ESCUELA POLITÉCNICA SUPERIOR.

Grado en Ingeniería Electrónica y doble grado Ingeniería Eléctrica y Electrónica.

Informática. Curso 2014-2015.

PRACTICA DE CURSO. SIMULADOR DE CPU

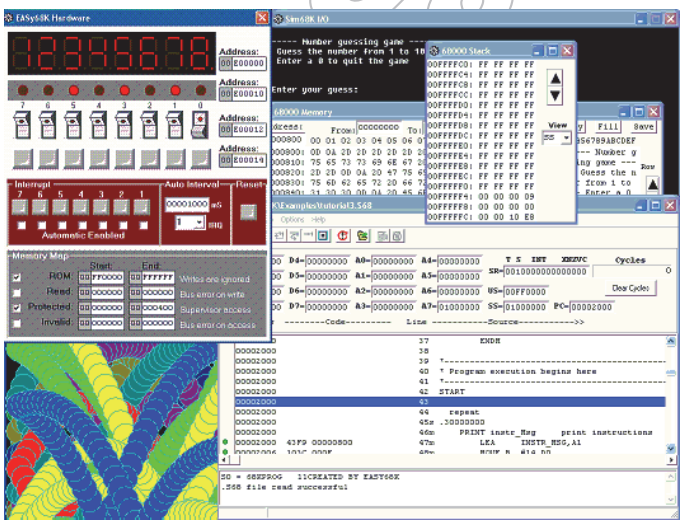
INDICE

Introducción....1
Recursos....3
Definición de la CPU a simular....3
Programa ejemplo....6
Funcionamiento del programa simulador....6
Formato del fichero de entrada al simulador....7
Control de excepciones de la CPU....8
Probando el ejemplo en el simulador gráfico....8
Ejercicios preliminares....10
Ejercicios con sentencias condicionales....11
Campos de una instrucción....11
Decodificación de instrucciones usando C....12
Ejercicios con bucles....15
Ejercicios con vectores....16
Ejercicios con ficheros....18

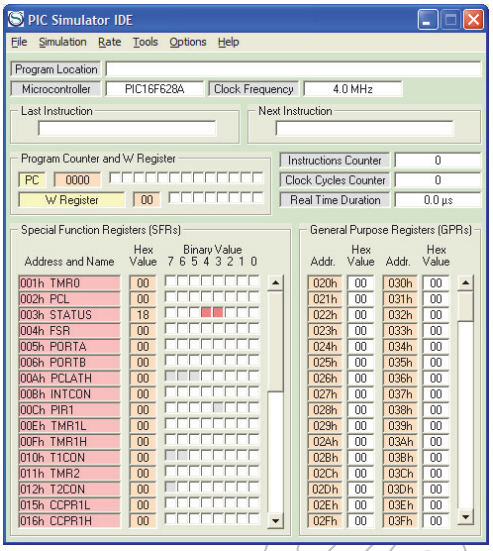
Introducción

Este boletín contiene la descripción completa de la práctica de curso de este año: un simulador de CPU.

Un simulador de CPU es un programa que actúa interpretando las instrucciones de una CPU y ejecutándolas en un entorno virtual. El sistema que se emula puede existir realmente, o bien ser un entorno completamente "inventado". El simulador es una herramienta de uso común cuando se realizan prototipos de sistemas basados en microcontroladores, ya que por lo general es más sencillo depurar el programa de control en un simulador del microcontrolador, y luego, una vez libre de errores, grabar dicho programa en el microcontrolador.



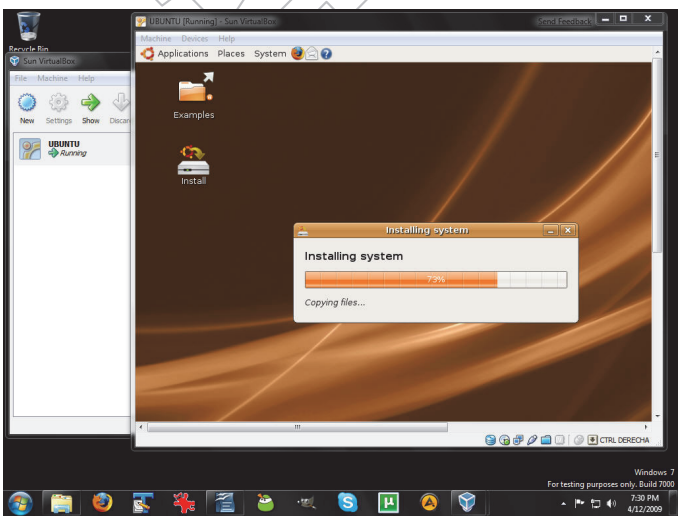
Entorno de programación Easy68K con ensamblador y simulador del procesador Motorola 68000



Simulador de PIC "PIC Simulator IDE".

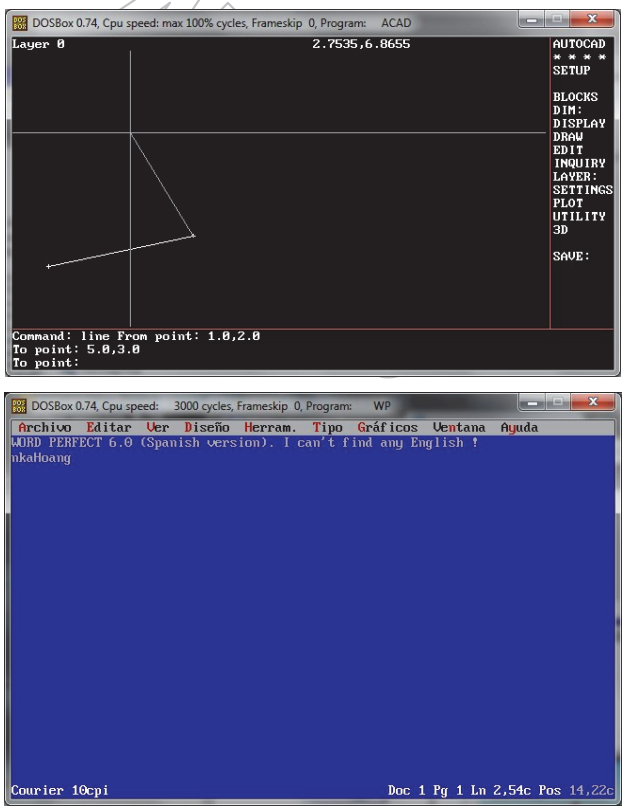
En la industria informática, la simulación va un paso más allá, y existen aplicaciones que pueden emular a un computador completo. Esto se consigue gracias a una técnica denominada "virtualización", y permite, por ejemplo, ejecutar un PC completo con Windows dentro de un ordenador Mac. También permite tener un único servidor, es decir, una única máquina física, ejecutando varias instancias de PC's con Linux o Windows. Dicho de otra forma: en el mismo espacio y con los mismos recursos con los que se mantiene un único servidor, pueden existir, de forma virtualizada, múltiples servidores.

VMWare Fusion ejecutando un PC virtualizado con Windows 7 dentro de un ordenador Mac con OS X como sistema anfitrión.



VirtualBox virtualizando Ubuntu Linux dentro de un PC que ejecuta Windows 7 como sistema anfitrión.

En el terreno empresarial y doméstico, los emuladores se utilizan para ejecutar aplicaciones que originalmente se escribieron para versiones obsoletas de un ordenador, o sistema operativo, y que deben seguir funcionando aunque los técnicos actualicen los ordenadores o el sistema operativo. Esto ocurre cuando la empresa que desarrolló la aplicación ya no da soporte para nuevas versiones del sistema operativo, o bien resulta que la propia empresa ya no existe. En el hogar, los emuladores se utilizan sobre todo en el campo del entretenimiento, al permitir ejecutar en un ordenador videojuegos que originalmente existían como circuitos electrónicos discretos, o grandes placas con memoria y microprocesador, pero que aún siguen siendo perfectamente válidos para jugar y divertirse.



DosBOX (emulador de MS DOS) en un PC con Windows XP ejecutando AutoCAD 2.5 (arriba) y Word Perfect 6.0 (abajo)

MAME (Multi-Arcade Machine Emulator) emulando en un PC con Windows todo el sistema de microprocesador, memoria, etc, que se observa en la placa de circuito impreso en la foto superior. El juego es Street Fighter II.

Recursos

Para ayudaros en la realización de esta práctica se dispone del siguiente material, disponible salvo que se diga otra cosa, en Enseñanza Virtual.

- Este boletín.
- Libro de teoría de la asignatura (Fundamentos de Informática para la Ingeniería Industrial) disponible en versión impresa en la biblioteca de la E.P.S.
- Todos los ejercicios y plantillas descritos en este boletín, resueltos (en formato ejecutable) para comprobar vuestras soluciones. Disponible para su descarga como fichero "*ejercicios_simulador.zip*". Cuando en un ejercicio hable de descargar un fichero o plantilla, aquí estará.
- Simulador gráfico de la Universidad de Oviedo. Es una versión avanzada del simulador que nosotros mismos haremos. Su archivo de ayuda contiene una descripción de la CPU que vamos a simular.
- Programa ensamblador y simulador (éste último, sólo ejecutable), para hacer vuestros propios programas, o ensamblar los que se os den ya escritos, para probar en vuestro simulador y en el que se os entrega, a fin de comprobar su correcto funcionamiento.

Definición de la CPU a simular.

La aplicación consiste en un simulador para una CPU sencilla que consta de 8 registros generales de 16 bits, denominados R0, R1, R2, ..., hasta R7; una serie de banderas binarias de condición, y un registro contador de programa PC, de tamaño suficiente para direccionar la memoria de programa disponible. La CPU es la misma que se describe en el Simulador de CPU de la Universidad de Oviedo, disponible en descarga en la web. También es la misma CPU que se describe al principio del tema 2 de teoría.

Esta CPU usa un sistema de memoria unificado para programa y datos (arquitectura Von Neumann). La memoria se implementará como un vector de elementos de 16 bits sin signo. El tamaño del vector será la cantidad de memoria disponible, y se podrá ajustar en tiempo de compilación asignando el valor adecuado a la constante TMEMORIA.

Se define el siguiente juego de instrucciones en forma tabulada:

- La primera columna indica el nmemotécnico de la instrucción. Esta es una abreviatura de la acción que realiza la instrucción, tal como MOV por "move", SUB por "substract", JMP por "jump", etc.
- El formato de la instrucción es la secuencia, escrita en formato binario, de esa instrucción. Es decir, describe qué campos tiene esa instrucción y qué hay en cada campo. En la práctica 3 se empezará a usar esto.
- La descripción indica qué hace la instrucción.
- La columna "Excepciones" indica si esta instrucción puede generar alguna excepción. Una excepción cualquier suceso no contemplado en el flujo normal de un programa, y habitualmente significa que algo está mal escrito o mal codificado en ese programa.
- La columna "Ejemplo" muestra un ejemplo de uso de esa instrucción, detallando los contenidos de los registros antes y después de ejecutar la instrucción. Cuando así sea necesario, también se detallará el contenido de la memoria y las banderas de condición.

