Práctica 1

Arquitectura ARM, Programación en Lenguaje Ensamblador e Introducción al Entorno de Desarrollo

1.1. Objetivos

El componente principal del equipo que vamos a utilizar en el laboratorio es un chip de Samsung, el S3C44B0X, que integra un microcontrolador ARM7TDMI junto con una serie de dispositivos que le dan gran versatilidad para el diseño de sistemas empotrados. En esta primera práctica trataremos de familiarizarnos con la arquitectura de este procesador ARM y su programación en lenguaje ensamblador. En concreto, los objetivos que perseguimos en esta práctica son:

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

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

La arquitectura ARM identifica cinco tipos de excepción y les asocia un modo privilegiado de ejecución a cada una de ellas. En total hay siete modos de ejecución: usuario (*usr*), supervisor (*svc*), interrupciones externas rápidas (*fiq*), interrupciones externas normales (*irq*), indefinido (*und*), abort (*abt*) y sistema (*sys*). Cuando se produce una excepción el procesador salta de forma controlada a una posición de memoria específica en función de su tipo. En esta primera práctica, así como en la siguiente, vamos a trabajar exclusivamente en modo usuario, que se diferencia de todos los demás por no ser privilegiado, con un acceso restringido a determinados recursos. El resto de modos, así como el cambio entre ellos, se abordará en prácticas siguientes.

El microcontrolador ARM7TDMI, y en general todos las procesadores ARM que incorporan la letra T en el nombre, incluyen un subconjunto de instrucciones denominado

Thumb cuyo propósito es reducir el tamaño del programa en memoria. Las instrucciones de este subconjunto tienen una formato reducido de 16 bits y presentan numerosas particularidades. En las prácticas que vamos a realizar durante el curso, no obstante, vamos a limitarnos exclusivamente al repertorio estándar ARM.

1.3. Memoria

La arquitectura ARM sigue un modelo Von Neumann con un mismo espacio de direcciones para instrucciones y datos. Esta memoria es direccionable por byte y está organizada en palabras de 4 bytes. Hasta la versión 5 incluida (ARM7TDI es ARM v4), los accesos deben ser alineados. Esto quiere decir que en todo acceso a memoria para la carga o el almacenamiento de un dato de 2, 4 u 8 bytes, la dirección accedida debe ser múltiplo de 2, 4 u 8 respectivamente. En concreto, las instrucciones deben almacenarse en posiciones de memoria alineadas a 4 bytes (32 bits), excepto en modo Thumb, en el que se alinean a 2 bytes (16 bits). La Arquitectura ARM permite los dos tipos de ordenación posibles, *little-endian* y *big-endian*, aunque no simultáneamente.

1.4. Registros

En total la arquitectura ARM dispone de 37 registros, 30 de propósito general y 7 de uso específico, aunque no todos ellos son visibles en un mismo modo de ejecución. Tener distintos registros para distintos modos permite acelerar el tratamiento de excepciones, como veremos más adelante.

En cada modo de ejecución 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). Además, aquellos modos que están asociados a excepciones, es decir todos menos *usr* y *sys*, disponen de un registro de salvaguarda del estado (SPSR). Todas las instrucciones pueden direccionar y escribir cada uno de los 15 registros en cada momento. Los registros de estado, CPSR y SPSR, deben ser manipulados por medio de instrucciones especiales.

Aunque el registro de 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. Por ejemplo, si se escribe sobre el, se producirá un salto en el flujo de instrucciones del programa. Además, es preciso tener en cuenta que, debido a la segmentación del procesador ARM7TDMI, cuando se realiza la lectura de este registro, el valor proporcionado es el de la dirección actual más 8 (es decir, dos instrucciones después de la actual).

Registro de estado

El registro de estado, CPSR, almacena información adicional necesaria para determinar el estado del programa, por ejemplo el signo del resultado de alguna operación anterior o el modo de ejecución del procesador. Es el único registro que tiene restricciones de acceso. Está estructurado en campos con un significado bien definido, como ilustra la Figura 1.1.

La Tabla 1.1 describe el significado de cada uno de estos campos. Como podemos ver, algunos bits están reservados para uso futuro y por tanto no son modificables (siempre se leen como cero). Los bits N, Z, C, V, Q, GE y E son modificables en modo usuario, mientras que los bits A, I, F y M sólo son modificables en los modos privilegiados. Del mismo modo, los bits J y T, que permiten cambiar de subconjunto de instrucciones (*Jazelle* o *Thumb*), sólo pueden ser modificados en los modos privilegiados, ignorándose su escritura cuando se está en modo usuario.

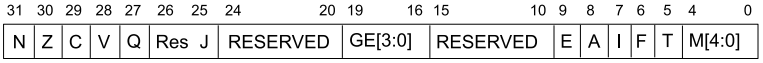


Figura 1.1: Registro de Estado (CPSR).

1.5. Repertorio de instrucciones

Dividiremos las instrucciones del repertorio en cinco grupos:

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

En este guión vamos a presentar únicamente un subconjunto de estas instrucciones. Para una mayor información debe consultarse el manual de referencia de la arquitectura [arm].

Una característica particular de la arquitectura ARM, es que todas las instrucciones soportan ejecución condicional. Para ello, el formato de instrucción contiene un campo, que indica la condición que deben cumplir los bits de Flag del CPSR. Si dicha condición se cumple, la instrucción se ejecuta normalmente, si no, la instrucción no surte ningún efecto sobre el estado arquitectónico (es decir, no modifica ningún registro ni palabra de memoria). La Tabla 1.2 muestra las alternativas posibles para este campo. Más adelante, veremos algunos ejemplos al tratar cada tipo de instrucción.

1.5.1. Instrucciones aritmético-lógicas.

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 TST, no escriben el resultado en registro, sólo modifican los bits del CPSR. En general, cualquier instrucción puede modificar los flags de CPSR además de escribir el resultado en el registro destino. Veremos algunos ejemplos más adelante.

Tabla 1.1: Descripción de los distintos campos del CPSR.

Campo	Bit	Significado
Bits de Flag	N	Indica si la última operación dio como resultado un valor negativo (N = 1) o positivo (N = 0).
	Z	Se activa (Z = 1) si el resultado de la última operación fue cero, de lo contrario permanece inactivo (Z = 0).
	C	Su valor depende del tipo de operación: ▪ Para una suma o una comparación (CMP), C = 1 si hubo *carry*. ▪ Para una resta o una comparación, C = 0 si hubo *underflow*. ▪ Para las operaciones de desplazamiento, toma el valor del bit saliente. ▪ El resto de operaciones no modifican su valor.
	V	En el caso de una suma o una resta, V = 1 indica que hubo un *overflow*.
Bit Q	Q	Indica si se produjo *overflow* y/o saturación en instrucciones de tipo DSP. Disponible en ARM v5 y versiones posteriores de la arquitectura.
Bits GE	GE[3:0]	Se emplean como bits de condición para instrucciones de tipo SIMD. Disponible en ARM v6 y versiones posteriores de la arquitectura.
Bit E	E	Indica el tipo de ordenación de memoria, *little-* o *big-endian*. Disponible en ARM v6 y versiones posteriores de la arquitectura.
Bits de Interrupción	A	Si $A = 1$ se deshabilitan las excepciones de tipo abort. Disponible en ARM v6 y versiones posteriores de la arquitectura.
	I	Si $I = 1$ se deshabilitan las interrupciones externas (IRQ).
	F	Si $F = 1$ se deshabilitan las interrupciones externas rápidas (FIQ).
Bits de Modo	M[4:0]	Indican el modo de ejecución del procesador: *usr*(10000), *fiq*(10001), *irq*(10010), *svc* (10011), *abt*(10111), *und*(11011), *sys*(11111).
Bits T y J	T,J	Indican el repertorio de instrucciones activo: *ARM*(00), *Thumb*(01), *Jazelle*(10). La combinación (11) está reservada.

