ARM: Documentación
Nociones básicas sobre la arquitectura ARM y el lenguaje ensamblador

Departamento de Ingeniería en Automática, Electrónica,

Arquitectura y Redes de Computadores

Universidad de Cádiz

José Alcántara Muñoz (Autor)
Mercedes Rodríguez García (Supervisora)

Índice general

Capítulo 1

ARM ayer y hoy

Hoy en día, muchos dispositivos móviles (teléfonos, tablets...) funcionan con procesadores ARM. Ahora bien, ¿qué es un procesador ARM?

Un procesador ARM (Advanced RISC Machine) es un RISC desarrollado originalmente en Acorn Computers Limited de Cambridge, Inglaterra, entre 1983 y 1985, y fue el primer microprocesador RISC desarrollado para uso comercial, teniendo algunas diferencias significativas con respecto a otras arquitecturas RISC.

En 1990, ARM Limited (tomando el nombre del procesador como el nombre de la empresa) se separó para dedicarse exclusivamente a la explotación de la tecnología ARM. Se ha convertido en un líder de mercado para aplicaciones embebidas de bajo consumo. Esta tecnología cuenta con un emulador del conjunto de instrucciones para modelado hardware, pruebas de software y benchmarking, un ensamblador, compiladores de C y C++ un linker y un depurador simbólico.

La arquitectura ARM sigue siendo similar a la originalmente creada por Acorn.

La razón por la que ARM ha ido ganando terreno es debido a su eficiencia energética. Se ha convertido en una arquitectura pensada para tecnologías móviles y es utilizada hoy en día en muchos teléfonos móviles y tablets. La familia ARM-Cortex es la más utilizada actualmente.

Capítulo 2

Características del procesador ARM

2.1. Herencia de otras arquitecturas

Para cuando la arquitectura ARM estaba siendo diseñada, los únicos ejemplos de RISC eran los productos del proyecto Berkeley RISC, RISC I y II, y el MIPS, de manera que ARM heredó algunas de las características de estas arquitecturas, principalmente las características propias de una arquitectura RISC:

- **Una arquitectura de carga-almacenamiento:** Se cuenta con instrucciones específicas para los accesos a memoria, siendo este el único medio para acceder a memoria.
- **Instrucciones de longitud fija:** 32 bits.
- **Solo tres formatos de instrucciones.**

Un aspecto muy importante a tener en cuenta es la simplicidad de esta arquitectura: se combina un hardware simple con un conjunto de instrucciones basado en las ideas RISC pero con unas pocas características CISC, alcanzando así una mayor densidad de código que un RISC puro, lo cual dota a esta arquitectura de una mayor eficiencia energética y un tamaño de núcleo más reducido.

2.2. Modelo de programador

Al desarrollar programas a nivel de usuario, sólo se tienen en cuenta 15 registros de propósito general y 32 bits (del r0 al r14), el contador de programa (r15) y el registro de estado del programa actual (CPSR, current program status register). El resto de registros sólo se usan para la programación a nivel del sistema y para el manejo de excepciones.

El CPSR es usado para almacenar los bits de código de condición (para grabar el resultado de una comparación o decidir si se toma un salto condicional).

Figura 2.1: Estructura del registro CPSR

El contenido del CPSR se muestra en la figura 2.1.

Los bits 4-0 indican el modo del procesador, el bit 5 (T) indica el conjunto o juego de instrucciones, el bit 7 indica si tiene lugar una interrupción normal (IRQ) y el 6 si tiene lugar una interrupción rápida (FIQ); estos bits están protegidos contra modificaciones por parte del programa de usuario. Por otra parte están los indicadores, los bits más significativos: N indica si el resultado de una ALUop es negativo (bit + significativo es 1), Z si el resultado es 0, C para indicar si hay acarreo, y V para indicar si hay desbordamiento.

En cuanto al uso de memoria, los datos pueden ser "bytes"de 8 bits, "halfwords"de 16 bits, o "words"de 32 bits. Las palabras siempre se alinean en límites de 4 bytes (por lo que los dos últimos bits de direcciones son 0), y las medias palabras hasta en un byte. La organización de memoria usada por ARM es el estándar little-endian, aunque puede ser configurado para trabajar con una organización big-endian. También cabe destacar la presencia del modo supervisor: el procesador ARM soporta un modo supervisor protegido, que asegura que el código de usuario no pueda adquirir privilegios de supervisor sin las condiciones adecuadas para asegurar que no se realicen operaciones ilegales.

