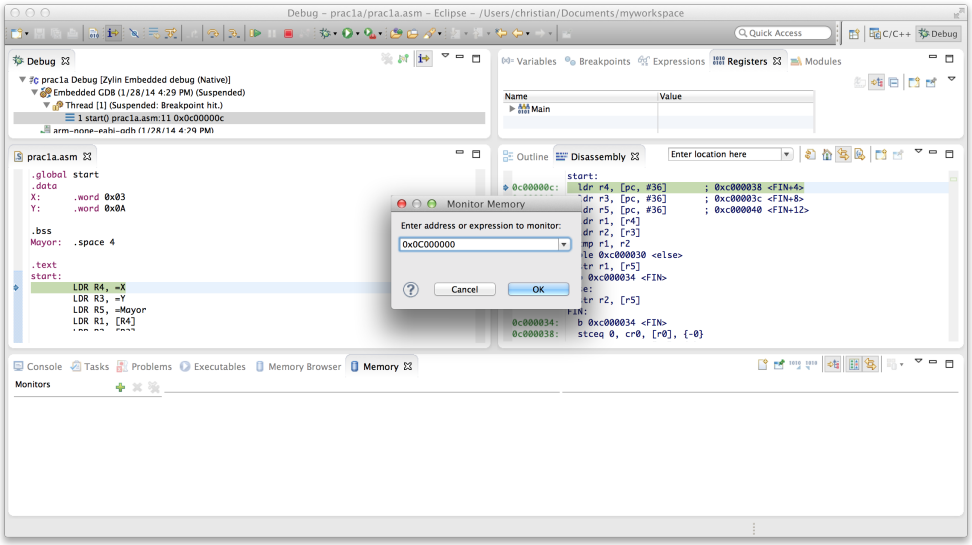


Figura 1.19: Ventana para indicar la dirección del monitor de memoria.

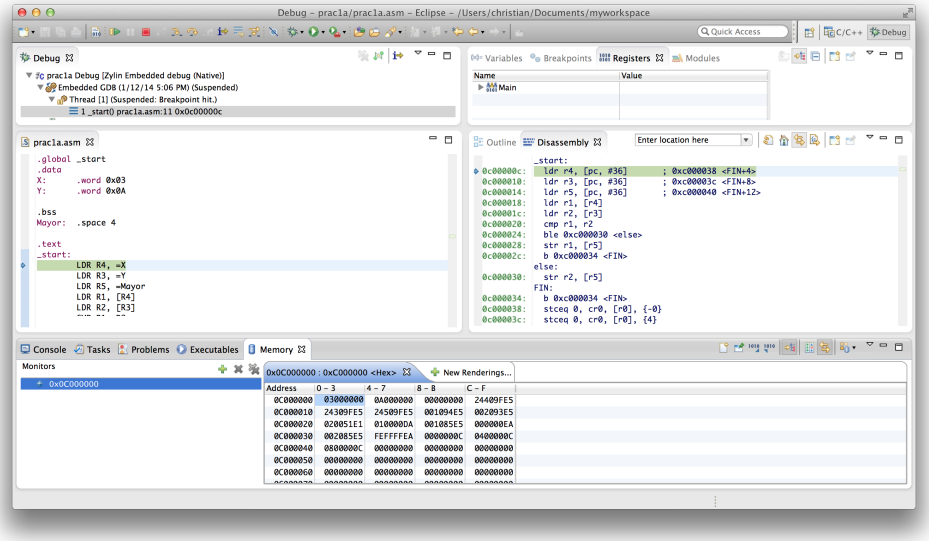


Figura 1.20: Ventana con el aspecto del monitor de memoria en la dirección `0x0C000000`.

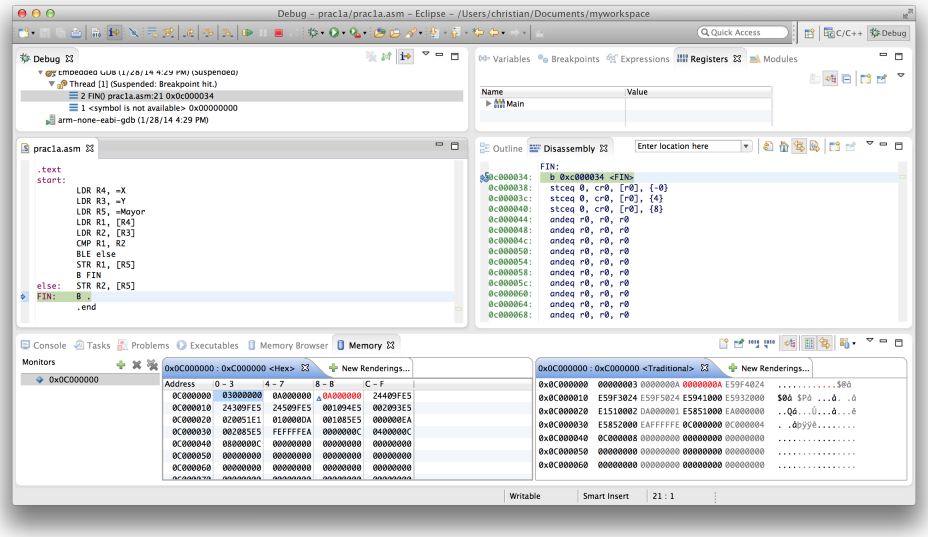


Figura 1.21: Ventana con el aspecto del monitor de memoria tras la simulación completa del programa.

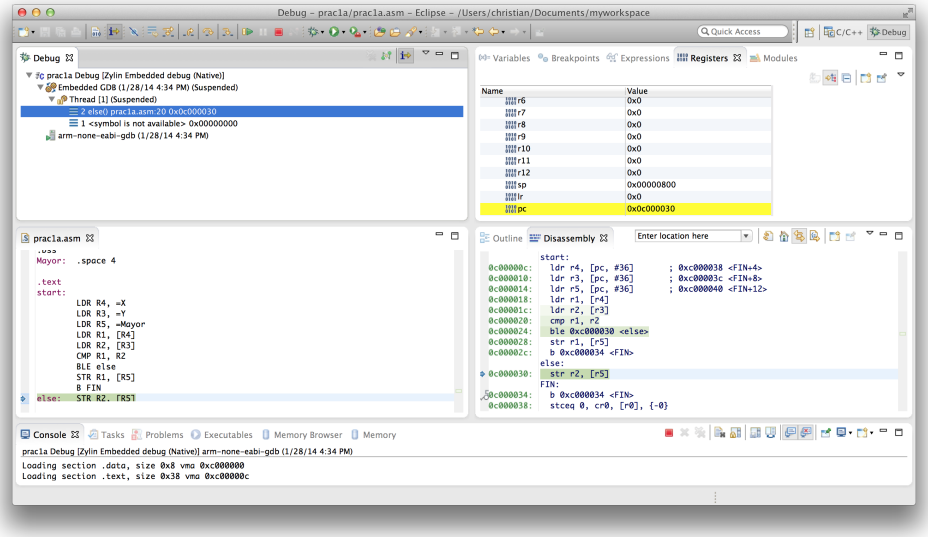


Figura 1.22: Ejemplo de una sesión de depuración con el visor de registros.

1.7. Desarrollo de la práctica

El alumno deberá presentar al profesor los siguientes apartados:

1. Desarrollo completo del ejemplo presentado en la sección 1.6.2
2. Desarrollar un programa en código ensamblador del ARM que divida dos números tamaño palabra A y B y escriba el resultado en el número C mediante restas parciales utilizando el algoritmo del cuadro 4.

Cuadro 4 Pseudocódigo para realizar la división A/B con restas parciales.

```
C = 0
while( A >= B ) {
    A = A - B
    C = C + 1
}
```

Práctica 2

Programación en ensamblador y etapas de compilación

2.1. Objetivos

Una vez familiarizados con el repertorio de instrucciones del ARM y con el entorno de desarrollo, en esta práctica se persiguen los siguientes objetivos:

- Comprender la finalidad de las etapas de compilación, ensamblado y enlazado.
- Conocer algunos modos de direccionamiento adicionales en instrucciones LDR/STR
- Adquirir práctica en el manejo de vectores de datos (arrays) conociendo la posición de inicio del vector y su longitud.
- Aprender a codificar en ensamblador del ARM sentencias especificadas en lenguaje de alto nivel.

2.2. Generación de un ejecutable

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 entonces 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 5 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 etiqueta `.equ` permite declarar constantes (que recordamos no se almacenarán en memoria).

Cuadro 5 Ejemplo de código con secciones.

```
        .global start

        .equ UNO, 0x01

        .bss
RES:    .word 0x03

        .data
A:      .word  0x03
B:      .word  0x02

        .text
start:
        MOV R0, #UNO
        LDR R2, =A      @ carga en R2 la dirección de A
        LDR R1, [R2]    @ carga el valor de A
        ADD R2, R0, R1
        LDR R3, =B      @ carga en R3 la dirección de B
        LDR R4, [R3]    @ carga el valor de B
        ADD R4, R2, R4
        LDR R3, =RES    @ carga en R3 la dirección de RES
        STR R4, [R3]    @ guarda el contenido de R4 en RES
FIN:    B .
        .end
```

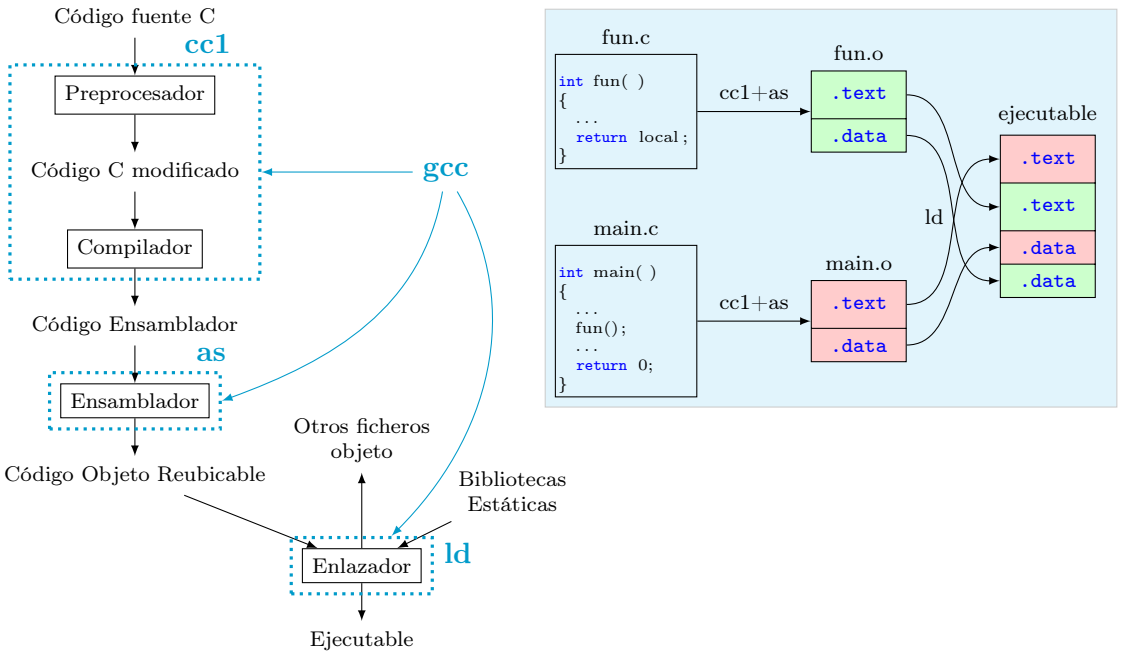


Figura 2.1: Proceso de generación de código ejecutable

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, como hicimos en la práctica anterior con el `ld_script.ld`.

2.3. Modos de direccionamiento de LDR/STR

Un modo de direccionamiento especifica la forma de calcular la dirección de memoria efectiva de un operando mediante el uso de la información contenida en registros y/o valores inmediatos contenidos en la propia instrucción de la máquina. En la práctica anterior

estudiamos dos de los modos de direccionamiento válidos en ARMv4 para las instrucciones `ldr`/`str`. Sin embargo, para el desarrollo de esta práctica nos será muy util considerar uno más:

- **Indirecto de registro con desplazamiento por registro** 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 o dividido por una potencia de 2. En realidad, hay cinco posibles operaciones que se pueden realizar sobre el registro de offset, pero sólo veremos dos de ellas:

  - Desplazamiento aritmético a la derecha (ASR). Equivalente a dividir por una potencia de 2. En ensamblador la instrucción se escribiría como:

    ```
    LDR Rd, [Rb, Ri, ASR #Potencia de dos]
    ```

  - Desplazamiento lógico a la izquierda (LSL). Equivalente a multiplicar por una potencia de 2. En ensamblador la instrucción se escribiría como:

    ```
    LDR Rd, [Rb, Ri, LSL #Potencia de dos]
    ```

  En la Figura 2.2 se ilustra el resultado de realizar la operación `ldr r2, [r1, r5, LSL #2]`

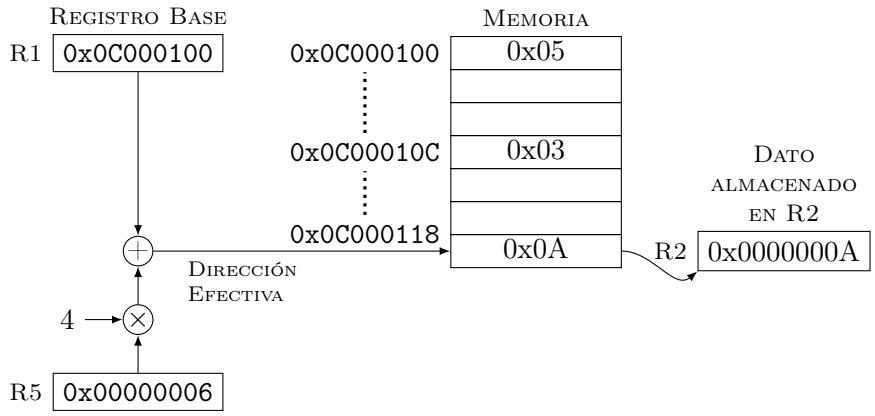


Figura 2.2: Ejemplo de ejecución de la instrucción `ldr r2, [r1, r5, LSL #2]`.

Como se ilustra en el cuadro 6, éste nuevo modo de direccionamiento es el más adecuado para recorrer arrays:

- En un registro mantendremos la dirección de comienzo del array y lo usamos como registro base.
- En otro registro mantenemos el índice del array que queremos acceder y será el offset. Bastará con multiplicar dicho registro (usando LSL) por el número de bytes que ocupa cada elemento del array (4 para números enteros) para conseguir la dirección del elemento deseado.

Cuadro 6 Ejemplo de recorrido de arrays con direccionamiento indirecto de registro con desplazamiento por registro.

Código C

```
int A[100]={0};

for (i=0; i < 100; i++){
  ... = A[i] ...;
}
```

Ensamblador

```
/* reservamos memoria para A, inicializada a 0 */
    .data
A:  .space 100, 0

    .text
/* almacenamos en r1 la dirección de comienzo de A
   */
     LDR  r1,=A
     MOV r2,#0   @ inicializamos i
for: CMP r2,#100
     BGE fin
...
/* leemos el elemento i-ésimo del vector A y lo
   almacenamos en r3 */
     LDR r3,[r1,r2,lsl#2]
...
     ADD r2,r2,#1
     B for
fin:
```

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 #3] @ Carga en R11 lo que hay en Mem[R3 + R5*8]
```

2.4. Pseudo-instrucción LDR

Como hemos visto en el ejemplo anterior, para recorrer los arrays necesitamos almacenar su dirección de comienzo en un registro (`r1`, en el Cuadro 6). Para ello resulta necesaria la pseudoinstrucción:

```
LDR R1,=A
```

La Figura 2.3 ilustra el funcionamiento de esta pseudo-instrucción, dónde suponemos que tras el enlazado la dirección asociada a `A` es `0x208` y que en algún momento el programa ha escrito nuevos valores en nuestro array `A`. Utilizando `LDR R1,=A` conseguimos cargar la dirección `0x208` en `R1`.

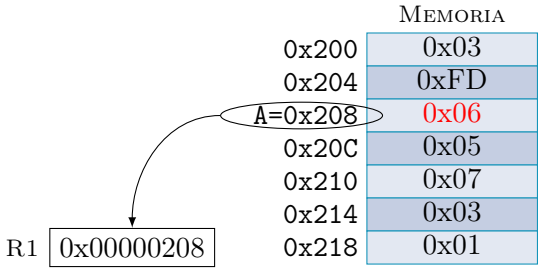


Figura 2.3: Ilustración del funcionamiento de la pseudoinstrucción `LDR R1,=A`.

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 `A`. De hecho, parecería mucho más lógico utilizar una instrucción para escribir un valor inmediato en un registro, algo como:

```
MOV R1, #A
```

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

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