Tabla 1.2: Condiciones asociadas a las instrucciones.

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

Tabla 1.3: Instrucciones aritmético-lógicas comunes

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

Modos de direccionamiento

Como hemos visto las instrucciones aritmético-lógicas tienen generalmente dos operandos fuente y un operando destino. De los dos operandos fuente, uno debe ser siempre un

registro. El segundo operando se denomina *shifter_ operand* y puede ser tanto un inmediato como un registro. En caso de que sea un registro, opcionalmente puede ser desplazado. La cantidad a desplazar puede a su vez indicarse con un valor inmediato o con un tercer registro. Hay cinco tipos de desplazamiento aplicables:

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

Ejemplos

```
ADD  R0, R1, #1          @ R0 = R1 + 1
ADD  R0, R1, R2          @ R0 = R1 + R2
ADD  R0, R1, R2 lsl #1   @ R0 = R1 + (R2 << 1)
ADD  R0, R1, R2 lsl R3   @ R0 = R1 + (R2 << r3)
BIC  R4, #0x05           @ Borra los bits 0 y 3 de r4
MOV  R11, #0             @ Escribe cero en r11
MOV  R15, R3 lsl #2      @ Copia en R15 (PC) el resultado de desplazar a la
                         @ izquierda r3 dos posiciones. ¡Provoca un salto!
SUB  R3, R2, R1          @ R3 = R2 - R1
SUBS R3, R2, R1          @ R3 = R2 - R1. Modifica los flags del registro de
                         @ estado en función del resultado
ADDEQ R7, R1, R2         @ Si el bit Z de CPSR está activo R7 = R1 - R2
```

1.5.2. 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. Las principales variantes se describen en la Tabla 1.4. Para completar la lista, es preciso consultar el manual de referencia de la arquitectura ARM [arm]. Todas las instrucciones pueden modificar opcionalmente los bits Z y N del registro de estado en función de que resultado sea cero o negativo.

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
```

Tabla 1.4: Instrucciones de multiplicación.

Mnemotécnico	Operación
MUL	Multiplica dos números de 32 bits y genera los 32 bits menos significativos del resultado.
MULA	Multiplica dos números de 32 bits, suma el valor de un tercer registro, trunca el resultado a 32 bits y lo guarda en un cuarto registro.
SMULL	Multiplica dos datos de 32 bits con signo almacenados en dos registros fuente y produce un resultado de 64 bits que queda almacenado el el tercer y cuarto registro.
UMULL	Multiplica dos datos de 32 bits sin signo almacenados en dos registros fuente y produce un resultado de 64 bits que queda almacenado el el tercer y cuarto registro.
SMLAL	Multiplica dos datos de 32 bits con signo almacenados en dos registros fuente y al resultado le suma el numero de 64 bits almacenado en el tercer y cuarto registro fuente, dejando el resultado final en el tercer y cuarto registro fuente.
UMLAL	Multiplica dos datos de 32 bits sin signo almacenados en dos registros fuente y al resultado le suma el numero de 64 bits almacenado en el tercer y cuarto registro fuente, dejando el resultado final en el tercer y cuarto registro fuente.

```
SMULL R4, R8, R2, R3     @ R4 = [R2 x R3]31..0
                         @ R8 = [R2 x R3]63..32
UMULL R6, R8, R0, R1     @ R8/R6 = R0 x R1
UMLAL R5, R8, R0, R1     @ R8/R5 = R0 x R1 + R8/R5
```

1.5.3. Instrucciones de consulta/modificación del Registro de Estado

Estas instrucciones permiten consultar o modificar el registro de estado (CPSR). Si nos hallamos en un modo privilegiado, también pueden trabajar con el registro de salvaguarda del estado (SPSR). La Tabla 1.5 resume su comportamiento.

Tabla 1.5: Instrucciones de manejo del Registro de Estado

Mnemotécnico	Operación
MRS	Copia el valor del CPSR/SPSR a un registro.
MSR	Copia el valor de un registro en el CPSR/SPRS. En modo usuario sólo podemos modificar los flags de condición (bits 31:24).
CPS	Cambia los bits de modo del procesador o los bits de control de interrupciones.
SETEND	Cambiar el bit E (endianness).

1.5.4. Instrucciones Load y Store

Las instrucciones de load (LDR) pueden cargar un dato de tamaño palabra, media palabra o byte desde la memoria a un registro, o un dato de tamaño doble palabra y almacenarlo en dos registros consecutivos. Además, permite que a los datos de tamaño media palabra o byte les sea extendido el bit de signo automáticamente.

Las instrucciones de store (STR) pueden almacenar en memoria un dato de tamaño palabra, media palabra o byte procedente de un registro, o uno de tamaño doble palabra procedente de dos registros consecutivos.

Modos de direccionamiento

La Figura 1.2 ilustra los modos de direccionamiento de las instrucciones de load/store. Para formar la dirección de memoria a la que se desea acceder, se utiliza un registro base y un desplazamiento (offset). Éste último puede ser un inmediato, otro registro o un registro desplazado. Hay tres variantes o tres formas de combinar el registro base y el desplazamiento:

- Normal (indirecto de registro con desplazamiento). El desplazamiento se suma o resta al registro base para formar la dirección de memoria.
- Pre-indexado. El desplazamiento se suma o resta al registro base para formar la dirección de memoria. El registro base se actualiza con este resultado. Es muy utilizado para recorridos de arrays.
- Post-indexado. La dirección de memoria se calcula sólo con el registro base. Después del acceso, el registro base y el desplazamiento se suman o restan y el resultado se almacena en el registro base.

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

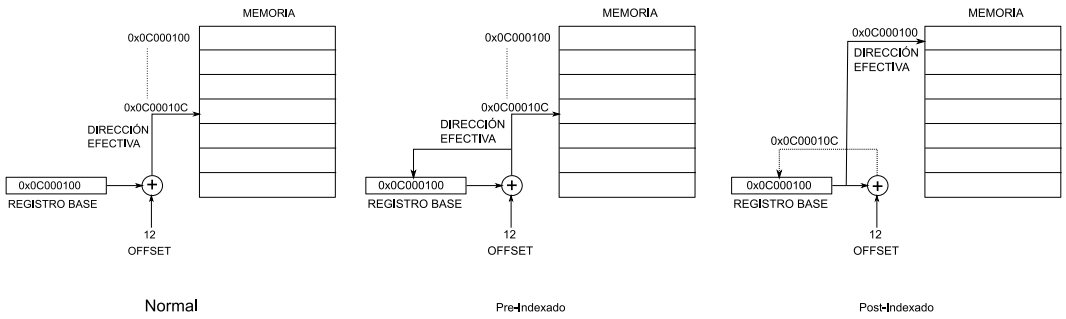


Figura 1.2: Modos de direccionamiento en las instrucciones de load/store.

Ejemplos