Además, se trata de una arquitectura de carga-almacenamiento, por lo que sólo se accederá a memoria mediante instrucciones específicas, las instrucciones de transferencia de datos, que por ahora no trataremos.

Capítulo 3

Instrucciones del lenguaje ensamblador

En este capítulo veremos los distintos tipos de instrucciones que utilizaremos.

Al leer los ejemplos nótese que, en ensamblador ARM, los comentarios se marcan con una ';' , todo lo que haya escrito después de un ';' en una línea es ignorado por el ensamblador.

3.1. Instrucciones de procesamiento de datos

Una instrucción de procesamiento de datos es una instrucción que modifica el contenido de los registros, ya sea realizando una operación aritmético-lógica, una comparación o un movimiento de datos de un registro a otro. Tenemos distintos tipos de instrucciones de procesamiento de datos según la función de las mismas:

3.1.1. Operaciones aritmético-lógicas

Operan sobre dos registros de 32 bits. Las instrucciones aritméticas y lógicas suelen tener el siguiente formato:

```
ADD r0, r1, r2
```

donde el primer registro es el destino, el segundo es el primer operando, y el tercero es el segundo operando.

Tenemos las operaciones aritméticas ADD (suma simple), ADC (suma con acarreo), SUB (resta), SBC (resta con acarreo), entre otras. A continuación vemos un ejemplo de cada instrucción, donde C es el acarreo, tomado del valor del indicador C del CPSR en el momento de realizar la operación:

```
ADD r0, r1, r2 ; r0 := r1 + r2
ADC r0, r1, r2 ; r0 := r1 + r2 + C
```

```
SUB r0, r1, r ; r0 := r1 - r2
SBC r0, r1, r2 ; r0 := r1 - r2 + C - 1
```

Operaciones lógicas (a nivel de bit):

```
AND r0, r1, r2 ; r0 := r1 and r2
ORR r0, r1, r2 ; r0 := r1 or r2
EOR r0, r1, r2 ; r0 := r1 xor r2 \Exclusive OR"
BIC r0, r1, r2 ; r0 := r1 and not r2 \Bit Clear"
```

3.1.2. Multiplicación

Una forma especial de instrucciones de procesamiento de datos soporta la multiplicación:
`MUL r4, r3, r2 ; r4 := (r3 x r2)`
Este tipo de instrucción no admite no admite operandos inmediatos y no permite que el registro fuente 1 sea a la vez el registro destino.

Aunque se obtiene un resultado de 64 bits, sólo los 32 bits menos significativos se almacenan en el registro destino, perdiéndose el resto (aunque ARM también soporta el almacenamiento de la parte más significativa en un segundo registro).

3.1.3. Operaciones de movimiento de registros

Mueven el contenido del segundo registro al primero (no deben confundirse con las instrucciones de transferencia, que mueven los datos de registros a memoria y viceversa):

```
MOV r0, r2 ; r0 := r2
MVN r0, r2 ; r0 := not r2 (invierte cada bit)
```

3.1.4. Operaciones de comparación

Éstas no producen un resultado, sino que sólo ponen los bits de código de condición (N, Z, C y V) en el CPSR de acuerdo a la operación en cuestión:

```
CMP r1, r2 ; r1 - r2 ÇoMPare"
CMN r1, r2 ; r1 + r2 ÇoMpare Negated"
TST r1, r2 ; r1 AND r2 "(bit) TeST"
TEQ r1, r2 ; r1 XOR r2 "Test EQual"
```

Para explicar algo mejor el funcionamiento de estas últimas instrucciones, ponemos como ejemplo la instrucción CMP, utilizada en el programa que veréis más abajo: ésta realiza una resta entre r1 y r2, de forma que si el resultado es 0, ambos son iguales.

Los cambios realizados en el CPSR afectan al funcionamiento de las instrucciones de ejecución condicional condicionales.

3.2. Instrucciones de control de flujo

Podemos definir una instrucción de control de flujo como una instrucción que determina cuál será la siguiente instrucción a ejecutar.
Las instrucciones de control de flujo por excelencia son B, que realiza un salto al lugar del programa indicado mediante una dirección de memoria, normalmente representada por una etiqueta, y BL, que antes de realizar el salto almacena el valor del contador del programa en r14 (veremos la utilidad de esto más en adelante).