```
dirA: .word A
```

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

```
ldr r1, dirA
```

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 `dirA` (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,=A`, que es más cómoda de utilizar.

Volveremos de nuevo sobre la pseudo instrucción `LDR R<n>, =Etiqueta` en la práctica 4, donde analizaremos en detalle la resolución de símbolos.

2.5. Preparación del entorno de desarrollo

En la práctica anterior aprendimos a crear un proyecto en eclipse con el plugin de GNU ARM. En esta práctica podríamos repetir el procedimiento para añadir un nuevo proyecto a nuestro `workspace` para cada uno de los programas que tendremos que desarrollar. Sin embargo resultará más conveniente crear los nuevos proyectos como copias de los anteriores. Para crear un nuevo proyecto `prac2a` a partir de un proyecto anterior de nuestro `workspace`, `prac1a`, procederemos de la siguiente manera:

1. Abrimos eclipse, seleccionando nuestro `workspace` en la ventana de `Workspace Launcher`.

2. Seleccionamos la perspectiva `C/C++`.

3. Pinchamos con el botón derecho del ratón sobre el proyecto a copiar, `prac1a`, con lo que se desplegará un menú como el que muestra la Figura 2.4, y seleccionamos la opción `Copy`. Alternativamente podemos usar la combinación de teclas del sistema (`Ctrl c` en windows y Linux, `⌘ c` en Mac Os X) para copiar el proyecto.

4. Pinchamos con el botón derecho del ratón sobre el explorador de proyectos, con lo que se despliega de nuevo el mismo menú, y seleccionamos la opción `Paste`. Alternativamente podemos usar la combinación de teclas del sistema (`Ctrl v` en windows y Linux, `⌘ v` en Mac Os X) para pegar el proyecto copiado. Se abrirá una ventana que nos permitirá seleccionar un nombre para el nuevo proyecto, como ilustra la Figura 2.5.

5. Será conveniente borrar la carpeta `Debug` del nuevo proyecto. Esta carpeta contendrá los ficheros objeto y el ejecutable del proyecto anterior. La primera compilación de nuestro proyecto volverá a crear la carpeta para alojar los nuevos ficheros objeto y el nuevo ejecutable.

6. Finalmente, pinchamos en el proyecto con el botón derecho y seleccionamos `Properties`→`C/C++ Build`→`Refresh Policy`. En el cuadro aparecerá el nombre de la práctica original, lo seleccionamos y pulsamos el botón `Delete`. Después pulsamos el botón `Add Resource...` y seleccionamos el nuevo proyecto en la ventana que se abre, como muestra la Figura 2.7. Pulsamos `Ok`. Esto hará que se refresque automáticamente el proyecto en el explorador de proyectos cuando compilemos.

Con esto tenemos creado un nuevo proyecto. Como ilustra la Figura 2.6, el proyecto contiene una copia de los ficheros del proyecto original. Generalmente nos interesará renombrar el fichero fuente, por ejemplo cambiar `prac1a.asm` por `prac2a.asm`. Esto podemos hacerlo seleccionando el fichero con el botón derecho del ratón y seleccionando `Rename...` en el menú desplegado. Otra alternativa es directamente borrar los ficheros fuente que queramos

cambiar, seleccionando la opción `Delete` en el mismo menú, y luego añadir nuevos ficheros como lo hicimos en la práctica anterior.

2.5.1. Copia de configuraciones de depuración

Aunque aún no hemos preparado el proyecto, cuando lo hagamos deberemos simularlo y depurarlo. Ya explicamos en la práctica anterior cómo crear una configuración de depuración, sin embargo existe una alternativa: copiar una configuración de depuración de otro proyecto de nuestro `workspace`. Para esto el proyecto del que queremos copiar la configuración debe estar abierto, de lo contrario sus configuraciones de depuración no serán visibles. Para hacer la copia haremos lo siguiente:

- Cambiamos a la perspectiva de `Debug`.
- Seleccionamos `Run`→`Debug Configurations...`. Se abrirá una ventana como la de la Figura 2.8, y podremos ver en su panel izquierdo la configuración de depuración que queremos copiar (`prac1a Debug` en nuestro ejemplo).
- Seleccionamos la configuración a copiar y pulsamos el botón  . Entonces se creará una nueva configuración como copia exacta de la anterior, en la que debemos cambiar el nombre del proyecto (`prac1a` por `prac2a`) y el nombre del ejecutable a depurar (`Debug/prac1a.elf` por `Debug/prac2a.elf`) y pulsamos el botón `Apply`. La Figura 2.9 ilustra el aspecto final que tendría la ventana.
- Finalmente pulsamos el botón `Close`. Si nos equivocamos y pulsamos `Debug` nos dirá que se ha producido un error. Esto es lógico porque estaremos intentando depurar un ejecutable que aún no hemos creado. Podemos ignorar el error.

Una vez hecho esto, podemos cerrar el proyecto original. Para ello, en el explorador de proyectos de la perspectiva `C/C++` pinchamos sobre el proyecto a cerrar y seleccionamos `Close Project` en el menú desplegado.

2.6. Desarrollo de la práctica

Es obligatorio traer realizados los apartados a) y b) de casa. Durante la sesión de laboratorio se solicitará una modificación de uno de ellos, que habrá que realizar en esas 2 horas. En **TODOS** los apartados será obligatorio definir tres secciones:

- `.data` para variables con valor inicial (declaradas como `.word`)
- `.bss` para variables sin valor inicial (declaras como `.space`)
- `.text` para el código

a) Codificar en ensamblador del ARM el siguiente código C, encargado de buscar el valor máximo de un vector de enteros positivos `A` de longitud `N` y almacenarlo en la variable `max`. Es **OBLIGATORIO** escribir en memoria el valor de `max` cada vez que éste cambie (es decir, no basta con actualizar un registro; hay que realizar una instrucción `str`).

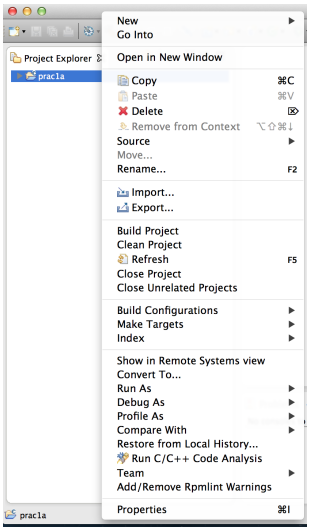


Figura 2.4: Menu desplegado al pinchar con el botón derecho del ratón sobre un proyecto de nuestro `workspace`.

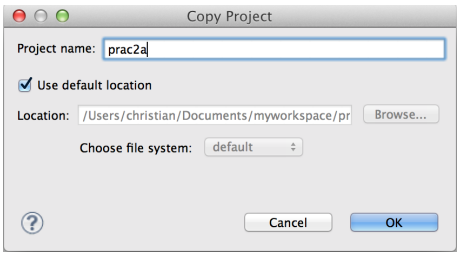


Figura 2.5: Ventana abierta al pegar el proyecto copiado. Nos permite seleccionar un nombre para el nuevo proyecto.

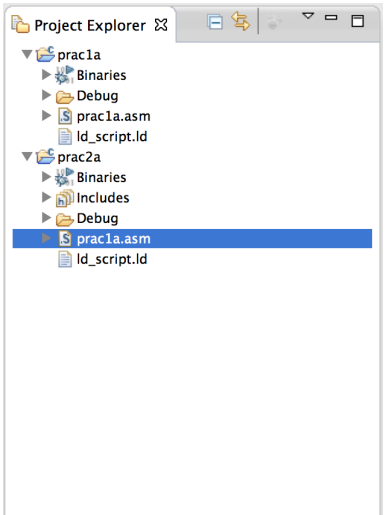


Figura 2.6: Aspecto que tendrá el explorador de proyectos cuando hayamos copiado el proyecto `prac1a` con el nombre `prac2a`.

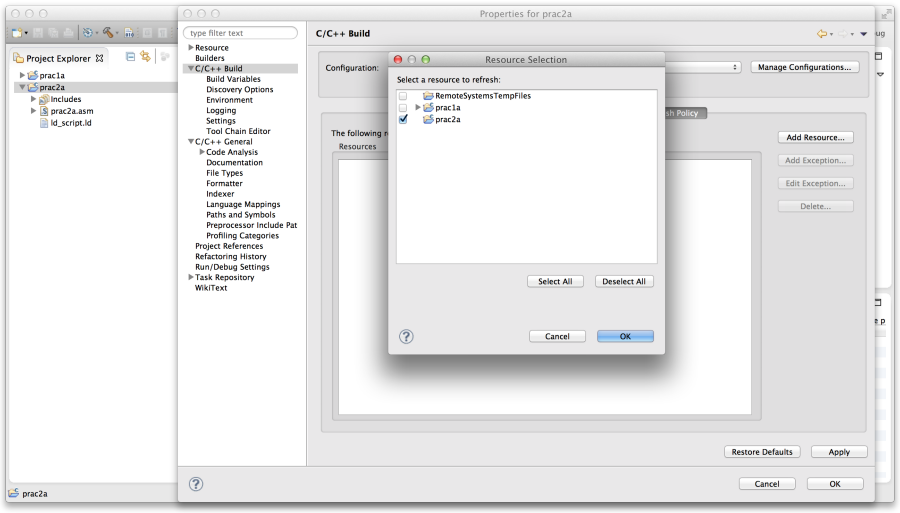


Figura 2.7: Ventana para seleccionar las acciones de refresco del proyecto.

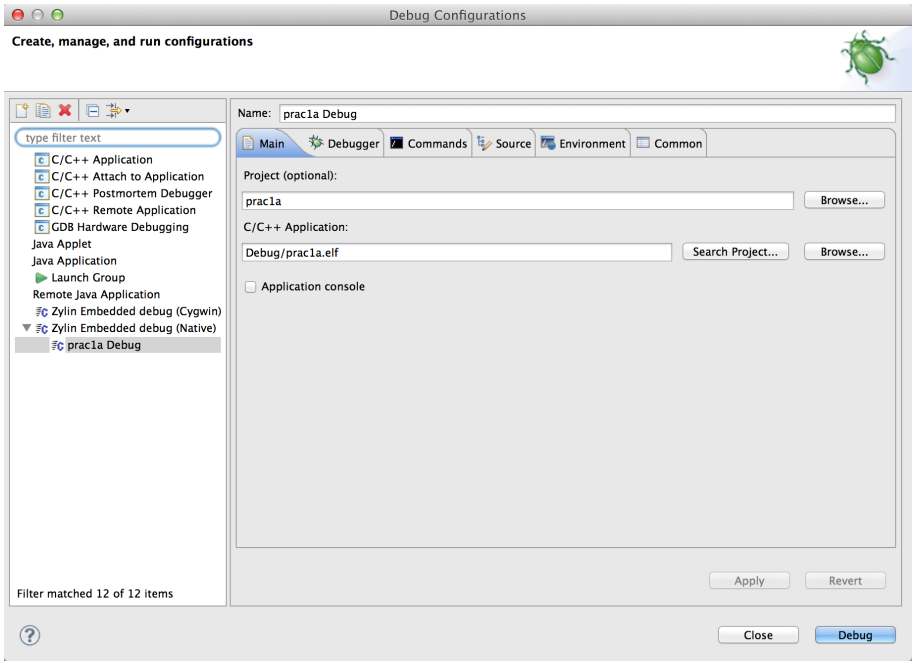


Figura 2.8: Aspecto que tendrá la ventana `Debug Configurations` cuando hayamos copiado el proyecto `prac1a`.

Código C

```
#define N 8
int A[N]={7,3,25,4,75,2,1,1};
int max;

max=0;
for(i=0; i<N; i++){
  if(A[i]>max)
    max=A[i];
}
```

Ensamblador

```
.global start

.EQU N, 8

.data
A:      .word 7,3,25,4,75,2,1,1

.bss
max:    .space 4

.text
start:
        mov r0, #0
        ldr r1,=max  @ Leo la dir. de max
        str r0,[r1]  @ Escribo 0 en max
        @ Terminar de codificar
```

b) Codificar en ensamblador del ARM un algoritmo de ordenación basado en el código del apartado anterior. Supongamos un vector `A` de `N` enteros mayores de 0, queremos rellenar un vector `B` con los valores de `A` ordenados de mayor a menor. Para ello nos podemos basar en el siguiente código de alto nivel. Es **OBLIGATORIO** escribir en memoria el valor de `ind` y `max` cada vez que cambien (es decir, no basta con actualizar un registro; hay que utilizar la instrucción `str` para actualizar su valor en memoria).

Código C

```
#define N 8
int A[N]={7,3,25,4,75,2,1,1};
int B[N];
int max, ind;

for(j=0; j<N; j++){
  max=0;
  for(i=0; i<N; i++){
    if(A[i]>max){
      max=A[i];
      ind=i;
    }
  }
  B[j]=A[ind];
  A[ind]=0;
}
```

Ensamblador

```
.EQU N, 8
.global start
.data
A:      .word 7,3,25,4,75,2,1,1

.bss
B:      .space N*4
max:    .space 4
ind:    .space 4

.text
start:
        @ Terminar de codificar
```

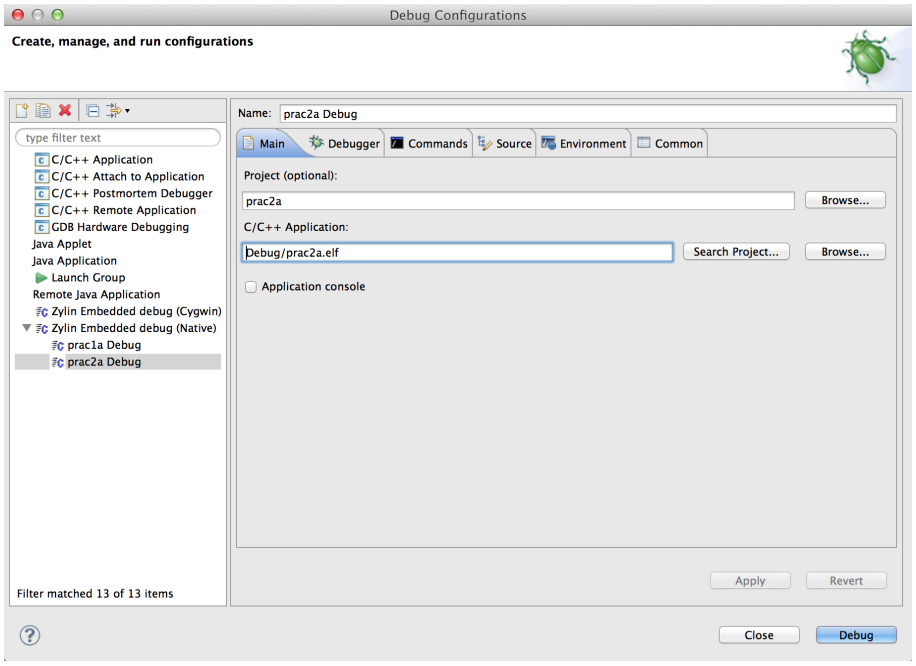


Figura 2.9: Aspecto que tendrá la ventana `Debug Configurations` tras la copia de la configuración `prac1a Debug`.

Práctica 3

Programación con subrutinas

3.1. Objetivos

El objetivo de esta práctica es introducir el concepto de subrutina y familiarizarse con su uso. Para ello resulta necesario conocer:

- El soporte del ensamblador del ARM para la gestión de subrutinas.
- Conceptos de pila de llamadas, marco de activación, su construcción y su destrucción (prólogo/epílogo).
- Convenio de paso de parámetros a subrutinas y valor de retorno.

3.2. Subrutinas y Pila de Llamadas

Se conoce como subrutina a 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 3.1 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, que podemos inicializar 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 inicializa este `N` con un valor. Se dice que se pasan los parámetros en la llamada o invocación. Una subrutina puede además retornar un valor.

Como ilustra Figura 3.1 sólo existe una copia del código de una subrutina, que debe colocarse fuera de la región del programa principal, y resulta conveniente identificar la primera instrucción de la subrutina con una etiqueta, ya que entonces para invocarla basta con hacer dos cosas:

- indicar de alguna manera la dirección de la instrucción por la que debe continuar la ejecución cuando termine la subrutina, y

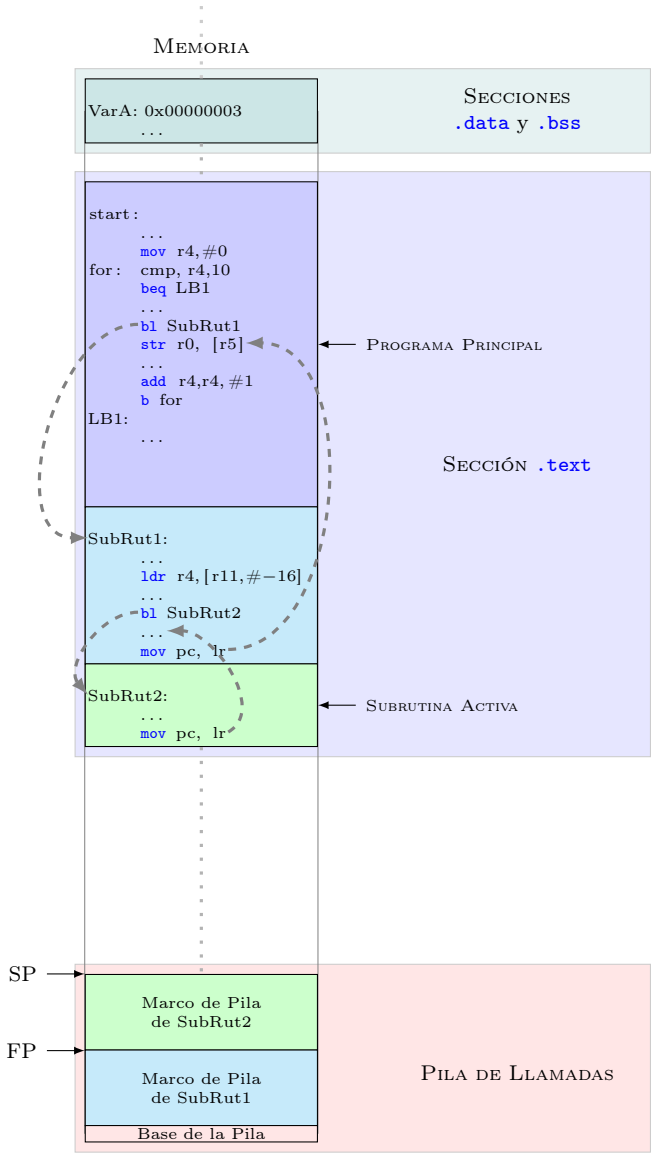


Figura 3.1: Subrutinas y Pila de Llamadas. El programa principal invoca la subrutina `SubRut1`, que a su vez invoca a `SubRut2`, que pasa a ser la subrutina activa. Los punteros `SP` y `FP` delimitan el marco de la subrutina activa.

- 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 el valor contenido en `LR` (`r14`) sobre `PC`. Esto suele hacerse como en la Figura 3.1 con la instrucción `MOV`:

```
mov pc, lr
```

o con la instrucción `BX` (*brach and exchange*):