```
LDR   R1,  [R0]            @ Carga en R1 lo que hay en Mem[R0]
LDR   R8,  [R3, #4]        @ Carga en R8 lo que hay en Mem[R3 + 4]
```

```
LDR   R12, [R13, #-4]         @ Carga en R12 lo que hay en Mem[R13 - 4]
STR   R2,  [R1, #0x100]       @ Almacena en Mem[R1 + 256] lo que hay en R2
STR   R4,  [PC, R1, LSL #2]   @ Almacena en Mem[PC + R1*4] lo que hay en R4
LDR   R11, [R3, R5, LSL #2]   @ Carga en R11 lo que hay en Mem[R3 + R5*4]
LDRB  R5,  [R9]               @ Carga un byte de Mem[R9] en R5
LDRSH R5,  [R9]               @ Carga en R5 la media palabra de Mem[R9] con
                              @ extensión de signo
LDR   R11, [R3, R5, LSL #2]   @ Carga R11 desde Mem[R3 + (R5 x 4)]
LDR   R1,  [R0, #4]!          @ Carga Mem[R0 + 4] en R1 y actualiza R0 = R0 + 4
STRB  R7,  [R6, #-1]!         @ Almacena el contenido de R7 en Mem[R6-1] y
                              @ actualiza R6 = R6 - 1,
LDR   R3,  [R9], #4           @ Carga R3 desde Mem[R9] y actualiza R9 = R9 + 4
STR   R2,  [R5], #8           @ Alamacena el contenido de R2 en Mem[R5] y
                              @ actualiza R5 = R5 + 8?
```

1.5.5. 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. El Load Múltiple (LDM) permite la carga de varios registros con datos ubicados en posiciones de memoria consecutivas. El Store Múltiple (STM) permite almacenar en posiciones de memoria consecutivas el valor de varios registros. La sintaxis en ensamblador de este tipo de instrucciones es la siguiente

```
ldr RegBase{!}, lista de registros entre llaves
str RegBase{!}, lista de registros entre llaves
```

Si se pone el signo ! tras el registro base, dicho registro quedará actualizado adecuadamente para encadenar varios accesos de este tipo.

Modos de direccionamiento

Los 4 modos de direccionamiento para estas instrucciones se describen en la Tabla 1.6. Existen dos nomenclaturas distintas para referirse ellos, la normal y la alternativa. Esta última se suele usar cuando las instrucciones LDM/STM se emplean para insertar/extraer datos en/de la pila. En estos casos el registro base suele ser R13, que actúa a modo de puntero de pila (*Stack Pointer* o SP) y se contemplan 4 alternativas en función dos características:

- Tipo de objeto apuntado por el puntero de pila:
  - *Full*: cuando el SP apunta a la última posición ocupada.
  - *Empty*: cuando el SP apunta a la primera dirección libre.
- Sentido en el que se apilan los datos en la memoria:
  - *Descending*: cuando la pila crece hacia direcciones menos significativas.
  - *Ascending*: cuando la pila crece hacia direcciones más significativas.

Cuando empleamos las instrucciones LDM/STM para gestionar una pila, es más sencillo pensar en el tipo de pila que queremos usar que en el modo de direccionamiento que debemos emplear en cada caso, de ahí que existe una nomenclatura alternativa.

Tabla 1.6: Modos de direccionamiento para las instrucciones LDM y STM.

Normal	IA	Increment After	El registro se carga con lo que hay en memoria en la dirección indicada por el registro base. Las direcciones subsiguientes se forman sumando 4 a la dirección anterior.
	IB	Increment Before	El registro se carga con lo que hay en memoria en la dirección resultante de sumar 4 al contenido del registro base. Las direcciones subsiguientes se forman sumando 4 a la dirección anterior.
	DA	Decrement After	El registro se carga con lo que hay en memoria en la dirección indicada por el registro base. Las direcciones subsiguientes se forman restando 4 a la dirección anterior.
	DB	Decrement Before	El registro se carga con lo que hay en memoria en la dirección resultante de restar 4 al contenido del registro base. Las direcciones subsiguientes se forman restando 4 a la dirección anterior.
Alternativa	FD	Full Descending.	$LDMFD = LDMIA$ y $STMFD = STMDB$
	FA	Full Ascending	$LDMFA = LDMDA$ y $STMFA = STMIB$
	ED	Empty Descending	$LDMED = LDMIB$ y $STMED = STMDA$
	EA	Empty Ascending	$LDMEA = LDMDB$ y $LDMEA = STMIA$

Ejemplos

```
STMFD R13!, {R0 - R12, LR}          @ Guarda a partir de la dirección indicada
                                    @ por R13 menos 4 lo que hay en R0-R12, LR y
                                    @ actualiza R13. Apila.
LDMFD R13!, {R0 - R12, PC}          @ Carga a partir de la dirección indicada
                                    @ por R13 lo que hay en R0-R12, PC y
                                    @ actualiza R13. Desapila y supone salto.
LDMIA R0,   {R5 - R8}               @ Carga a partir de la dirección indicada
                                    @ por R0 lo que hay en R5-R8
STMDA R1!,  {R2, R5, R7 - R9, R11} @ Guarda a partir de la dirección indicada
                                    @ por R1 lo que hay en R2,R5, R7-R9, R11 y
                                    @ actualiza R1.
```

1.5.6. Instrucciones de Salto

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

La dirección a la que se salta se calcula como [1]:

$$Dir = PC - 8 + Inmediato$$

BL además de saltar almacena en el registro R14 (*Link Register* o LR) la dirección de la instrucción siguiente al salto. Se utiliza para realizar saltos a subrutina, con lo que LR contendrá la dirección de retorno de la subrutina.

Existen dos variantes, BX y BLX, que activan el cambio de repertorio de instrucciones en el salto. En estos casos, si la dirección destino del salto es múltiplo de 32 (termina en 00) pasan a repertorio *ARM*, y si es sólo múltiplo de 16 (termina en 10) pasan a repertorio *Thumb*. Nosotros no vamos a utilizar este tipo de saltos.

Los saltos condicionales se consiguen modificando los bits de condición de la instrucción. En lenguaje ensamblador se indica añadiendo al mnemotécnico (B o BL) el sufijo de la condición: EQ,NE,GE,GT,LE,LT,....

Ejemplos

