

## PRÁCTICA 1: DESARROLLO DE CÓDIGO PARA EL PROCESADOR ARM

### OBJETIVOS:

- Conocer la estructura segmentada en tres etapas del procesador ARM 7 que actualmente es uno de los más utilizados en los sistemas empuotrados.
- Familiarizarse con el entorno Eclipse sobre Windows7, con la generación cruzada de código para ARM y con su depuración.
- Desarrollar código en ensamblador con los dos repertorios de instrucciones que soporta este procesador:
  - ARM: adecuado para optimizar el rendimiento.
  - Thumb: adecuado para reducir el tamaño del código y el consumo de energía.
- Optimizar código a nivel de ensamblador.
- Combinar de manera eficiente código en ensamblador con código en C.
- Depurar código en ensamblador, siguiendo el contenido de los registros internos del procesador y de la memoria.

### CONOCIMIENTOS PREVIOS NECESARIOS:

En esta asignatura el estudiante necesitará manejar con soltura los contenidos impartidos en las asignaturas previas, en particular, Arquitectura y Organización de Computadores I y II.

### ENTORNO DE TRABAJO:

Esta práctica se realiza con Eclipse, que está instalado en el laboratorio de la asignatura, junto con las herramientas de gcc para compilación cruzada y el complemento (*plug-in*) para simular procesadores ARM7TDMI.

### MATERIAL DISPONIBLE:

En la página web de la asignatura (moodle) puede encontrarse el siguiente material de apoyo:

- Cuadernos de prácticas de una asignatura de la Universidad Complutense que usa el mismo entorno de trabajo y las mismas placas
- Breve resumen del repertorio de instrucciones ARM.
- Breve resumen del repertorio de instrucciones Thumb.
- Manual de la arquitectura ARM.
- Fuentes para los apartados A y B

## ESTRUCTURA DE LA PRÁCTICA:

La práctica consta de dos partes, la primera es una toma de contacto con el entorno de trabajo y el procesador ARM7, mientras que en la segunda el estudiante deberá realizar un pequeño proyecto de desarrollo de código.

La práctica completa se debe realizar en unas 3 semanas y será la base para la práctica siguiente (las fechas concretas se anunciarán en la web de la asignatura).

## PARTE A: EJEMPLO DE DESARROLLO DE CÓDIGO PARA EL PROCESADOR ARM7

### DURACIÓN: 1 SEMANA

**TRABAJO PREVIO.** Antes de realizar esta práctica se debe repasar la arquitectura ARM y cómo trabajar en un entorno mixto C/ensamblador.

El objetivo de este apartado es desarrollar una función que copie diez datos de una zona de memoria. Deben implementarse tres versiones distintas:

- Ensamblador ARM: se proporciona un esqueleto al cual deben añadirse algunas instrucciones.
- Ensamblador Thumb: se deben añadir algunas instrucciones al esqueleto facilitado.
- C: en este caso se suministra la función en C.

La función tiene los siguientes parámetros:

- src: dirección de inicio de los datos a copiar
- dst: dirección de destino de los datos copiados

El estudiante deberá integrar las tres versiones de la función en el esqueleto que se proporciona y comprobar que el código se ejecuta correctamente (se recomienda no integrar una función hasta haber depurado las anteriores). Se debe comparar el rendimiento de las tres implementaciones, para ello cada grupo deberá medir el tamaño del código de cada una de las tres versiones así como el número de instrucciones que ejecuta cada una.

Los pasos a seguir para configurar el entorno de trabajo (*workspace*) están detallados en el archivo: "guía del entorno" (el código que se usa es distinto, pero los pasos a seguir son los mismos). En concreto, en la página 3 se indica la creación de un proyecto y en la página 16 "depuración sobre el simulador". Dadas las características concretas de nuestro laboratorio existen pequeñas diferencias:

- Es preciso añadir a la variable de entorno PATH el camino correcto al compilador antes del punto 9 de la página 4 y al depurador al final del punto 2 de la página 17 (ver detalle en documentación en la web de la asignatura).

- No es recomendable trabajar en local (que es lo que indica la guía del entorno en el punto 2). Debéis trabajar en un directorio en el que tengáis permisos de lectura y escritura y trabajar con *paths* cortos (Eclipse no reconoce los *paths* largos). Por ejemplo D:\Workspace es una buena elección.