Nemotécnico	*Formato de la instrucción (binario)*	*Descripción*	*Excepciones*	*Ejemplo de uso*
Instrucciones de movimiento (código de clase de instrucción 00) Todas estas instrucciones están encaminadas a dar valores a los registros, bien leyendo el valor desde memoria, desde otro registro, o desde la propia instrucción. También hay instrucciones que escriben datos de un registro a memoria. Un caso especial en esta clase de instrucción es la instrucción STOP, ya que no es una instrucción de movimiento propiamente dicha.				
MOV Rd,Rs	00 000 Rd Rs 00000	Copia el valor que esté almacenado en el registro Rs, al registro Rd.	No tiene.	MOV R3,R5 Binario: 00 000 011 101 00000b Hexadecimal: 03A0h Antes de esta instrucción: R3 = 1C40h R5 = 9B23h. Después de esta instrucción: R3 = 9B23h
MOV Rd,[Ri]	00 001 Rd Ri 00000	Lee un valor de 16 bits de la memoria, en la dirección indicada por el valor del registro Ri, y lo guarda en el registro Rd.	Excepción de memoria, si la dirección desde la que se quiere leer está fuera de rango. En tal caso, el registro Rd no se altera.	MOV R7,[R1] Binario: 00 001 111 001 00000b Hexadecimal: 0F20h Antes de esta instrucción: R1 = 0300h R7 = 1234h La posición 0300h de memoria contiene el valor 8D01h. Después de esta instrucción: R7 = 8D01h

Nemotécnico	*Formato de la instrucción (binario)*	*Descripción*	*Excepciones*	*Ejemplo de uso*
MOV [Ri],Rs	00 010 Ri Rs 00000	Escribe el valor de 16 bits que está en el registro Rs, a la dirección de memoria indicada por el valor del registro Ri	Excepción de memoria, si la dirección a la que se pretende escribir está fuera de rango.	MOV [R4],R0 Binario: 00 010 100 000 00000b Hexadecimal: 1400h Antes de esta instrucción: R0 = FF12h R4 = 12ABh La posición 12ABh de memoria contiene el valor 2855h. Después de esta instrucción: La posición 12ABh de memoria contiene el valor FF12h
MOVL Rd,inm8	00 100 Rd inm8	Guarda en el byte menos significativo del registro Rd el valor inm8.	No tiene.	MOVL R5,5Ah Binario: 00 100 101 01011010b Hexadecimal: 255Ah Antes de esta instrucción: R5 = 1812h Después de esta instrucción: R5 = 185Ah
MOVH Rd,inm8	00 101 Rd inm8	Guarda en el byte más significativo del registro Rd el valor inm8	No tiene.	MOVH R0,EFh Binario: 00 101 000 11101111b Hexadecimal: 28EFh Antes de esta instrucción: R0 = 6B03h Después de esta instrucción: R0 = EF03h
STOP	00 111 00000000000	Para la ejecución del programa. Se termina la simulación.	No tiene.	STOP Binario: 00 111 00000000000 Hexadecimal: 3800h
Instrucciones lógico-aritméticas (código de clase de instrucción 01). Todas estas instrucciones actualizan las banderas de condición después de realizar la operación. - Si el resultado de la operación es 0, ZF vale 1, en otro caso, 0. - El valor de SF será el valor del bit más significativo del resultado. - Si al realizar la operación hubo desbordamiento (el resultado no cabe en un entero de 16 bits en complemento a 2) entonces OF vale 1, en otro caso, vale 0. - Si al realizar la operación hubo acarreo (el resultado no cabe en un entero de 16 bits en formato de binario natural) entonces CF vale 1, en otro caso, vale 0.				
ADD Rd,Rs1,Rs2	01 00000 Rd Rs1 Rs2	Suma el contenido del registro Rs1 con el de Rs2, y el resultado lo guarda en el registro Rd.	No tiene.	ADD R2,R6,R7 Binario: 01 00000 010 110 111b Hexadecimal: 40B7h Antes de esta instrucción: R2 = 883Eh R6 = 12FFh R7 = A003h Después de esta instrucción: R2 = B302h ZF=0, SF=1, CF=0, OF=0
SUB Rd,Rs1,Rs2	01 00001 Rd Rs1 Rs2	Resta el contenido del registro Rs1 menos el de Rs2, y el resultado lo guarda en el registro Rd.	No tiene.	SUB R0,R1,R4 Binario: 01 00001 000 001 100b Hexadecimal: 420Ch Antes de esta instrucción: R0 = 0A1Ch, R1 = 5982h, R4 =9BB0h Después de esta instrucción: R0 = BDD2h ZF=0, SF=1, CF=0, OF=0
OR Rd,Rs1,Rs2	01 00010 Rd Rs1 Rs2	Realiza la operación OR (suma lógica, en C es el operador |) entre el contenido del registro Rs1 y Rs2. El resultado se guarda en el registro Rd.	No tiene.	OR R2,R3,R4 Binario: 01 00010 010 011 100b Hexadecimal: 449Ch Antes de esta instrucción: R2 = 9B10h R3=E004h R4=38C8h Después de esta instrucción: R2=F8CCh ZF=0, SF=1, CF=0, OF=0
AND Rd,Rs1,Rs2	01 00011 Rd Rs1 Rs2	Realiza la operación AND (producto lógico, en C es el operador &) entre el contenido del registro Rs1 y Rs2. El resultado se guarda en el registro Rd.	No tiene.	AND R2,R3,R4 Binario: 01 00011 010 011 100b Hexadecimal: 469Ch Antes de esta instrucción: R2 = 9B10h R3=E004h R4=38C8h Después de esta instrucción: R2=2000h ZF=0, SF=0, CF=0, OF=0
XOR Rd,Rs1,Rs2	01 00100 Rd Rs1 Rs2	Realiza la operación XOR (o exclusivo lógico, en C es el operador ^) entre el contenido del registro Rs1 y Rs2. El resultado se guarda en el registro Rd.	No tiene.	XOR R2,R3,R4 Binario: 01 00100 010 011 100b Hexadecimal: 489Ch Antes de esta instrucción: R2 = 9B10h R3=E004h R4=38C8h Después de esta instrucción: R2=D8CCh ZF=0, SF=1, CF=0, OF=0

Nemotécnico	*Formato de la instrucción (binario)*	*Descripción*	*Excepciones*	*Ejemplo de uso*
COMP Rs1,Rs2	01 00111 Rs1 Rs2 000	Compara el contenido del registro Rs1 con el del registro Rs2. En realidad lo que hace es restar Rs1 menos Rs2, pero no guarda el resultado en ningún registro de destino, solamente actualiza las banderas de condición.	No tiene.	COMP R2,R7 Binario: 01 00111 010 111 000b Hexadecimal: 4EB8h Antes de esta instrucción: R2 = AA15h R7 = 79F0h Después de esta instrucción: ZF=0, SF=0, CF=1, OF=1
NOT Rds	01 01000 Rds 000000	Realiza la operación NOT (complemento a 1) del valor del registro Rds, y lo almacena en el mismo registro.	No tiene.	NOT R4 Binario: 01 01000 0100 000000b Hexadecimal: 5080h Antes de esta instrucción: R4 = 1234h Después de esta instrucción: R4 = EDCBh ZF=0, SF=1, CF=0, OF=0
INC Rds	01 01001 Rds 000000	Incrementa el valor del registro Rds	No tiene	INC R1 Binario: 01 01001 001 000000b Hexadecimal: 5240h Antes de esta instrucción: R1 = 90BFh Después de esta instrucción: R1 = 90C0h ZF=0, SF=1, CF=0, OF=0
DEC Rds	01 01010 Rds 000000	Decrementa el valor del registro Rds	No tiene	DEC R7 Binario: 01 01010 111 000000b Hexadecimal: 55C0h Antes de esta instrucción: R7 = 4098h Después de esta instrucción: R7 = 4097h ZF=0, SF=0, CF=0, OF=0
NEG Rds	01 01011 Rds 000000	Realiza el complemento a 2 del valor del registro Rds	No tiene	NEG R6 Binario: 01 01011 110 000000b Hexadecimal: 5780h Antes de esta instrucción: R6 = 80B2h (-32590 en decimal) Después de esta instrucción: R6 = 7F4Eh (32590 en decimal) ZF=0, SF=0, CF=0, OF=0
CLR Rds	01 01100 Rds 000000	Pone a 0 el registro Rds	No tiene	CLR R3 Binario: 01 01100 011 000000b Hexadecimal: 58C0h Antes de esta instrucción: R3 = F12Fh Después de esta instrucción: R3 = 0000h ZF=1, SF=0, CF=0, OF=0
Instrucciones de salto incondicional (código de clase de instrucción 10)				
JMP desplaz	10 desplaz	El valor de 14 bits *desplaz* es extendido con signo a 16 bits, y se le suma al valor actual del registro PC, obteniendo la dirección de la siguiente instrucción a ejecutar.	Excepción de memoria si la dirección destino de salto está fuera del rango de la memoria disponible.	JMP -23 Binario: 10 11111111101001b Hexadecimal: BFE9h Antes de esta instrucción: PC = 0120h Después de esta instrucción: PC = 0109h
Instrucciones de salto condicional (código de clase de instrucción 11)				
BR cond desplaz	11 cond desplaz	*cond* codifica la condición que ha de cumplirse: 000 C, Salto si CF=1 001 NC, Salto si CF=0 010 O, Salto si OF=1 011 NO, Salto si OF=0 100 Z, Salto si ZF=1 101 NZ, Salto si ZF=0 110 S, Salto si SF=1 111 NS, Salto si SF=0 Si la condición se cumple, el valor de 11 bits *desplaz* se extiende con signo a 16 bits, y se le suma al valor actual del registro PC, obteniendo la dirección de la siguiente instrucción a ejecutar. Si la condición no se cumple, no se modifica PC.	Excepción de memoria si la dirección destino de salto está fuera del rango de la memoria disponible.	BR S,+42 Binario: 11 110 00001000010b Hexadecimal: F042h Antes de esta instrucción: PC = 3200h SF=0 Después de esta instrucción: PC = 3200h (no se modifica PC, al no cumplirse la condición) BR NZ,+42 Binario: 11 101 00001000010b Hexadecimal: E842h Antes de esta instrucción: PC = A004h ZF=0 Después de esta instrucción: PC = A02Eh (se modifica PC, al cumplirse la condición)

Programa ejemplo

Para comprobar el correcto funcionamiento del simulador, puede emplear el programa de ejemplo mostrado en la siguiente tabla, que al ejecutarse, guarda en las posiciones de memoria 0000h a 000Fh las potencias de 2, desde 2^0 hasta 2^{15} inclusive.

El código de este programa está pensado para que comience a ejecutarse a partir de 0010h. En la siguiente tabla mostramos el código (sólo el código, no los datos) del programa, que comienza en 0010h. En las direcciones anteriores, 0000h a 000Fh estarán los datos que este programa escribe.

Las direcciones de memoria se dan habitualmente en hexadecimal, así que hemos empleado ese formato en este ejemplo.

Dirección	Nmemotécnico	Formato binario	Hex.	Comentario
0010	MOVH R0,00h	00 101 000 00000000	2800	Estas dos instrucciones cargan el registro R0 con el valor 0000h. Como sólo puede cargarse un valor inmediato a una mitad del registro, usamos dos instrucciones, una para cada mitad. R0 contendrá la dirección de memoria donde iremos escribiendo cada potencia.
0011	MOVL R0,00h	00 100 000 00000000	2000	
0012	MOVH R1,00h	00 101 001 00000000	2900	Usamos el mismo método para cargar el registro R1 con el valor 0001h. R1 es el valor de cada potencia. Irá cambiando a 2,4,8,16,… Al principio vale 1, que es 2^0.
0013	MOVL R1,01h	00 100 001 00000001	2101	
0014	MOVH R2,00h	00 101 010 00000000	2A00	Y el mismo método para que R2 valga 0010h (que es 16 en decimal). R2 es un contador que va decrementándose cada vez que se escriba una potencia. Vamos a escribir 16 resultados.
0015	MOVL R2,10h	00 100 010 00010000	2210	
0016	CLR R3	01 01100 011 000000	58C0	Ponemos a 0 el registro R3. Lo usaremos más tarde.
0017	MOV [R0],R1	00 010 000 001 00000	1020	Guardamos el valor actual del registro R1 en la dirección de memoria apuntada por R0
0018	ADD R1,R1,R1	01 00000 001 001 001	4049	Hacemos R1←R1+R1, o lo que es lo mismo, R1←R1*2, con lo que R1 contiene ahora la siguiente potencia de 2 a escribir.
0019	INC R0	01 01001 000 000000	5200	Incrementamos R0 para que apunte a la próxima dirección de memoria en donde hay que escribir.
001A	DEC R2	01 01010 010 000000	5480	Decrementamos R2 para indicar que ya hemos escrito una potencia.
001B	COMP R2,R3	01 00111 010 011 000	4E98	Comparamos R2 con cero (R3). R2 será cero cuando hayamos escrito las 16 potencias.
001C	BR NZ,-6	11 101 11111111010	EFFA	Si no es cero, saltamos 6 posiciones hacia atrás contando desde la posición de memoria siguiente a esta (001D). Es decir, saltamos a 001D-0006=0017h. Con esto volvemos hacia atrás al punto donde escribimos la siguiente potencia en memoria.
001D	STOP	00 111 00000000000	3800	Fin de la ejecución. Salimos del simulador.

