Práctica 2: Introducción a los modos de direccionamiento

2.1 Objetivos

Una vez familiarizados con el repertorio de instrucciones del ARM y con el entorno de desarrollo EmbestIDE, en esta práctica se pretenden reforzar los siguientes aspectos de la programación en ensamblador:

- Adquirir práctica en el manejo de vectores de datos, cargar y almacenar datos de memoria conociendo la posición de inicio de un vector y su longitud.
- Codificar en ensamblador del ARM sentencias especificadas en lenguaje de alto nivel.

2.2 Modos de direccionamiento

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 constantes, contenida dentro de una instrucción de la máquina.

La arquitectura ARM dispone de un amplio conjunto de modos de direccionamiento. Revisaremos brevemente a continuación los tres modos de direccionamiento de las instrucciones de load/store que son más útiles para el desarrollo de esta práctica:

- **Indirecto de registro**. La dirección de memoria a la que deseamos acceder se encuentra en un registro del banco de registros. En la Figura 2.1 se ilustra el resultado de realizar la operación `ldr r2,[r1]`

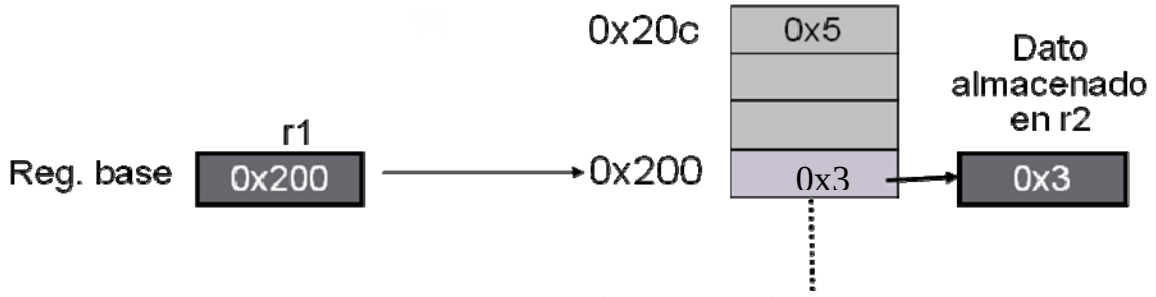

Figura 2.1 Ejemplo de ejecución de instrucción ldr r2,[r1]

- **Indirecto de registro con desplazamiento constante**. 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. El desplazamiento se codifica con 12 bits en el convenio *complemento a 2*.

En la Figura 2.2.2 se ilustra el resultado de realizar la operación `ldr r2,[r1,#12]`

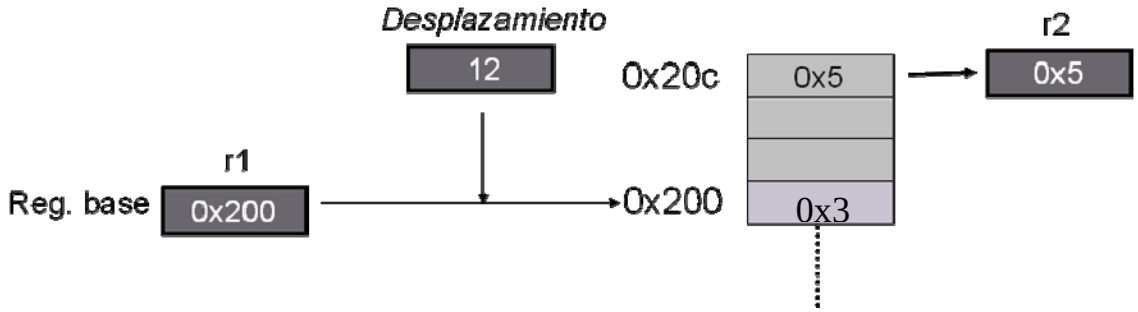

Figura 2.2 Ejemplo de ejecución de la instrucción ldr r2,[r1,#12]

- **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 almacenado en otro registro (*offset*). Este segundo registro puede estar opcionalmente multiplicado/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
  - Desplazamiento lógico a la izquierda (LSL). Equivalente a multiplicar por una potencia de 2. En la Figura 2.2.3 se ilustra el resultado de realizar la operación `str r3,[r5,r1,LSL#2]`