Este apartado no hace falta presentarlo, pero es imprescindible realizarlo para poder continuar. Su objetivo es que la primera toma de contacto con ARM sea sencilla, y que el siguiente apartado resulte más sencillo. Además tenéis que aprender a configurar los proyectos de Eclipse, dado que **el siguiente apartado se configurará de manera similar a éste**. De todas formas recomendamos que se lo enseñéis a uno de los profesores para comprobar que lo habéis hecho todo bien.

## PARTE B: ACELERACIÓN DE UN JUEGO DE SUDOKU.

### DURACIÓN: 2 SEMANAS

El Sudoku (en japonés: 数独 *sūdoku*) es uno de los juegos matemáticos más populares en el mundo. La clave de su éxito es que las reglas son muy sencillas, aunque resolverlo puede llegar a ser muy difícil.

El sudoku se presenta normalmente como un tablero o **cuadrícula** de  $9 \times 9$ , compuesta por subtablas de  $3 \times 3$  denominadas recuadros o **regiones**. Algunas **celdas** ya contienen números, conocidos como números dados (o a veces **pistas**).

El objetivo es rellenar las celdas vacías, con un número en cada una de ellas, de tal forma que cada columna, fila y región contenga los números 1–9 solo una vez.

Las reglas son sencillas:

- Se parte de una cuadrícula donde ciertas celdas ya contienen una pista
- Para completar el sudoku se deben rellenar las celdas vacías de forma que cada fila, cada columna y cada región contenga todos los números del 1 al 9.
- Cada número de la solución aparece solo una vez en cada una de las tres "direcciones", de ahí el "los números deben estar solos" que evoca el nombre del juego.

Los métodos de resolución son múltiples.

La mayoría de sudokus de nivel medio hacia arriba son difíciles de resolver y requieren de técnicas sofisticadas, estas técnicas implican que de alguna manera llevemos la cuenta en cada momento de los valores o **candidatos** posibles para cada celda. Se trata de reducir el número de candidatos hasta encontrar una celda con un solo candidato. A partir de esta nueva información, se selecciona otra celda y se repite el proceso de reducir candidatos hasta encontrar la solución.

	C1	C2	C3	C4	C5	C6	C7	C8	C9
R1	9	6					7		8
R2	8					4	3		
R3	1			5					
R4							1	7	6
R5	2				9	3			5
R6	7		8						
R7			7		3	2		4	
R8	3	8	2	1		5	6		
R9		4	1			9	5	2	

Examinemos los posibles candidatos de la celda R5C8 (fila 5, columna 8). Mirando la fila 5 observamos que los valores 2, 3, 5 y 9 ya están en uso, como solamente pueden aparecer una vez por fila, podemos eliminarlos de la lista de candidatos. Inspeccionando la columna podemos eliminar además el 4 y el 7. Centrándonos en la región vemos que tampoco pueden ser el 1, ni el 6.

	C1	C2	C3	C4	C5	C6	C7	C8	C9
R1	9	6					7		8
R2	8					4	3		
R3	1			5					
R4							1	7	6
R5	2				9	3			5
R6	7		8						
R7									
R8	3	8							
R9		4							

	C1	C2	C3	C4	C5	C6	C7	C8	C9
R1	9	6					7		8
R2	8					4	3		
R3	1			5					
R4							1	7	6
R5	2				9	3			5
R6	7		8						
R7									
R8	3	8							
R9		4							

¿Qué nos queda en la lista de candidatos?, solo hay una posibilidad: el 8.

Normalmente este procedimiento se realiza escribiendo los posibles candidatos con un lápiz sobre el papel.

Cierta empresa quiere lanzar un sistema, que se ejecutará en un procesador ARM7, para facilitar que una persona resuelva el rompecabezas. PH\_sudoku permitirá al jugador/a ahorrar este rutinario trabajo escribiendo los candidatos de cada celda, permitiéndonos la edición y relleno de celdas con nuevos valores y la correspondiente actualización de la cuadrícula. Esto facilitará la resolución de sudokus.

#### TRABAJO A REALIZAR

A lo largo del curso os encargareis de la implementación de PH\_sudoku realizando un prototipo sobre un sistema de desarrollo disponible en el laboratorio. El dispositivo ayudara en las tareas de escaneo y marcado de las celdas, pero el análisis para la resolución recae íntegramente sobre el jugador humano.