```
bx lr
```

Sin embargo, para que una subrutina pueda ser invocada con seguridad desde cualquier punto del programa, es imprescindible que la subrutina preserve el estado arquitectónico, es decir, el valor de los registros visibles al programador. La Figura 3.1 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 saber cómo está implementada.

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 3.1 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.

Las subrutinas suelen estructurarse en tres partes:

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

Comienzan con un prólogo, un código encargado de construir el marco de activación de la subrutina y de insertarlo en la pila. Finalizan con un epílogo, un código encargado de extraer el marco de activación de la pila, dejándola tal y como estaba antes de la ejecución del prólogo.

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`) y habitualmente es almacenado en un

registro arquitectónico, que es inicializado a la base de la región de pila cuando comienza el programa (pila vacía). Para facilitar el acceso a la información contenida en el marco, es habitual utilizar un puntero denominado *frame pointer* (`FP`) que apunta a una posición preestablecida del marco, generalmente la base.

La subrutina usará el 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.

El marco de pila se utiliza también para pasar por memoria los parámetros a la subrutina. El programa copia los parámetros en la cima de la pila. La subrutina inserta su marco de activación en la pila, justo por encima de los parámetros de la llamada, pudiendo acceder a ellos con desplazamientos relativos a `FP`. Existe una alternativa al paso de parámetros por pila, pasarlos por registro.

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 este caso, en el estándar de llamadas a subrutinas, que forma parte de lo que se denomina el *Application Binary Interface* (ABI). En la siguiente sección estudiaremos el estándar de ARM.

3.3. 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. Las secciones siguientes 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.

3.3.1. 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 3.1 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 3.1 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 3.1: 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.

3.3.2. Modelo de pila

La figura 3.2 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.

3.3.3. 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]`,
- ...

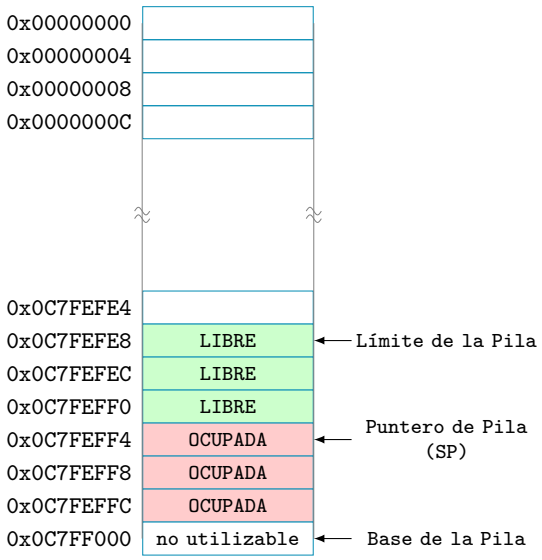


Figura 3.2: Ilustración de una pila *Full Descending*.

De esta forma la subrutina siempre puede acceder a los parámetros que se le pasan por memoria con desplazamientos positivos a su frame pointer.

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 3.3 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`.

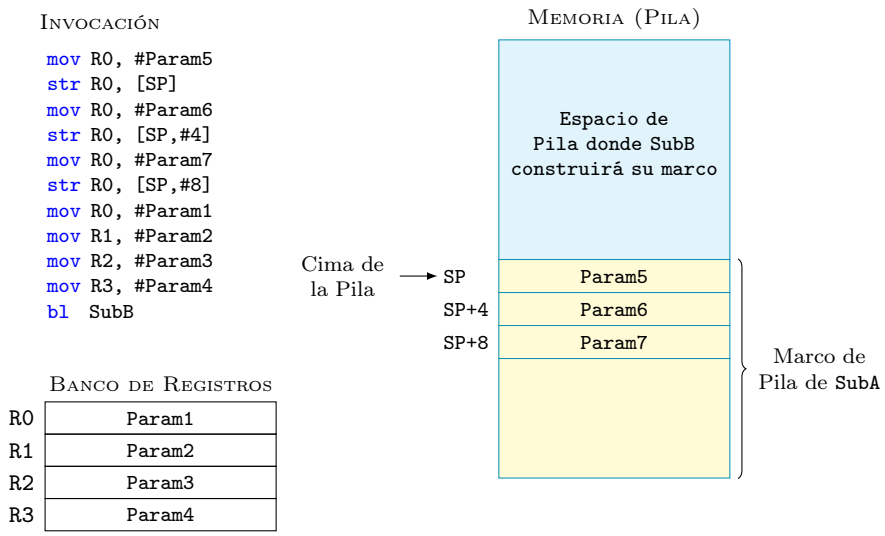


Figura 3.3: Paso de los parámetros `Param1-7` desde `SubA` a `SubB`. El código de invocación asume que los parámetros son constantes (valores inmediatos), en caso de ser variables habría que cargarlos con instrucciones `ldr` en lugar de `mov`.

3.3.4. 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].

3.4. 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 AAPC desaparecen estas restricciones. Cualquier rutina que cumpla con las condiciones presentadas en las secciones anteriores estaría conforme al estándar. No obstante, en esta asignatura 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 pero lo tendremos que usar para pasar a su vez la dirección de retorno a las subrutinas invocadas.
- Vamos a usar siempre frame pointer, para facilitar la depuración y el acceso al marco de activación. Por lo tanto, tendremos siempre que almacenar el valor anterior de `FP` en la pila.
- 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` a partir del valor de `FP`.
- De los registros `r4-r10` insertaremos sólo aquellos que modifiquemos en el cuerpo de la subrutina.

La Figura 3.4 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`, aparte de la facilidad de codificación y depuración, es que se forma en la pila una lista enlazada de marcos de activación. Como ilustra la Figura 3.4, 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.

Para insertar el marco de activación en la pila (*push*) suelen utilizarse en ARM instrucciones de store múltiple, capaces de almacenar varios registros en memoria. Del mismo modo, para extraer el marco de la pila (*pop*) una vez terminada la subrutina suelen utilizarse instrucciones de load múltiple. En la siguiente sección analizaremos este tipo de instrucciones antes de presentar el código del prólogo y el epílogo que utilizaremos en nuestras subrutinas.

3.4.1. 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 y las copias de bloques de datos en memoria. Además permiten reducir el tamaño del código.

[1] Esta es una simplificación válida para valores de tamaño palabra o menor. El resto de los casos queda fuera del objetivo de este documento

[2] Con los flags -g -O0

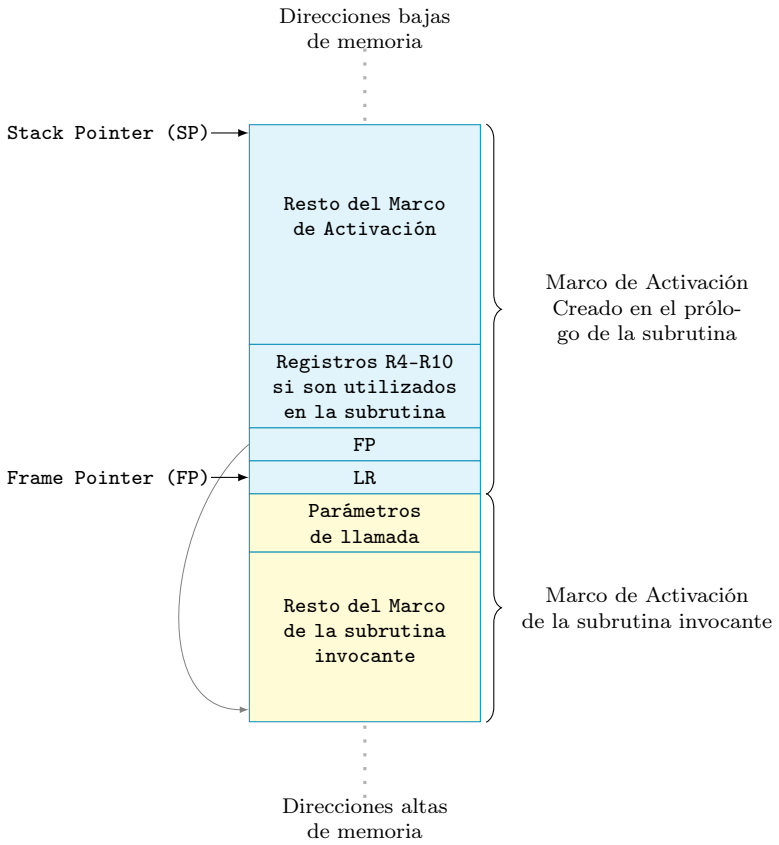


Figura 3.4: Estructura de Marco de Activación recomendada. En el caso de subrutinas hoja podremos omitir la copia de `LR`.

El Load Múltiple (`LDM`) permite la carga simultánea de varios registros 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. La codificación de estas instrucciones en ensamblador es:

```
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.

Existen cuatro modos de direccionamiento para estas instrucciones, dando lugar a 4 versiones de cada instrucción. En esta práctica vamos a limitarnos a estudiar sólo los dos modos que nos interesan para codificar el prólogo y el epílogo de nuestras rutinas, que son:

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

  ```
  dirección de comienzo = Rb
  ```

- *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*NúmeroDeRegistros
```

El funcionamiento de las cuatro instrucciones a las que dan lugar se ilustra en la Figura 3.5. 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.

Las dos instrucciones que nos permiten implementar las operaciones de inserción (*push*) y extracción (*pop*) sobre una pila *full descending* son:

Inserción: `STMDB Rb!, {lista de registros}`. Esta instrucción es equivalente a un *push* de la lista de registros si utilizamos `SP` como registro base. De hecho podemos referirnos a esta instrucción con dos posibles alias:

- `STMFD Rb!, {lista de registros}`, donde `FD` deriva de full descending.
- `PUSH {lista de registros}`. En este caso no podemos elegir el registro base, que forzosamente es `SP`. Éste es el formato que usaremos en los prólogos de las subrutinas.

Extracción: `LDMIA Rb!, {lista de registros}`. Esta instrucción es equivalente a un *pop* de la lista de registros si utilizamos `SP` como registro base. De hecho podemos referirnos a esta instrucción con dos posibles alias:

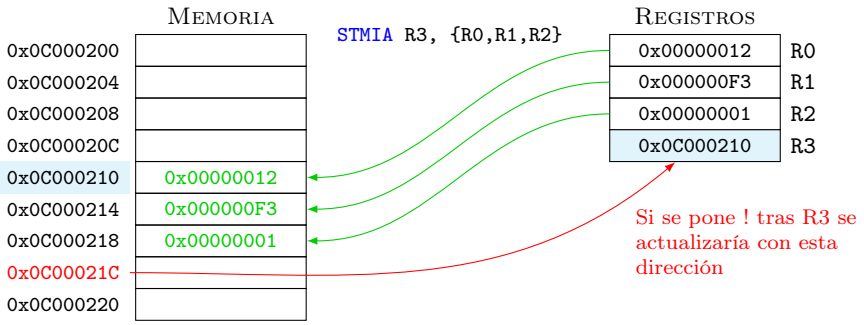
- `LDMFD Rb!, {lista de registros}`. Igual que antes `FD` deriva de full descending.
- `POP {lista de registros}`. En este caso no podemos elegir el registro base, que forzosamente es `SP`. Este es el formato que utilizaremos en los epílogos de las subrutinas.

Ejemplos

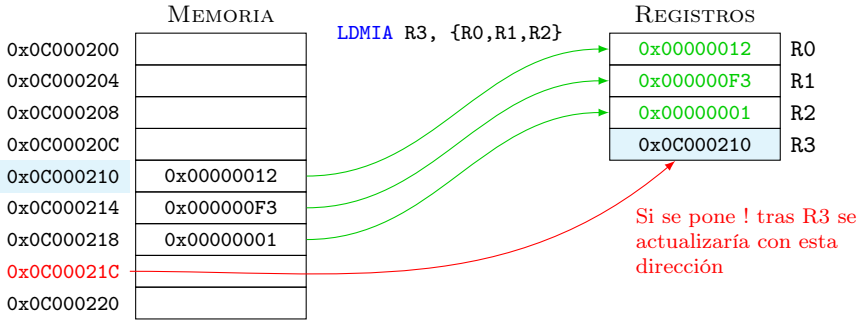
```
STMDB SP!, {R4-R10,FP,LR}   @ Empezando en la dirección SP-4*9 copia el
                            @ contenido de los registros R4-R10, FP y LR.
                            @ SP queda con la dirección de comienzo (SP-4*9)
                            @ Se apilan los registros.
STMFD SP!, {R4-R10,FP,LR}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

3.4.2. Prólogo y Epílogo

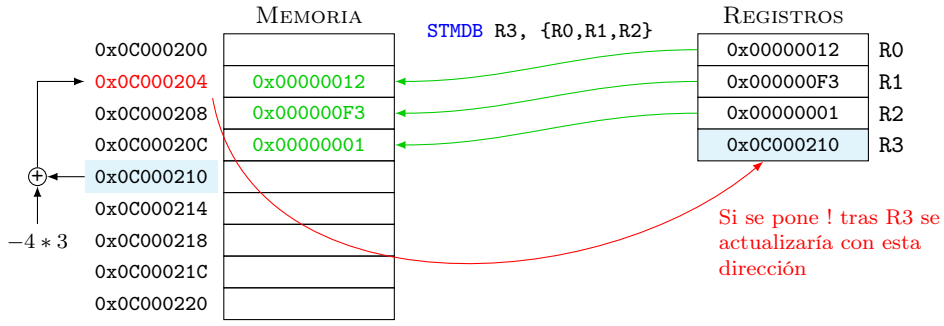
El cuadro 7 describe un posible prólogo para la construcción del marco de activación representado en la Figura 3.4. La primera instrucción apila los registros a preservar sobre el marco de la rutina invocante, dejando `SP` apuntando a la nueva cima. Debemos incidir en que no es necesario apilar siempre todos los registros `R4-R10`, sólo aquellos que se vayan a modificar en la rutina. La siguiente instrucción deja el `FP` apuntando a la base del nuevo



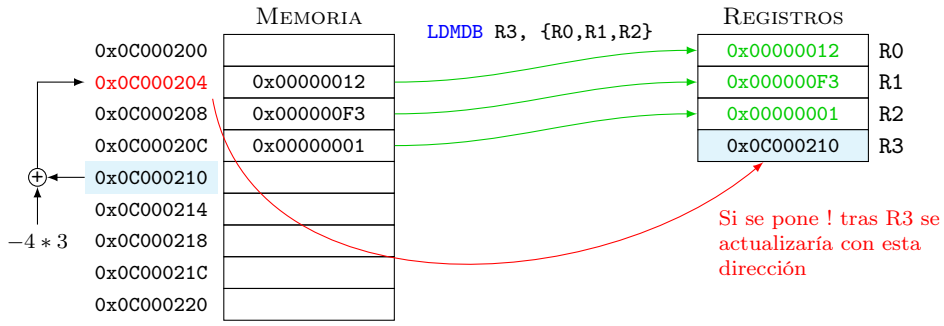
(a) STMIA



(b) LDMIA



(c) STMDB



(d) LDMDB

Figura 3.5: Instrucciónes de load y store múltiple

marco de activación. La última instrucción reserva nuevo espacio en el marco, el necesario para los registros que deba pasar por pila a las subrutinas que invoque. Existen algunos casos especiales en los que podemos simplificar este prólogo:

- Si no pasamos más de cuatro parámetros a ninguna subrutina no tendremos que reservar espacio extra. Podemos entonces eliminar la tercera instrucción.
- Si sólo apilamos un registro (sucedería en una subrutina hoja que no modifique ninguno de los registros `r4-r10`) la segunda instrucción se convertiría en `ADD FP, SP, #0`. En este caso quizá quede más claro codificarla como `MOV FP, SP`.

Cuadro 7 Prólogo para insertar en la pila el Marco de Activación de la Figura 3.4.

```
PUSH {R4-R10,FP,LR}                       @ Copiar registros en la pila
ADD  FP, SP, #(4*NumRegistrosApilados-4)  @ FP ← dirección base del marco
SUB  SP, SP, #4*NumPalabrasExtra          @ Espacio extra necesario
```

El cuadro 8 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. Otra alternativa sería hacer lo contrario de lo que hace la tercera instrucción del prólogo, asumiendo que no hemos alterado el valor de `SP` en el cuerpo de la subrutina, es decir, `ADD SP, SP, #4*NumPalabrasExtra`. Las dos instrucciones son igual de eficientes por lo que no hay un motivo a priori por el que elegir una de ellas. Hemos optado por la instrucción que aparece en el cuadro 8 porque es la que utiliza el compilador gcc-4.7 que usamos en el laboratorio, y de esta forma el alumno estará más familiarizado con el código que genera el compilador y le resultará más fácil comprenderlo.

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-R10`, `FP` 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 de la rutina invocante.

Cuadro 8 Epílogo para extraer de la pila el Marco de Activación de la Figura 3.4.

```
SUB  SP, FP, #(4*NumRegistrosApilados-4) @ SP ← dirección del 1er registro apilado
POP  {R4-R10,FP,LR}                      @ Desapila restaurando los registros
BX   LR                                  @ Retorno de subrutina
```

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

- Si sólo apilamos un registro la primera instrucción del epílogo se reduce a `SUB SP, FP, #0`. En este caso quizá quede más claro codificarla como `MOV SP, FP`.
- Si no hemos reservado espacio extra (`NumPalabrasExtra=0`) y no hemos alterado el valor de `SP` en el cuerpo de la subrutina, podemos incluso eliminar esta misma instrucción del epílogo, ya que `SP` tendrá el valor que le asignó la primera instrucción del prólogo.

3.5. Un ejemplo

El código de la Figura 3.6 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`, proporcionado por el enlazador al seguir las indicaciones del script de enlazado `ld_script.ld`. También se inicializa `fp` a cero, para que al llamar a la primera subrutina se meta 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.

3.6. Desarrollo de la práctica

Los alumnos deberán preparar en casa las soluciones a los apartados que se presentan a continuación. En el laboratorio se pedirá una **modificación** de estos apartados y se realizará un examen online para comprobar que el estudiante ha entendido correctamente todos los conceptos asociados a la práctica desarrollada.

a) Codificar en ensamblador del ARM la siguiente función en C encargada de buscar el valor máximo de un vector `A` de enteros positivos de longitud `longA` y devolver la posición de ese máximo (el índice):

```
int i, max, ind;
int max(int A[], int longA){
  max=0;
  ind=0;
  for(i=0; i<longA; i++){
    if(A[i]>max){
      max=A[i];
      ind=i;
    }
  }
  return(ind);
}
```

NOTA: El código de este apartado se corresponde (excepto por las instrucciones de gestión de subrutina) con el bucle interno del apartado b de la práctica anterior. Nótese que la subrutina es de tipo hoja, por tanto sólo necesita salvar y restaurar el registro `fp` y los registros `r4` a `r10` que utilice.

b) Codificar en ensamblador del ARM un algoritmo de ordenación basado en el código del apartado anterior. Supongamos un vector `A` de `N` enteros mayores de 0, queremos rellenar un vector `B` con los valores de `A` ordenados de mayor a menor. Para ello nos podemos basar en el siguiente código de alto nivel:

```
#define N 8
int A[N]={7,3,25,4,75,2,1,1};
int B[N];
int j;
void main(){
  for(j=0; j<N; j++){
```

Código C

```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;
}
```

Ensamblador

```
    .extern _stack
    .global start
    .equ N, 4
    .data
A:  .word 7,3,25,4
    .bss
B:  .space N*4

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

Recorre:
    push {r4-r8,fp,lr}
    add fp, sp, #24 @ 24=4*7-4
    mov r4, r0      @ R4, A
    mov r5, r1      @ R5, B
    sub r6, r2, #1  @ R6, M-1
    mov r7, #0      @ R7, j
for:
    cmp r7, r6
    bge Ret1
    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 for
Ret1:
    pop {r4-r8,fp,lr}
    mov pc, lr

Mayor:
    push {fp}
    mov fp, sp  @ SP - 0
    cmp r0, r1
    bgt Ret2
    mov r0, r1
Ret2:
    pop {fp}
    mov pc, lr
    .end
```

Figura 3.6: Ejemplo con subrutinas

```
    ind=max(A,N)
    B[j]=A[ind];
    A[ind]=0;
  }
}
```

NOTA: El código de este apartado se corresponde (excepto por las instrucciones de llamada a subrutina) con el bucle externo del apartado b de la práctica anterior.

Práctica 4

Mezclando C y ensamblador

4.1. Objetivos

En esta práctica profundizaremos en nuestros conocimientos sobre la programación en ensamblador sobre ARM creando proyectos con varios ficheros fuente, algunos de ellos escritos en lenguaje C y otros escritos en lenguaje ensamblador. Los objetivos concretos son:

- Comprender la diferencia entre variables locales y variables globales, en su almacenamiento a bajo nivel.
- Comprender el significado de símbolos estáticos (variables y funciones).
- Analizar los problemas que surgen cuando queremos utilizar varios ficheros fuente y comprender cómo se realiza la resolución de símbolos.
- Comprender la relación entre el código C que escribimos y el código máquina que se ejecuta.
- Saber utilizar desde un programa escrito en C variables y rutinas definidas en ensamblador, y viceversa.
- Comprender el código generado por el compilador *gcc*.
- Conocer la representación de los tipos estructurados propios de los lenguajes de alto nivel.

En esta práctica el alumno tendrá que crear un programa, escribiendo unas partes en C y otras partes en ensamblador, de forma que las rutinas escritas en C puedan invocar rutinas escritas en ensamblador y viceversa.

4.2. Variables locales

Para introducir el concepto de variables locales vamos a partir del ejemplo de la siguiente declaración de una función C:

```
int funA( int a1, int a2, int a3, int a4, int a5);
```

Como sabemos, esta declaración nos dice que la función tendrá, al menos, 5 *variables locales* llamadas `a1-a5` respectivamente. No debemos confundir las variables locales con los parámetros de la llamada, estos son los valores iniciales que debemos dar a estas variables. Por ejemplo, una posible llamada en C podría ser `a = funA(1,3,6,4,b);`, donde pasamos los valores 1,3,6 y 4 con los que se deberá inicializar las variables locales `a1-4`, mientras que a `a5` le asignaremos como valor inicial lo que contenga la variable `b`.