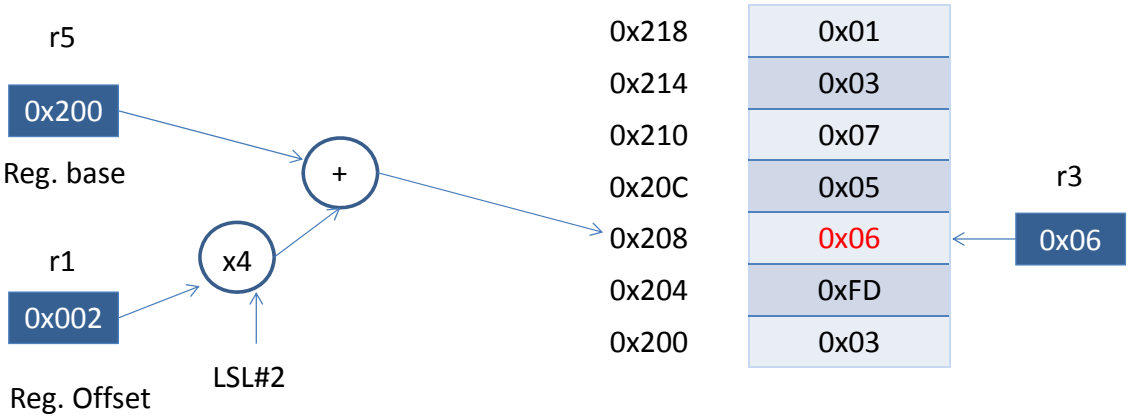

Figura 2.3 Ejemplo de ejecución de la instrucción str r3,[r5,r1,LSL#2]

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,[R5,R1,LSL#2]`	`Almacena en Mem[R5+R1*4] lo que hay en R4`
`LDR R11,[R3,R5,LSL#3]`	`Carga en R11 lo que hay en Mem[R3+R5*8]`

Este último modo de direccionamiento es el más adecuado para recorrer *arrays*: en un registro mantendremos la dirección de comienzo del *array* mientras en otro registro mantenemos el índice del *array* que queremos acceder. Bastará con multiplicar dicho registro (usando *LSL)* por el tamaño adecuado del dato (4 para números enteros) para conseguir la dirección del elemento deseado. El siguiente ejemplo ilustra la utilidad de este modo direccionamiento al recorrer un *array:*

```
int v[100];

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

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

En el caso de los *arrays* para poder recorrerlos es necesario tener almacenada en un registro la dirección de comienzo del *array* (*r1*, en el ejemplo anterior). Para ello resulta necesaria la pseudoinstrucción:

LDR R<n>,=ETIQUETA

Guarda en el registro R<n> la dirección asociada a *Etiqueta*, como podemos ver en la Figura 2.4. La variable *A* ha sido previamente inicializada usando la directiva *A: .word 6*, que guarda en la dirección de memoria 0x208 el valor 6 y asocia esa dicha dirección a la etiqueta *A*.

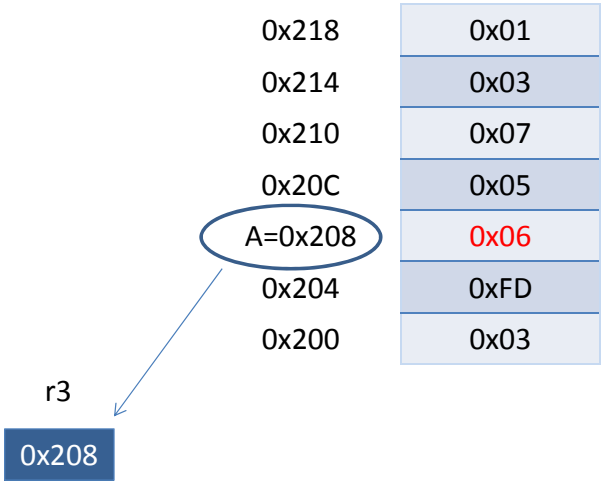

Figura 2.4 Ejemplo de ejecución de la instrucción ldr r3,=A

2.3 Secciones de un ejecutable

La Figura 2.5 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. A partir de los diferentes ficheros en código ensamblador, se genera el código objeto para la arquitectura destino (ARM en nuestro caso).

Sin embargo, el programa en ensamblador, generado por el compilador o a mano por un programador, puede hacer referencia a ciertos símbolos aún no definidos, por ejemplo, la dirección de comienzo de alguna subrutina que esté definida en otro fichero. Esto hará que el ensamblador cree una entrada en la tabla de símbolos del código objeto resultante indicando que dicho símbolo debe ser importado desde otro fichero objeto. Esta tabla de símbolos también contendrá una entrada por cada símbolo exportado, normalmente serán funciones o variables globales que queramos exportar para utilizarlas desde otros ficheros.

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.

El enlazador es el responsable de generar el código ejecutable final. Para ello tomará varios ficheros de código objeto, procedentes de bibliotecas externas y/o procedentes de la compilación de nuestros ficheros fuente en C o ensamblador, cada uno con sus propias secciones de código y datos. Tomará estas secciones de entrada y las reubicará en distintas direcciones de memoria para formar las secciones del ejecutable final, determinando así el

mapa de memoria del programa. En este proceso todos los símbolos habrán quedado resueltos.

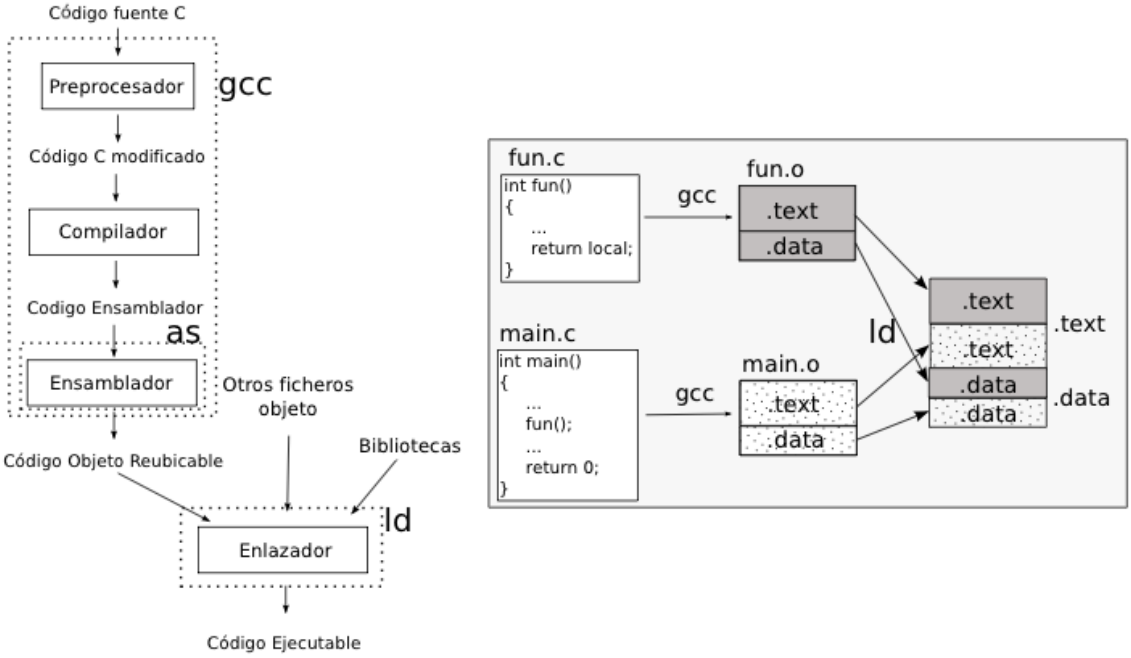

Figura 2.5 Proceso de generación de un ejecutable

Como ya hemos explicado en la práctica 1, 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. Por ejemplo, el programa de la Figura 2.6 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).

Otro aspecto relevante del código de ejemplo es el uso de la instrucción *LDR R2,=A*.. En este caso es **imprescindible** usar esta pseudo-instrucción, ya que la variable *A* está en una sección diferente del código, y no es posible generar su dirección durante la compilación.