El usuario debe ser capaz de introducir cualquier número del 1 al 9 en las celdas que no contengan una pista inicial, rellenando una celda vacía o modificando un valor previo. Debe, así mismo, ser capaz de borrar un número de una casilla suya poniendo un cero (volverá al estado inicial). Para terminar la partida el usuario introducirá un valor en la casilla no existente R0C0. Tras cada movimiento el sistema debe comprobar que no hay errores y calcular la lista de candidatos de cada celda.

El dispositivo PH\_sudoku debe tener un coste reducido, por ello están preocupados por las necesidades computacionales y tiempo de ejecución y el tamaño de la memoria necesaria. En esta primera práctica nos centraremos en la implementación eficiente de las que en principio se consideran las funciones más críticas.

Para ello os proponen acelerar las funciones más críticas del código: `sudoku_recalcular()` y su hija `sudoku_candidatos()`. También se plantean incluir una pequeñísima memoria muy rápida en la que guardar estas funciones, por lo que es importante conseguir que ocupen lo mínimo posible.

Por tanto tenéis que:

- a) **Estudiar el código en inicial en C y la especificación de las funciones `sudoku_recalcular_c()` y `sudoku_candidatos_c()`.** En el código inicial en C os encontrareis con la definición de las funciones y el pseudocódigo del comportamiento de ambas funciones.
  - Debéis **implementar ambas en C verificando** sobre la cuadrícula en memoria **la correcta ejecución**, comprobando a mano que el cálculo de los candidatos de todo el tablero ha sido realizado satisfactoriamente.
- b) **Realizar `sudoku_candidatos_arm()` en ensamblador ARM** para tratar de mejorar el rendimiento y el tamaño del código. Realiza un programa que llame a ésta nueva función desde C y **verifique** automáticamente la correcta ejecución.
- c) **Realizar `sudoku_recalcular_arm()` en ensamblador ARM.** Modificar el código para poder ejecutar las cuatro posibles combinaciones (C-C, C-ARM, ARM - C y ARM - ARM) verificando automáticamente el resultado.
- d) Como también estamos interesados en reducir el tamaño en bytes de la zona crítica, tenéis que hacer **`sudoku_candidatos_thumb()` en ensamblador Thumb**. Esta nueva función debe ser llamada desde C y desde ARM. Debéis comparar el **tamaño del código** de cada versión de `sudoku_candidatos` (C original, ARM y Thumb) y el **número de instrucciones ejecutadas** para por ejemplo realizar el cálculo de candidatos de la fila 1 - columna 1 (R1C1) (Más adelante en el guion os mostramos una forma fácil de realizarlo).

En total tenemos 6 configuraciones distintas (2 para recalcular por 3 de candidatos). Vuestro código debe recalcular el tablero inicial para cada una de las 6 combinaciones y **verificar de forma automática que todas las configuraciones generan la misma salida**.

Queremos evaluar el tiempo de ejecución de cada una de estas configuraciones pero aún no tenemos el sistema sobre el que finalmente se ejecutará y no disponemos de una manera fina de medir tiempo. La ejecución se está realizando sobre un simulador en un ordenador con instrucciones Intel x86 mientras realiza otras múltiples tareas. Pero podemos hacernos una idea si medimos manualmente el tiempo de ejecución. **Calcular el tiempo de ejecución de las 6 combinaciones y razonar los resultados obtenidos.** (Más adelante en el guion os mostramos un truco para medir los tiempos).

A la hora de evaluar este trabajo se valorará especialmente que cada estudiante haya sido capaz de optimizar al máximo tanto el código ARM, como el código Thumb. **Cuanto más pequeño sea el código y menos instrucciones se ejecuten, mejor será la valoración de la práctica.** También se valorará que se reduzcan los accesos a memoria.

#### REQUISITOS ADICIONALES

Los códigos ARM y Thumb deben tener la misma estructura que el código C: cada función, cada variable o cada condición que exista en el código C debe poder identificarse con facilidad en la versión de ensamblador. Para ello, los códigos en ARM y Thumb deberán incluir los comentarios oportunos. Si se quiere hacer una versión distinta, se puede hacer como apartado opcional.

**IMPORTANTE: DEBE MANTENERSE LA MODULARIDAD DEL CÓDIGO.** Es decir no se puede eliminar la llamada a la función `sudoku_candidatos()` e integrarla dentro de la función `sudoku_recalcular()`. No se puede eliminar la llamada a ninguna función, o eliminar el paso de parámetros. Y no se puede acceder a las variables locales de una función desde otra. Esto último es un fallo muy grave que implica un suspenso en la práctica.