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, en una ejecución de `funA` tendremos varias *instancias activas* de `funA`. 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 (recordemos que una variable tiene reservado un espacio en memoria para almacenar su valor). El espacio más natural para ello es el marco de activación de la subrutina.

Por otro lado, las variables que hemos venido utilizando hasta ahora se conocen como variable globales. Recordemos que tienen un espacio de memoria reservado en las secciones *.data* o *.bss* desde que se inicia el programa, y persisten en memoria hasta que el programa finaliza. En cambio, las variables locales se crean en el prólogo de la subrutina, que reservará espacio para ellas en el marco de activación y se destruyen cuando finaliza la llamada a la subrutina. Como vemos debemos ampliar las funciones que asignamos en la práctica anterior al marco de activación.

4.2.1. Revisión del Marco de Activación

La figura 4.1 muestra un esquema detallado de la organización del marco de activación de una subrutina, incluyendo el espacio para las variables locales y el paso de parámetros por pila. Cuando se crea el marco de activación y tras salvar el contexto, se reserva espacio para las variables locales. Para que se cumplan las restricciones del estándar este espacio tendrá un tamaño múltiplo de 4 bytes. La colocación de las variables locales en este espacio la decide el programador. En el cuerpo de la subrutina las variables locales suelen direccionarse con desplazamientos fijos relativos a `FP` (por tanto desplazamientos negativos).

Hay una excepción habitual a este emplazamiento de variables locales. 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. En este caso la variable se direcciona en el cuerpo de `SubB` con desplazamientos positivos a `fp`.

El prólogo y el epílogo que presentamos en la práctica anterior son compatibles con el marco de la Figura 4.1. Sólo hay que hacer una matización respecto de la tercera instrucción del prólogo, que como recordaremos era:

```
SUB SP, SP, #4*NumPalabrasExtra @ Espacio extra necesario
```

Esta instrucción es la encargada de reservar el espacio extra necesario en el marco de activación. Como ilustra la Figura 4.1, ahora debemos sumar al espacio necesario para los parámetros que la subrutina debe pasar por memoria el numero de palabras que necesita para almacenar sus variables locales.

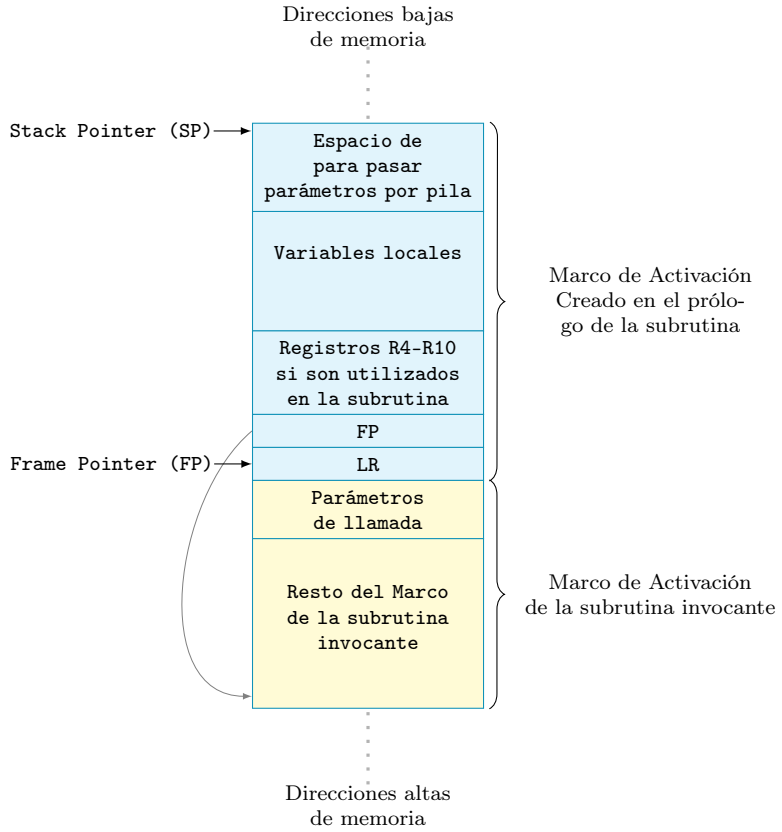


Figura 4.1: Estructura detallada del marco de activación de una subrutina.

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

4.2.2. Ejemplo

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

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

La función tiene tres variable locales, luego necesitaremos reservar 12 bytes adicionales en el marco para almacenarlas. Colocaremos por ejemplo `X` en los primeros 4 (`[fp, #-4]`), `Y` en los siguientes 4 (`[fp, #-8]`) y `m` en los últimos 4 (`[fp, #-12]`). El código ensamblador equivalente sería:

```
1 Mayor:
```

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

Como vemos hemos generado un código que es una traducción línea a línea del código C, siguiendo exactamente la semántica de cada línea del código C. Este código visto en ensamblador parece redundante y poco eficiente. Por ejemplo la línea 9 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.

4.3. Pasando de C a Ensamblador

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 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 4.2 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 4.2, 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.

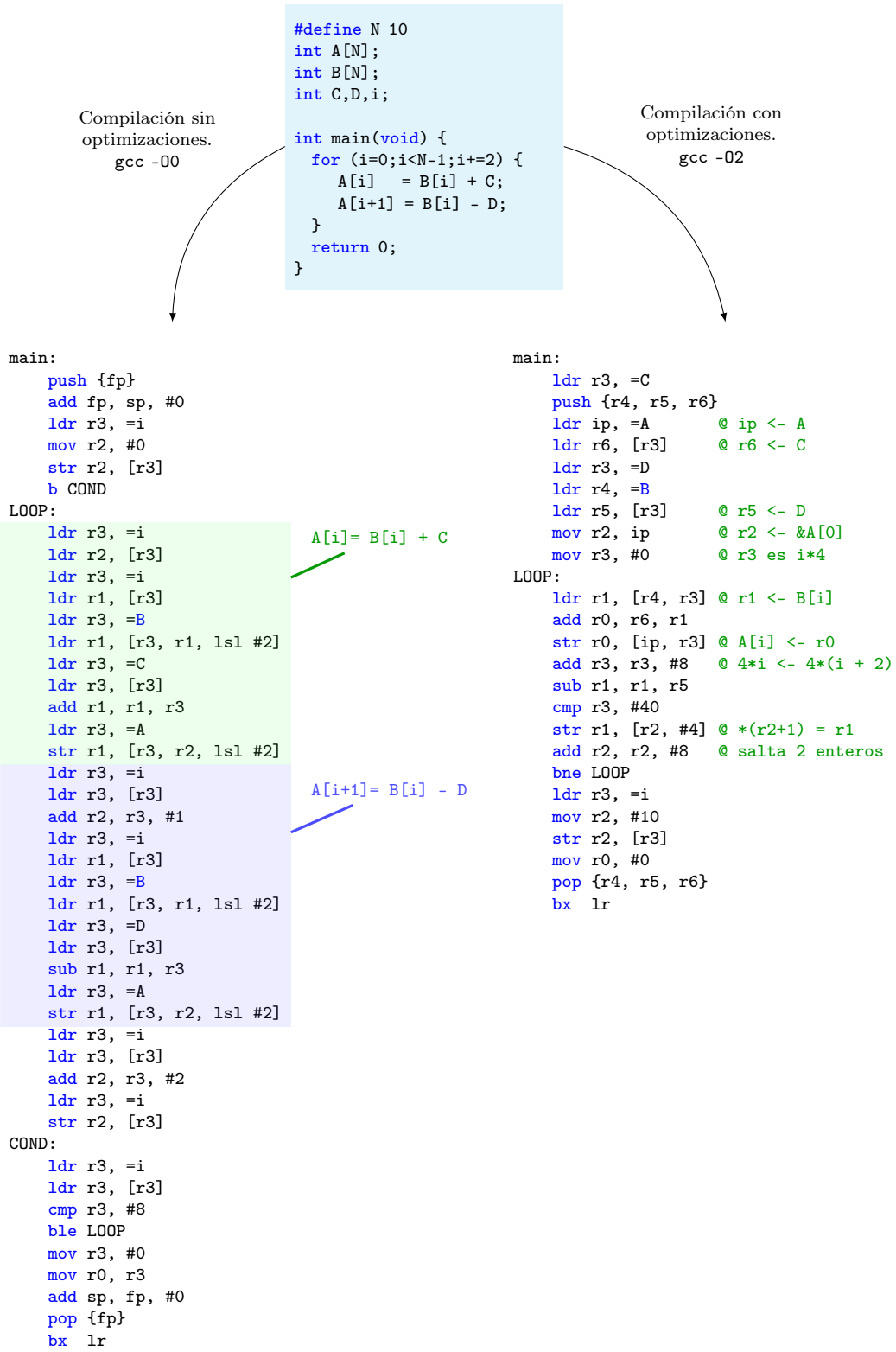
Otro problema típico, producido por la reorganización de instrucciones derivada de la optimización, es que podemos llegar a ver saltos extraños en el depurador. Por ejemplo, si el compilador ha desplazado hacia arriba la primera instrucción ensamblador generada por la traducción de una instrucción C, 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, en esta asignatura es preferible que el alumno se acostumbre a escribir código ensamblador similar al mostrado en la parte izquierda de la Figura 4.2, a menos que se indique expresamente lo contrario. Cada variable que se modifique debería actualizarse en memoria tan pronto como se modifique su valor, evitando hacer por ejemplo lo que hace el código de la derecha con la variable global `i`. Del mismo modo se recomienda cargar de memoria el valor de cada variable cuando se vaya a utilizar, aunque ya tengamos su valor en algún registro por la ejecución de una operación anterior.

4.4. Accesos a memoria: tamaño y alineamiento

Hasta ahora todos los accesos a memoria que hemos venido haciendo con las instrucciones de load/store, han sido accesos tamaño palabra, es decir, llevarnos de un registro a memoria un dato de 32 bits o viceversa. Sin embargo los lenguajes de programación suelen ofrecer tipos básicos de otros tamaños, que en el caso de C son: `char` de tamaño byte, `short int` de 16 bits (media palabra), `long int` de 32 bits (equivalente a `int` en arquitecturas de 32 bits o más) y `long long int` de 64 bits. En todos los casos C admite el modificador `unsigned` para indicar que son enteros sin signo, ya sea de 8, 16, 32, o 64 bits.

Para trabajar con estos tipos de datos de forma eficiente el lenguaje máquina debe proporcionar instrucciones para realizar los accesos a memoria del tamaño apropiado. En ARMv4 el modo de direccionamiento de las instrucciones `ldr`/`str` permite seleccionar accesos de tamaño: byte, media palabra (*half word*, 16 bits), palabra (*word*, 32 bits) y doble palabra (*double word*, 64 bits). Para indicar el tamaño del acceso en lenguaje ensamblador se añaden sufijos a las instrucciones `ldr`/`str` normales, dando lugar a las siguientes variantes:




```
#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;
}
```

```
main:
    push {fp}
    add fp, sp, #0
    ldr r3, =i
    mov r2, #0
    str r2, [r3]
    b COND
LOOP:
    ldr r3, =i
    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
    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
```

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

- LDRB: load de un entero sin signo de tamaño byte. Al escribirse el dato sobre el registro destino se extiende con ceros hasta los 32 bits.
- LDRSB: load de un entero con signo de tamaño byte. Al escribirse el dato sobre el registro destino se extiende con el bit de signo hasta los 32 bits.
- STRB: se escribe en memoria un entero de tamaño byte obtenido de los 8 bits menos significativos del registro fuente.
- LDRH: load de un entero sin signo de 16 bits. Al escribirse el dato sobre el registro destino se extiende con ceros hasta los 32 bits.
- LDRSH: load de un entero con signo de 16 bits. Al escribirse el dato sobre el registro destino se extiende con el bit de signo hasta los 32 bits.
- STRH: se escribe en memoria un entero de 16 bits obtenido de los 16 bits menos significativos del registro fuente.
- LDRD: carga dos registros consecutivos con dos palabras (32 bits) consecutivas de memoria. Para facilitar la codificación de la instrucción se pone la restricción de que el primer registro debe ser un registro par y el segundo el registro impar siguiente, por ejemplo `R8` y `R9`.
- STRD: es la operación contraria a la anterior.

En todos los casos se soportan los modos de direccionamiento indirecto de registro, indirecto de registro con desplazamiento inmediato e indirecto de registro con desplazamiento en registro que estudiamos en las prácticas anteriores.

Sin embargo, la arquitectura ARMv4 impone restricciones de alineamiento en los accesos a memoria. Se dice que un acceso está alineado si la dirección de memoria que se accede es múltiplo del tamaño del dato accedido en bytes. Así, un acceso tamaño byte no tiene ninguna restricción, ya que todas las direcciones serán múltiplo de 1. En cambio, un acceso tamaño media palabra sólo será válido si la dirección de memoria es múltiplo de 2, mientras que un acceso tamaño palabra o doble palabra necesitan direcciones múltiplo de 4 y 8 respectivamente. Si se viola la restricción de alineamiento, se produce una excepción *data abort*. El mecanismo y tratamiento de excepciones queda fuera del alcance de esta asignatura introductoria. Si el alumno comete un error de este tipo tendrá que reiniciar la simulación.

Ejemplos

```
LDRB R5,   [R9]          @ Carga en el byte 0 de R5 Mem(R9).
                         @ y pone los bytes 1,2 y 3 a 0
LDRB R3,   [R8, #3]      @ Carga en el byte 0 de R3 Mem(R8 + 3).
                         @ y pone los bytes 1,2 y 3 a 0
STRB R4,   [R10, #0x200] @ Almacena el byte 0 de R4 en Mem(R10+0x200)
STRB R10, [R7, R4]       @ Almacena el byte 0 de R10 en Mem(R10+R4)
LDRH R1,   [R0]          @ Carga en los byte 0,1 de R1 Mem(R0).
                         @ y pone los bytes 2,3 a 0
LDRH R8,   [R3, #2]      @ Carga en los byte 0,1 de R8 Mem(R3+2).
                         @ y pone los bytes 2,3 a 0
STRH R2,   [R1, #0X80]   @ Almacena los bytes 0,1 de R2 en Mem(R1+0x80)
```

```
LDRD R4,  [R9]          @ Carga en R4 una palabra de Mem(R9)
                        @ Carga en R5 una palabra de Mem(R9+4)
STRD R8,  [R2, #0x28]   @ Almacena R8 en Mem(R2+0x28)
                        @ y almacena R9 en Mem(R2+0x2C)
```

4.5. Utilizando varios ficheros fuente

Hasta ahora todos 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 nos tendremos que dividir necesariamente el programa en varios ficheros fuente, ya que los ficheros en C deben ser compilados mientras que los ficheros en ensamblador sólo deben ser ensamblados.

Cuando tenemos un proyecto con varios ficheros fuente, utilizaremos en alguno de ellos variables o subrutinas definidas en otro. Como vimos en la práctica 2, 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. En ésta última etapa se deben resolver todas las *referencias cruzadas* entre los ficheros objeto. El objetivo de esta sección es estudiar este mecanismo y su influencia en el código generado.

4.5.1. Tabla de Símbolos

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:

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.

4.5.2. Símbolos globales en C

El cuadro 9 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:

```
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.

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.

4.5.3. Símbolos globales en ensamblador

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 siempre ser declarado global. De este modo además, se evita que pueda haber más de un símbolo `start` en varios ficheros fuente.

Cuadro 9 Ejemplo de exportación de símbolos.

// fichero fun.c

```
//declaración de variable global
//definida en otro sitio
extern int var1;

//definición de var2
//sólo accesible desde func.c
static int var2;

//declaración adelantada de one
void one(void);

//definición de two
//al ser static el símbolo no se
//exporta, está restringida a este
//fichero
static void two(void)
{
   ...
   var1++;
   ...
}

void fun(void)
{
   ...
   //acceso al único var1
   var1+=5;
   //acceso a var2 de fun.c
   var2=var1+1;
   ...
   one();
   two();
   ...
}
```

// fichero main.c

```
//declaración de variable global
//definida en otro sitio (más abajo)
extern int var1;

//definición de var2
//sólo accesible desde main.c
static int var2;

//declaración adelantada de one
void one(void);

//declaración adelantada de fun
void fun(void);

int main(void)
{
   ...
   //acceso al único var1
   var1 = 1;
   ...
   one();
   fun();
   ...
}

//definición de var1
int var1;

void one(void)
{
   ...
   //acceso al único var1
   var1++;
   //acceso a var2 de main.c
   var2=var1-1;
   ...
}
```

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
    ...
```

4.5.4. Mezclando C y ensamblador

Para utilizar un símbolo exportado globalmente desde un fichero ensamblador dentro de un código C, debemos hacer una declaración adelantada del símbolo, como ya hemos visto en la sección 4.5.2.

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`.

4.5.5. Resolución de símbolos

La Figura 4.3 ilustra el proceso de resolución de símbolos 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 fuentes 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 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, guardando el resultado en `r2`. Finalmente se vuelve a cargar en `r3` 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 4.3 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

0x0C000004. La entrada 0x44 corresponde ahora a la dirección 0x0C000048 y contiene el valor 0x0c000000. 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.

4.6. 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 esta asignatura) 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 estamos utilizando herramientas de compilación para un sistema *bare metal*, es decir, sin sistema operativo. En este caso, debemos proporcionar nosotros el código de arranque. El cuadro 10 presenta el código de arranque que utilizaremos de aquí en adelante.

Cuadro 10 Ejemplo de rutina de inicialización.

```
    .extern main
    .extern _stack
    .global start

start:
    ldr sp,=_stack
    mov fp,#0

    bl main
End:
    b End
.end
```

4.7. 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.