Funcionamiento del programa simulador

La aplicación comenzará mostrando un mensaje de bienvenida con el título "SIMULADOR DE CPU. E.P.S. CURSO 2014-2015." y en la línea siguiente, el nombre completo del alumno.

A continuación, leerá desde disco un fichero llamado *memoria.txt* que contiene los contenidos iniciales de la memoria del procesador. Este fichero podrá estar en la misma carpeta que la aplicación, por lo que no hay que especificar la trayectoria completa del mismo. En estos contenidos se incluye el código del programa que ha de ejecutarse, y los datos iniciales si los hubiera, y las direcciones de memoria de carga y ejecución. Su formato se detalla en la siguiente sección.

El fichero debe existir, pero no es obligatorio que el contenido de dicho fichero cubra por completo la memoria del procesador. A este fin, las posiciones de memoria no usadas se rellenan con 0's. Más adelante en este documento se muestra el formato concreto que debe tener el fichero *memoria.txt* y un ejemplo. Si no encuentra este fichero, el simulador emitirá un mensaje de error en pantalla y terminará su ejecución.

Una vez leído el fichero y cargados sus datos a la memoria de la CPU, la simulación comienza estableciendo el valor de todos los registros a 0, e inicializando el registro PC con la dirección de ejecución leída desde el fichero de memoria. La simulación irá leyendo y ejecutando cada instrucción hasta ejecutar una instrucción STOP. Cuando esto ocurre, la simulación terminará, y se mostrará por pantalla un mensaje indicando en qué dirección de memoria de programa se ha detenido el programa, junto con un listado de todos los registros y sus valores actuales en formato hexadecimal. Por último se escribirá el fichero *resultados.txt* con el contenido de la memoria actual. Hecho esto, la aplicación terminará. El formato de este fichero será el mismo que el usado para *memoria.txt*. La dirección de carga y ejecución en *resultados.txt* serán las mismas que se leyeron inicialmente de *memoria.txt*.

Al ejecutar el simulador desde C-Free o desde la consola, usando el programa de ejemplo, aparecerá algo como esto en pantalla:

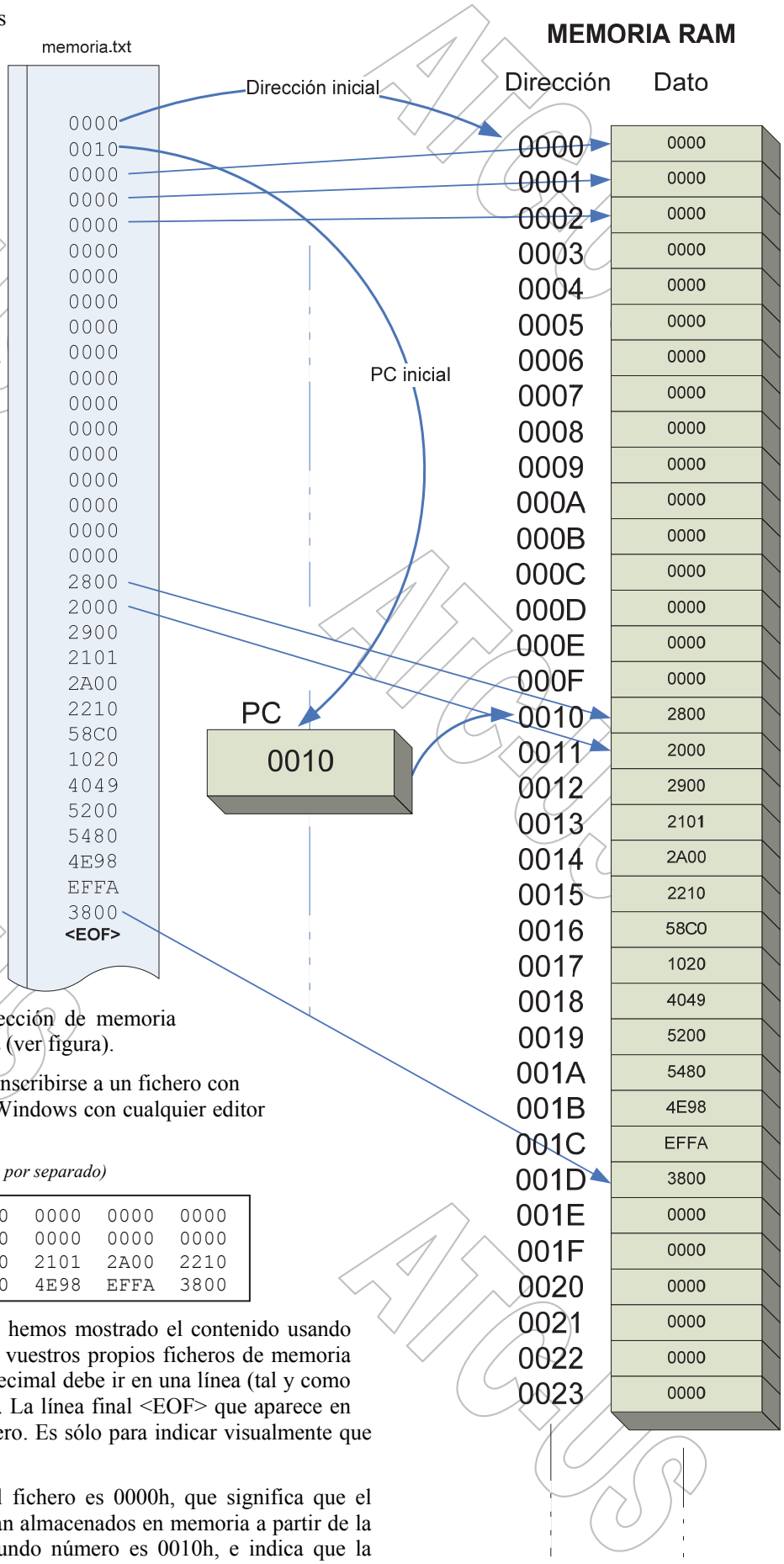
```
SIMULADOR DE CPU. E.P.S. CURSO 2014-2015
Miguel Angel Rodriguez Jodar.

Programa terminado con normalidad. Se encontro STOP en 001D.
R0 = 0010
R1 = 0000
R2 = 0000
R3 = 0000
R4 = 0000
R5 = 0000
R6 = 0000
R7 = 0000
ZF=1  SF=0  CF=0  OF=0
```

Formato del fichero de entrada al simulador

Es un fichero de texto con números de 16 bits escritos en formato hexadecimal. Cada número estará en una línea.

- El primer número indica la dirección de memoria donde se va a cargar la información del fichero. Puede tomar cualquier valor desde 0000h hasta TMEMORIA-1. En el simulador, supondremos que este número es una dirección de memoria válida, es decir, no obligamos a que vuestro simulador tenga que realizar este chequeo.
- El segundo número indica la dirección de memoria donde comenzará la ejecución del programa que se encuentra en el fichero. Puede tomar cualquier valor desde 0000h hasta TMEMORIA-1. Este segundo número no tiene por qué ser igual que el primero ya que un fichero de memoria puede comenzar definiendo un bloque de datos y después el código, como de hecho pasa con el programa de ejemplo. Este segundo número es precisamente el valor inicial que tendrá el registro PC (contador de programa).
- El tercer número y siguientes son los datos que se encuentran consecutivamente en la memoria, comenzando en la dirección de memoria especificada por el primer número (ver figura).



Así, el programa anterior puede transcribirse a un fichero con este contenido, que puede crearse en Windows con cualquier editor de textos, como el Bloc de Notas:

memoria.txt *(cada número en una línea de texto por separado)*

```
0000  0010  0000  0000  0000  0000  0000  0000
0000  0000  0000  0000  0000  0000  0000  0000
0000  0000  2800  2000  2900  2101  2A00  2210
58C0  1020  4049  5200  5480  4E98  EFFA  3800
```

Para ahorrar espacio en la página, hemos mostrado el contenido usando ocho números por línea, pero al crear vuestros propios ficheros de memoria para el simulador, cada número hexadecimal debe ir en una línea (tal y como se muestra en la figura de la derecha). La línea final <EOF> que aparece en la figura no debe escribirse en el fichero. Es sólo para indicar visualmente que ahí termina el fichero de texto.

El primero número que hay en el fichero es 0000h, que significa que el código y los datos del programa estarán almacenados en memoria a partir de la dirección 0000h de memoria. El segundo número es 0010h, e indica que la primera instrucción que ha de ejecutarse es la que esté en la dirección 0010h. Al

ser esta una máquina Von Neumann, los datos y el código comparten la memoria, así que el contenido de memoria.txt contiene tanto el código, como las posiciones de memoria donde irán los datos de salida. Los siguientes 16 valores son todos 0000h, y son los valores iniciales de las posiciones de memoria desde la 0000h hasta 000Fh. En la siguiente posición de memoria, en la 0010h se ubicará el valor 2800h, que es la versión en hexadecimal de la instrucción MOVH R0,00h tal y como se puede ver en la tabla del programa de ejemplo. El último número que aparece en el archivo es 3800h, que es el código de STOP, la última instrucción de nuestro programa.

A efectos de simplificar la lectura de este fichero, el simulador asumirá que el fichero, de existir, es correcto. Es decir, que se ajusta al formato descrito. Esto significa que una vez abierto el fichero siempre podremos asegurar que habrá al menos dos valores en él: el valor de la primera dirección de memoria, y el valor inicial del PC, y en ese orden.

Control de excepciones de la CPU

Si durante la ejecución del programa la CPU se encuentra con una excepción, la simulación terminará en ese mismo momento, mostrando por pantalla un mensaje describiendo el tipo de excepción que ha ocurrido y el código completo (la instrucción en formato hexadecimal) de la instrucción que lo ha producido. También se mostrarán en pantalla el contenido actual de todos los registros, pero no se creará el fichero *resultados.txt*.