Cuando hagáis la llamada en ensamblador debe ser de verdad una llamada a una función, por tanto, hay que pasar los parámetros, utilizando el estándar AATPCS (*ARM Application Procedure Call Standard*), a través de los registros correspondientes. No sirve hacer un salto sin pasar parámetros.

Para las llamadas a ARM se debe crear un marco de pila (o bloque de activación) tal y como se explica en las transparencias de la asignatura. Cuando hagáis la llamada en Thumb, podéis reducir el tamaño del bloque de activación por ejemplo eliminando PC y FP si lo deseáis y / o podéis usar `r7` como *frame pointer* para que su uso sea más sencillo.

Además en las **funciones hoja** tanto de ARM como Thumb podéis reducir el bloque de activación al máximo tal y como hemos hecho en el ejemplo del apartado A.

En todos los casos se debe garantizar que una función no altere el contenido de los registros privados de las otras funciones.

#### ALGUNOS CONSEJOS DE PROGRAMACIÓN EFICIENTE PARA OPTIMIZAR EL CÓDIGO

- No comencéis a escribir el código en ensamblador hasta tener claro cómo va a funcionar. Si diseñáis bien el código a priori, el número de instrucciones será mucho menor que si lo vais escribiendo sobre la marcha.

- Optimizar el uso de los registros. Las instrucciones del ARM trabajan principalmente con registros. Para operar con un dato de memoria debemos cargarlo en un registro, operar y por último, y sólo si es necesario, volverlo a guardar en memoria. Mantener en los registros algunas variables que se están utilizando frecuentemente permite ahorrar varias instrucciones de lectura y escritura en memoria. Cuando comencéis a pasar del código C original a ensamblador debéis decidir qué variables se van a guardar en registros, tratando de minimizar las transferencias de datos con memoria.
- Utilizar instrucciones de transferencia de datos múltiples como LDMIA, STMIA, push o pop que permiten que una única instrucción mueva varios datos entre la memoria y los registros.
- Utilizar instrucciones con ejecución condicional, también llamadas instrucciones predicadas. En el repertorio ARM gran parte de las instrucciones pueden predicarse. Por ejemplo el siguiente código:

```
if (a == 2) { b++ }  
  
else { b = b - 2 }
```

Con instrucciones predicadas sería:

```
CMP    r0, #2           #compara con 2  
  
ADDEQ  r1,r1,#1         #suma si r0 es 2  
  
SUBNE  r1,r1,#2         #resta si r0 no es 2
```

Mientras que sin predicados sería:

```
CMP    r0, #2           #compara con 2  
  
BNE    resta           # si r0 no es 2 saltamos a la resta  
  
ADD    r1,r1,#1         #suma 1  
  
B      cont            #continuamos la ejecución sin restar  
  
Resta: SUBNE           r1,r1,#2     #resta 2  
  
Cont :
```

Hay otros ejemplos útiles en las transparencias de la práctica.

- Utilizar instrucciones que realicen más de una operación. Por ejemplo la instrucción MLA r2,r3,r4,r5 realiza la siguiente operación:  $r2 = r3 * r4 + r5$ .
- Utilizar las opciones de desplazamiento para multiplicar. Las operaciones de multiplicación son más lentas (introducen varios ciclos de retardo). Para multiplicar/dividir por una potencia de dos basta con realizar un desplazamiento que además puede ir integrado en otra instrucción. Por ejemplo:

o  $A = B + 2 * C$  puede hacerse sencillamente con `ADD R1, R2, R3, lsl #1`

- $A = B + 10 * C$  puede hacerse sencillamente con `ADD R1, R2, R3, lsl #3` y `ADD R1, R1, R3, lsl #1` ( $A = B + 8 * C + 2 * C$ )
- Sacar partido de los modos de direccionamiento registro base + offset en los cálculos de la dirección en las instrucciones load/store. Por ejemplo, para acceder a `A[4]` podemos hacer `LDR R1, [R2, #4]` (si es un array de elementos de tipo char) o `LDR R1, [R2, #16]` (si es un array de enteros).

**NOTA:** como el código a desarrollar es pequeño es probable que alguna de estas optimizaciones no sean aplicables a vuestro código, pero muchas sí que lo serán y os animamos a utilizarlas.