4.7.1. 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

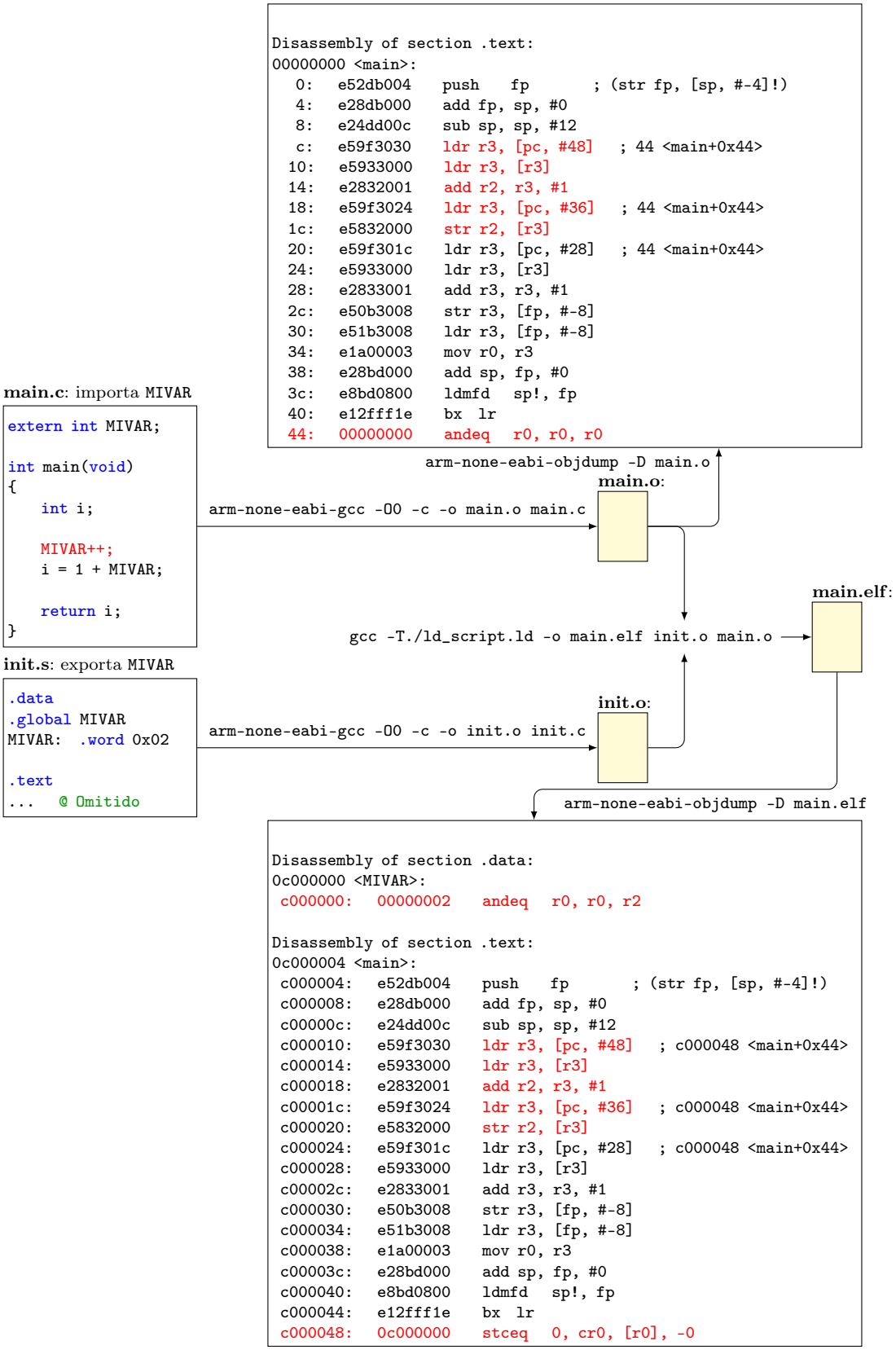


Figura 4.3: Ejemplo de resolución de símbolos.

ser accedido por el nombre de la variable del array seguido de uno o más subíndices encerrados entre corchetes.

Si declaramos una variable global cadena como:

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

el compilador reservará doce bytes consecutivos en la sección `.data`, asignándoles los valores como indica la figura 4.4. 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`).

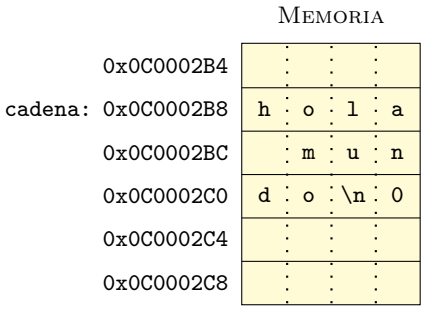


Figura 4.4: Almacenamiento de un array de caracteres en memoria.

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.

4.7.2. 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:

```
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 4.5.

4.7.3. 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

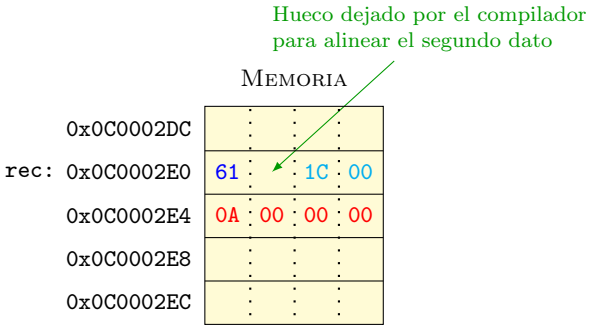


Figura 4.5: Almacenamiento de la estructura *rec*, con los valores de los campos primero, segundo y tercero a `0x61`, `0x1C` y `0x0A` respectivamente.

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 4.7.2. Sin embargo su huella de memoria (*memory footprint*) es muy distinta, como ilustra la figura 4.6.

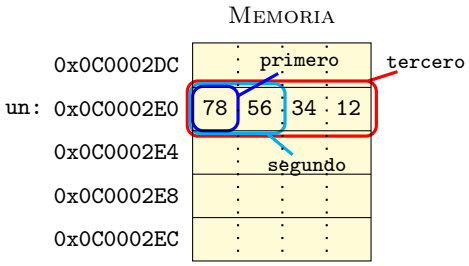


Figura 4.6: Almacenamiento de la union `un`, con los campos primero, segundo y tercero. Un acceso al campo primero da como resultado el byte `0x78`, es decir el carácter `x`. Un acceso al campo segundo da como resultado la media palabra `0x5678` (asumiendo configuración *little endian*), es decir el 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`.

4.8. Eclipse: depurando un programa C

Eclipse ofrece ciertas facilidades para la depuración de un programa C que conviene conocer. Como ejemplo, la Figura 4.7 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 `0x0C000000`, y se han mostrado dos representaciones del mismo, una como enteros y otra como ASCII. Para ver las dos representaciones simultaneamente se ha pulsado el botón  . La pestaña `New Renderings` permite seleccionar distintos formatos de representación.

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`.

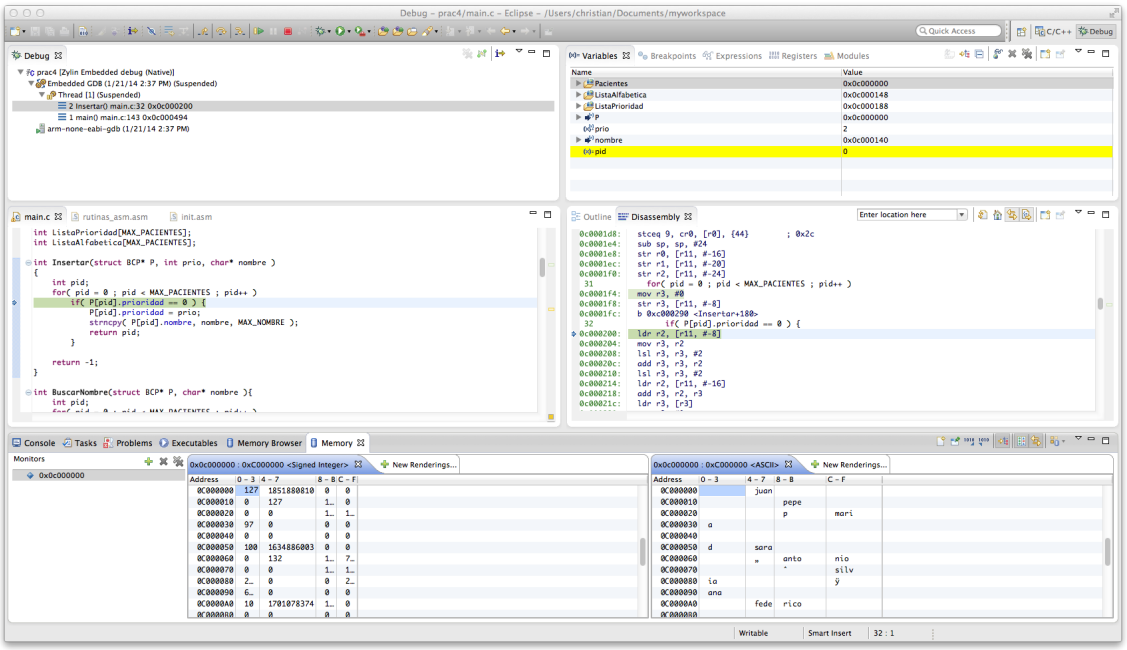


Figura 4.7: Depurando un programa C: variables y visor de memoria.

4.9. Desarrollo de la práctica

En esta práctica partiremos de un proyecto que permite ejecutar un programa C en nuestro sistema de laboratorio. Es un sencillo programa de ejemplo que declara en memoria un

array `Pacientes`. Cada elemento del array es un Bloque de Control de Paciente, representado por una estructura (`struct BCP`) con dos campos: `prioridad`, que debe ser mayor o igual que 1, siendo 1 el más prioritario, y `nombre`, una cadena de caracteres con el nombre del paciente. El array tiene `MAX_PACIENTES` entradas, pudiendo estar algunas libres. Se considera que una entrada está libre cuando el campo prioridad es 0. Los pacientes son identificados por un identificador de paciente (`pid`), que se corresponde con su posición en el array `Pacientes`.

El programa comienza con el array `Pacientes` inicializado en memoria. El array tiene una serie de entradas rellenas y otras vacías. El programa crea dos listas de pacientes, ordenados según dos criterios diferentes: orden de prioridad, y orden alfabético.

Para obtener una lista ordenada se rellena primero la lista con las posiciones ocupadas de la tabla, en el orden original. Después se ordena la lista por inserción de acuerdo al criterio escogido. Para conocer el valor de cada elemento de la lista, se accede al array `Pacientes` con el índice almacenado en la lista. Pongamos un ejemplo para una ordenación por prioridad. Al comenzar tendremos:

```
Pacientes={{5,n1},{3,n2},{0,0},{7,n3},{4,n4},{8,n5},{9,n6},{7,n7}}
Lista={}
```

dónde los `ni` representan los nombres de los pacientes. Empezamos rellenando la lista con los índices de las posiciones ocupadas, obteniendo:

```
Lista={0,1,3,4,5,6,7}
```

Entonces buscamos el elemento de la lista completa con menor valor de prioridad. Este será el elemento en la segunda posición (`Lista(1)=1`), ya que `Pacientes(1) = {3,n2}`. Intercambiamos este elemento con el de la primera posición, resultando:

```
Lista={1,0,3,4,5,6,7}
```

Ya tenemos el primer elemento de `Lista`, pasamos ahora a ordenar el resto repitiendo el proceso anterior. Buscamos el elemento con menor prioridad entre los restantes elementos: `{0,3,4,5,6,7}`. El menor es `Lista(3)=4`, ya que `Pacientes(4)={4,n4}`. Intercambiamos este elemento con el segundo de la lista, quedando:

```
Lista={1,4,3,0,5,6,7}
```

Ahora tenemos los dos primeros elementos de la lista ordenados. Buscamos entonces el elemento con menor prioridad entre los restantes: `{3,0,5,6,7}`. El menor es `lista(3)=0` ya que `Pacientes(0)={5,n1}`. Intercambiamos este con el de la tercera posición, quedando:

```
Lista={1,4,0,3,5,6,7}
```

Si continuamos con el mismo proceso, los cambios sucesivos serían:

```
Lista={1,4,0,3,5,6,7}

Lista={1,4,0,3,7,6,5}

Lista={1,4,0,3,7,5,6}
```

Como vemos, si recorremos el array `Pacientes` en el orden dado por los índices almacenados en `Lista`, iremos de menor a mayor valor del campo prioridad.

El algoritmo descrito se implementa muy fácilmente con dos funciones de apoyo:

`int PosMinPrioridad(int* Lista, struct BCP* P, int ini, int num );`
que nos da la posición en `Lista(ini...num-1)` del elemento con menor prioridad, y

`void Intercambiar(int Lista, int i, int j);`
que intercambia los elementos `i` y `j` en `lista`.

Al comenzar se dan estas todas las funciones implementadas en C, excepto `Intercambiar`, que se da en ensamblador. Hay que analizar el código y depurarlo paso a paso con el fin de afianzar los conceptos expuestos en la parte teórica. Después se proponen algunos ejercicios al alumno. Los pasos a seguir son:

1. Crear un proyecto nuevo en el workspace. Podemos hacerlo copiando alguno de los de las prácticas anteriores.

2. Añadirle un fichero nuevo con el nombre `init.s`. Este fichero será el encargado de inicializar el registro de pila arquitectura para la ejecución de nuestro programa escrito en lenguaje C y de invocar la función de entrada a nuestro programa. Incluir en este fichero el código del cuadro 10.

3. Añadir otro fichero fuente, con el nombre `main.c` y copiar el siguiente contenido:

```
#include <string.h>
#define MAX_PACIENTES 16
#define MAX_NOMBRE 12

struct BCP {
    unsigned int prioridad;
    char nombre[MAX_NOMBRE];
};

struct BCP Pacientes[MAX_PACIENTES] = {
        {127,"juan"},
        {127,"pepe"},
        {112,"maria"},
        {0,""},
        {100,"sara"},
        {132,"antonio"},
        {136,"silvia"},
        {255,"ana"},
        {10,"federico"},
        {0,""}};

#define CRIT_PRIO 0x00
#define CRIT_ALFA 0x01

int ListaPrioridad[MAX_PACIENTES];
int ListaAlfabetica[MAX_PACIENTES];

int PosMinPrioridad(int* Lista, struct BCP* P, int ini, int num )
{
    int i,pid, pos, prio;

    pos  = ini;
    pid  = Lista[ini];
    prio = P[ pid ].prioridad;
```

```
    for( i = ini+1; i < num; i++) {
        pid = Lista[i];
        if( P[pid].prioridad < prio ){
           pos = i;
           prio = P[pid].prioridad;
        }
    }

    return pos;
}

int PosMinAlfabetico(int* Lista, struct BCP* P, int ini, int num )
{
    int i,pid, pos;
    char *nombre;

    pos    = ini;
    pid    = Lista[ini];
    nombre = P[ pid ].nombre;

    for( i = ini+1; i < num; i++) {
        pid = Lista[i];
        if( strncmp( P[pid].nombre, nombre, MAX_NOMBRE ) < 0 ){
           pos    = i;
           nombre = P[pid].nombre;
        }
    }

    return pos;
}

void Intercambiar(int* Lista, int i, int j);

int OrdenaPacientes(int* Lista, struct BCP*  P, unsigned char criterio)
{
    int pid,num,i,j;

    if( (criterio != CRIT_PRIO) && (criterio != CRIT_ALFA) )
        return -1; //error

    // Copiamos los indices de los BCPs ocupados
    // a la lista
    for( pid = 0, num = 0; pid < MAX_PACIENTES ; pid++ )
        if( P[pid].prioridad != 0 ) {
            Lista[num] = pid;
            num++;
        }

    // Ordenamos la lista
    if( criterio == CRIT_PRIO )
        for( i = 0; i < num ; i++ ) {
            j = PosMinPrioridad(Lista, P,i,num);
```

```
            Intercambiar(Lista, i, j );
        }
    else
        for( i = 0; i < num ; i++ ) {
            j = PosMinAlfabetico(Lista, P,i,num);
            Intercambiar(Lista, i, j );
        }

    return num;
}

int main(void)
{
    int num;

    num = OrdenaPacientes( ListaPrioridad, Pacientes, CRIT_PRIO );
    if( num == -1 )
        return -1;

    num = OrdenaPacientes( ListaAlfabetica, Pacientes, CRIT_ALFA );
    if( num == -1 )
        return -1;

    return 0;
}
```

4. Añadir al proyecto otro fichero `rutinas_asm.asm` con el siguiente código:

```
    .global Intercambiar

Intercambiar:
    push {fp}
    mov fp, sp
    sub sp, #4

    ldr r3, [r0, r2, lsl #2]
    str r3, [fp, #-4]
    ldr r3, [r0, r1, lsl #2]
    str r3, [r0, r2, lsl #2]
    ldr r3, [fp, #-4]
    str r3, [r0, r1, lsl #2]