Añadiéndole un código de dos letras, podemos hacer que el salto sea condicional (ejecución condicional de la instrucción B), o bien decir que existen instrucciones de control de flujo como las siguientes:

Instrucción	Interpretación	Usos
B	Incondicional (Branch)	Siempre toma esta rama
BL	Subrutina (Branch and Link)	Salta al lugar indicado, guardando el pc en r14.
BEQ	Igual (Equal)	Comparación igual o resultado igual a 0
BNE	Diferente (Not Equal)	Comparación diferente o resultado diferente a 0
BPL	Positivo (Plus)	Resultado ¿= 0
BMI	Negativo (Minus)	Resultado negativo
BCC	Carry clear	Operación no ha dado acarreo
BCS	Carry set	Operación ha dado acarreo
BVC	Overflow clear	No sobrecarga tras operación de enteros
BVS	Overflow set	Sobrecarga tras operación de enteros
BGT	Greater than	Comparación de enteros dio ¿
BGE	Greater or equal	Comparación de enteros dio ¿=
BLT	Less than	Comparación de enteros dio ¡
BLE	Less or equal	Comparación de enteros dio ¡=

Estas “condiciones” se basan en el estado de los indicadores del procesador, que son modificados tras realizar operaciones arimético-lógicas para señalar si ha habido acarreo, el resultado de una comparación...

Cabe destacar una curiosa propiedad del ensamblador ARM: todas las instrucciones pueden ejecutarse de forma condicional (no solo la instrucción B, como pasa por ejemplo en MIPS), sin necesidad de utilizar explícitamente instrucciones condicionales de control de flujo (éstas son las instrucciones que reciben el nombre de instrucción condicional), para lo cual se añade un código de dos letras a la instrucción. Por ejemplo, si tenemos la instrucción `CMP # 0, # 1`, y justo después la instrucción `ADDEQ r0, r1, r2`, como 0 y 1 no son iguales, la suma no se ejecutará. Así, se puede apreciar que el código “EQ” ejecuta la instrucción si una comparación ha resultado en igualdad.

Uso de subrutinas

Muchos programas necesitan hacer un salto que lleve a una subrutina de manera que sea posible recuperar la secuencia de código original cuando ésta se haya completado. Para esto se deberá guardar el valor del contador del programa antes de tomar dicha rama. En el ensamblador ARM se puede hacer esto

mediante la instrucción de control de flujo BL (Branch and Link), que además de realizar un salto almacena el valor del contador de programa (pc o r15) en el registro de enlace (r14).

Para volver a la rutina desde la cual llamamos a la subrutina, debemos copiar el valor de r14 en el PC. Para esto podremos utilizar una instrucción MOV. A continuación tenemos un ejemplo de subrutina:

```
          BL SUBRUT       ; salta a SUBRUT, r14 apunta a...
          ...             ; ... esta instrucción
SUBRUT    ...
          MOV pc, r14     ; copiamos r14 en r15 para volver
```

La instrucción BL también puede ejecutarse condicionalmente, utilizando los mismos códigos que en la tabla de la instrucción B.

3.3. Instrucciones de transferencia de datos

Estas mueven datos entre registros y memoria principal. Hay tres tipos de instrucciones de este tipo:

- **Instrucciones de carga-almacenamiento de un registro:** proveen la forma más flexible de transferir datos entre los registros y la memoria, pudiendo ser el dato una palabra, una media palabra (aunque no en procesadores más antiguos), o un byte.
- **Instrucciones de carga-almacenamiento de varios registros:** menos flexibles que las anteriores (sólo funcionan con palabras), pero permiten transferir cantidades mayores de datos de forma eficiente.
- **Instrucciones de intercambio de un registro:** intercambian el valor de un registro con el de una dirección de memoria.

Las instrucciones de transferencia de datos del ARM están basadas en el direccionamiento indirecto de registros, que usa el valor de un registro como dirección de memoria. Las instrucciones principales de este tipo son:

```
   LDR r0, [r1] ; r0 := mem32[r1]
STR r0, [r1] ; mem32[r1] := r0
```