```
B    label          @ Salto incondicional a la dirección dada por la etiqueta
BCC  label          @ Salto a la dirección de la etiqueta si el bit de carry
                    @ esta a cero
BEQ  label          @ Salto a la etiqueta si el flag Z está activo
MOV  PC, #0         @ R15 = 0, salto absoluto a la dirección 0
BL   func           @ Salto relativo al PC a la rutina func
MOV  LR, PC         @ almacena lo que se lee de PC en LR (PC apunta dos
                    @ instrucciones hacia delante)
func:               @ rutina func
MOV  PC, LR         @ R15=R14, retorno de subrutina
```

1.6. Entorno de Desarrollo Embest IDE

El entorno de desarrollo Embest IDE se basa en el ensamblador (as), el compilador (gcc) y el enlazador (ld) de GNU para ARM. Por lo tanto, se debe emplear la sintaxis y reglas de programación propias de estas herramientas. En las siguientes secciones, repasaremos brevemente las principales características del ensamblador y el enlazador. Para una mayor información es preciso consultar los manuales de estas herramientas.

1.7. Ensamblador GNU para ARM

El ensamblador es un programa que lee un fichero de texto con instrucciones de lenguaje ensamblador y directivas de ensamblado, y genera un fichero que llamamos de *código objeto*. Posteriormente, el enlazador formará el fichero ejecutable final uniendo varios ficheros de código objeto (ver Sección 1.8).

[1] Cuando la dirección de salto se especifica con una etiqueta, el ensamblador traducirá automáticamente la dirección de la etiqueta al inmediato necesario para el salto, teniendo en cuenta que se le va a sumar el PC. Si el inmediato no puede ser representado con 24 bits se producirá un error.

Tabla 1.7: Directivas de ensamblado (GNU ARM).

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

Las instrucciones del lenguaje ensamblador no son más que mnemotécnicos que nos permiten expresar de forma más sencilla instrucciones del repertorio de la arquitectura destino. Estas instrucciones se traducen generalmente uno a uno por una instrucción máquina.

Además de instrucciones, el código en lenguaje ensamblador puede contener directivas para guiar el proceso de ensamblado. Estas directivas son órdenes para el propio programa ensamblador, y permiten inicializar posiciones de memoria, definir símbolos que hagan más legible el programa, marcar el inicio y el fin del programa, etc. Las más relevantes están descritas en la Tabla 1.7. Para completar esta información puede consultarse la guía de referencia rápida del ensamblador [gnu].

Un programa escrito en lenguaje ensamblador está compuesto por líneas de texto con la siguiente sintaxis:

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

El ensamblador de GNU no impone indentación, la etiqueta se reconoce por el símbolo `":"` que la sucede. Además los comentarios pueden ponerse como comentarios de C, entre `/* ... */`.

1.7.1. Modos de direccionamiento

Aunque conceptualmente son próximos, en la Arquitectura ARM el abanico de modos de direccionamiento es amplio, y por lo tanto la sintaxis del lenguaje ensamblador es compleja. En la Tabla 1.8 mostramos tan sólo algunos de los modos más frecuentes. Para una información más detallada es preciso consultar el manual de referencia de la Arquitectura ARM [arm].

Tabla 1.8: Modos de direccionamiento (GNU ARM).

Sintaxis	**Descripción**
#Inmediato	Inmediato
Rm	Directo registro
Rm, <shift>#inm	Directo registro con desplazamiento por inmediato
Rm, <shift>Rs	Directo registro con desplazamiento por registro
[Rn, #+/-offset]	Indirecto registro con desplazamiento inmediato
[Rn, +/-Rm]	Indirecto registro con desplazamiento en registro
[Rn, +/-Rm, <shift>#Inm]	Indirecto registro con desplazamiento en registro y escalado
[Rn, +/-Rm]!	Indirecto registro con deplazamiento en registro de tipo pre-indexado
[Rn], +/-Rm	Indirecto registro con desplazamiento en registro de tipo post-indexado

Además, es preciso tener en cuenta que los símbolos definidos en el ensamblador se sustituyen por su valor. Esto quiere decir que si queremos asignar el valor de un símbolo a un registro con la instrucción `MOV` debemos precederlo por el símbolo #:

```
.equ UNO, 0x01
...
MOV r1, #UNO
...
```

1.7.2. Secciones

Todo programa está dividido en secciones. Por defecto se crearán tres secciones, código (`.text`), datos con valor inicial (`.data`) y datos sin valor inicial (`.bss`). El ensamblador permite indicar en qué sección debemos colocar un determinado bloque utilizando la directiva `.section`. También podemos definir secciones nuevas. Cuando escribamos el código de

nuestro programa deberemos especificar en qué sección `.text` debe colocarse, y, si queremos reservar memoria para datos con valor inicial, será conveniente marcar el bloque como perteneciente a la sección `.data`.

1.7.3. Entrada al progama

Por lo general, el comienzo del programa se indica mediante la etiqueta `start`, que suele además exportarse como símbolo externo. No obstante, como veremos en la Sección 1.8, el enlazador puede cambiar este punto de entrada.

1.7.4. Fin de ensamblado

El ensamblador dará por concluido el programa cuando encuentre la directiva `.end`. El texto situado detrás de esta directiva de ensamblado será ignorado.

1.7.5. Tabla de literales y macros de LDR

Por ahora hemos visto que la única manera de asignar un valor inmediato a un registro es con la instrucción `MOV`. Sin embargo, el modo de direccionamiento de esta instrucción nos limita el rango de valores del inmediato.

¿Cómo podríamos asignarle un valor inmediato de 32 bits a un registro? La única forma de conseguirlo consiste en almacenar el valor en una posición de memoria, no muy lejana a la actual, y posteriormente cargarlo con un load (`LDR`) relativo al PC. Por ejemplo:

```
...
.word 0xAAAAAAAA
... #n bytes
LDR r1, [PC,-#n]
```

La zona de memoria en la que se almacenan estos valores inmediatos se conoce con el nombre de tabla de literales o *literal pool*, y, para facilitar su gestión, el ensamblador ofrece la directiva `.ltorg` y una macros asociadas a la instrucción LDR:

LDR Rd, =Etiqueta

LDR Rd, =Símbolo

LDR Rd, =Inmediato
: El ensamblador crea un símbolo con el valor de la etiqueta o el inmediato (en el caso del símbolo, ya existe y no es necesario). Además se encarga de almacenar una nueva entrada en la tabla de literales más próxima a la posición actual. La instrucción es entonces sustituida por un load relativo a PC que carga el valor correspondiente de la tabla de literales.

LDR Rd, Etiqueta
: En este caso, simplemente se carga el valor de la dirección indicada por la etiqueta mediante direccionamiento relativo al PC y no es necesario el uso de la tabla de literales. Para poder usar esta variante es necesario que la etiqueta esté definida

dentro de la misma sección y no se encuentre alejada de la posición actual (es decir, el offset debe estar dentro del rango representable por la instrucción).

Así, el ejemplo anterior se reduciría a:

```
LDR r1, =0xAAAAAAAA
```

Debemos tener en cuenta que en el caso de tener etiquetas cuya dirección no pueda ser determinada en tiempo de compilación (por ejemplo por estar en distintas secciones), la resolución del símbolo se aplaza hasta el enlazado. Es entonces cuando el enlazador puede calcular su valor, en función de dónde haya colocado la instrucción correspondiente a la etiqueta, y será entonces cuando la tabla de literales se actualizará con el valor final del símbolo.

Saltos Globales

Frecuentemente querremos saltar a una determinada posición de memoria descrita por una etiqueta, por ejemplo para saltar a una función o al comienzo de un bloque. Si lo indicamos con una instrucción `B etiqueta` estamos limitados a indicar el destino del salto como un desplazamiento relativo al PC. Si el desplazamiento es muy largo o no puede determinarse al realizar el ensamblado, se producirá un error. La única solución en este caso es emplear un salto absoluto escribiendo un valor de 32 bits directamente en el PC, y para ello es preciso utilizar las macros que acabamos de describir: `LDR PC, =etiqueta`. De esta manera podremos hacer saltos absolutos a cualquier dirección de memoria dentro del rango de 4GB direccionable.

Los saltos absolutos son estrictamente necesarios para saltar a subrutinas definidas en distintos ficheros, ya que, como veremos en la sección 1.8, hasta el momento del enlazado no podemos resolver la dirección destino, y por lo tanto es imposible hacer un salto relativo al PC en tiempo de compilación/ensamblado.

1.7.6. Ejemplo

Veamos un ejemplo de programa escrito para el ensamblador GNU-ARM:

```
        .global start

        .equ UNO, 0x01

        .bss
RES:    .space 4

        .text
DOS:    .word  0x02
start:
        MOV r0, #UNO
        LDR r1, DOS
        ADD r2, r0, r1
        LDR r3, =RES        @ = obligatorio por estar en otra sección
```

```
        STR r2, [r3]
FIN:    B .
        .ltorg              @ Aquí va el pool de literales que
                            @ almacena la dirección de RES
        .end
```

1.8. Enlazador GNU para ARM

El enlazador es la última pieza de la cadena. Toma una serie de ficheros de código objeto, generados por un compilador o ensamblador y forma con ellos un binario ejecutable.

El enlazador de GNU, conocido como ld, tomará los ficheros objeto y tratará de resolver los símbolos no resueltos, es decir, aquellos que se definieron como externos en alguno de los ficheros. Si estos símbolos no han sido definidos y exportados (mediante la directiva `.global`) en alguno de los otros ficheros objeto se producirá un error de enlazado. Lo mismo sucederá si se encuentran dos símbolos exportados con el mismo nombre. Si el ensamblador consigue resolver todos los símbolos, unirá las secciones con el mismo nombre una detrás de otra, y las colocará en un binario con formato ELF.

Este comportamiento por defecto puede modificarse con un fichero de configuración o script ld. La sintaxis de este fichero es sencilla y emplea la siguiente nomenclatura. Las secciones de los ficheros objeto a enlazar se denominan secciones de entrada y las secciones del binario (ELF) a formar se denominan secciones de salida. El fichero de configuración (script) indica esencialmente cómo formar las secciones de salida a partir de las secciones de entrada, y cómo han de colocarse las secciones de salida en memoria. Asimismo, permite definir algunos símbolos, que pueden ser referenciados en el propio código objeto.

Las secciones de salida se definen dentro de una construcción `SECTIONS`, cuyo formato es el siguiente:

```
SECTIONS {

    ...

    secname start BLOCK(align) (NOLOAD) : AT ( ldadr )
        { contents } >region =fill
    ...

}
```

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

secname
: Es el nombre de la sección de salida.

start
: Es la dirección de emplazamiento de la sección. Puede representarse con una expresión. Por ejemplo, para colocar la sección output a partir de la dirección 0x40000000:

```
SECTIONS {
    ...
    output 0x40000000: {
           ...
    }
    ...
}
```

BLOCK(align)
Se utiliza para emplazar un bloque con un determinado alineamiento, dado por la expresión *align*. Lo que sucede es que el puntero de posición actual `"."` avanza hasta la siguiente posición que respete el alineamiento deseado antes de colocar la sección.

(NOLOAD)
Evita que la sección sea cargada en memoria en tiempo de ejecución. Se utiliza por ejemplo cuando hemos definido un contenido para una ROM y este contenido ya se cargó una vez. Al ejecutar el programa la sección no se cargará.

AT(ldadr)
Indica la dirección de carga de la sección. Si no se especifica, la dirección de carga será la dirección de emplazamiento de la sección. Esto permite cargar una sección en un lugar (por ejemplo una ROM) pero resolviendo los símbolos como si estuviese en otro lugar (dirección de emplazamiento).

>region
Le asigna la sección a una región de memoria previamente definida en una entrada `MEMORY`. Para más detalle consultar [ld-].

=fill
Especifica un valor de relleno que asignar a los huecos encontrados en la sección (direcciones no asignadas).

contents
Permite definir el contenido de la sección de salida así como los símbolos globales. En el siguiente ejemplo, se define el contenido de la sección de salida `.text` como la concatenación de las secciones de entrada `.text` de todos los ficheros (mediante la expresión `*.text`) y se define el símbolo `\_etext` (representa el final de la sección de salida).

```
SECTIONS
{
     .text 0x1000 : { *(.text) _etext = . ; }
}
```

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

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

Para completar la información sobre el enlazador de GNU puede consultarse la documentación de ld [ld-].

1.9. Desarrollo de la práctica

Vamos a dividir la práctica en cuatro apartados. El primero será completamente guiado y muy sencillo, su objetivo es aplicar los conocimientos adquiridos sobre la estructura del programa y la formación del binario y servirá para familiarizarse con el entorno EmbestIDE. Los otros tres apartados supondrán modificaciones de dificultad creciente sobre el anterior.

1.9.1. Preparación y Depuración de un programa sencillo

Vamos a crear un programa muy sencillo, que inicialice un registro con una variable global con valor inicial, almacenada en la sección `.data`, y sume cinco veces 1 a este registro y al finalizar guarde el resultado en una posición de memoria reservada en la sección `.bss`. El objetivo es afianzar los conocimientos sobre la estructura del programa, la división en secciones, los modos de direccionamiento y el proceso de compilación y enlazado. También aprenderemos a utilizar el depurador en circuito.

Los pasos a seguir, que se ilustran en las figuras 1.3-1.6, son los siguientes:

1. Abrir el EmbestIDE y crear un nuevo workspace, por ejemplo `Prac1` en `c:\hlocal`.
2. Crear un nuevo fichero que se llame `prog1.s` y añadirlo como fichero fuente al workspace.
3. Copiar el siguiente código en el fichero:

```
        .global start

        .equ UNO, 0x01
        .equ CINCO, 0x05

        .bss
RES:    .space 4

        .text
VAR:    .word 0x02
```

```
start:  MOV  r0, #UNO
        LDR  r1, VAR
        MOV  r3, #CINCO

BUCLE:  ADD  r1, r1, r0
        SUBS r3, r3, #1
        BNE BUCLE

        LDR  r2, =RES
        STR  r1, [r2]
FIN:    B .
        .ltorg
        .end
```

4. Crear otro fichero nuevo con el nombre `ldscript.ld`. Este será el fichero de configuración del enlazador. Queremos crear un binario que emplace el código a partir de la dirección 0x0C000000 y la sección de datos sin valor inicial a partir de la dirección 0x0C100000. Por ello copiamos el siguiente código a nuestro fichero:

```
SECTIONS
{
    .text 0x0C000000: {
        _btext = .;
        *(.text)
        _etext = .;
    }
    .bss 0x0C100000: {
        _bbss = .;
        *(.bss)
        _ebss = .;
    }
}
```

5. Añadir una carpeta *common* al workspace y añadir en ella el fichero `ldscript.ld` previamente creado.

6. Configurar el proyecto:

   a) Seleccionamos menú *Project*, *settings*.

   b) En la pestaña *Processor* seleccionamos *CPU Module arm7*, *CPU family ARM7* y *Member ARM7*. Como herramientas de construcción seleccionamos *GNU Tools for ARM*.

   c) En la pestaña Remote seleccionamos *Remote device UNetICE*, en *Speed* seleccionamos *Medium Speed* y en *Communication type USB*.

   d) En la pestaña *Debug*, en *Symbol file* seleccionamos el fichero `.\debug\Prac1.elf`, que será generado en la compilación.

e) En la misma pestaña seleccionamos la categoría *Download* y como fichero de descarga el mismo `.\debug\Prac1.elf`. Marcamos la casilla *Download verify* y escribimos la dirección de descarga que deseamos en *Download address*, que coincidirá con la de la primera sección 0x0C000000. En el grupo *Execute program from* marcamos *Program entry point*.

f) En la pestaña *Linker*, categoría *general*, escogemos el fichero *ldscript.ld* como *Linker script file*.

7. Construimos el proyecto. En menú *Build* seleccionamos *Build Prac1* o pulsando F7.

8. Conectamos el EmbestIDE con el depurador UNetICE seleccionando *Remote Connect* en el menú *Debug* (o pulsamos F8).

9. Descargamos el programa seleccionando *Download* en el menú *Debug*.

La figura 1.7 muestra una captura de pantalla del depurador mientras se depura el programa que hemos creado, indicando la función de cada una de las ventanas. Ejecutar paso a paso el programa. En la ventana de memoria podemos localizar las dos posiciones de memoria de nuestras variables, las señaladas por las etiquetas VAR y RES. Es necesario responder a las siguientes preguntas:

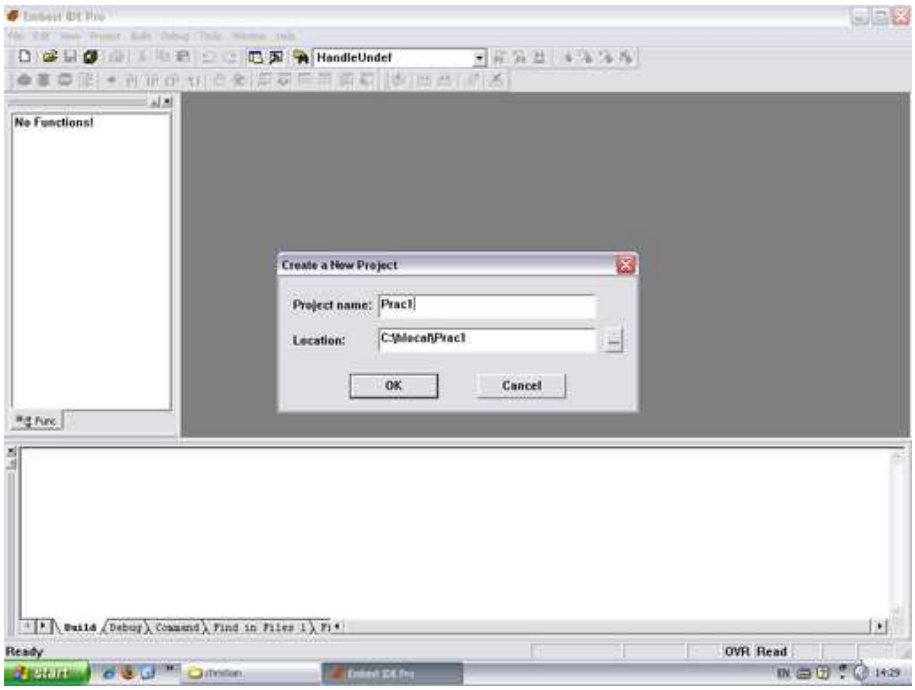
- ¿En qué direcciones de memoria está cada una?
- ¿Que valor inicial y final tiene cada una?
- ¿Cómo se carga la dirección del resultado en el registro r2, de dónde se saca?

1.9.2. Copia de una tabla con load/store múltiple

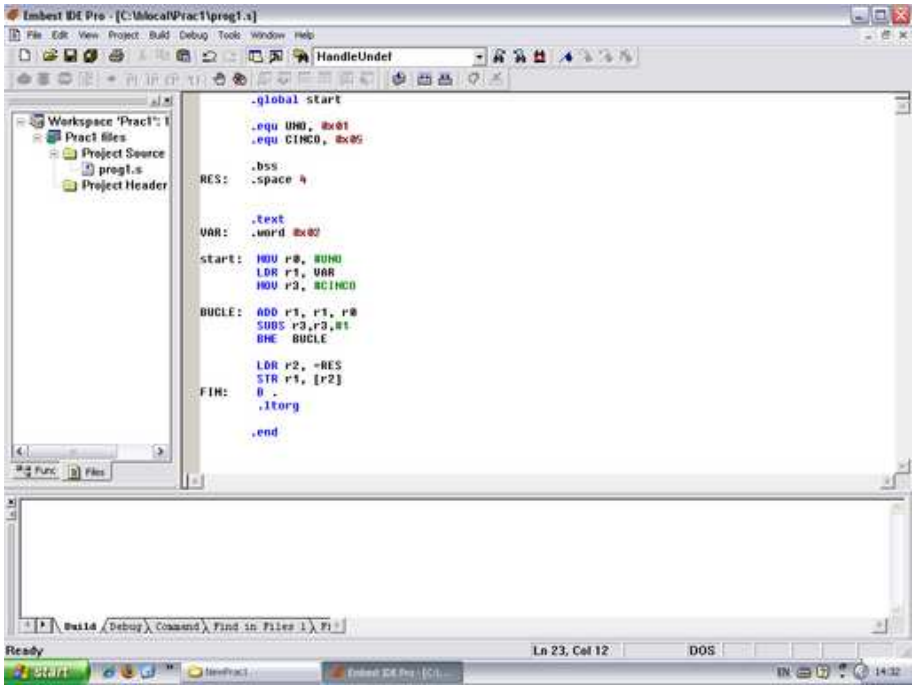
En esta parte vamos a diseñar un programa que copie una tabla desde una posición de memoria a otra posición de memoria utilizando una subrutina y las operaciones de load y store múltiple. La tabla de datos estará en la sección de datos con valor inicial. También colocaremos allí la tabla destino con el fin de poder inicializarla a cero y comprobar posteriormente que se ha copiado correctamente la tabla origen. Diseñaremos la rutina de forma que los registros que utilice sean recuperados al salir con su valor inicial. Para ello utilizaremos una pila que comenzará en la posición 0x0C7FF000.

Damos un esquema del código, indicando las zonas que deben ser completadas por los alumnos.

```
        .global start

        .equ N, 22                @ Palabras a copiar
        .equ STACK, 0x0c7ff000    @ Valor inicial para el puntero de pila