En la siguiente captura se ha modificado el programa cambiando la instrucción de salto condicional que está en la dirección 001C, de EFFA a EBFF). El cambio afecta al valor de "*desplaz*" de esta instrucción. Al ejecutar de nuevo el programa con el simulador ocurre esto:

```
SIMULADOR DE CPU. E.P.S. CURSO 2014-2015
Miguel Angel Rodriguez Jodar.

La instruccion (EBFF) en 001C intento saltar a una direccion de memoria ilegal.
R0 = 0001
R1 = 0002
R2 = 000F
R3 = 0000
R4 = 0000
R5 = 0000
R6 = 0000
R7 = 0000
ZF=0  SF=0  CF=0  OF=0
```

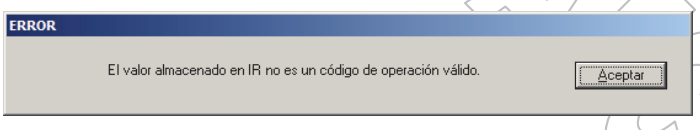
Las posibles excepciones que pueden ocurrir durante la ejecución de un programa, y que nuestro simulador debe detectar, son:

- Que se intente ejecutar una instrucción que no existe.
- Que se intente leer un valor desde una posición de memoria ilegal (fuera de rango).
- Que se intente escribir un valor a una posición de memoria ilegal (fuera de rango).
- Que se intente realizar un salto (condicional o incondicional) a una posición de memoria ilegal (fuera de rango).

Para las excepciones que tienen que ver con direcciones ilegales, recordar que la memoria que maneja nuestro simulador está limitada a TMEMORIA elementos, y TMEMORIA puede ser un número pequeño. En la captura de pantalla precedente, el valor de TMEMORIA que tiene el simulador estaba fijado en 1000 posiciones de memoria. ¿A qué dirección de memoria pretendía saltar la instrucción modificada?

Probando el ejemplo en el simulador gráfico

El Simulador de CPU de la Universidad de Oviedo es una versión gráfica del simulador objeto de esta práctica. Aquél permite ejecutar el programa paso a paso, o ciclo a ciclo. Nuestro simulador no requiere ejecución paso a paso, ni ciclo a ciclo, sino que ejecutará el programa que se le cargue de forma no interactiva, es decir, a la mayor velocidad posible. Por ello, nuestro simulador tiene una característica que no posee la versión gráfica: la instrucción STOP. Esta instrucción no existe en el simulador gráfico, y cuando la intente decodificar, mostrará esta ventana de error en pantalla:

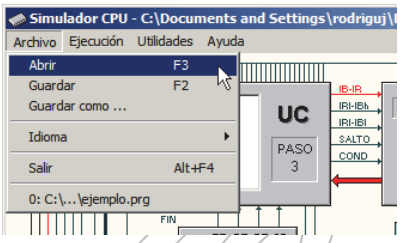


Es decir, para el simulador gráfico, la instrucción STOP es una instrucción inválida y por ello muestra una **excepción**. ¡OJO! es una excepción para el simulador gráfico, para nuestro simulador, STOP es una instrucción válida. El resto de instrucciones que no sean válidas en nuestro simulador tampoco lo serán en el simulador gráfico.

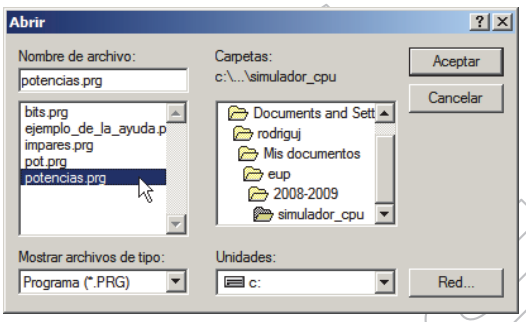
Para probar el ejemplo de la sección anterior, crea un fichero de texto con el Bloc de Notas conteniendo el programa de ejemplo: un número hexadecimal en cada línea. Grábalo en disco con un nombre sencillo, como por ejemplo *potencias.prg*. Este simulador espera que los ficheros de entrada tengan extensión PRG en lugar de TXT como usaremos nosotros.

Una vez tengas este fichero creado, procede de esta manera:

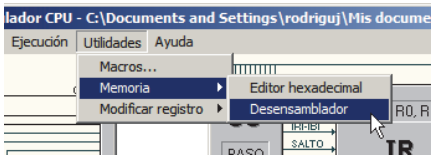
Abre el simulador gráfico y elige el menú Archivo – Abrir.



Busca la ubicación del fichero *potencias.prg* y especifica que quieres buscar archivos con la extensión PRG.

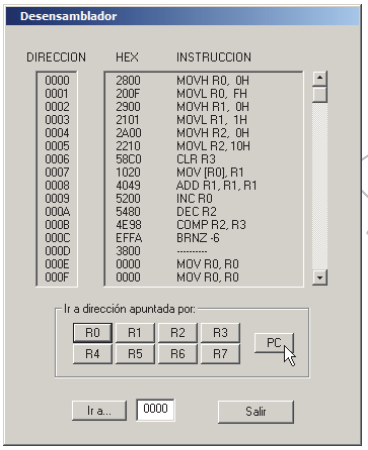


Pulsa en Aceptar para cargar el programa, y de nuevo en la ventana principal del programa, elige Utilidades – Desensamblador. El desensamblador es una utilidad que convierte el código del programa en nmemotécnicos.

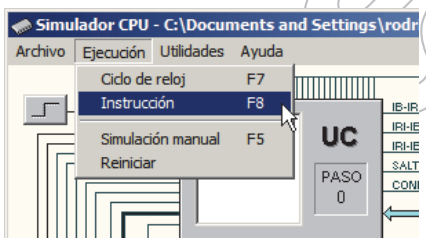


Esta es la ventana del desensamblador. Se puede elegir qué zona de memoria se quiere desensamblar. En nuestro caso, nos interesa ver el código que hay en la zona de memoria cuya dirección está en el registro PC, que es el registro que indica cuál es la dirección de la próxima instrucción a ejecutar. Nótese como en el listado del desensamblador, la instrucción cuyo código es 3800h no tiene traducción nmemotécnica. En el simulador de la práctica esta instrucción es STOP, pero en esta versión dicha instrucción no existe, por lo que no se puede desensamblar.

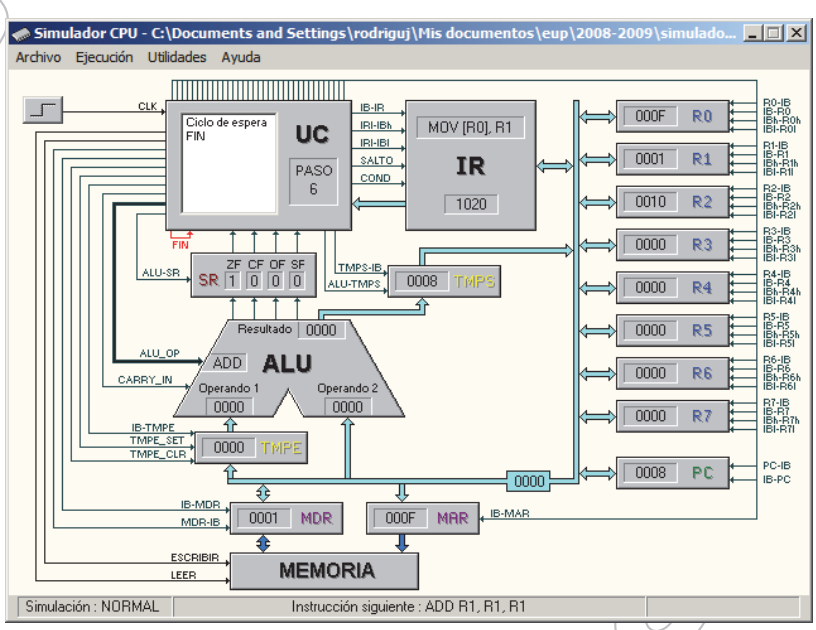
Existe una ventana similar, que es el Editor Hexadecimal. Permite mostrar, en hexadecimal, cualquier porción de la memoria de la CPU.



Pulsando en el botón Salir salimos del desensamblador. Para ejecutar el programa, en este simulador sólo puede hacerse paso a paso, bien por instrucciones (se ejecuta una instrucción cada vez que pulsemos la tecla F8) o bien por ciclos de reloj (se ejecuta un ciclo de reloj de la instrucción actual pulsando F7).



A efectos de esta práctica, nos interesa la ejecución por instrucciones. Pulsando F8 vemos como la Unidad de Control UC) decodifica y ejecuta cada instrucción que lee. El registro de instrucción IR contiene dicha instrucción, la cual podemos ver en formato hexadecimal y en formato nmemotécnico. En el gráfico se puede ver el momento en el que se está a punto de ejecutar la instrucción MOV [R0],R1, después de haber ejecutado (usando varias veces F8) todas las instrucciones anteriores MOVL,MOVH y CLR, así que los registros R0,R1,R2 y R3 contienen los valores con los que el programa los ha iniciado.



Ejercicios preliminares

Estos ejercicios están pensados para realizarse después de la explicación en clase de teoría de la organización de una CPU. No se necesita programar nada en C, pero sí usaremos el simulador gráfico de CPU disponible en Enseñanza Virtual.

EJERCICIO 1.

Haz de CPU y usando lápiz y papel para representar la memoria y los registros, ejecuta una a una las instrucciones que hay en el programa de ejemplo de este boletín. Llega al menos dos veces hasta la instrucción BR NZ (a esto se le llama "hacer la traza de un programa").

Este ejercicio, aun pareciendo el más pueril, es quizás el más importante para comprender cómo funciona esta CPU, y por ende, cómo debe simularse. Trabaja siempre que puedas en hexadecimal. Aunque no lo parezca, es más cómodo que trabajar en decimal, al menos para CPU y este ejemplo.

EJERCICIO 2.

Carga y realiza la simulación gráfica del programa de ejemplo tal como se ha descrito. Realiza la simulación hasta que se ejecute la instrucción STOP y el simulador gráfico muestre el error de instrucción no válida. Entonces, usa la ventana del Editor Hexadecimal para comprobar que el programa ha escrito los valores que se esperaban a partir de la posición 0000h de memoria. Recuerda que, si el programa se ha ejecutado correctamente, a partir de esa posición verás potencias de 2, es decir: 1, 2, 4, 8, 16, 32, 64, etc, pero como el editor hexadecimal muestra contenidos en hexadecimal, se verán como 0001, 0002, 0004, 0008, 0010, 0020, etc.

EJERCICIO 3.

En la tabla de instrucciones se especifica que la instrucción ***BR cond desplaz*** suma el valor *desplaz* al valor actual del registro PC. ¿Qué valor tiene el registro PC cuando se realiza esa suma? ¿Sigue apuntando el valor de PC a la instrucción que estamos ejecutando (el salto condicional) o ya no? Para averiguarlo, durante la simulación, al llegar a la instrucción BR NZ, ejecútala ciclo a ciclo (F7).

Ejercicios con sentencias condicionales.

A continuación se proponen una serie de ejercicios, que están destinados a sentar las bases del programa que implementa el Simulador de CPU, objeto de la práctica de este año.

Al principio, propondremos pequeños programas y funciones, que solucionan algunos de las cuestiones que se exponen en el enunciado de dicha práctica. Estos programas, quizás no estén relacionados entre sí, al menos al principio.

Poco a poco, iremos haciendo programas y funciones cada vez más grandes y complejos, re-usando código de los ejercicios que se han ido haciendo, hasta eventualmente, conseguir el objetivo final, de haber escrito un programa que simula el comportamiento de una CPU.

Para empezar, y dado que hemos hecho ejercicios en donde el programa hace una cosa u otra en función de cierta condición, vamos a aprovechar lo aprendido para resolver algunos pequeños problemas que se plantean en la práctica del simulador. Nos estamos refiriendo a discernir qué hacer una vez que tenemos metido en una variable, el código de la instrucción que se va a ejecutar. Dependiendo del valor que tenga ese código de operación, habrá que hacer unas cosas, u otras. Esta fase de la simulación se corresponde con la fase de decodificación de una instrucción.

Es recomendable guardar los resultados de cada sesión en un pendrive o dispositivo similar. Usaremos los resultados de estos pequeños ejercicios para escribir porciones de la práctica, más grandes.

Campos de una instrucción.