## ¿CÓMO FUNCIONA EL CÓDIGO QUE NOS HAN DADO?

La información del tablero esta guardada en una estructura de datos denominada *cuadrícula*, definida en `init_b.asm`, consistente en una matriz de celdas. Los datos necesarios para la celda están codificados en 16 bits. Los 4 bits de mayor peso contienen el valor actual (0...9). El siguiente bit indica si se trata de una pista inicial (1) o una celda inicialmente vacía (0). Los dos siguientes bits están libres. Más adelante el de mayor peso se utilizará para marcar si se ha detectado un error en esa celda. Los 9 bits de menor peso se reservan para ir almacenando el mapa de bits con los posibles candidatos calculados.

1. Mira en tú código dónde está el *array cuadrícula* (puedes ir ejecutando paso a paso y mirando los valores de los registros, por ejemplo) y pon un monitor de memoria en esa dirección, te da varias opciones para representar los datos de la memoria elige **hex integer**. Pula el botón derecho sobre el monitor y elige **format** (Figura 1). Para simplificar la visualización en el entorno, *cuadrícula* se ha definido de 9 filas por 16 columnas, las 9 primeras contienen valores validos de celdas, el resto se han añadido para alinear el tablero. Configúralo tal y como aparece en la Figura 2. Se deben ver 16 columnas. Si la dirección asignada por el enlazador no es múltiplo de 32 el tablero no se verá bien. Se puede solucionar añadiendo datos antes del tablero (que está definido en `init_b`), o con una directiva de alineamiento (`.align`). De esta forma el tablero es visible para el jugador.

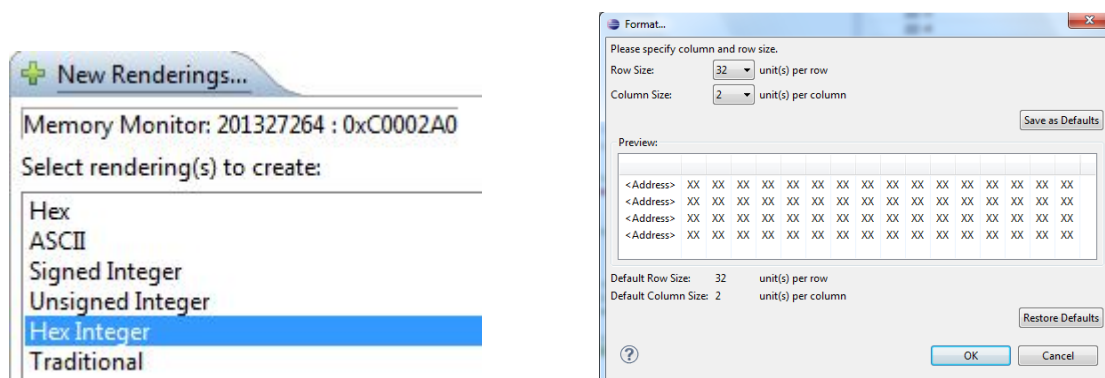



Figura 1: configurando el tablero en Eclipse



201327264 <Hex> 201327264 : 0xC0002A0 <Hex Integer>  New Renderings...

Address	0	2	4	6	8	A	C	E	10	12	14	16	18	1A	1C	1E
0C0002A0	9800	6800	0000	0000	0000	0000	7800	0000	8800	0000	0000	0000	0000	0000	0000	0000
0C0002C0	8800	0000	0000	0000	4800	3800	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
0C0002E0	1800	0000	0000	5800	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
0C000300	0000	0000	0000	0000	0000	0000	1800	7800	6800	0000	0000	0000	0000	0000	0000	0000
0C000320	2800	0000	0000	0000	9800	3800	0000	0000	5800	0000	0000	0000	0000	0000	0000	0000
0C000340	7800	0000	8800	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
0C000360	0000	0000	7800	0000	3800	2800	0000	4800	0000	0000	0000	0000	0000	0000	0000	0000
0C000380	3800	8800	2800	1800	0000	5800	6800	0000	0000	0000	0000	0000	0000	0000	0000	0000
0C0003A0	0000	4800	1800	0000	0000	9800	5800	2800	0000	0000	0000	0000	0000	0000	0000	0000

Figura 2: tablero inicial tal y como debe verse

2. Contar el número de instrucciones puede resultar algo tedioso. Utilizando comandos del depurador podemos simplificarlo. En la pestaña de la **consola de depuración** (normalmente abajo a la izquierda) nos muestra información sobre la ejecución del depurador y también nos permite introducir comandos, por ejemplo, si tecleamos el comando `stepi` (*step in*) se realizará un paso de ejecución.