    mov sp, fp
    pop {fp}
    mov pc, lr

    .end
```

5. Configurar el proyecto tal y como se hizo en las prácticas anteriores.

6. Compilar el proyecto y crear el ejecutable.

7. Pasar a la perspectiva `Debug` y crear una configuración de depuración para el proyecto copiando la de la práctica anterior.

8. Comenzar la depuración.

9. Ejecutamos el programa paso a paso analizando lo que sucede.

10. Para evaluar esta práctica se pide al alumno:

    - Obligatorio: considerando que `MAX_NOMBRE` será siempre 12, es decir, que la estructura `struct BCP` ocupará 4 palabras (16 bytes):
      a) Obtener una función C equivalente a `Intercambiar`.
      b) Reemplazar la función `OrdenaPacientes` por una rutina codificada en ensamblador.
    - Opcional (para subir nota):
      c) Reemplazar la función `PosMinPrioridad` por una rutina codificada en ensamblador, asumiendo que `MAX_NOMBRE` será siempre 12.
      d) Modificar las subrutinas ensamblador anteriores considerando que `MAX_NOMBRE` puede tomar cualquier valor entero positivo. La dificultad de este ejercicio radica en calcular la dirección de memoria del i-ésimo elemento de `Pacientes`, que depende del número de bytes ocupado por la estructura `BCP`. El primero de los campos de la estructura es de tamaño palabra, mientras que el segundo es un array de bytes. Por lo tanto la estructura estará alineada a 4 bytes y su tamaño será el múltiplo de 4 bytes más pequeño que nos permita almacenar todos los campos. Por tanto, el número de bytes ocupado por la estructura `BCP` será:

      ```
      4 + ((MAX_NOMBRE+3)/4)*4
      ```

      donde la división es entera.
      En C este cálculo se hace fácilmente aprovechando las operaciones a nivel de bit. Haciendo una `AND` con una máscara de bits que tenga todos los bits a 1 excepto los dos menos significativos (el complemento a 1 de `3`) forzamos los dos bits menos significativos a cero, quedándonos en definitiva con el múltiplo de 4 inmediatamente inferior. La operación quedaría:

      ```
      4 + ( (MAX_NOMBRE+3) & ~0x3 )
      ```

      Si `MAX_NOMBRE` es 12, el resultado sería 16 bytes, como consideramos en la parte obligatoria. En este apartado debemos considerar cualquier valor y por tanto deberemos realizar estas operaciones en ensamblador para indexar correctamente el array.
      e) Obtener una subrutina en ensamblador que implemente la función de la librería estándar de C `strncmp`. Esta función compara dos cadenas de caracteres terminadas en un byte a 0 (`'\0'`). Se pasan las direcciones de comienzo de las dos cadenas como los dos primeros parámetros de la llamada. La función recibe también un tercer argumento entero, que representa el número máximo de caracteres que la función debe comparar. La función devuelve un valor negativo si la primera cadena es *alfabéticamente menor* que la segunda, un número positivo en el caso contrario y 0 si las cadenas son iguales.
      Recordemos que una variable de tipo `char` no es más que un entero (con signo) representado con un sólo byte. Un carácter se representa por un número entero entre 0 y 127, que se corresponde con su código ASCII. Los caracteres aparecen en este código ordenados alfabéticamente, primero en mayúsculas y

luego en minúsculas. Por tanto para comparar las cadenas basta con ir restando a cada carácter de la primera el correspondiente carácter de la segunda. Si el resultado de la resta es negativo la primera cadena es lexicográficamente menor que la segunda, y si es positivo es justo al revés. Sólo en caso de que el resultado de la resta sea 0 deberemos continuar con el siguiente carácter.
En este caso hay que tener en cuenta que los accesos a las cadenas no serán tamaño palabra, sino tamaño byte. Remarcar además que se deben comparar las cadena hasta que se encuentre un byte a 0 en alguna de ellas (indica final de cadena) o hasta que se hayan comparado un número máximo de caracteres indicado por el tercer parámetro de la llamada, lo que suceda antes.

Práctica 5

Introducción al sistema de Entrada/Salida

5.1. Objetivos

El objetivo de esta práctica es tener un primer contacto con los elementos fundamentales de un sistema de entrada/salida. Para ello utilizaremos la placa S3CEV40 descrita en clase, y únicamente haremos operaciones de E/S mediante encuesta. El estudio del sistema de interrupciones queda por tanto expresamente excluido de los objetivos de esta asignatura. Como ejemplo práctico usaremos los dispositivos más sencillos de los que disponemos: LEDs, pulsadores y el display de ocho segmentos. Los principales objetivos de la práctica son:

- Conocer el sistema de E/S de la placa S3CEV40 usada en el laboratorio
- Comprender el mecanismo de E/S localizado en memoria
- Aprender a programar dispositivos básicos mediante encuesta

5.2. Sistemas de Memoria y Entrada/Salida

La figura 5.1 representa de forma esquemática los sistemas de memoria y de entrada/salida de la placa S3CEV40, donde hemos omitido deliberadamente los componentes relacionados con el sistema de interrupciones. Los principales componentes son el procesador ARM7TDMI, el controlador de memoria, los controladores de E/S (tanto internos como externos al chip principal de la placa, Samsung S3C44B0X), y la lógica de selección, que se sirve de las señales generadas por el controlador de memoria para seleccionar el chip externo con el que se desea trabajar. A continuación describiremos brevemente algunas de las características que nos interesan de cada uno de estos componentes.

5.2.1. El ARM7TDMI

El ARM7TDMI es un procesador de la familia ARMv4, cuyo repertorio de instrucciones hemos venido estudiando en las prácticas anteriores. Tiene un bus de direcciones de 32 bits, por lo que es capaz de direccionar potencialmente un espacio total de 4GB de memoria. Este espacio es compartido por el sistema de E/S (*E/S localizada en memoria*), por lo que las direcciones pueden referirse tanto a posiciones de memoria principal como a elementos internos de los controladores de E/S (registros o memoria local). En ambos casos el ARM7TDMI

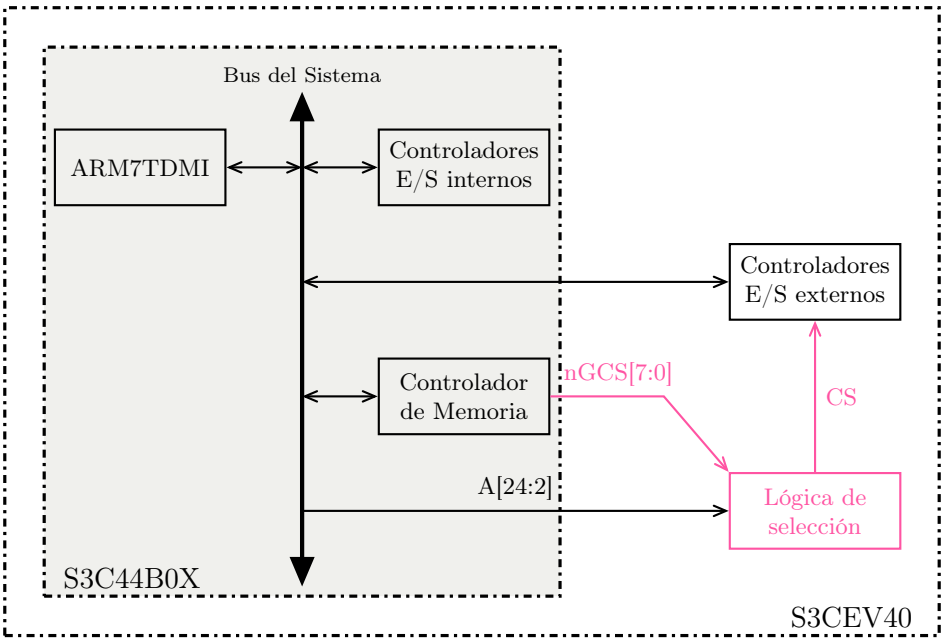


Figura 5.1: Sistema de E/S de la placa de laboratorio.

realiza el acceso del mismo modo y es responsabilidad del programador saber a que dispositivo está asociada cada dirección. Esta asignación de rangos de direcciones a dispositivos suele recibir el nombre de *mapa de memoria del sistema*.

5.2.2. El controlador de memoria

El controlador de memoria es el responsable de actuar de interfaz entre los módulos de memoria externos (ROM o RAM) y el bus del sistema. Es fácil adivinar que el *mapa de memoria del sistema* viene determinado en gran medida por las características y configuración de este componente del sistema de memoria.

El comportamiento del controlador de memoria podría resumirse del siguiente modo. Cuando el procesador pone una dirección en el bus:

- Si esta se encuentra dentro del rango del propio controlador de memoria, se encarga de generar las señales necesarias para realizar el acceso al módulo de memoria que corresponda.
- Si, por el contrario, la dirección queda fuera de su rango de competencia, el controlador se inhibe, ya que supuestamente es responsabilidad de algún controlador de E/S atender a dicho acceso (lectura/escritura de un registro o memoria local).

Si el acceso a memoria falla (p.ej. se realiza el acceso a un banco para el que no hay asignado módulo de memoria), es responsabilidad del controlador detectar el error y generar una excepción de *Abort*. El tratamiento de excepciones queda fuera del alcance de esta asignatura.

En la placa S3CEV40, tanto el procesador ARM7TDMI como el controlador de memoria, así como otros controladores y dispositivos de E/S, se encuentran integrados dentro un mismo chip, el *System on Chip* S3C44B0X de Samsung, como ilustra la Figura 5.1.

El controlador de memoria del S3C44B0X reduce el espacio de direcciones efectivo a 256MB y lo divide en 8 regiones o rangos independientes, denominados *bancos*, tal y como ilustra la figura 5.2. Cada banco puede ser asignado a un chip de memoria externo distinto (módulo), aunque sólo los dos últimos bancos admiten cualquier tipo de memoria (SROM, SRAM o SDRAM).

Cuando se accede a una dirección dentro del rango de uno de estos bancos, además de las señales necesarias para realizar el acceso al módulo al que está asociado, el controlador de memoria activa una señal nGCS[1]. Estas señales se utilizan externamente para seleccionar/activar el chip correspondiente.

De todo el espacio de memoria, la parte alta del primer banco, correspondiente al rango `0x01C00000-0x01FFFFFF`, está reservada para los puertos de E/S de los controladores integrados dentro del S3C44B0X. Como hemos mencionado previamente, cuando el procesador realiza un acceso dentro este rango el controlador de memoria se inhibe (i.e. no genera señal alguna).

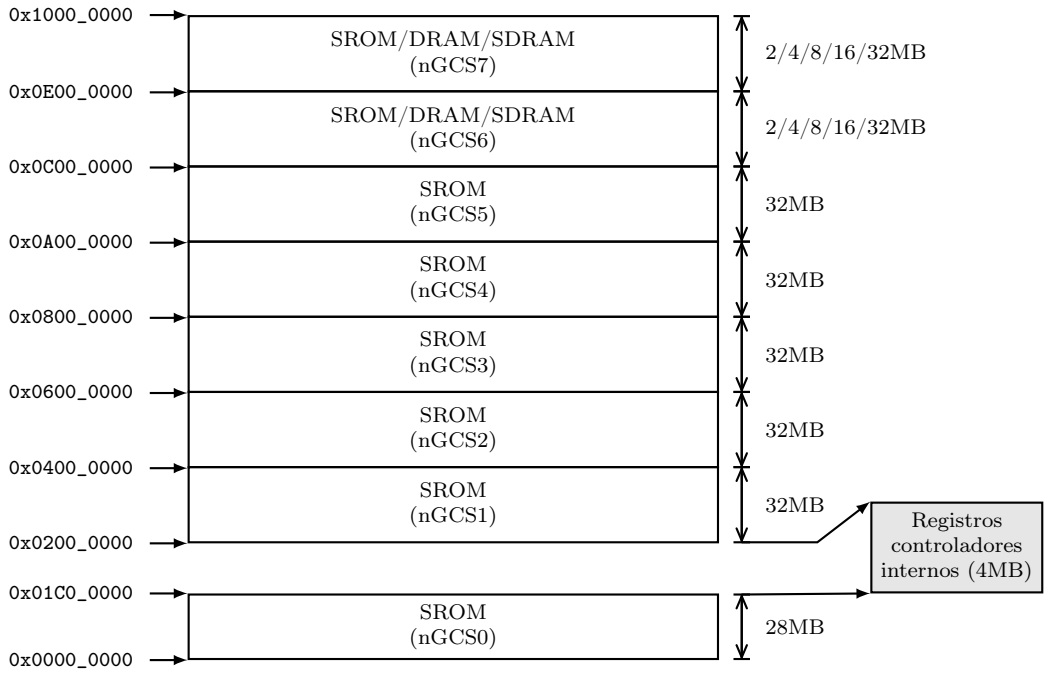


Figura 5.2: Mapa de memoria del S3C44B0X (SROM se refiere tanto a ROM como a SRAM).

La placa S3CEV40 dispone tan sólo de dos módulos de memoria de distinto tipo:

- Una memoria ROM Flash de 1Mx16bits, ubicada en el banco 0 del S3C44B0X y cuyo rango de direcciones es `[0x00000000-0x001FFFFF]`.
- Una memoria SDRAM de 4Mx16bits, ubicada en el banco 6 del S3C44B0X y cuyo rango de direcciones es `[0x0C000000-0x0C7FFFFF]`.

Como podemos ver, del espacio de 256MB que el controlador de memoria es capaz de gestionar, en nuestra placa sólo tenemos disponibles 10MB, 2 de ROM Flash y 8 de SDRAM.

Antes de poder usar estos módulos de memoria debe configurarse el controlador de manera adecuada. Cabe preguntarse entonces, cómo puede llevarse a cabo esta configuración si

[1]Por convenio, las señales cuyo nombre comienza con "n" se activan en baja.

al encender o reiniciar el sistema no se dispone aún de memoria. Afortunadamente la configuración mínima para poder acceder al Banco-0, en el que está ubicada la ROM Flash, se puede preestablecer externamente mediante diversos pines de entrada del S3C44B0X[2] cuyo valor se fija en la propia placa de laboratorio[3]. El resto de la configuración del controlador de memoria puede realizarlo con posterioridad el código de arranque (*boot*) contenido en la ROM Flash.

En fase de desarrollo es habitual que la configuración la realice el depurador al conectarse a la placa, escribiendo directamente en los registros del controlador de memoria a través del puerto de depuración. El chip que utilizamos en el laboratorio tiene un puerto JTAG para la depuración en circuito. Para conectar nuestro entorno de desarrollo a este puerto utilizaremos un programa intermediario llamado Open On Chip Debugger (OpenOCD). Este programa abrirá un puerto al que se podrá conectar Eclipse para realizar la depuración mientras el código se ejecuta directamente sobre el procesador. Veremos más adelante que en la configuración de depuración de Eclipse deberemos escribir algunas órdenes para OpenOCD que se encargarán de preparar el procesador, entre otras cosas, configurando adecuadamente el controlador de memoria.

5.2.3. Controladores de dispositivos de E/S y lógica de selección

Salvo contadas excepciones, los dispositivos de E/S se gestionan a través de controladores específicos de E/S. Estos son los que actúan de interfaz entre el dispositivo por un lado, y el bus de sistema por otro lado. Para comunicarse con el dispositivo, el procesador lee y escribe en los registros internos de estos controladores, a los que nos referiremos habitualmente como puertos de E/S.

Como mencionamos antes, el rango `0x01C00000-0x01FFFFFF` está reservado para los registros correspondientes a los controladores internos al S3C44B0X (E/S de propósito general, de memoria,...), y el controlador de memoria se inhibe cuando el procesador realiza una acceso en este rango.

Sin embargo, los puertos de E/S correspondientes a controladores externos al S3C44B0X se ubican fuera del rango de direcciones correspondiente a los controladores internos y requieren de una lógica adicional externa para su selección (ver figura 5.1). Esta lógica es la encargada de activar la señal de *Chip Select* (CS) correspondiente al dispositivo que se pretende acceder, en función de la dirección que el procesador pone en el bus.

Obviamente, no pueden asignarse puertos de E/S a bancos que tengan asignados módulos de memoria. La tabla 5.1 muestra el rango de direcciones y la señal de selección que el fabricante de la placa ha asignado a cada uno de ellos. Como podemos ver, cada dispositivo tiene asignado un rango de direcciones dentro del Banco 1 (ver Figura 5.2).

Para simplificar la lógica de selección de dispositivos externos, el fabricante de la placa se ha servido de las señales nGCS0-7 que genera el controlador de memoria, como ilustra la Figura 5.3. El circuito consiste en un simple decodificador de 3 a 8 (chip 74LV138), que se activa cuando se realiza un acceso al Banco 1 (debido a la conexión de la señal nCGS1 a la entrada *enable* del decodificador). Las salidas del decodificador activan entonces una de las señales CS de alguno de los dispositivos ubicados sobre el Banco 1. Podemos ver también

[2] Al realizar un Reset, el valor de los pines ENDIAN y OM[1:0] determinan la ordenación de los bytes (*little-* o *big-endian*) y el ancho del bus del Banco-0 (8/16/32bits).

[3] El valor de la señal ENDIAN viene determinado por la posición del *jumper* SW5 (por defecto *little-endian* y el valor de OM[1:0] se fija mediante conexiones cableadas a `b01`.

Tabla 5.1: Rango asignado a cada uno de los dispositivos ubicados en el Banco-1.

Dispositivo	CS	Dirección
USB	CS1	0x0200_0000 - 0x0203_FFFF
Nand Flash	CS2	0x0204_0000 - 0x0207_FFFF
IDE (IOR/W)	CS3	0x0208_0000 - 0x020B_FFFF
IDE (KEY)	CS4	0x020C_0000 - 0x020F_FFFF
IDE (PDIAG)	CS5	0x0210_0000 - 0x0213_FFFF
8-SEG	CS6	0x0214_0000 - 0x0217_FFFF
ETHERNET	CS7	0x0218_0000 - 0x021B_FFFF
LCD	CS8	0x021C_0000 - 0x021F_FFFF

que el dispositivo se selecciona en función de los bits 18-20 del bus de direcciones, dando lugar a los ocho rangos de direcciones de la Tabla 5.1.

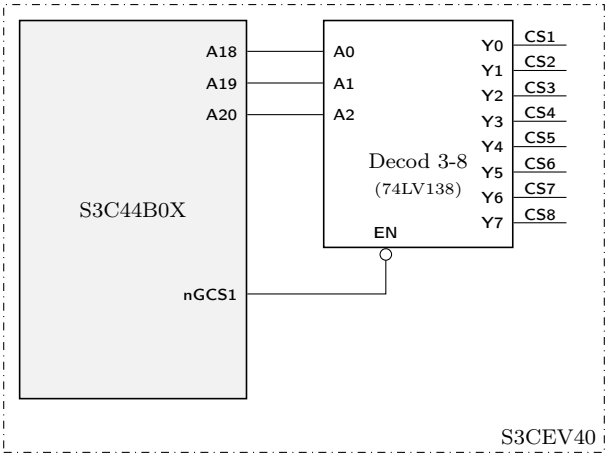


Figura 5.3: Lógica para la selección de dispositivos externos. Es conveniente recordar que la señal nGCS1 del controlador de memoria se activa a baja cuando se referencia una dirección del Banco-1.

5.3. Controlador de pines de E/S

El controlador de pines de E/S de propósito general (GPIO) mantiene el control de 71 pines multifuncionales, es decir, cada uno de estos pines puede ser utilizado para más de una función distinta. El GPIO permite configurar la funcionalidad escogida para cada uno de estos pines.

Los 71 pines están divididos en 7 grupos, a los que denominamos puertos, que son:

- 2 grupos de 9 bits de E/S (Puertos E y F)
- 2 grupos de 8 bits de E/S (Puertos D y G)
- 1 grupo de 16 bits de E/S (Puerto C)
- 1 grupo de 11 bits de E/S (Puerto B)

- 1 grupo de 10 bits de E/S (Puerto A)

Cada puerto es gestionado mediante 2-4 registros. El número concreto depende del puerto. Al realizar un *Reset* estos registros se cargan a un valor seguro para no dañar ni el sistema ni el hardware que pueda haber conectado a estos pines.

En esta práctica vamos a utilizar sólo algunos de los pines de E/S de los puertos B y G. La tabla 5.2 recoge una relación de sus registros de datos y control, que describiremos en detalle a continuación. Para obtener una descripción completa es preciso consultar el manual del S3C44B0X [um-].

Tabla 5.2: Registros de los puertos B y G

Registro	Dirección	R/W	Valor de Reset
PCONB	0x01D20008	R/W	0x7FF
PDATB	0x01D2000C	R/W	Undef
PCONG	0x01D20040	R/W	0x00
PDATG	0x01D20044	R/W	Undef
PUPG	0x01D20048	R/W	0x00

5.3.1. Puerto B

El puerto B tiene 11 bits que admiten dos funcionalidades: podemos utilizar estos pines como pines de salida (para escribir un valor directamente en ellos) o podemos conectarlos a algunas señales generadas por el controlador de memoria (*i.e.* hacemos que estas señales salgan por estos pines fuera del S3C44B0X).

La configuración del puerto se realiza a través de dos registros del GPIO:

- *PCONB*, registro de control que selecciona la funcionalidad de cada uno de los pines. Su descripción está en la tabla 5.3.
- *PDATB*, registro de datos que permite escribir los bits que se pretenden sacar por el puerto (sólo útil para aquellos pines configurados como salida).

En esta práctica utilizaremos dos pines del puerto B configurados como salida, para activar o desactivar dos LEDs conectados a ellos.

Es importante hacer notar que las señales del controlador de memoria salen fuera del chip a través del puerto B. Por lo tanto debemos tener mucho cuidado de no tocar la configuración de los pines que no estén conectados a los LEDs, ya que un error nos puede dejar sin memoria y tendríamos que resetear el sistema.

5.3.2. Puerto G

El puerto G tiene ocho bits que pueden configurarse de cuatro formas diferentes. La configuración del puerto se realiza mediante tres registros del GPIO:

- *PCONG*, registro de control que permite seleccionar la funcionalidad de cada pin, tal y como se describe en la tabla 5.4.
- *PDATG*, registro de datos que permite escribir o leer del puerto.

Tabla 5.3: Configuración de los pines del puerto B.

PCONB	Bit	Descripción
PB10	[10]	0 = Salida 1 = nGCS5
PB9	[9]	0 = Salida 1 = nGCS4
PB8	[8]	0 = Salida 1 = nGCS3
PB7	[7]	0 = Salida 1 = nGCS2
PB6	[6]	0 = Salida 1 = nGCS1
PB5	[5]	0 = Salida 1 = nWBE3/nBE3/DQM3
PB4	[4]	0 = Salida 1 = nWBE2/nBE2/DQM2
PB3	[3]	0 = Salida 1 = nSRAS/nCAS3
PB2	[2]	0 = Salida 1 = nSCAS/nCAS2
PB1	[1]	0 = Salida 1 = SCLK
PB0	[0]	0 = Salida 1 = SCKE

- *PUPG*, registro de configuración que permite activar (bit a '0') o no (bit a '1') una resistencia de *pull-up*[4] por cada pin.

Tabla 5.4: Configuración de los pines del puerto G.

PCONG	Bits	Descripción
PG7	15:14	00 = Entrada 01 = Salida 10 = IISLRCK 11 = EINT7
PG6	13:12	00 = Entrada 01 = Salida 10 = IISDO 11 = EINT6
PG5	11:10	00 = Entrada 01 = Salida 10 = IISDI 11 = EINT5
PG4	9:8	00 = Entrada 01 = Salida 10 = IISCLK 11 = EINT4
PG3	7:6	00 = Entrada 01 = Salida 10 = nRTS0 11 = EINT3
PG2	5:4	00 = Entrada 01 = Salida 10 = nCTS0 11 = EINT2
PG1	3:2	00 = Entrada 01 = Salida 10 = VD5 11 = EINT1
PG0	1:0	00 = Entrada 01 = Salida 10 = VD4 11 = EINT0

n esta práctica utilizaremos dos pines del puerto G que están conectados a dos pulsadores de la placa. Por tanto tendremos que seleccionar la configuración 00 (Entrada) para estos pines.

[4]Una resistencia de *pull-up* pone la entrada a uno cuando está desconectada, evitando que se quede en un valor indefinido. De forma similar, una resistencia de *pull-down* pone la entrada a cero cuando está desconectada.

5.4. LEDs y pulsadores

En la placa S3CEV40 hay dos pulsadores y dos LEDs conectados al sistema S3C44BOX como indica la figura 5.4.

Los dos pulsadores se conectan a los pines 6 y 7 del puerto G. Configurando estos pines como entrada leeremos un 0 en el bit correspondiente del registro *PDATG* cuando el pulsador esté pulsado. Para asegurarnos que los pines tomen el valor 1 lógico (Vdd) cuando el pulsador no esté pulsado es necesario activar las resistencias de *pull-up* de estos pines mediante el registro *PUPG*. Sin embargo, cuando se produzca una pulsación en realidad no se producirá una bajada *limpia* de la tensión de Vdd a 0, sino que la tensión oscilará (entre Vdd y 0) hasta que finalmente se estabilice en 0. Solemos decir que se producen rebotes de la señal. Esto puede hacer que detectemos varias pulsaciones cuando sólo se ha producido una. Para filtrar estos rebotes debemos esperar un tiempo cuando detectemos una pulsación. El tiempo a esperar puede ajustarse empíricamente, pero 100ms suele ser suficiente. En esta práctica haremos esta espera con una función `Delay` que básicamente consiste en dos bucles anidados que no hacen nada (cuerpo del bucle interno vacío)[5].

Por otro lado, los LEDs se conectan a los pines 9 y 10 del puerto B, como ilustra la Figura 5.4. Para encender los leds deberemos configurar los pines 9 y 10 como salida (ver tabla 5.3) y escribir en ellos un '0' por medio del registro *PDATB*, ya que los ánodos se conectan a la alimentación (Vdd) y los cátodos a los pines.

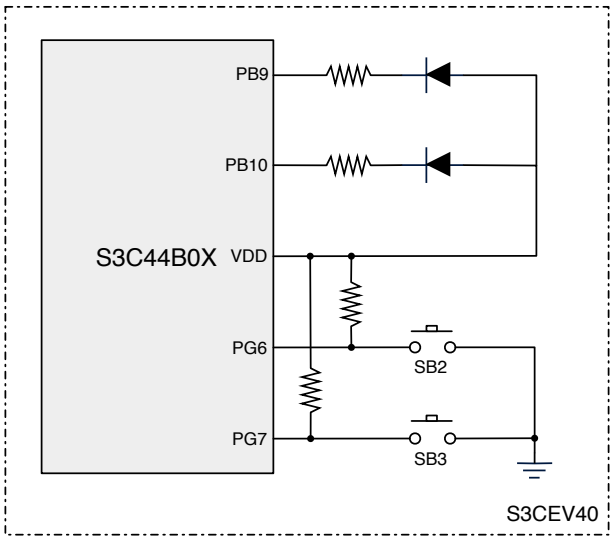


Figura 5.4: Esquema de conexión de pulsadores y LEDs de la placa Embest S3CEV40.

5.5. Display 8-segmentos

En la placa S3CEV40 también tenemos un display de 8 segmentos, que consta de 7 LEDs para conformar cualquier dígito hexadecimal y otro LED adicional para el punto decimal.

[5]La función `Delay` que se proporciona utiliza un contador hardware del chip para calibrarse, determinando el número de iteraciones que debe hacer su bucle interno para que la espera se pueda especificar en múltiplos de $100\mu s$.

Este display es de tipo ánodo común, lo significa que para encender los LEDs hay que ponerlos a cero.

La Figura 5.5 muestra la conexión del display a la lógica de selección externa. Como podemos ver, el latch conectado al display de 8 segmentos se activa con la señal CS6, cada vez que accedemos a alguna dirección del rango `0x0214_0000-0x0217_FFFF`. Mientras la entrada LE (*Latch Enable*) permanezca activa el latch dejará pasar el byte definido por los 8 bits menos significativos del bus de datos. Cuando esta señal se desactiva el latch mantiene el último valor que le ha llegado.

Sin embargo, el banco 1 tiene una anchura de 8 bits. Esto quiere decir que si hacemos una escritura de tamaño palabra el controlador de memoria enviará por separado los 4 bytes que componen la palabra. Como el procesador se configura como little-endian se escribirían los bytes en orden de su peso (0,1,2 y 3), quedando en el latch el byte más significativo, en lugar del menos significativo. **Por este motivo es muy importante que siempre se realicen escrituras de tamaño byte en el display de 8 segmentos, utilizando la instrucción** `strb`.

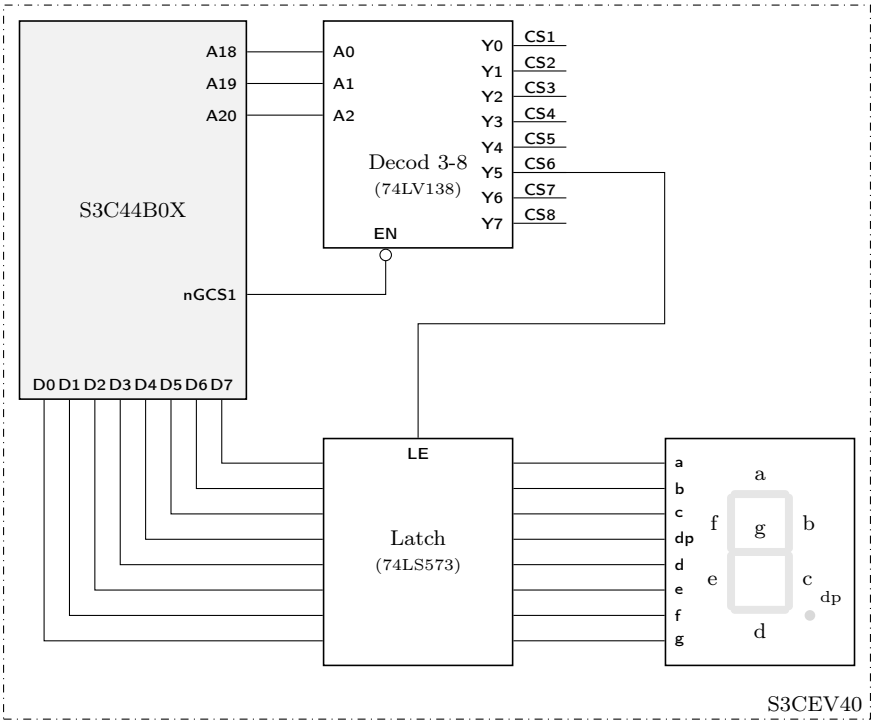


Figura 5.5: Esquema de conexión del display de 8 segmentos en la placa S3CEV40. A[20:18] son bits de direcciones y D[7:0] son bits de datos del bus del sistema del S3C44B0X. Es conveniente recordar que la señal nGCS1 del controlador de memoria se activa a baja cuando se referencia una dirección del Banco-1.

5.6. 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*.

5.6.1. 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 ver qué bits 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
```