Lo primero que aprenderemos es a decodificar instrucciones de la CPU. Una instrucción es en esencia un valor de 16 bits que, por comodidad, expresaremos en hexadecimal (4 dígitos hexadecimales). Dentro de esos 16 bits está codificada toda la información que necesita la CPU para ejecutar la instrucción (tipo de instrucción, operandos fuente, y operandos destino): son los *campos* de la instrucción.

Decodificar una instrucción es extraer la información de todos los campos que están dentro de ella. Hay campos que existirán o tendrán un significado diferente en función del valor de otro campo.

Los diferentes campos de una instrucción son los siguientes:

- Código de clase de instrucción (existe en todas las instrucciones y siempre está situado en el mismo sitio).
- Código de instrucción (sólo si el código de clase define más de una instrucción).
- Operando destino (hay instrucciones en que este operando también es operando fuente).
- Operandos fuente (puede haber ninguno, uno o dos).
- Valor inmediato (sólo en las instrucciones en las que se defina este valor).

El código de clase de instrucción es el único código que debe existir en todas las instrucciones. Ocupa siempre los dos bits más significativos de la instrucción, y por tanto su valor puede ser 00, 01, 10 ó 11 (0,1,2 ó 3). En el enunciado de la práctica se ofrece más información sobre los códigos de clase de una instrucción.

El código de instrucción existe sólo si el código de clase agrupa más de una instrucción (en nuestra CPU esto sucede con los códigos de clase 00 y 01, ya que los códigos de clase 10 y 11 definen, cada uno, una única instrucción). El tamaño del código de instrucción depende de cuál sea el código de clase: para el código de clase 00, el código de instrucción ocupa tres bits; para el código de clase 01, ocupa cinco bits. El tamaño del código de instrucción indica cuántas instrucciones diferentes pueden definirse dentro de una clase de instrucción. Así, podemos ver que pueden existir hasta 8 instrucciones de clase 00, y 32 instrucciones de clase 01. En la CPU, no todas las combinaciones posibles del código de instrucción tienen por qué estar usadas. Puede haber códigos de instrucción que no identifiquen a ninguna instrucción conocida de la CPU. Por ejemplo, no existe una instrucción con código de clase 00 y código de instrucción 110. Si la CPU se encontrara con una instrucción así, tiene que informar de una *excepción*.

El código de tipo de instrucción, si existe, aparece en la instrucción ocupando los siguientes bits después del código de clase.

Por ejemplo, la instrucción 258Ah se puede expresar en binario como 0010 0101 1000 1010. Los dos bits más significativos son 00, así que ésta es una instrucción de movimiento (ver enunciado de la práctica).

Si es una instrucción de movimiento, el código de tipo de instrucción existe y ocupa los siguientes tres bits. En nuestro caso esos tres bits tienen el valor 100. Según la tabla de instrucciones de la práctica, es una instrucción MOVL Rd,Inm8.

Sabiendo en este punto qué instrucción es, podemos decodificar los operandos que necesita: en este caso, esta instrucción necesita un operando que es un registro destino (Rd) cuyo código ocupa tres bits, y un valor inmediato de 8 bits, que ocupa como es de esperar, ocho bits. En la tabla de instrucciones de la práctica vemos dónde están posicionados cada uno de estos campos. Para este ejemplo, el valor de Rd será 101 (5) y el valor inmediato de 8 bits, 1000 1010 (8Ah).

Así, la instrucción decodificada es: MOVL R5,8Ah

Decodificación de instrucciones usando C.

Para extraer cada uno de los campos usaremos los operadores de C que proveen operaciones lógicas. Concretamente usaremos el operador AND, y además, un operador existente en C para realizar operaciones de desplazamiento de bits.

El operandor AND lógico en C se denota con el símbolo &. Es un operador binario, pero a diferencia del operador &&, que opera con valores booleanos, éste opera bit a bit. Es decir, que por ejemplo, la expresión 6 & 3 en C realiza el AND binario entre el número 6 (110) y el número 3 (011), así:

$6 \,\&\, 3 = 110_{(2} \,\&\, 011_{(2} = 2$

110

011 &

$\overline{010} = 2$

El operador de desplazamiento de bits se denota en C con el símbolo >> si el desplazamiento es a la derecha, y << si es a la izquierda. Es un operador binario. En C la expresión a<<b significa "desplaza el valor *a* hacia la izquierda *b* bits.

Por ejemplo, si *a,b* son variables de 16 bits (short int), y *a* tiene el valor 0C82h, ó 0000 1100 1000 0010, al hacer *b=a<<3*, obtenemos en *b* el siguiente valor (expresado en binario): 0110 0100 0001 0000.

Este valor viene de haber desplazado tres lugares a la izquierda todos los bits que forman el número. Al desplazar los tres bits más significativos del número original, se pierden (se "caen" del número). Por la derecha entran bits nuevos, que son siempre 0. El nuevo valor, expresado en hexadecimal es 6410h.

EJERCICIO 4

Escribe un programa que lea un número entero por teclado, lo desplace 1 vez a la izquierda y guarde el resultado de dicho desplazamiento a una variable. Imprime el valor original y desplazado, en pantalla. Modifica el programa para que el desplazamiento sea, por ejemplo de 2 lugares a la izquierda, y luego vuelve a modificarlo para que sean 3 lugares. En cada caso, anota los valores antes y después del desplazamiento. A la vista de los resultados, contesta: Cuando desplazamos una variable N posiciones a la izquierda, ¿qué estamos haciendo en realidad? NOTA: cuando ejecutes el programa y pida un valor por teclado, usa valores pequeños y positivos, para ver mejor qué está pasando.

El desplazamiento a la derecha es similar. Al desplazar a la derecha, los bits menos significativos se pierden. Los bits que entran por la izquierda no obstante, pueden ser 0 ó 1:

- Si la variable en la que se realiza el desplazamiento tiene signo (char, short int, o int), entonces el bit que entra por la izquierda es el mismo que el bit de signo del valor original de la variable.
- Si la variable en la que se realiza el desplazamiento no tiene signo (unsigned char, unsigned short int, o unsigned int), entonces el bit que entra por la izquierda es un 0.

Ejemplo: una variable *a* de tipo *short int* tiene el valor 843Fh, que expresado en binario es 1000 0100 0011 1111. La variable *b* también es de tipo short int. Al hacer *b = a>>2*, b toma el valor en binario: 1110 0001 0000 1111, que en hexadecimal es E10Fh.

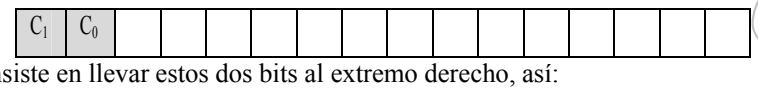
EJERCICIO 5

Re-escribe el programa del ejercicio 1, pero ahora en lugar de desplazar a la izquierda, desplazamos a la derecha. A la vista de los resultados, contesta: cuando desplazamos una variable N posiciones a la derecha, ¿qué estamos haciendo en realidad? ¿Qué sentido tiene aquí distinguir el caso en que la variable tenga signo, o no lo tenga?

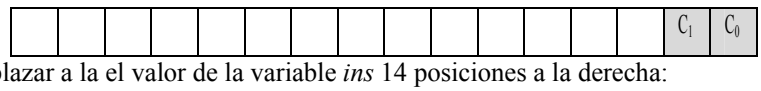
En la extracción de campos de una instrucción usaremos una mezcla de instrucciones AND y desplazamientos. La extracción de un campo tiene dos operaciones: posicionamiento del campo, y enmascaramiento. Lo veremos con un ejemplo:

Usaremos una variable de tipo *unsigned short int* (16 bits sin signo) llamada *ins* que guarda la instrucción completa. En nuestro ejemplo, el valor que tiene guardado en este momento esa variable es 40B7h. Queremos saber cuál es la clase de instrucción, y si es posible, el tipo de instrucción.

El código de clase de instrucción ocupa los dos bits más significativos del valor. Gráficamente, serían los dos espacios sombreados de la figura. C_1 sería el bit más significativo del código de clase, y C_0 el menos significativo.



Posicionar el campo consiste en llevar estos dos bits al extremo derecho, así:



Para ello habrá que desplazar a la el valor de la variable *ins* 14 posiciones a la derecha:

```
ins>>14
```

Una vez que tenemos el campo posicionado, hay que enmascarar los bits no usados. El enmascaramiento es una operación AND aplicada a un número que aísla determinados bits de dicho número, los cuales quedan con su valor original. El resto de bits queda a 0.

En nuestro caso, nos interesa dejar con su valor original a los dos bits que contienen el código de clase, y que previamente hemos llevado hasta el extremo derecho del valor de 16 bits. Es decir, en este ejemplo nos interesa dejar intactos a los bits 0 y 1. El resto estarán a 0. Para ello usaremos este valor de máscara:

0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1

Que en decimal sería el número 3.

Cuando hagamos la operación AND entre el valor desplazado, conseguido anteriormente, y esta máscara, obtendremos:

														C_1	C_0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1
0	0	0	0	0	0	0	0	0	0	0	0	0	0	C_1	C_0

En C, esto se aplicaría así (partiendo de la expresión anterior donde desplazamos el campo a la derecha):

```
(ins>>14)&3
```

Supongamos que queremos guardar el código de clase de instrucción en la variable *clase*. Lo haríamos así:

```
clase = (ins>>14)&3;
```

Ahora podemos preguntar por el valor de clase, y según qué valor tenga (0,1,2 ó 3) sabremos si hay o no código de instrucción y cómo se extrae.

Por ejemplo, supongamos que queremos extraer el código de instrucción para el caso en que el código de clase sea 0, y guardar el valor de dicho código en la variable *codins*. Miramos la tabla de instrucciones y vemos que el código de instrucción en este caso ocupa 3 bits (I_2, I_1, I_0), así:

		I_2	I_1	I_0											

El desplazamiento necesario aquí será de 11 posiciones.

```
ins>>11
```

Y la máscara necesaria será:

0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1

Con lo que la operación final queda en C:

```
codins = (ins>>11)&7;
```

EJERCICIO 6

Escribe un programa en C (descargable en la web como *simej6.c*) que incluya una función que llamaremos *obtclase* (obtener clase). Esta función toma como argumento un valor entero de 16 bits, sin signo, que representa una instrucción, y devuelve un número entero sin signo como resultado de las operaciones descritas anteriormente para hallar la clase de una instrucción. Este número entero, por fuerza, estará comprendido entre 0 y 3.

Para probar esta función, añade al programa en C (a *main*) dos variables enteras, de 16 bits, sin signo, llamadas: *ins*, y *clase*.

El programa pedirá al operador por teclado un número en formato hexadecimal (es decir, usa `"%x"` dentro de *scanf*), que se almacenará en una variable entera (por ejemplo, la podemos llamar *aux*). A continuación, el contenido de *aux* se asigna a la variable *ins*. Este número guardado en *ins* será una instrucción de 16 bits de la CPU virtual. El programa usará la función *obtclase* para guardar en la variable *clase* el código de clase de la instrucción, devuelto por la función. Finalmente, y en función del valor de *clase* (que será 0, 1, 2, ó 3) mostrará por pantalla un mensaje indicando a qué clase pertenece la instrucción entrada por teclado (0: movimiento, 1: artimética, 2: salto incondicional o 3: salto condicional).

La razón de usar la variable *aux* (por "auxiliar") en el *scanf()* es porque *scanf()* no reconoce las variables de tipo *short int*, como *ins*, y para ella, una variable de tipo *short int* es como si fuera de tipo *int*. Lo malo es que una variable de tipo *int* ocupa 4 bytes, y una *short int*, 2, con lo que al guardar el valor que se ha leído de teclado, en el caso de una variable *short int*, *scanf* escribe fuera del espacio asignado a esa variable. Usando una variable auxiliar que sea *int* evitamos este error. La asignación *ins=aux* almacena en la variable *ins*, de forma segura, lo que se ha leído en *aux*.