3.3.1. Inicializando un puntero a dirección

De lo anterior deducimos que, para cargar o almacenar desde una dirección de memoria, debemos inicializar un registro en el que almacenar dicha dirección o, en el caso de instrucciones de transferencia de un solo registro, una dirección dentro de los 4 KB de esa localización.
Si la localización está cerca del código que está siendo ejecutado es posible, frecuentemente, explotar el hecho de que el contador de programa, r15, está cerca de la dirección deseada: podríamos sumar un pequeño offset al r5 mediante una

instrucción de procesamiento de datos, aunque el cálculo del offset puede no ser tan sencillo. Para estos casos, contamos con la pseudoinstrucción ADR:

```
COPY      ADR r1, TABLA      ; r1 apunta a TABLA
TABLA                        ; sección de código a la que apunta r1
```

Se puede observar en este ejemplo el uso de etiqueta, que son nombres dados a puntos particulares del código.
Con la instrucción ADR, hemos conseguido que el registro r1 contenga la dirección de los datos que siguen a la etiqueta TABLA.

3.3.2. Instrucciones de carga/almacenamiento de un solo registro

Estas instrucciones calculan una dirección para la transferencia usando un registro base que debe contener una dirección cercana a la dirección deseada, y un offset que puede estar en otro registro o ser un valor inmediato. Ya vimos anteriormente un ejemplo de estas instrucciones:

```
    LDR r0, [r1] ; r0 := mem32[r1]
STR r0, [r1] ; mem32[r1] := r0
```

La dirección de palabra en r1 debería estar en un límite de 4 bytes, por lo que los dos bits menos significativos de la dirección deberían ser 0. Usando estas instrucciones, vamos a copiar la primera palabra de una tabla a otra:

```
COPY      ADR r1, TABLA1       ; r1 apunta a TABLA1
          ADR r2, TABLA2       ; r2 apunta a TABLA2
          LDR r0, [r1]         ; cargamos el primer valor de la tabla 1 en r0
          STR r0, [r2]         ; y lo almacenamos en la tabla 2
TABLA1                         ; fuente de los datos
TABLA2                         ; destino
```

Si quisiéramos coger más datos de una tabla y/o añadir mas datos en la otra tabla, tendríamos que sumarle el tamaño de nuestros datos (en nuestro caso, palabras completas de 4 bytes) en bytes al registro base.
En el siguiente ejemplo transferimos dos palabras de una tabla a otra, en vez de una:

```
COPY      ADR r1, TABLA1   ; r1 apunta a TABLA1
          ADR r2, TABLA2   ; r2 apunta a TABLA2
          LDR r0, [r1]     ; cargamos el primer valor de la tabla 1 en r0
          STR r0, [r2]     ; y lo almacenamos en la tabla 2
          ADD r1, r1, #4   ; avanzamos una palabra en TABLA1
          ADD r2, r2, #4   ; avanzamos una palabra en TABLA2
          LDR r0, [r1]     ; cargamos el segundo valor de la tabla 1 en r0
          STR r0, [r2]     ; y lo almacenamos en la tabla 2
TABLA1                     ; fuente de los datos
TABLA2                     ; destino
```

3.3.3. Direccionamiento de base + offset

Aunque la forma de usar direcciones anterior funciona en todos los casos, ARM cuenta con más modos de direccionamiento que permiten hacer un código más eficiente. A continuación vemos algunos ejemplos.

Si el registro base no contiene exactamente la dirección de memoria que necesitamos, podemos sumarle o restarle en el momento un offset de hasta 4 KB usando un modo de direccionamiento pre-indexado, de la siguiente forma:

```
LDR r0, [r1, #4] ; r0 := mem32[r1+4]
```

También podemos usar el direccionamiento pre-indexado con autoindexado, que tras calcular la dirección de memoria rehace el mismo cálculo para actualizar el registro base. Este modo se utiliza de la siguiente forma:

```
LDR r0, [r1, #4]!; r0 := mem32[r1+4]
                 ; r1 := r1 + 4
```

También está el post-indexado:

```
LDR r0, [r1], #4 ; r0 := mem32[r1]
                 ; r1 := r1 + 4
```

Finalmente hay que indicar que el offset puede ser tanto un valor inmediato como un registro.
Si queremos trabajar con Bytes en vez de palabras, sólo tenemos que añadir una B al nombre de la instrucción:

```
LDRB r0, [r1] ; r0 := mem8[r1]
```

3.3.4. Transferencia de datos de múltiples registros

Cuando tengamos que transferir cantidades considerables de datos es aconsejable utilizar estas instrucciones, que transfieren varios registros a la vez, se trata de las instrucciones LDMIA y STMIA, que sólo permiten transferencias de palabras completas. Aquí vemos un ejemplo de su funcionamiento:

```
LDMIA r1, {r0,r2,r5}    ; r0 := mem32[r1]
                        ; r2 := mem32[r1+4]
                        ; r5 := mem32[r1+8]
STMIA r1, {r0,r2,r5}    ; mem32[r1] := r0
                        ; mem32[r1+4] := r2
                        ; mem32[r1+8] := r5
```

LDMIA y STMIA funcionan por incremento post-indexado. También están LDMIB y STMIB (incremento pre-indexado), LDMDA y STMDA (decremento post-indexado) y LDMDB y STMDB (decremento pre-indexado).