5.6.2. 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 consultar ejecutar un código sólo si el valor bit 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;
}
```

5.7. Entrada/Salida en C

Como hemos venido explicando a lo largo de la práctica, un programa dedicado a atender dispositivos de E/S tendrá que leer y escribir en los registros de los controladores. Como en nuestra plataforma destino la E/S está localizada en memoria, cada puerto (registro de E/S) tiene asignada una dirección de memoria. Por tanto para leer del registro nos bastará con utilizar una instrucción `ldr` y para escribir en él un `str`. ¿Quiere esto decir que sólo podemos hacer la E/S en Ensamblador? Por fortuna la respuesta es no, gracias a que C incluye un tipo de datos para manejar direcciones de memoria, el puntero.

5.7.1. 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 `ldr` 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 `str` 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 `ldr/str` será la adecuada para el tamaño del dato apuntado. Por ejemplo, si es un puntero `char*` se utilizarían las instrucciones `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, supongamos que queremos poner a 0 el bit 6 del registro `PDATB` para encender el led conectado al pin 6 del puerto B. En ensamblador haríamos lo siguiente:

```
ldr r0,=0x1d2000c @ Cargamos en r0 la dirección de PDATB
ldr r1, [r0]
and r1, r1, #~(0x1<<6)
str r1, [r0]
```

y en cambio en C haríamos:

```
// declaramos un puntero rPDATB y lo inicializamos
unsigned int* rPDATB = (unsigned int*) 0x1d2000c;
// lo usamos para leer el registro
// y volver a escribir un nuevo valor
*rPDATB = *rPDATB & ~( 0x1 << 6);
```

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/str`. 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 más `i` veces el número de bytes ocupados por el tipo apuntado, es decir, es equivalente a poner `*(p+i)`.

5.7.2. 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 de los pulsadores

conectados a los pines 6 y 7 del puerto G, esperando a que se pulse uno de ellos. El código podría ser el siguiente:

```
#define BUTTONS (0x3 << 6)
unsigned int* rPDATG = (unsigned int*) 0x1d20044;
unsigned int status;

...

do {
    status = ~( *rPDATG );
    status = status & BUTTONS;
} while( status == 0 );
```

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

```
    .equ rPDATG, 0x1d20044

    .text
...

    ldr r0, =rPDATG
DO:
    ldr r1,[r0]
    mvn r1, r1
    and r1, r1, #(0x3 << 6)
    cmp r1, #0
    beq DO
```

Fijémonos que dentro del bucle no se modifica lo que hay en la dirección `0x1d20044`, sólo se lee. El compilador podría entonces *optimizar* el código sacando el `ldr` fuera del bucle, con lo que quedaría:

```
    .equ rPDATG, 0x1d20044

    .text
...

    ldr r0, =rPDATG
    ldr r1,[r0]
DO:
    mvn r1, r1
    and r1, r1, #(0x3 << 6)
    cmp r1, #0
    beq DO
```

Entonces si no hay ningún pulsador 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 `0x1d20044` 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 los pulsadores sería:

```
#define BUTTONS (0x3 << 6)
volatile unsigned int* rPDATG = (unsigned int*) 0x1d20044;
```

```
unsigned int status;

...

do {
    status = ~( *rPDATG );
    status = status & BUTTONS;
} while( status == 0 );
```

En este caso el compilador nunca sacará el `ldr` fuera del bucle.

5.7.3. Uso de macros para acceder a los puertos de E/S

Cuando se programa la E/S en C, es frecuente utilizar macros del preprocesador para el acceso a los puertos, en lugar de definir punteros. Esto permite al compilador generar un código más eficiente. ¿Pero no hemos dicho que necesitamos los punteros? Así es, pero C permite convertir un literal a un puntero mediante un *casting*. Esto podemos hacerlo directamente en una macro del preprocesador. Además incluyendo la desreferenciación en la macro podemos obtener un código más sencillo de leer. Por ejemplo, el código anterior para el escaneo de los pulsadores podría quedar así:

```
#define BUTTONS (0x3 << 6)
#define rPDATG (*(volatile unsigned int*) 0x1d20044)

unsigned int status;

...

do {
    status = ~( rPDATG );
    status = status & BUTTONS;
} while( status == 0 );
```

En esta práctica incluiremos todas las macros para el acceso a los puertos B y G y al display de 8 segmentos en un fichero de cabecera `ports.h`. Bastará entonces con incluir este fichero (con la directiva `#include`) en los ficheros fuente en los que queramos trabajar con alguno de los puertos de E/S.

5.7.4. Estructura de un programa dedicado a Entrada/Salida

Determinadas aplicaciones están enfocadas únicamente a realizar tareas de E/S: leer valores de sensores (temperatura, luz, posición), teclado u otros, y actuar sobre elementos externos al procesador (motores, displays, leds) en función de la información obtenida en la lectura. En esos casos, la estructura del código de dichas aplicaciones, extremadamente simplificada y asumiendo una E/S basada en encuesta y espera activa, podría esquematizarse como sigue:

```
función principal (   ) {

        // Configurar dispositivos como entrada y/o salida
        inicialización_de_dispositivos;

        while (true) {
```

```
                // Rutina(s) para leer los valores de cada dispositivo
                // A su vez, contendrán bucles de espera para el dispositivo
                lectura_dispositivos_de_entrada;

                calcular_salidas_en_función_de_etradas;