Es decir, que cada vez que queramos leer de teclado (o en su momento, de fichero) un dato en formato hexadecimal con la intención de guardarlo en una variable que sea de tipo *short int*, haremos esto:

```
short int ins;
int aux;
......
scanf ("%x", &aux); /* leemos de teclado y guardamos en variable INT */
ins = aux; /* el valor de la variable entera lo copiamos en la variable de tipo SHORT INT */
```

EJERCICIO 7

Escribe otra función a la que llamaremos *obtcodigo* (obtener código de instrucción). Esta función toma dos argumentos: el primero es un valor entero de 16 bits, sin signo, que representa una instrucción, y el segundo es un número entero que representa su código de clase (que habremos averiguado previamente usando la función *obtclase* con la misma instrucción), y devuelve un número entero con signo como resultado de las operaciones descritas anteriormente para hallar el código de una instrucción.

En esta primera versión de *obtcodigo*, sólo vamos a calcular el código de la instrucción suponiendo que la clase de la instrucción sea siempre 0. Esta suposición es necesaria para simplificar este ejercicio, ya que el código de una instrucción se calcula de distinta manera según cuál sea la clase de dicha instrucción. Así, para esta primera versión de *obtcodigo* (la mejoraremos en siguientes ejercicios), sencillamente ignoraremos el segundo argumento por ahora, el que representa el código de clase de la instrucción.

Incorpora esta función al programa que estás escribiendo en el ejercicio 6. La nueva versión del ejercicio 6 es un programa que pide por teclado una instrucción, calcula su código de clase, y si dicho código de clase es 0, entonces calcula además su código de instrucción. Para esto es para lo que usamos las dos funciones que hemos definido. Una vez calculado el código, lo imprimirá en hexadecimal en pantalla. Probad este programa únicamente con instrucciones de la clase 0, ya que para otras clases, su comportamiento estará indefinido.

EJERCICIO 8

Amplía la función *obtcodigo* para que calcule también los códigos de las instrucciones cuyo código de clase es 1, 2 y 3, e decir, el de todas las instrucciones. Ahora sí que usamos el segundo parámetro de *obtcodigo*, que en la versión primera era ignorado, para según qué clase tenga la instrucción, calcular el código de instrucción de una forma u otra.

Para las instrucciones de la clase 2, en las que no hay código, sencillamente se devuelve 0.

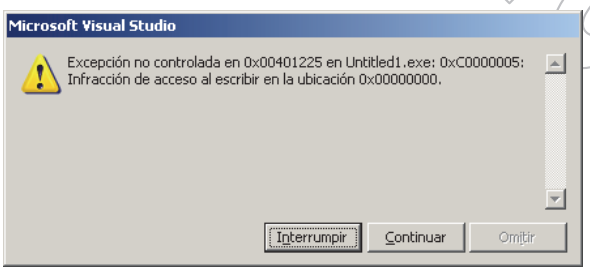
Para las instrucciones de clase 3, el código que se calcula y devuelve será el código de la condición. Como este código ocupa 3 bits, igual que en el caso de las instrucciones de clase 0, en realidad las instrucciones de clase 0 y 3 se tratarán de la misma formas.

Una vez realizada la ampliación modifica el programa del ejercicio 6 para que muestre en pantalla la clase y el código de la instrucción tecleada.

Para probar los programas de los ejercicios 6, 7 y 8, usa los ejemplos de instrucciones que hay en la tabla de instrucciones del enunciado de la práctica.

Los programas escritos en estos ejercicios, sobre todo el del ejercicio 8, nos permiten discernir, dada una instrucción leída, saber de qué clase es, y dentro de la clase, saber cuál de ellas es. Una vez que tengamos una instrucción identificada, podemos seguir decodificando el resto de sus campos para obtener los operandos que necesita. Una vez se saben los operandos, estamos listos para "ejecutar" la instrucción, es decir, hacer los cambios y operaciones necesarias en los registros de la CPU según la instrucción que hemos leído.

En una CPU real, las instrucciones se leen directamente de memoria y se van decodificando. Por diversas causas, pueden aparecer instrucciones desconocidas, o instrucciones cuyos operandos no son correctos. Cuando esto pasa, la CPU no puede ejecutar la instrucción y ocurre una excepción. Las excepciones ocurren con muchísima frecuencia: la mayoría de dichas excepciones son "benignas", es decir, no conducen a tener que abortar ningún programa que se esté ejecutando en el ordenador, pero hay veces en que esto no es así, y una excepción obliga al sistema operativo a terminar bruscamente un programa. Quizás el ejemplo más conocido sea la ventana que aparece en Windows cuando un programa sufre un error (casi siempre, por estar mal escrito).



Ejercicios con bucles.

EJERCICIO 9

Usando como partida la plantilla de nombre *simej9.c* , escribe una función que se llamará *simulador*. De momento, esta función no tiene argumentos de entrada, ni devuelve ningún valor a la salida. Es decir, su prototipo será:

```
void simulador (void);
```

Esta función usará un bucle que pedirá instrucciones (números hexadecimales de 4 dígitos, o 16 bits) por teclado al usuario. Incorpora también al programa las funciones *obtclase* y *obtcodigo*, para que la nueva función pueda calcular el código de clase y el código de instrucción de la instrucción recién introducida y mostrarlo por pantalla. El programa repetirá las operaciones de pedir instrucción y mostrar código y clase mientras la instrucción leída no sea STOP. Cuando la instrucción leída sea STOP, el bucle terminará y mostrará un mensaje como éste:

```
Fin de programa, Codigo de operacion leído es STOP
```

Para que el programa sea más claro, se aconseja definir una constante (usando `#define`), de nombre STOP y de valor 0x3800 (que es el código de operación de STOP en hexadecimal, tal y como se puede comprobar en la tabla de instrucciones del enunciado de la práctica.)

EJERCICIO 10

Modifica la función *simulador* del ejercicio anterior para que declare una variable entera llamada **`pc`**, inicializada a 0, y la incremente dentro del bucle. Dentro del bucle debe mostrar un mensaje como:

```
Dame instrucción en la dirección %4.4X:
```

Al ejecutarse, *%4.4X* se sustituirá por el valor de la variable **`pc`** (aparecerá como un número hexadecimal de 4 dígitos, relleno con ceros a la izquierda). La idea es que mientras que el programa se ejecuta, el usuario hace el papel de "memoria" de la CPU, y va suministrando instrucciones a medida que la CPU las pide. El número que aparece en el mensaje de cortesía sería la dirección de memoria donde está la instrucción que la CPU quiere leer. Al comenzar esta dirección en 0, el valor de **`pc`** se convierte, de facto, en una cuenta de instrucciones.

Cuando el bucle termina, el mensaje final cambia a éste:

```
Fin de programa, Codigo de operacion leído es STOP. Se leyeron n instrucciones
```

Donde *n* es el valor de **`pc`** tras el bucle (por ahora, coincide con el número de instrucciones leídas)

EJERCICIO 11

Vamos a darle algo de "cuerpo" al simulador: tenemos un programa, el del ejercicio 10, que constituye el *bucle principal de emulación de la CPU*: mientras no se termine la emulación, la CPU debe seguir leyendo instrucciones. Una vez que tenemos un nuevo código de operación de instrucción, tendremos que preguntar cuál es, para según sea uno u otro, realizar las operaciones pertinentes.

Modificar la función "simulador" para que, dentro del bucle y una vez que tenemos una nueva instrucción introducida por teclado, el programa la decodifique e imprima por pantalla el mnemotécnico de dicha instrucción (STOP, MOVH, MOVL, etc…). Para ello será muy útil la construcción *if-else if-else if...* vista en la práctica "*Condiciones*". Al principio, decodifica solamente las instrucciones cuyo código de clase sea 0 (las instrucciones de movimiento). Luego ve ampliando la función, añadiendo más *if-else if* para decodificar el resto de instrucciones. El esquema de esta decodificación, que estaría en el cuerpo del bucle, sería así:

```
LEE instrucción en variable ins /* usa en realidad, aux para leer la instrucción como se ha avisado */
codclase = obtclase (ins)
codins = obtcodigo (ins, codclase)
SEGÚN codclase HACER
  CASO 0: SEGÚN codins HACER /* instrucciones de carga y almacenamiento */
            CASO 0: ...
            CASO 1: ...
            CASO 2: ...
            CASO 4: ...
            CASO 5: ...
            CASO 7: fin_simulacion=1 /* se ha leido la instrucción STOP */
            POR DEFECTO: EXCEPCION
          FIN SEGUN
  CASO 1: /* instrucciones aritmético-lógicas */
  CASO 2: /* salto incondicional */
  CASO 3: /* salto condicional */
FIN SEGUN
```

Recuerda que las secuencias if-else if terminan con un "else" (sin un "if" a continuación). Ese último "else" es el caso que se ejecuta cuando ninguna de las anteriores condiciones se cumple. Para nosotros esto pasa cuando nos encontramos con un código de instrucción para el que no hay una instrucción definida. Entonces, el programa debe imprimir la palabra EXCEPCION.

Ejercicios con vectores

En simulador tenemos una memoria. Esta memoria, según las especificaciones, es de elementos de 16 bits. En C un elemento de 16 bits sin signo es de tipo *unsigned short*.

Vamos a añadir la memoria a nuestra CPU. Además, vamos a añadir también los registros generales R0 a R7.

EJERCICIO 12

El programa de este ejercicio lo haremos en un fichero aparte, no nos basaremos en ejercicios anteriores. Más tarde, juntaremos el código de este ejercicio con los anteriores. Usaremos como plantilla el fichero *simej12a14.c*

En el programa, define primero una constante (usando *#define*) que dé un valor a TMEMORIA. Para empezar, un valor de 100 no está mal. Declara un vector de TMEMORIA elementos, donde cada elemento será de 16 bits. Por ejemplo, así:

```
#define TMEMORIA 100  // ← Esto se pone justo después de los #include
```

Ya dentro del código, hay que declarar e inicializar este vector. Para declarar el vector, llámalo *mem*. Según las especificaciones de la práctica, los elementos no usados de la memoria estarán a 0, así que comenzaremos por inicializar toda la memoria a 0. Como son muchos elementos, conviene hacerlo con un bucle, que inicialice cada elemento.

Escribe una función que llamaremos *InicializarMemoria*, con este prototipo:

```
void InicializarMemoria (unsigned short m[], int tmem);
```

Dentro de la función, *m* será el vector que contiene a la memoria, y *tmem*, un valor entero que indica cuántos elementos tiene el vector. La función se encargará de poner un 0 en cada elemento de *m*.

A esta función la llamaremos desde *main()* de esta forma:

```
InicializarMemoria (mem, TMEMORIA);
```

Nuestra CPU necesita 8 registros, que en la especificación están nombrados como R0, R1, R2, etc. ¿Cómo es mejor hacerlo? ¿8 variables llamadas *r0,r1,...,r7*? ¿O un vector de 8 elementos *r[8]*? Antes de decidirte por uno u otro, piensa en lo que tendrías que hacer para, por ejemplo, implementar una instrucción como *ADD Rd,Rs1,Rs2*, donde hay que coger el valor del registro número *s1*, sumarlo con el valor del registro número *s2*, y el resultado guardarlo en el registro de número *d*. Los valores de *s1*, *s2* y *d* estarían en sendas variables.

Una vez que lo hayas pensado, declara los registros generales, e inicialízalos todos al valor 0.