3.3.5. Intercambio entre memoria y registros

Por último, trataremos la instrucción SWP (swap) o SWPB (si queremos trabajar con bytes). El formato de la instrucción es:

```
SWPB Rd, Rm, [Rn]
```

La instrucción carga la palabra o byte en la dirección de memoria a la que apunta Rn en Rd, y almacena el valor de Rm en la dirección de memoria apuntada por Rn. Rd y Rm pueden ser el mismo registro, con lo cual conseguiríamos intercambiar el valor de un registro con el de una dirección de memoria.

Capítulo 4

Técnicas de implementación

En este capítulo se presentan algunas técnicas de uso frecuente al escribir programas.

4.1. Bloques condicionales

Escribir un bloque condicional (lo que vendría siendo un if en un lenguaje de alto nivel) es muy parecido a escribir una subrutina. Para escribir un bloque de ejecución condicional utilizaremos una instrucción de salto o control de flujo (B) estableciendo alguna condición añadiéndole dos letras a la B, según la tabla del apartado anterior. Ahora bien, en esta ocasión, al salto se le debe indicar lo que se debe cumplir para NO ejecutar el bloque condicional, indicando el lugar donde termina un bloque con una etiqueta que pondremos en la primera línea posterior al bloque. Así, un bloque condicional, normalmente, tiene la siguiente estructura:

```
          ...                    ;Operación o comparación(CMP, por ejemplo)
          B<condicion>ETIQUETA   ;salta a ETIQUETA si se cumple la condicion
          ...
ETIQUETA  ...
          ...
```

Otra posible forma de implementar un bloque condicional es, precisamente, mediante una subrutina.

4.2. Bucles

Los bucles, al igual que los bloques condicionales, utilizan etiquetas e instrucciones de control de flujo. Se procede a explicar distintas implementaciones típicas para bucles.

4.2.1. Bucles do-while

Los bucles do-while suelen seguir la siguiente estructura en ensamblador:

```
BUCLE     ...              ; cuerpo del bucle
          ...              ; operación o comparación
          B<condicion>BUCLE
```

A continuación, un ejemplo de bucle do-while:

```
BUCLE     ADD r1,r1,#1
          CMP r1,#10
          BLT BUCLE        ; salto si r1<10
```

Este bucle equivaldría, en C o C++ a lo siguiente:
`do{ r1=r1+1; }while(r1<10)`

4.2.2. Bucles while

El esquema típico de los bucles while:

```
          B COND
BUCLE     ...              ; cuerpo del bucle
          ...
COND      ...              ; operación o comparación
          B<condicion>BUCLE
```

Ejemplo:

```
          B COND
BUCLE     ADD r1, r1, #1
COND      CMP r1,#10
          BLT BUCLE        ; salto si r1<10
```

En C o C++ equivaldría a:
`while(r1<10) r1=r1+1;`

4.2.3. Bucles for

A continuación, un esquema para los bucles for, aunque en este caso el esquema variará según lo que queramos inicializar y según como queramos ïncrementar.º modificar la variable utilizada para controlar el número de iteraciones:

```
          MOV rn, #x       ;inicialización
BUCLE     ...              ; operación o comparación
          B<condicion>HECHO
          ...              ; cuerpo del bucle
          ADD rn, rn, #y   ;incremento/decremento
```

```
          B BUCLE
HECHO     ...              ; fin del bucle
```

Un ejemplo de bucle for:

```
          MOV r1, #0
BUCLE     CMP r1, #10
          BGE HECHO        ; salto si r1>=10
          ADD r1, r1, #1
          B BUCLE
HECHO     ...              ; fin del bucle
```

Este ejemplo equivaldría a algo tan sencillo como lo siguiente:
`for(int r1=0;r1<10;r1++);`

Capítulo 5

Directivas usadas frecuentemente

Para poder escribir programas completos, necesitaremos usar ciertas directivas. A continuación veremos las directivas de uso más frecuente, muchas de ellas fundamentales para escribir programas completos.