Una forma de contar las instrucciones es parar la ejecución antes de la primera instrucción a contar y anotar el PC de la última que queremos contar. A continuación crearemos e inicializaremos una variable contador introduciendo el siguiente comando:

```
set $count=0
```

Ya estamos preparados para ir ejecutando el código hasta el PC de la última instrucción contándolas. Realizaremos un pequeño bucle introduciendo el siguiente comando:

```
while ($pc != 0xpc_ultima)
```

Nos saldrá como un pequeño cursor sobre el que iremos introduciendo las órdenes que queremos dentro del bucle, en nuestro caso, ejecutar paso a paso e ir incrementando el contador:

```
stepi
set $count=1+$count
end
```

Según el número de instrucciones el simulador tardara más o menos en procesar el bucle, puede tardar varios segundos. Cuando acabe solo nos quedará ver cuánto vale el contador.

```
print $count
```

3. La ejecución completa del cálculo del tablero en cualquiera de las seis combinaciones es relativamente rápida. Si queremos **medir manualmente el tiempo** de la ejecución no nos queda otro remedio que ejecutarla múltiples veces seguidas y ver cuanto tarda. Por ejemplo, podemos ponerla en un bucle para que se ejecute mil veces, si medimos el tiempo de ejecución del bucle solamente tendremos que dividir por 1000 el tiempo total para tener el de una ejecución. Aun así, como estamos sobre un sistema donde se están ejecutando múltiples cosas simultáneamente que no controlamos, conviene realizar esta medida varias veces para ver que es consistente.

#### APARTADO OPCIONAL 1:

Estudiar el impacto en el rendimiento del código C de las opciones compilación con optimización (-O1, -O2...) y **después estudiar el ensamblador que genera el compilador y explicar qué cambia en cada versión.**

#### APARTADO OPCIONAL 2:

Hemos optimizado las funciones en ARM y THUMB. ¿Eres capaz de optimizar las funciones en C?

#### EVALUACIÓN DE LA PRÁCTICA

La práctica se presentará **aproximadamente** el 22 de Octubre.

La memoria habrá que presentarla **aproximadamente** el 27 de Octubre.

**Las fechas definitivas se publicarán en la página web de la asignatura (moodle).**

#### ANEXO 1: REALIZACIÓN DE LA MEMORIA

La memoria de la práctica tiene diseño libre, pero es obligatorio que incluya los siguientes puntos:

1. Resumen ejecutivo (una cara como máximo). El resumen ejecutivo es un documento independiente del resto de la memoria que describe brevemente qué habéis hecho, por qué lo habéis hecho, qué resultados obtenéis y cuáles son vuestras conclusiones.
2. Código fuente del apartado B comentado. Además cada función debe incluir una cabecera en la que se explique cómo funciona, qué parámetros recibe y dónde los recibe y para qué usa cada registro (ej. En el registro 4 se guarda el puntero a la primera matriz...).
3. Descripción de las optimizaciones realizadas al código ensamblador.
4. Resultados de la comparación entre las distintas versiones de las funciones.
5. Descripción de los problemas encontrados en la realización de la práctica y sus soluciones.
6. Conclusiones

Se valorará que el texto sea **claro y conciso**. Cuánto más fácil sea entender el funcionamiento del código y vuestro trabajo, mejor. Revisa el documento de recomendaciones para la redacción de una memoria técnica disponible en Moodle.

#### ANEXO 2: ENTREGA DE LA MEMORIA

La entrega de la memoria será a través de la página web de la asignatura (*moodle* en <http://add.unizar.es>). Debéis enviar un fichero comprimido en formato ZIP con los siguientes documentos:

1. Memoria en formato PDF
2. Código fuente de los apartados A y B en formato texto

Se debe mandar un único fichero por pareja. El fichero se nombrará de la siguiente manera:

p1\_NIP-Apellidos\_Estudiante1\_NIP-Apellidos\_Estudiante2.zip

Por ejemplo: p1\_345456-Gracia\_Esteban\_45632-Arribas\_Murillo.zip