```
.global start

.equ UNO, 0x01

.bss
RES: .space 4

.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
     STR R4, RES
     FIN: B .
.end
```

Figura 2.6 Ejemplo código con secciones

2.4 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 **A** de enteros positivos 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*).

```
#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];
}
```

```
.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 realizar una instrucción *str*).

```
#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;
}
```

```
.EQU N, 8

.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
```

c. Modificación de alguno de los apartados a) y b) que se dará a conocer en el laboratorio.

2.5 Procedimiento de entrega y evaluación

Antes del final de la sesión de laboratorio el alumno deberá:

- Entregar, a través del campus virtual, el fichero de texto correspondiente al tercer apartado de la práctica (2.5.c). Dicho fichero deberá llamarse **prog3.s**
- Contestar a las **cuestiones** planteadas a través del Campus Virtual. Las cuestiones podrán ser sobre cualquiera de los tres apartados. No es necesario responderlas en orden, pero es **IMPRESCINDIBLE ENVIAR** las respuestas antes del final de la sesión. Las respuestas deberán tener el mismo formato que en la primera práctica. Cualquier respuesta que NO se ajuste al formato pedido, será considerada incorrecta. Ejemplos:
  - Si se pregunta en qué registro está almacenado un determinado valor, se responderá únicamente "R<n>", siendo <n> el índice del registro tal y como aparece en la ventana del simulador.
    - Repuesta válida: `R2`
    - Respuesta NO válida: `registro 2`
    - Respuesta NO válida: `El dato está en el registro 2`
  - Si se pregunta por una dirección de memoria se responderá con la dirección expresada en hexadecimal usando **8 dígitos.** Si se desea, se puede preceder la dirección de "0x", pero no es necesario.
    - Respuesta válida: `0X0C000008`
    - Respuesta válida: `0C000008`
    - Respuesta NO válida: `0XC000008`
    - Respuesta NO válida: `C000008`
    - Respuesta NO válida: la dirección es `0x0C000008`
  - Se pueden usar tanto mayúsculas como minúsculas.