EJERCICIO 13

La memoria está inicializada a 0, pero esto sólo, no nos vale para el simulador. La memoria debe contener el programa que se va a ejecutar (y el resto, a 0). En este ejercicio, vamos a rellenar parte de la memoria con un programa. En la práctica final esto se hace leyendo un fichero. Nosotros de momento vamos a leer desde el teclado.

Escribe una función a la que llamaremos *CargarMemoria*, con este prototipo:

```
void CargarMemoria (unsigned short m[]);
```

Que será llamada desde *main()* así:

```
CargarMemoria (mem);
```

Esta función necesita poder leer y modificar una serie de variables que también se necesitan en otros puntos del programa. Si definimos esas variables dentro de la función, no podrán ser usadas por otras funciones. Para remediarlo, usaremos variables globales, que son variables que pueden ser leídas y modificadas por cualquier función.

Una variable global es toda variable que se declare fuera de una función. Nosotros definiremos las variables globales después de los #includes y #defines, y antes de los prototipos.

Añade una variable global que llamaremos *pc*. Ya sabes para qué sirve. Añade como variables globales a los registros y al vector *mem*.

La función *CargarMemoria* pide por teclado (con su correspondiente mensaje de cortesía) un valor en hexadecimal, que será la posición de memoria a partir de la cual se va a almacenar el programa. Este valor deberá guardarse también en una variable global, que llamaremos *pos_origen*. Copiamos su valor a otra variable (ésta, local), que llamaremos *pos_actual*.

A continuación, pediremos al usuario por teclado la dirección, en hexadecimal, de la primera instrucción a ejecutar una vez que comience la simulación. ¿Dónde hay que guardar este valor? En una variable que represente a un registro del procesador que señale a la próxima instrucción a ejecutar. Esa también es una variable global. Mantendremos además una copia de este valor en otra variable global.

A continuación, el programa irá pidiendo al usuario valores hexadecimales. Cada valor se irá guardando en la memoria, en el elemento (dirección) que indique la variable *pos_actual*. A cada nuevo valor introducido en memoria, la dirección (*pos_actual*) donde se almcena, se irá incrementando. Esto se repetirá **MIENTRAS QUE** haya aún datos que introducir.

Cuando se lee de un fichero, existe una forma de detectar que no hay más datos para introducir. Este sistema también se puede usar para el teclado. Simplemente hay que saber que el teclado se llama *stdin* . La forma de saber si ya no hay más datos para introducir por teclado es usando la función *feof()* con el parámetro *stdin*. Por ejemplo, el siguiente programa:

```
#include <stdio.h>

int main()
{
  int n;

  printf ("Dame un numero (pulsa Ctrl-Z para terminar): ");
  fflush (stdin);
  scanf ("%d", &n);
  while (!feof(stdin))
  {
    printf ("Tu numero es: %d\n", n);
    printf ("Dame otro numero (pulsa Ctrl-Z para terminar): ");
    fflush (stdin);
    scanf ("%d", &n);
  }
}
```

Pide números por teclado al usuario y los va imprimiendo en pantalla, hasta que el usuario, en respuesta a una petición de un número pulsa las teclas Ctrl y Z simultáneamente. Esta pulsación va a ser nuestra forma de decirle al ordenador que no queremos introducir más números (en otros sistemas operativos, como Linux u OS X, se usa la combinación Ctrl-D en lugar de Ctrl-Z).

Al pulsar esta combinación de teclas, aparecerá en pantalla la secuencia **^z** (el carácter **^** se asocia a "control"). Pulsando *Intro* se envía la secuencia a *scanf*.

La llamada a la función *feof(stdin)* devuelve VERDADERO si la última vez que se pidieron datos por teclado, el usuario pulso Ctrl-Z. Devuelve FALSO si no fue así. De esa forma, el bucle se lee como "Mientras NO se haya pulsado Ctrl-Z…"

El algoritmo a implementar en la función *CargarMemoria* es éste:

```
LEER direccion origen de los datos en variable pos_origen
pos_actual ← pos_origen
LEER direccion inicial de ejecución en variable pc
pc_inicial ← pc
LEER instrucción
MIENTRAS QUE en la ultima lectura no se haya pulsado ctrl-z  HACER
  Copiar a la direccion pos_actual de memoria el valor de instrucción
  Incrementar pos_actual
  LEER instrucción
FINMIENTRAS
```

Al finalizar este bucle, en la variable *pos_actual* estará el valor de la primera dirección de memoria libre, tras haber introducido el programa y los datos. Este valor lo guardaremos en una variable global a la que llamaremos *pos_final*. Ahora tenemos acotada la porción de memoria que hemos inicializado con los valores que se han leído de teclado: es la porción que va desde la dirección guardada en *pos_origen* hasta (sin llegar) a la dirección guardada en *pos_final*. Esta información la usaremos en el siguiente ejercicio.

Atención: no olvidarse de que las lecturas por teclado de valores que vayan a guardarse en variables de tipo *short int* deben hacerse usando una variable auxiliar que sea entera, para acto seguido, asignar el valor de esa variable entera a la que realmente usaremos, de tipo *short int*. Ver ejercicio 6 para más detalles.

EJERCICIO 14

Escribe la función *GrabarMemoria*, con el siguiente prototipo:

```
void GrabarMemoria (unsigned short m[], unsigned short pc, unsigned short origen, unsigned short
fin);
```

Esta función imprime por pantalla en hexadecimal, y un número en cada línea, los siguientes valores:

- El valor de *origen*
- El valor de *pc*
- Todos los valores almacenados en memoria, desde la posición *origen*, hasta la posición justo anterior a *fin*.

Desde *main()* la llamaremos así:

```
GrabarMemoria (mem, pc_inicial, pos_origen, pos_final);
```

Prueba este programa introduciendo el código escrito en el recuadro titulado como *"memoria.txt"* de la página 5 de este boletín. Los valores hexadecimales que allí hay han de introducirse de izquierda a derecha, y de arriba abajo. Los dos primeros valores corresponden a la dirección de origen, y dirección inicial de ejecución. Los contenidos que van a memoria comienzan en realidad en el tercer número (0000). Después de introducir el último número (3800), cuando pida el siguiente, pulsa Ctrl-Z para indicar que se han terminado los datos de entrada. En ese momento se mostrará la salida impresa generada por *GrabarMemoria*, que debe coincidir con el contenido del recuadro.

Ejercicios con ficheros.

EJERCICIO 15

Partiendo del programa del ejercicio 14, vamos a cambiar el comportamiento de la función *GrabarMemoria*. Ahora mismo, los datos se vuelcan en la pantalla. Modifícalo para que dichos datos se vuelquen en un fichero de nombre *"resultados.txt"* tal y como se pide en los requisitos del simulador (ver boletín del simulador). Al probar esta nueva versión veremos que ya no se imprimen los datos de memoria al final, como pasaba en el ejercicio 14, sino que aparecen en el fichero mencionado. Comprueba que el fichero se ha generado correctamente abriéndolo con el Bloc de Notas.

EJERCICIO 16

Haremos otra modificación al programa, partiendo del ejercicio 14. La función *CargarMemoria* ahora no inicializa el vector de memoria desde teclado, sino desde fichero.

Cambiaremos también su prototipo: originalmente esta función no devuelve nada. Ahora sí devuelve un entero. Modifica el prototipo para reflejar este cambio. El valor que devuelve se explica a continuación:

Puede ocurrir que el fichero *memoria.txt* no pueda abrirse para lectura. En ese caso, la función *CargarMemoria* debe devolver 0 para indicar error. Si la lectura se realizó con éxito, devolverá 1. Puedes tomar como guía el diagrama de flujo de la sección "Programación modular" del tema 3 de teoría, donde se ha refinado esta función.

Para probar esta nueva función junto con la modificada en el ejercicio 13, crea un fichero *memoria.txt* usando el Bloc de Notas. Hay un ejemplo de este fichero en el boletín de la práctica del Simulador. También puedes crear el fichero *memoria.txt* usando el ensamblador suministrado en las herramientas de testeo del simulador, en Enseñanza Virtual.

Si todo va bien, al ejecutarse el programa se creará un fichero *resultados.txt* cuyo contenido debe ser idéntico a *memoria.txt*

Ha llegado el momento de unir las funciones *GrabarMemoria*, *CargarMemoria*, e *InicializarMemoria* en el programa cuya plantilla se llama *simcpu.c* . A partir del siguiente ejercicio se irán incorporando otras funciones que ya teníamos escrita a este mismo programa. Esta plantilla es la última que usaremos, y contiene al simulador completo.

EJERCICIO 17

Del ejercicio 11, rescata la función *simulador*. Esta función, recordemos, iba pidiendo instrucciones por teclado al usuario, y mostrándolas por pantalla, hasta que el usuario tecleaba la instrucción STOP, o bien se producía una excepción.

Vamos a hacer algunos cambios a esa función. Para empezar, copia la función a la nueva plantilla donde ya hemos copiado el resto de funciones. Copia también las funciones *obtclase()* y *obtcodigo()* que son usadas por esta función. En la plantilla ya están definidos los prototipos de todas estas funciones, así que sólo hace falta copiar las implementaciones.

En segundo lugar, vamos a cambiar el prototipo de la función. El original era:

```
void simulador (void);
```

El nuevo prototipo será éste:

```
int Simulador (void);
```

Lo que devuelve la nueva versión de *Simulador* es un 0 si la simulación se ha producido sin excepciones, y distinto de 0 si hubo una excepción. Ahora mismo, la única excepción que estamos tratando es la de instrucción incorrecta, así que a esa excepción le asignaremos el valor 1. En la función *Simulador* por tanto añadiremos una variable *excepcion*, que inicialmente valdrá 0, y ocasionalmente, cuando nos encontremos con un código de instrucción que no tiene correspondencia con ninguna instrucción definida, valdrá 1. Luego veremos otros tipos de excepciones que se pueden producir en la ejecución, que harán que esta variable tome otros valores diferentes de 0 y 1.

A continuación viene el cambio más grande de todos: la nueva función *Simulador* ya no lee instrucciones desde el teclado, sino que las leerá de la memoria, es decir, el vector *mem*. Eso significa que dentro de la función *Simulador*, donde aparece un código similar a éste:

```
printf ("Dame instrucción en la dirección %04.4X: ", pc);
fflush (stdin);
scanf ("%04.4X", &aux);
ins = aux;
```

Debe cambiarse por una instrucción que lea el elemento *pc*-ésimo del vector *mem*. Por supuesto, **ya no hay mensajes de cortesía ni lectura de teclado en esta función**. El elemento leído se pasa directamente a la variable *ins*. Ya no hace falta aquí la variable entera *aux* para poder leer números hexadecimales de teclado sin peligro puesto que ya no leemos de teclado en esta función.

La variable *pc*, por otra parte, ya no se inicializará a 0, sino que se inicializará al valor que tenga la variable global *pc_inicial*, y cuyo valor fue leído del fichero en la función *CargarMemoria*.

Al final de todos estos cambios, lo que tenemos es una función que se encarga de, dentro de un bucle, ir leyendo de forma secuencial instrucciones desde el vector *mem*. El índice que se usa para acceder a *mem* es el valor de *pc*, que se incrementará en 1 justo después de haber leído la instrucción.

La instrucción leída es decodificada, y su mnemotécnico se imprime en pantalla.

Esto continúa así hasta que la instrucción leída es STOP, o bien se produjo una excepción. En el primer caso, *Simulador* devuelve 0. En el segundo caso, devuelve 1. El bucle *while* que controla la lectura de instrucciones debe terminar no solamente si se encuentra STOP, sino también si la última instrucción generó una excepción.