5.1. AREA

Los programas en ensamblador de la arquitectura ARM suelen empezar con la directiva AREA, que marca el inicio de una sección de una aplicación o programa. La directiva AREA sirve para dividir el código en secciones ELF (todos los códigos con el mismo nombre indicado por la directiva se agrupan en la misma sección ELF) ejecutables.
AREA le da un nombre a la sección de código e indica sus atributos, indicándose en primer lugar el nombre y señalando posteriormente los atributos separados por comas. Algunos atributos válidos son:

- ALIGN=expresión: Por defecto las secciones ELF están alineadas en un límite de 4 bytes. Con este atributo modificamos el valor dicho límite, que se establece mediante la fórmula $2^{\texttt{expresión}}$ (en bytes). `expresión` puede tomar valores entre 0 y 31, ambos incluidos. Sin embargo, para secciones de código `expresión` no puede valer 0 o 1, pues esto equivale al tamaño de un byte o de media palabra, respectivamente.
- CODE: La sección contiene instrucciones. Por defecto este atributo implica que la sección tiene el atributo READONLY, explicado más en adelante.
- DATA: La sección contiene sólo datos, no instrucciones. Por defecto implica el atributo READWRITE.
- READONLY: Indica que de esta sección sólo se puede leer código.
- READWRITE: Indica que de esta sección se puede tanto leer como escribir.

Usa la directiva AREA para subdividir los ficheros de código en secciones. Por ejemplo, se debe subdividir un fichero para separar las secciones de datos y de código del mismo.

5.2. RN

La directiva RN es utilizada para definir un nombre para un determinado registro, y así recordar mejor la función que se le va a dar. El formato de la directiva es:

```
nombre RN expresión
```

donde `expresión` es un valor entre 0 y 15 correspondientes a los registros r0 a r15. Un ejemplo de esto sería:

```
contPrograma RN 15
```

5.3. EQU

La directiva EQU es utilizada de manera similar al #define de C: le otorga un nombre simbólico a una constante numérica o a un valor relativo a un registro o un programa. La sintaxis es la siguiente:

```
nombre EQU expresión, tipo
```

donde `expresión` es una dirección relativa a un registro, a un programa, una dirección absoluta, o una constante entera de 32 bits. El parámetro `tipo` es opcional y puede ser una de las siguientes opciones: CODE16, CODE32, DATA. Para las prácticas de esta asignatura haremos un uso básico de la directiva EQU, luego en la mayoría de los casos solo será necesario especificar un entero de 32 bits.

5.4. ENTRY

La directiva ENTRY declara un punto de entrada a un programa. Debe especificarse un punto de entrada para cada fichero de código. Se declara un punto de entrada a un programa escribiendo simplemente `ENTRY`.

5.5. DCB, DCW y DCD

La directiva DCB asigna uno o más bytes de memoria y define el contenido inicial de la memoria. La sintaxis es:

```
etiqueta DCB expresión, expresión...
```

donde `expresión` es una expresión numérica que devuelve un entero entre -128 y 255 o una cadena de caracteres especificada entre comillas. Los caracteres de la cadena son cargados en bytes consecutivos de memoria (recordando que si queremos que la cadena tenga terminación nula, debemos añadir una `expresión` detrás: el entero 0.

Si se va a utilizar una instrucción justo después de esta directiva, se debe utilizar una directiva ALIGN para asegurar que la instrucción sea alineada.

DCW realiza la misma tarea de DCB, pero asignando medias palabras (halfwords-2 bytes). Si escribimos DCWU, el alineamiento de memoria será arbitrario. La sintaxis es la siguiente:

```
etiqueta DCWU expresión, expresión...
```

donde `expresión` puede tomar valores entre -32768 y 65535.

DCD, en cambio, trabaja con palabras completas (4 bytes). `expresión` puede ser una expresión numérica o relativa al programa. DCD inserta hasta 3 bytes de relleno antes de la primera palabra definida, si es necesario, para alcanzar el alineamiento de 4 bytes. Usa DCDU si no necesitas alineamiento. El formato aparece a continuación:

```
etiqueta DCDU expresión, expresión
```

5.6. ALIGN

La directiva ALIGN alinea la ubicación actual a un límite específico rellenado con ceros. La sintaxis es:

```
ALIGN expresión, offset
```

donde `expresión` es una expresión numérica equivalente a una potencia de 2 desde 2^0 hasta 2(31), y `offset` puede ser cualquier expresión numérica. La ubicación actual es alineada a la siguiente dirección de la forma `offset` $+ n *$ `expresión`.
Si `expresión` no se especifica, la directiva moverá la ubicación actual al siguiente límite de 4 bytes (una palabra).