                escritura_dispositivos_de_salida;
        }
}
```

En el desarrollo de esta práctica seguiremos fielmente ese modelo.

5.8. Configuración de un proyecto para la placa S3CEV40

Para esta práctica añadiremos como de costumbre un nuevo proyecto a nuestro `workspace`. Toda la configuración de compilación se hace igual que siempre. Sin embargo, en esta ocasión queremos ejecutar nuestro código sobre el arm7tdmi de la placa S3CEV40, en lugar de utilizar el simulador, por lo que no nos vale la configuración de depuración que hemos venido utilizando.

Lo primero que debemos hacer es configurar Eclipse para utilizar una herramienta externa. 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 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 5.6 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/arm-fdi-ucm.cfg`) tal y como se indica en la Figura 5.6. 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.

Además de añadir OpenOCD como herramienta externa debemos crear una nueva configuración de depuración. Los pasos iniciales se dieron en la práctica 1. Lo único que cambia es que en esta ocasión en lugar de seleccionar `Zylin` debemos seleccionar `GDB Hardware Debugging`, como muestra la Figura 5.7. Como en las prácticas anteriores, damos un nombre a la configuración, seleccionamos el proyecto y el ejecutable. En este caso debemos además pinchar en un enlace azul que aparece en la base de la ventana que pone Select Other.... Se abrirá entonces una ventana como la mostrada en la Figura 5.8, en donde deberemos marcar la casilla `Use configuration specific settings` y seleccionar `Standard GDB Hardware Debbuging Launcher`.

A continuación debemos seleccionar la pestaña `Debugger`, y rellenarla como indica la Figura 5.9. Primero deberemos seleccionar como depurador el `arm-none-eabi-gdb`. Después, en la parte baja de la ventana, deberemos seleccionar a qué *gdb server* debe conectarse. Este servidor lo pone OpenOCD, y está configurado para escuchar en el puerto 3333. Por ello seleccionaremos como JTAG Device `Generic TCP/IP`, como dirección IP `localhost` y como puerto 3333.

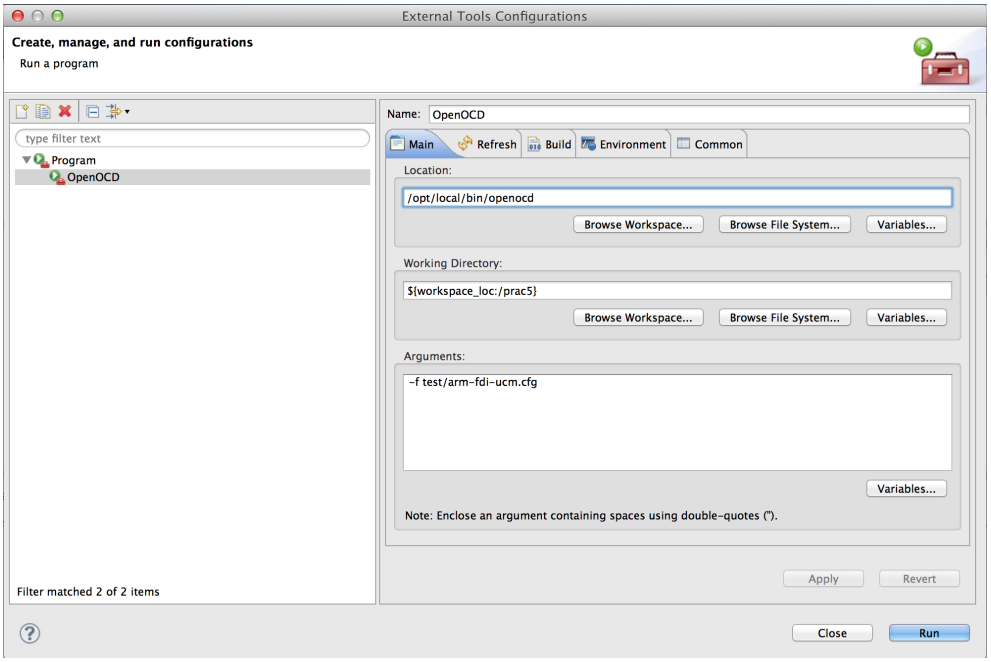


Figura 5.6: Ventana de configuración de OpenOCD como herramienta externa.

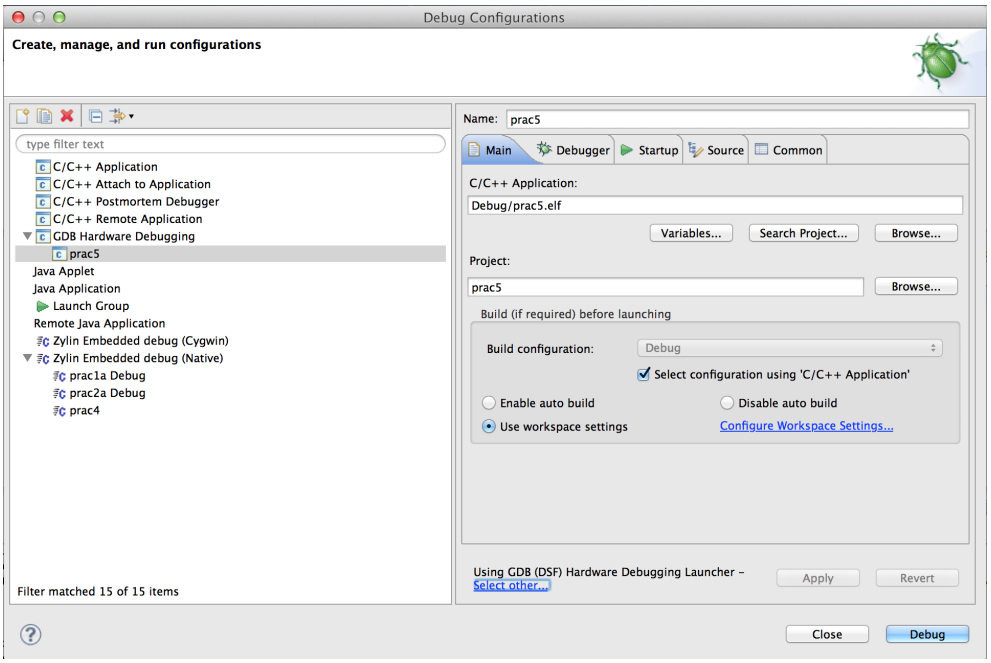


Figura 5.7: Configuración de depuración para conexión a placa: pestaña Main.

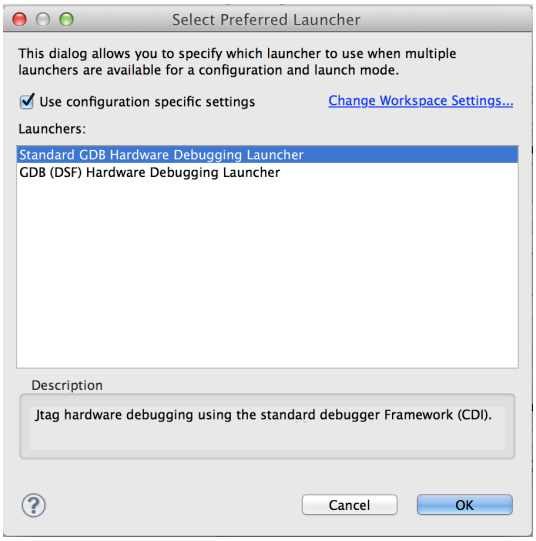


Figura 5.8: Configuración de depuración para conexión a placa: selección de *Standard GDB Hardware Debbuging Launcher*.

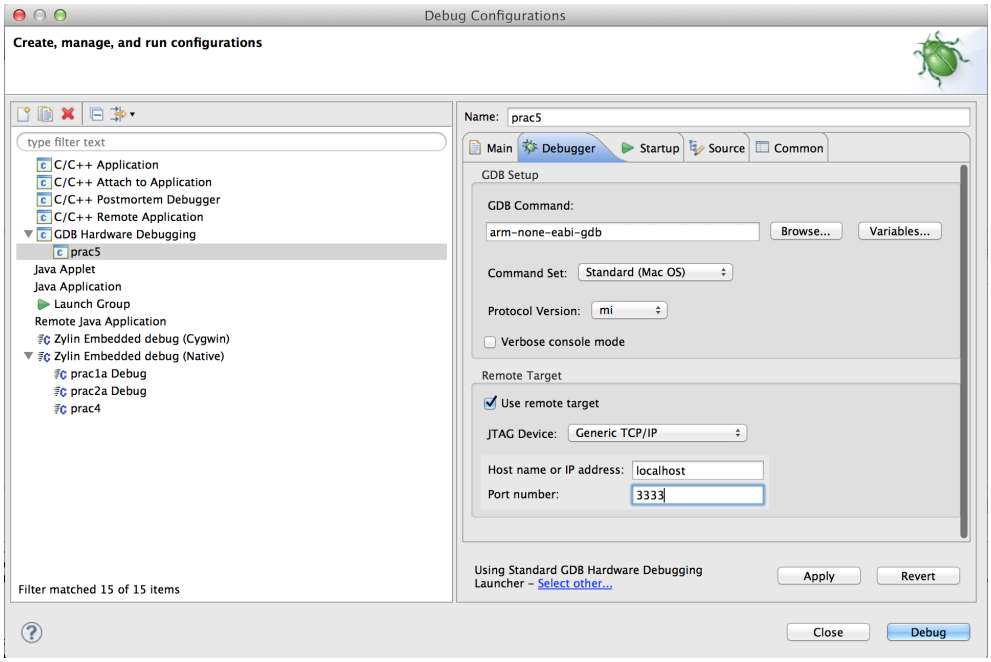


Figura 5.9: Configuración de depuración para conexión a placa: pestaña Debugger.

Finalmente deberemos rellenar la pestaña `Startup`, tal y como se ilustra en la Figura 5.10. En la parte superior desmarcamos las casillas de `Reset` y `Halt`, y escribimos en el cuadro `monitor reset init`. En la parte inferior de la ventana marcamos las casillas `Set Breakpoint at` y `resume`, y ponemos `*start` en la casilla para la dirección del breakpoint. Pinchamos en `Apply` y luego en `Close`. La configuración de depuración queda entonces lista para ser utilizada cuando queramos depurar.

5.9. Desarrollo de la Práctica

En esta práctica vamos a partir de un programa que hace que los dos leds de la placa parpadeen constantemente a una frecuencia de 1Hz aproximadamente. El programa estará compuesto de varios ficheros fuente, la mayor parte de ellos implementan funciones para actuar sobre un dispositivo de la placa. Sin embargo, algunas de estas funciones no estarán implementadas. El alumno deberá completar la implementación de estas funciones. Una vez completadas, se propone al alumno una modificación del programa principal que utilice todas estas funciones. Los pasos a seguir son:

1. Añadimos al proyecto los siguientes ficheros:

   - `ports.h`: definición de macros para el acceso a los puertos de E/S.

     ```
     #ifndef PORTS_H_
     #define PORTS_H_

     // GPIO
     #define rPDATB    (*(volatile unsigned int*) 0x1d2000c)
     #define rPCONB    (*(volatile unsigned int*) 0x1d20008)
     #define rPCONG    (*(volatile unsigned int*) 0x1d20040)
     #define rPDATG    (*(volatile unsigned int*) 0x1d20044)
     #define rPUPG     (*(volatile unsigned int*) 0x1d20048)

     // DISPLAY
     #define LED8ADDR (*(volatile unsigned char*) 0x2140000)

     // Watchdog
     #define rWTCON      (*(volatile unsigned *) 0x1d30000)
     #define rWTDAT      (*(volatile unsigned *) 0x1d30004)
     #define rWTCNT      (*(volatile unsigned *) 0x1d30008)

     #endif
     ```

   - `button.h`: declaración de funciones para manejar los pulsadores.

     ```
     #ifndef BUTTON_H_
     #define BUTTON_H_

     #define PULSADOR1 0x01
     #define PULSADOR2 0x02

     void button_init( void );
     unsigned int read_button( void );

     #endif
     ```

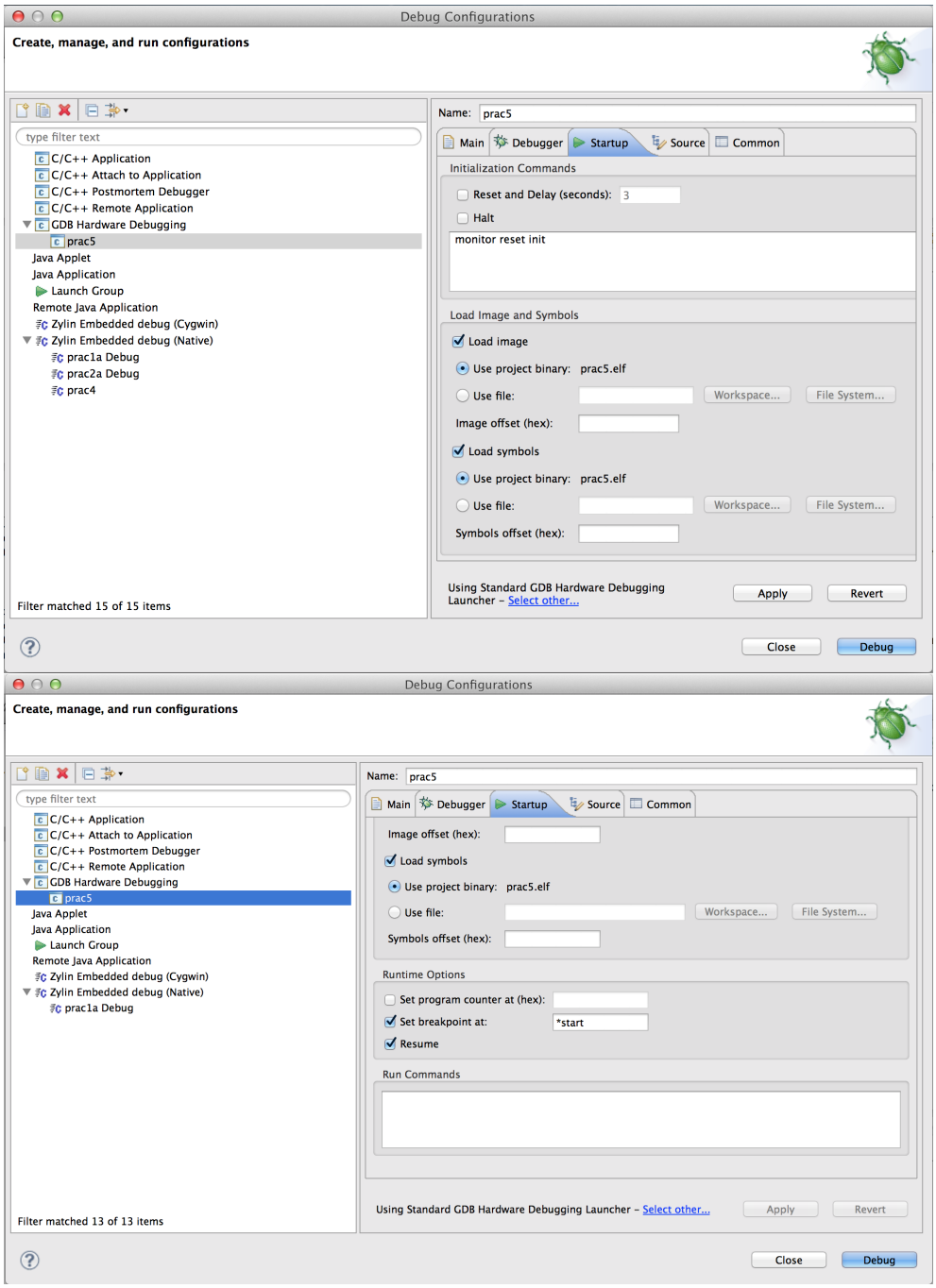


Figura 5.10: Configuración de depuración para conexión a placa: pestaña Startup.

- `button.c`: implementación de las funciones para manejar los pulsadores.

```
#include "ports.h"
#include "button.h"

// macros para seleccionar
// los bits 6 y 7 del puerto
#define BUTTONS (0x3 << 6)


void button_init( void )
{
    rPCONG &= ~( 0xF << 12 );
}

unsigned int read_button( void )
{
    unsigned int status;
    do {
        status = ~( rPDATG );
        status = status & BUTTONS;
    } while( status == 0 );

    status = (status >> 6) & 0x3;

    return status;
}
```

- `leds.h`: declaración de las funciones para manejar los leds.

```
#ifndef _LEDS_H_
#define _LEDS_H_

void leds_init( void );
void led1_on( void );
void led1_off( void );
void led2_on( void );
void led2_off( void );
void led1_switch( void );
void led2_switch( void );
void leds_switch( void );
void leds_display( unsigned int leds_status );

#endif
```

- `leds.c`: implementación de las funciones para manejar los leds.

```
#include "ports.h"
#include "leds.h"

// Mascaras de bits
#define LED1 0x01
#define LED2 0x02
#define BIT_LED1 (0x1 << 9)
#define BIT_LED2 (0x1 << 10)

// status representa el estado de los leds
```

```
// sólo tienen sentido los dos bites menos
// significativos, cada uno para un led
// 1 encendido y 0 apagado
static unsigned int status = 0;

void leds_init( void )
{
    rPCONB = 0x1ff;
    leds_display( status );
}


void led1_on( void )
{
    // COMPLETAR:
}

void led1_off( void )
{
    // COMPLETAR:
}

void led2_on( void )
{
    // COMPLETAR:
}

void led2_off( void )
{
    // COMPLETAR:
}

void led1_switch( void )
{
    // COMPLETAR:
}

void led2_switch( void )
{
    // COMPLETAR:
}

void leds_switch( void ){
    status ^= (LED1 | LED2);
    leds_display( status );
}

void leds_display( unsigned int leds_status )
{
    status = leds_status;

    // LED 1
    if( status & LED1 )
```

```
        rPDATB &= ~BIT_LED1;
    else
        rPDATB |= BIT_LED1;

    // LED 2
    if( status & LED2 )
        rPDATB &= ~BIT_LED2;
    else
        rPDATB |= BIT_LED2;
}
```

- `D8Led.h`: declaración de las funciones para manejar el display de 8 segmentos.

```
#ifndef D8LED_H_
#define D8LED_H_

void D8Led_init(void);
void D8Led_segment(int value);
void D8Led_digit(int value);

#endif
```

- `D8Led.c`: implementación de las funciones para manejar el display de 8 segmentos.

```
#include "ports.h"
#include "D8Led.h"

/*
 * Mascaras utiles para el uso del display de 8 segmentos
 * Cada bit representa un segmento. En la mascara ponemos
 * un 1 si queremos que se encienda dicho segmento. Como
 * el display funciona con logica invertida, nos toca
 * invertir el valor al escribir en el puerto.
 */
#define SEGMENT_A       0x80
#define SEGMENT_B       0x40
#define SEGMENT_C       0x20
#define SEGMENT_D       0x08
#define SEGMENT_E       0x04
#define SEGMENT_F       0x02
#define SEGMENT_G       0x01
#define SEGMENT_P       0x10

/* COMPLETAR:
 * ESTAS MACROS DETERMINAN LOS LEDS A ENCENDER PARA
 * MOSTRAR LOS DIGITOS HEXADECIMALES EN EL DISPLAY
 */
#define DIGIT_0
#define DIGIT_1
#define DIGIT_2
#define DIGIT_3
#define DIGIT_4
#define DIGIT_5
#define DIGIT_6
#define DIGIT_7
#define DIGIT_8
```

```
#define DIGIT_9
#define DIGIT_A
#define DIGIT_B
#define DIGIT_C
#define DIGIT_D
#define DIGIT_E
#define DIGIT_F


/* Tablas para facilitar el uso del display */

static unsigned int Segments[] =
    { SEGMENT_A, SEGMENT_B, SEGMENT_C, SEGMENT_D,
      SEGMENT_E, SEGMENT_G, SEGMENT_F, SEGMENT_P };

static unsigned int Digits[] = { DIGIT_0, DIGIT_1, DIGIT_2, DIGIT_3,
                                 DIGIT_4, DIGIT_5, DIGIT_6, DIGIT_7,
                                 DIGIT_8, DIGIT_9, DIGIT_A, DIGIT_B,
                                 DIGIT_C, DIGIT_D, DIGIT_E, DIGIT_F };

void D8Led_init(void)
{
    LED8ADDR = 0 ;
}

void D8Led_segment(int value)
{
    if( (value >= 0) && (value < 8) )
        // COMPLETAR: Escribir el valor correcto
        LED8ADDR = 0;
}

void D8Led_digit(int value)
{
    if( (value >= 0) && (value < 16) )
        // COMPLETAR: Escribir el valor correcto
        LED8ADDR = 0 ;
}
```

- `utils.h`: declaración de la función `Delay` que usaremos para realizar esperas activas.

```
#ifndef UTILS_H_
#define UTILS_H_

void Delay(int time);

#endif
```

- `utils.c`: implementación de la función `Delay`.

```
#include "ports.h"

/* Función que realiza una espera activa
 * de time milisegundos
 */
```

```
void Delay(int time)
{
    static int delayLoopCount=400;
    int i;

    // Bucle de espera activa
    for( ; time>0; time--)
        for( i=0; i < delayLoopCount;i++ );
}
```

- `main.c`: programa principal

```
#include "leds.h"
#include "button.h"
#include "D8Led.h"
#include "utils.h"

int main(void)
{
    leds_init();
    button_init();
    D8Led_init();

    while( 1 ){
        leds_switch();
        // espera 1s
        Delay( 1000 );
    }

    return 0;
}
```

- `init.asm`: programa de inicialización

```
    .extern main
    .extern _stack
    .global _start

_start:
    ldr sp,=_stack
    mov fp,#0

    bl main
End:
    B End
    .end
```

- `ld_script.ld`: script de enlazado

```
SECTIONS
{
    . = 0x0C000000;
    .data : {
      *(.data)
      *(.rodata)
    }
```

```
    .bss : {
      *(.bss)
      *(COMMON)
    }
    .text : {
      *(.text)
    }
    PROVIDE( end = . );
    PROVIDE (_stack = 0x0C7FF000);
}
```

2. Configuramos la compilación del proyecto como en las prácticas anteriores y compilamos.

3. Cambiamos a la perspectiva de Debug.

4. Conectamos la placa a un puerto USB libre del equipo de laboratorio (PC).

5. Ejecutamos la herramienta externa OpenOCD, creada previamente siguiendo los pasos descritos en la Sección 5.8.

6. Ejecutamos la configuración de depuración creada previamente siguiendo los pasos descritos en la Sección 5.8.

7. Si todo ha ido bien estaremos parados en el símbolo `start`, al inicio de nuestro programa. Dejamos correr el programa. Los leds deberían parpadear con una frecuencia de 1Hz, los dos a la vez.

8. Para la evaluación de la práctica el profesor suministrará a los alumnos el código del guión completado, y el alumno deberá:

   a) Analizar la solución, volcarla a la placa y comprobar el funcionamiento haciendo pequeñas pruebas con los dispositivos implicados.

   b) Realizar un pequeño programa que haga algo con los dispositivos. Lo que haga el programa se deja a la libre elección de los alumnos, teniéndose en cuenta la originalidad y dificultad de lo que hagan en la nota final de la práctica. A modo de ejemplo se dan las siguientes ideas:

   - Programa haga que el display de 8 segmentos muestre un sólo segmento encendido, girando en sentido horario a una frecuencia de 1Hz. Cada vez que pulsemos el botón 1 el led cambia el sentido del giro. Si pulsamos el botón 2 el segmento se detiene. Si lo volvemos a pulsar reanuda su movimiento. Obsérvese que en este caso debe cambiarse la función de lectura de los pulsadores, porque no podemos quedarnos bloqueados indefinidamente esperando a que el usuario pulse alguno. Esto podemos hacerlo añadiendo un parámetro a la función que nos diga si queremos esperar o no, haciendo sólo una iteración del bucle si no queremos esperar. Una solución sencilla sería muestrear unas 10 veces por segundo, usando `Delay` para esperar 100ms después de comprobar el estado de los pulsadores, de forma que cada 10 muestreos (1Hz) hagamos el cambio de posición del segmento si está en movimiento y hacemos parpadear los leds. Esta espera además nos serviría para filtrar los rebotes (aunque podemos detectar varias pulsaciones seguidas si dejamos el dedo en el pulsador, ya que no esperamos a que el usuario deje de pulsar el botón).

- Programa que muestre ciclicamente y letra a letra la palabra `hola` en el display de 8 segmentos. El botón 1 permite alternar entre dos velocidades, de 1Hz y 2Hz (1 o 2 letras por segundo). El botón 2 permite alternar entre dos palabras `hola` y `adios`. Recordar que debe darse soporte en `D8led.c` para mostrar los caracteres que necesitamos.

Bibliografía

[aap] The arm architecture procedure call standard. Accesible en http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0042d/index.html. Hay una copia en el campus virtual.

[um-] S3C44B0X RISC Microprocessor Product Overview. Accesible en http://www.samsung.com/global/business/semiconductor/productInfo.do?fmly_id=229&partnum=S3C44B0. Hay una copia en el campus virtual.

GNU Free Documentation License

Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "**Document**", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "**you**". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "**Modified Version**" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "**Secondary Section**" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "**Invariant Sections**" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "**Cover Texts**" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "**Transparent**" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "**Opaque**".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "**Title Page**" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "**publisher**" means any person or entity that distributes copies of the Document to the public.

A section "**Entitled XYZ**" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "**Acknowledgements**", "**Dedications**", "**Endorsements**", or "**History**".) To "**Preserve the Title**" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See `http://www.gnu.org/copyleft/`.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright © YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with ... Texts." line with this:

> with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.