.data

/* Tabla origen */
SRC: .word 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22
```

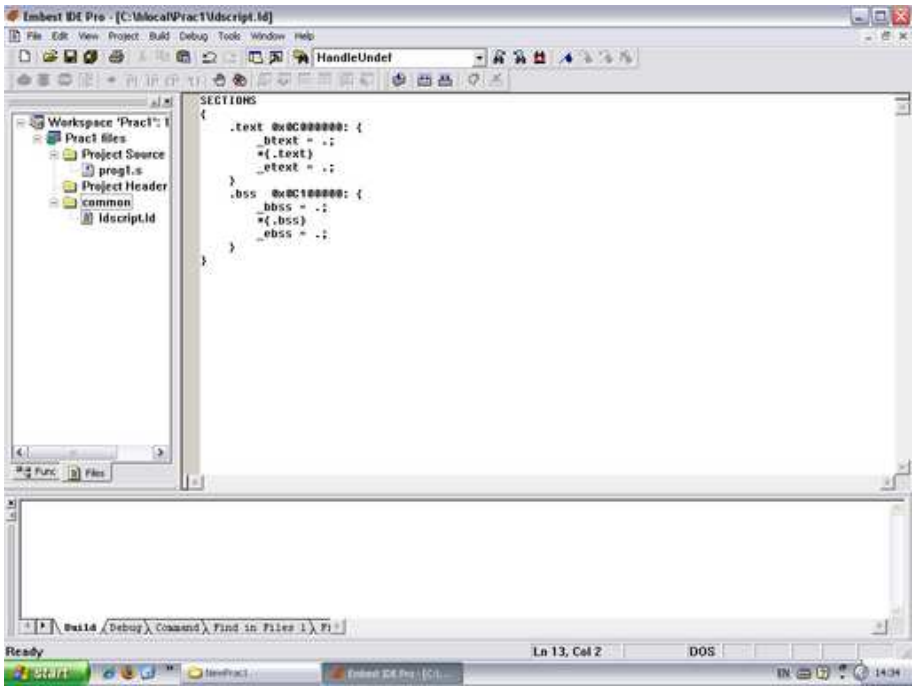


(a)

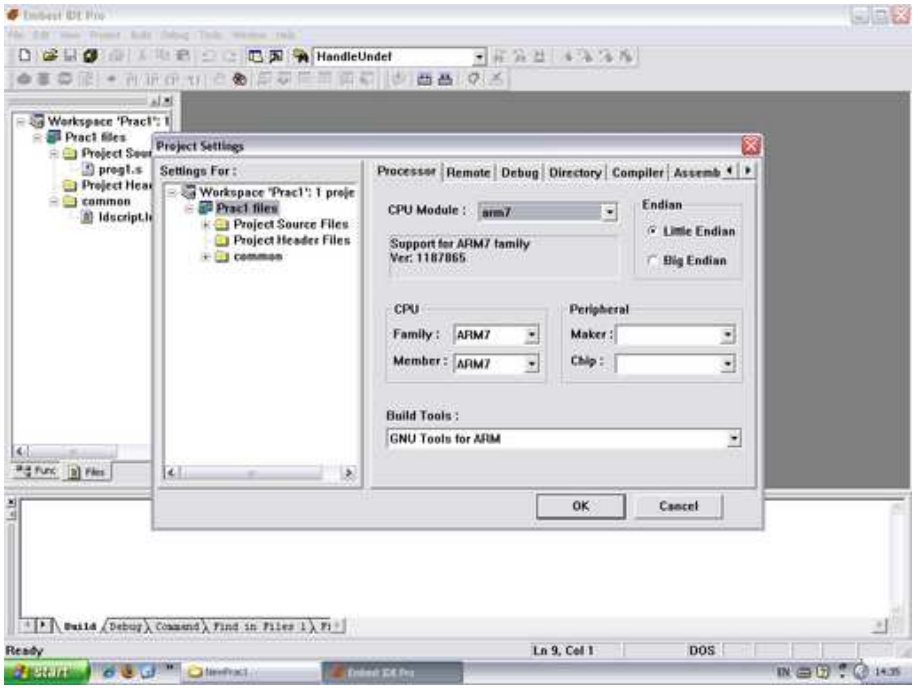


(b)

Figura 1.3: Pasos 1-3 para la creación del workspace de la práctica 1

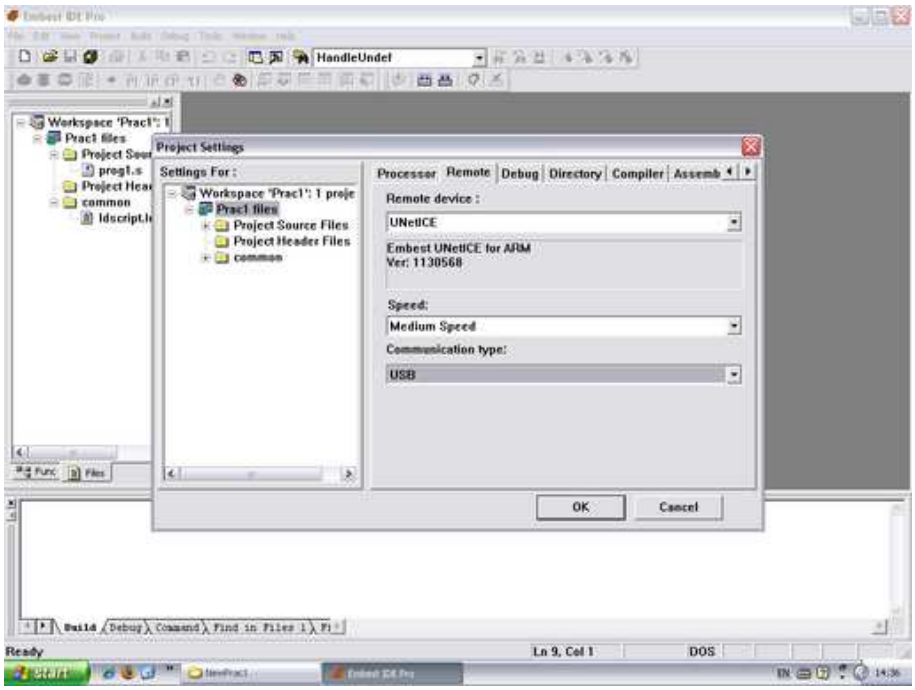


(a)

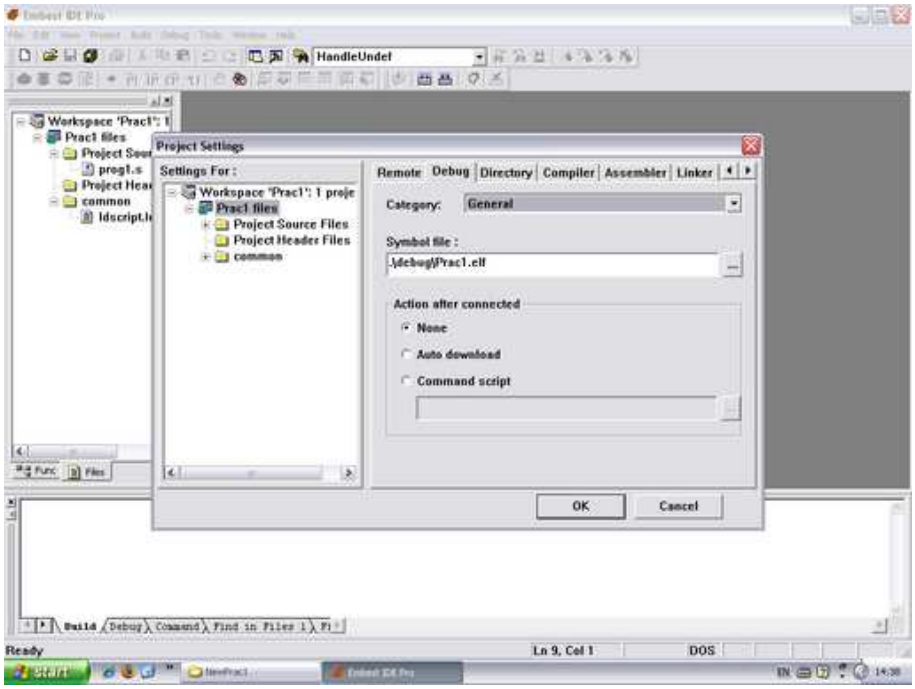


(b)

Figura 1.4: Pasos 4-6b para la creación del workspace de la práctica 1

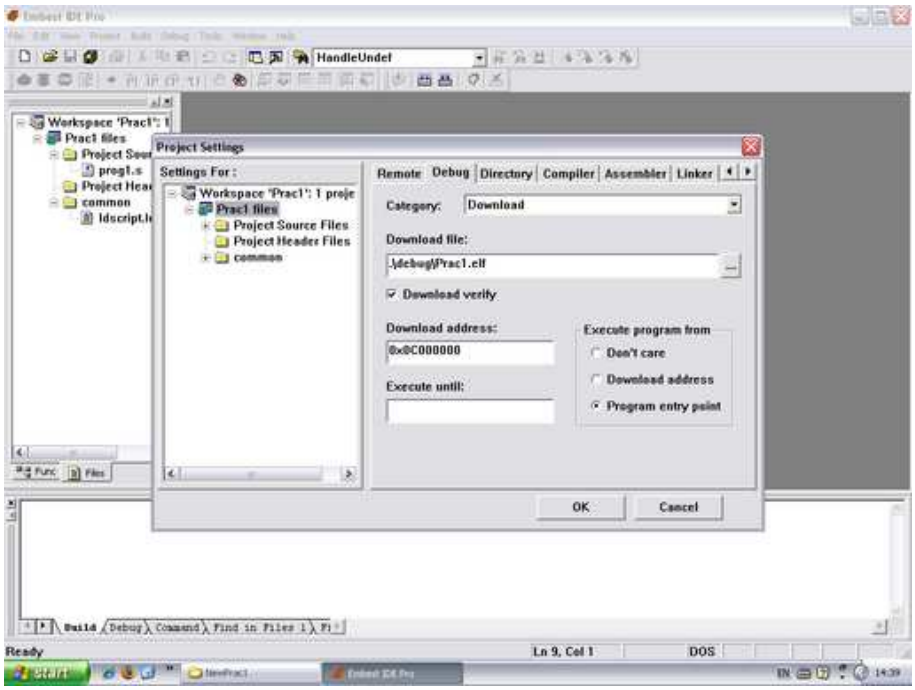


(a)

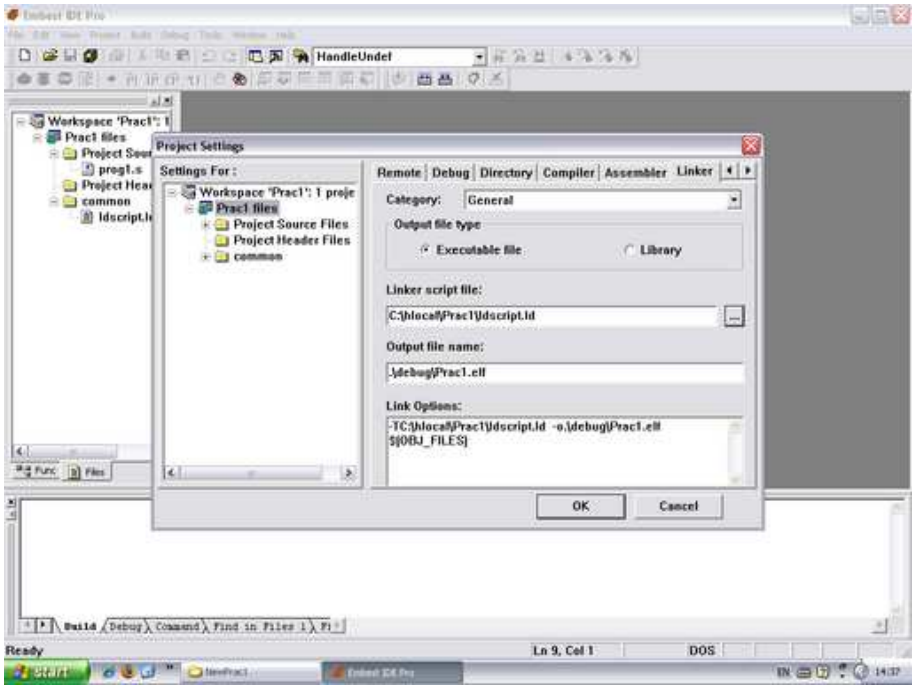


(b)

Figura 1.5: Pasos 5c-5d para la creación del workspace de la práctica 1



(a)



(b)

Figura 1.6: Pasos 5e-5f para la creación del workspace de la práctica 1

Figura 1.7: Depuración en circuito con el EmbestIDE conectado al UNetICE