5.7. SPACE

Esta directiva reserva un bloque de memoria inicializado a cero. Sigue el siguiente formato:

```
etiqueta SPACE expresión
```

donde `expresión` es el número de bytes inicializados a cero.

5.8. END

La directiva END informa al ensamblador de que se ha alcanzado el final de un fichero. Todo fichero en lenguaje ensamblador debe terminar con la directiva END escrita en una línea por separado.

Capítulo 6

Nuestras herramientas: ARM Software Development Toolkit

Para programar con el lenguaje ensamblador ARM, trabajaremos con dos programas: *ARM Project Manager* y *ARM debugger*, que utilizan ARMulator, un sistema capaz de simular procesadores de arquitectura ARM.

6.1. Creando el proyecto

Para poder comenzar nuestra práctica, necesitaremos cargar en ARM Project Manager el código ensamblador que veréis más abajo. Para hacerlo:

1. File / New... / Project y pulsamos OK
2. Aparecerá una ventana: Escribimos el nombre del proyecto y dejamos el resto de campos tal cual están. Aceptamos.
3. Una vez creado el proyecto, habrá que crear el fichero que contendrá el código en ensamblador: File / New... / Assembler source y pulsamos OK-
4. Copiamos o escribimos el código en el fichero y guardamos el fichero con extensión ".s".
5. Acto seguido nos vamos a Project / Add files to project y seleccionamos el fichero creado.

6.2. Compilando el proyecto

Para compilar el proyecto, nos vamos a Project / Build nombre_proyecto.apj "Debug" o pulsamos Shift+F8. Nos aparecerá una ventana como la de la figura 6.1

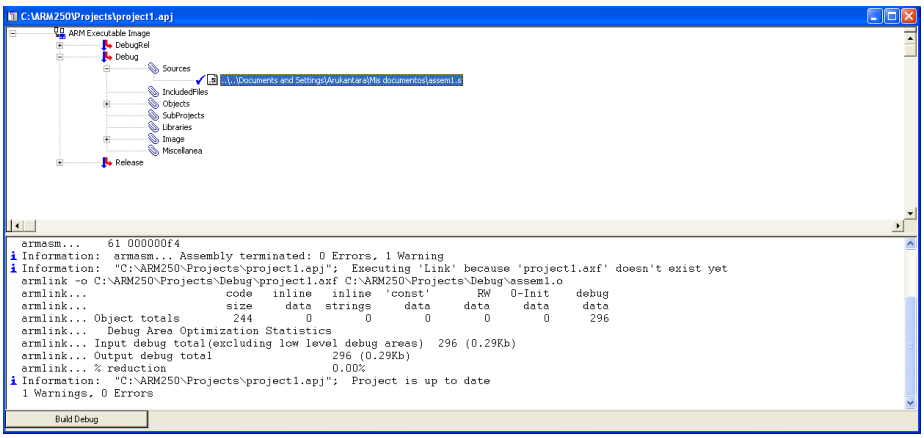

Figura 6.1: Ventana del proyecto mostrando resultados de la compilación

En la parte superior se pueden visualizar los ficheros del proyecto y, en la parte inferior, los errores de compilación. Si hay errores de compilación que impidan compilar el código por completo, habrá que editar el código desde la otra ventana del ARM Project Manager.

6.3. Ejecutando/Depurando el proyecto

Para ello podemos ir a Project / Execute nombre_proyecto.apj “Debug” o pulsar Ctrl+F5, lo cual nos llevará al ARM Debugger. Otra opción es abrir el ARM Debugger y seleccionar el proyecto desde File / Load Image...

Una vez abierto el programa y el proyecto, tenemos tres ventanas, de las cuales utilizaremos dos, principalmente:

- La ventana superior izquierda, que contiene el código y nos muestra en qué línea de código estamos cuando lo ejecutamos.
- La ventana inferior, que muestra las salidas del programa y permite escribir las entradas del mismo.