```
/* Tabla destino */
DST: .word 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0

.text

start:  ldr sp, =STACK              @ inicialización del puntero de pila
        ldr r0, =SRC                @ puntero origen
        ldr r1, =DST                @ puntero destino
        mov r2,#0                   @ valores que deben ser recuperados
        mov r3,#0
        mov r4,#0
        mov r5,#0
        mov r6,#0
        mov r7,#0

        bl copy                     @ salto a rutina
FIN:    b .                         @ bucle infinito = FIN

copy:                               @ RUTINA de copia
        stmfd sp!,{r0-r7,sp,lr,pc}  @ guardamos los registros r0-r7, sp, lr y
                                    @ pc en la pila

blockcopy:
        mov r3, #N                  @ Palabras a copiar
        movs r3, r3, lsr #2         @ Bloques de 4 palabras a copiar
beq copywords

quadcopy:
        /* Rellenar con código que copie los bloques de
        4 palabras indicados por el contenido de r3
        utilizando load/store múltiple con los registros r4-r7*/

copywords:
        /* Rellenar con código que calcule el número de palabras
        que quedan por copiar y lo escriba en r3. Si quedan palabras
        debe saltar a wordcopy y si no a RET.*/

wordcopy:
        /* Rellenar con código que copie palabra a palabra
        lo que queda por copiar y al terminar vaya a RET*/

RET:    ldmfd sp!, {r0-r7,sp,pc}    @ restauramos los registros r0-r7, sp y
                                    @ hacemos que pc tome el valor que tenía lr

.ltorg
.end
```

Además deberemos modificar el script del ld de forma que se indique dónde vamos a poner la sección `.data`. En depuración podremos comprobar que las tablas se han inicializado correctamente. Por ejemplo podemos usar:

```
SECTIONS
{
    .text 0x0C000000: {
        _btext = .;
        *(.data)
        *(.text)
        _etext = .;
    }
    .bss 0x0C100000: {
        _bbss = .;
        *(.bss)
        _ebss = .;
    }
}
```

Completar el código, ejecutar paso a paso y comprobar que funciona correctamente, si no depurarlo. Responder a las siguientes preguntas:

- ¿En qué direcciones están las tablas origen y destino?
- ¿Cómo se consigue el retorno de subrutina?

1.9.3. Copia de cadenas de caracteres

Esta parte no será guiada. Se pide modificar el programa anterior para que sea capaz de copiar una cadena de caracteres, byte a byte para simplificar, que haya sido escrita con valor inicial en la sección de datos. No debe haber ninguna constante que indique el tamaño de la cadena, sino que la cadena se inicializará con un byte a cero al final de forma que podamos detectar el final de la cadena fácilmente. Recordar que en ensamblador podemos utilizar para ello la directiva `.asciz`.

1.9.4. Ordenación de un vector

Esta parte no será guiada. Se pide crear un programa que realice la ordenación de un vector almacenado en memoria. El algoritmo de ordenación es a elección del alumno.

Bibliografía

[arm] Arm architecture reference manual. Accesible en http://www.arm.com/miscPDFs/14128.pdf. Hay una copia en el campus virtual.

[gnu] Gnu arm assembler quick reference. Accesible en http://microcross.com/GNU-ARM-Assy-Quick-Ref.pdf. Hay una copia en el campus virtual.

[ld-] Using ld, the gnu linker. Accesible en http://www.zap.org.au/elec2041-cdrom/gnutools/doc/gnu-linker.pdf. Hay una copia en el campus virtual.