Asimismo, en el desarrollo de las prácticas se abrirá una ventana con el valor de los registros (figura 6.3), así como una para acceder al contenido de memoria (figura 6.4).

La activamos pulsando View / Registers / User mode. Se puede cambiar los valores de los registros haciendo doble click en uno de ellos, escribiendo el nuevo valor y pulsando Enter. Esto será muy importante para el desarrollo de las prácticas.

Podremos activar también la ventana de contenido de memoria tecleando Ctrl+M (o clicando en View / Memory), y clicando en OK.

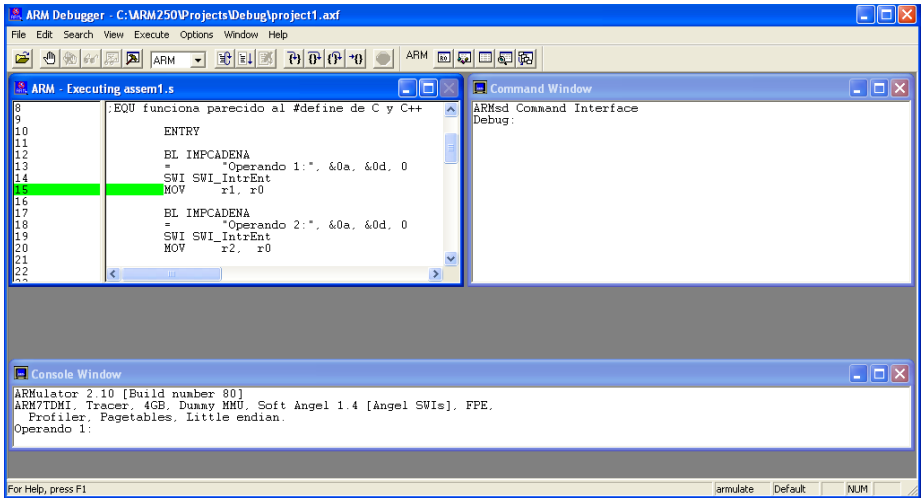

Figura 6.2: ARM Debugger

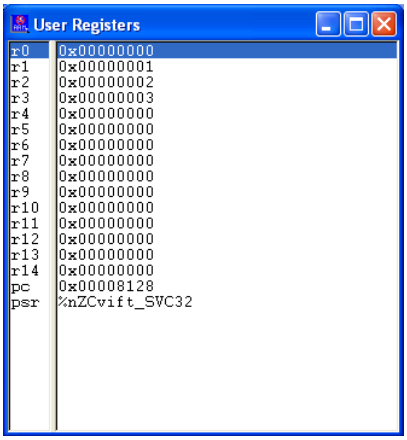

Figura 6.3: Ventana de registros

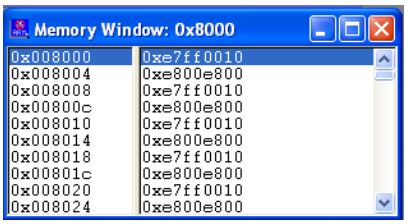

Figura 6.4: Ventana de registros

6.3.1. Uso de breakpoints

Para el desarrollo de esta práctica también será necesario el uso de breakpoints (importantes también cuando un código no funciona como esperamos).

Un breakpoint es una marca en una línea de código que hará que la ejecución se pause antes de ejecutar dicha línea. Para poner un breakpoint, hacemos click derecho en la línea de código correspondiente (no puede ser una línea vacía) y en "Toggle breakpoint" o bien hacemos click izquierdo y pulsamos F9.

¿No funciona algún breakpoint? Por desgracia, en ARM Debugger a veces pasan estas cosas. Para solucionar el problema, sigue los siguientes pasos en los breakpoint que no funcionen:

1. Pulsa Ctrl+I
2. Busca la línea de código (tendrá el mismo número de línea que antes a la izquierda)
3. Dos líneas más abajo encontrarás una línea de código con el mismo contenido.
4. Pon un breakpoint en esa línea de la misma forma que lo hiciste antes.
5. Vuelve a pulsar Ctrl+I para volver a visualizar el código normalmente.

Capítulo 7

Bibliografía

- **ARM system-on-chip architecture** (segunda edición).
  Autor: Steve Furber
  Editorial: Addison-Wesley
  ISBN: 0-201-67519-6

- **ARM Assembly Language Fundamentals and Techniques**.
  Autor: William Hohl
  Editorial: CRC Press
  ISBN: 978-1-4398-0610-4