Al terminar el bucle de simulación se mostrará un mensaje de que se ha terminado porque se encontró STOP (como teníamos hasta ahora) o bien mostrará que se ha terminado porque ha ocurrido una excepción. El mensaje de excepción contendrá la dirección de memoria donde ha ocurrido dicha excepción (¡OJO! Porque la variable *pc* estará apuntando no a la dirección de la instrucción incorrecta, sino a la siguiente) y la instrucción en sí (en formato hexadecimal ambos valores).

Ya tenemos un programa que es capaz de cargar un programa para nuestra CPU desde un fichero. El programa, una vez cargado, es leído por el simulador que va mostrando las instrucciones a medida que las lee y decodifica. Como las instrucciones no se ejecutan, el registro PC siempre va incrementándose en 1 cada vez, ya que aún no se están ejecutando las instrucciones de salto que pudieran cambiar su valor.

Por último, nuestro programa es capaz de generar un fichero con el contenido de la memoria. En este momento, repetimos, como el programa realmente no se ejecuta, el contenido de la memoria no ha cambiado, así que el fichero *resultados.txt* debería tener el mismo contenido que *memoria.txt*, salvo que durante la lectura e identificación de alguna de las instrucciones se cometiera alguna excepción. Si esto pasó, verás que el fichero *resultados.txt* no se genera (mira en *main* como la función *GrabarMemoria* sólo es llamada cuando el resultado de la función *Simulador* es igual a 0, es decir, cuando no se produjo ninguna excepción durante la simulación).

El resto de los ejercicios hasta el último muestran cómo implementar la ejecución de las instrucciones del simulador. Mostraremos paso a paso cómo implementar la ejecución de alguna de las instrucciones de movimiento (clase 0) y una de las instrucciones aritmético-lógicas (clase 1). El resto tendrás que hacerlas tú por tu cuenta.

Todas las instrucciones que componen nuestra CPU van a operar sobre registros y/o memoria. Cuando una instrucción necesite leer o escribir a un registro, accederá a la posición adecuada del vector de registros. Cuando tenga que escribir o leer a memoria, tendrá que hacer lo propio con el vector *"mem"*.

Además, las instrucciones aritmético-lógicas, tras su ejecución, tendrán que actualizar las banderas de la ALU según el resultado de la operación.

Las instrucciones de salto operarán además sobre el registro PC, modificándolo si es necesario.

EJERCICIO 18

Vamos a implementar el comportamiento de las instrucciones MOV, MOVH y MOVL. Estas son instrucciones de clase 0. Las modificaciones se harán sobre el código de la función *Simulacion* del ejercicio 17.

Localiza el punto en tu código, dentro de la cadena de if-else if (o *switch/case* si has optado por implementarlo así) donde se ha identificado correctamente la instrucción MOV. Seguramente será la primera instrucción que identificas. El punto donde se ha identificado es precisamente donde haces el *printf ("MOV Rd, Rs\n");* . Pon ese *printf* en un comentario (por ejemplo, poniendo dos barras // delante de él, o encerrando toda la orden entre un /* y un */). Es mejor dejarlo como comentario que borrarlo, ya que al dejarlo de comentario recordamos que ahí es donde se ejecuta esa instrucción. Si ocurre algún error y tenemos la necesidad de saber qué instrucciones se están ejecutando en nuestro programa, podemos descomentar la línea con *printf* y volver a ver esa información.

Tenemos que averiguar los valores de los operandos *Rd* y *Rs* para la instrucción en curso. *Rd* y *Rs* hacen referencia a números de registros, por lo que los valores de cada uno de estos estará entre 0 y 7. Un valor entre 0 y 7 se codifica con 3 bits.

Repasando la tabla con el juego de instrucciones encontramos:

Nemotécnico	***Formato***	***Descripción***	***Excepciones***	***Ejemplo***
MOV Rd,Rs	00 000 Rd Rs 00000	Copia el valor que esté almacenado en el registro Rs, al registro Rd.	No tiene.	MOV R3,R5 Binario: 00 000 011 101 00000b Hexadecimal: 03A0h Antes de esta instrucción: R3 = 1C40h R5 = 9B23h. Después de esta instrucción: R3 = 9B23h

Partiendo del formato de la instrucción en binario, obtenemos que *Rd* y *Rs* ocupan las posiciones de bit 8 a 10 (siendo la posición 0 la de más a la derecha) para *Rd*, y las posiciones 5 a 7 para *Rs*. Usaremos la misma técnica que para averiguar el código de clase y de instrucción: desplazamientos y máscaras.

Para averiguar el valor de *Rd*, habrá que desplazar la instrucción leída (que suponemos que está en la variable ins) 8 lugares a la derecha, y seguidamente aplicarle una máscara que deje sólo 3 bits. Usaremos una variable entera llamada *rd* para guardar este valor, que quedaría así:

```
rd = (ins>>8)&7;
```

Análogo procedimiento para obtener el valor de *Rs*, que guardaremos en la variable *rs*:

```
rs = (ins>>5)&7;
```

Ahora ya tenemos los números de registro para el registro fuente y el registro destino. La instrucción MOV lo que hace es copiar el valor almacenado en el registro número *Rs* al registro número Rd. Nuestros registros son elementos de un vector que se llama R, así que la acción de MOV se implementaría así:

```
R[rd] = R[rs];
```

Esta instrucción no modifica las banderas de la ALU ni genera excepciones, con lo que no hay que hacer nada más para implementarla. Ahora nuestro simulador es capaz de ejecutar una instrucción: MOV.

Para MOVL y MOVH (y de hecho, para el resto de instrucciones) el procedimiento es análogo: se decodifican los operandos de la instrucción usando desplazamientos y máscaras, y escribiremos en C las acciones que comportan la ejecución de la instrucción.

MOVH y MOVL se parecen a MOV en tanto que escriben un nuevo valor en el registro marcado como *Rd*. Se diferencian ambas de MOV en que ésta última modifica el registro completo, mientras que MOVL y MOVH sólo modifican una mitad (la mitad baja L, o la mitad alta H). Esto también lo resolveremos con máscaras y desplazamientos.

MOVH. Tiene dos operandos: un número de registro *Rd*, y un valor inmediato de 8 bits. El primero está en las posiciones de bit 8 a 10, y el segundo está en las posiciones de bit 0 a 7.

Decodificar ambos operandos es sencillo. Más incluso para el valor inmediato ya que no requiere desplazamiento, sólo la máscara. Supongamos que tenemos los operandos ya decodificados en las variables enteras *rd* e *inm8*. La acción de modificar la parte alta (los bits 8 a 15) del registro de destino se haría así:

```
R[rd] = (R[rd]&0x00FF) | (inm8<<8);
```

Recuerda lo que hace el operando | en C (OR bit a bit). La secuencia de operaciones es:

- Se coge el valor actual del registro cuyo número es *Rd* y se borran sus 8 bits superiores usando una máscara AND con ceros en los bits 8 a 15 (0x00FF).
- Se coge el valor de *inm8* y se desplaza 8 lugares a la izquierda (<<8). Ahora, el valor de *inm8* en lugar de estar ocupando las posiciones de bit 0 a 7, está ocupando las posiciones 8 a 15. Las posiciones de bit 0 a 7 se rellenan con ceros durante el desplazamiento.
- Se combinan ambos valores con la operación OR. Los valores de los bits 8 a 15 de *inm8* desplazado se copian en las posiciones de bit 8 a 15 del registro *Rd*. Las posiciones de bit 0 a 7 no se modifican al combinarse con ceros, provenientes de *inm8* desplazado.

En el caso de MOVL, para borrar sus 8 bits inferiores y dejar los 8 superiores intactos usaremos 0xFF00 en lugar de 0x00FF. Haz el resto de operaciones, para implementar esta instrucción.

EJERCICIO 19

Para el resto de instrucciones habrá que hacer más o menos lo mismo, así que para organizarte mejor en la implementación de cada instrucción, comienza por determinar la secuencia de máscaras y desplazamientos que necesitarás para decodificar cada operando, así como la acción a realizar para su implementación. Para ello, puedes rellenar la tabla que te proponemos a continuación:

Instrucción	Formato binario	Operando 1	Operando 2	Operando 3	Acción
MOV Rd,Rs	00 000 Rd Rs 00000	rd = (ins>>8)&7;	rs = (ins>>5)&7;	NO TIENE	R[rd] = R[rs];

MOV Rd,[Ri]	00 001 Rd Ri 00000	rd =	ri =	NO TIENE	mar = R[ri];
					if (mar < TMEMORIA)
					R[rd] = mem[mar];
					else
					excepcion = 2;

MOV [Ri],Rs	00 010 Ri Rs 00000	ri =	rs =	NO TIENE	
MOVL Rd,inm8	00 100 Rd inm8	rd =	inm8 =	NO TIENE	
MOVH Rd,inm8	00 101 Rd inm8	rd =	inm8 =	NO TIENE	R[rd]=(R[rd]&0x00FF) | (inm8<<8);

ADD Rd,Rs1,Rs2	01 00000 Rd Rs1 Rs2	rd =	rs1 =	rs2 =	op1 = R[rs1];
					op2 = R[rs2];
					temp = op1 + op2;
					R[rd] = temp;
					ActualizaALU(...);

SUB Rd,Rs1,Rs2	01 00001 Rd Rs1 Rs2	rd =	rs1 =	rs2 =	op1 = R[rs1];
					op2 = (~R[rs2])+1;
					temp = op1 + op2;
					R[rd] = temp;
					ActualizaALU(...);

OR Rd,Rs1,Rs2	01 00010 Rd Rs1 Rs2	rd =	rs1 =	rs2 =	
AND Rd,Rs1,Rs2	01 00011 Rd Rs1 Rs2	rd =	rs1 =	rs2 =	
XOR Rd,Rs1,Rs2	01 00100 Rd Rs1 Rs2	rd =	rs1 =	rs2 =	(ver ejercicio 20)
COMP Rs1,Rs2	01 00111 Rs1 Rs2 000	rs1 =	rs2 =	NO TIENE	
NOT Rds	01 01000 Rds 000000	rds =	NO TIENE	NO TIENE	
INC Rds	01 01001 Rds 000000	rds =	NO TIENE	NO TIENE	
DEC Rds	01 01010 Rds 000000	rds =	NO TIENE	NO TIENE	
NEG Rds	01 01011 Rds 000000	rds =	NO TIENE	NO TIENE	
CLR Rds	01 01100 Rds 000000	rds =	NO TIENE	NO TIENE	
JMP desplaz	10 desplaz	desplaz = *(extender en signo de 14 a 16 bits. Ver ejercicio 19)*	NO TIENE	NO TIENE	
BR cond desplaz	11 cond desplaz	cond =	desplaz = *(extender en signo de 11 a 16 bits. Ver ejercicio 19)*	NO TIENE	

Se aconsejan los siguientes tipos de datos en la definición de las variables que aparecen en esta tabla y el resto de los ejercicios:

- *ri, rs, rs1, rs2, rd, rds, inm8, cond, excepcion, temp* : de tipo entero (int).
- *pc, mar, ins, op1, op2, elementos de mem, elementos de R* : de tipo entero corto sin signo (unsigned short).
- *desplaz* : de tipo entero corto con signo (short)

Hemos añadido un ejemplo de cómo se codifica una instrucción que podría generar una excepción: en la instrucción de lectura de memoria, si se lee más allá del límite de la memoria. La variable *excepcion* aquí ya no tiene únicamente 0 ó 1, sino que toma más valores. Aconsejamos estos:

- 0 para indicar que no hay excepción.
- 1 para indicar una excepción por instrucción inválida.
- 2 para indicar una excepción durante una lectura a memoria (dirección fuera de rango , mostrada ya en la tabla).
- 3 para indicar una excepción durante una escritura a memoria (dirección fuera de rango).
- 4 para indicar una excepción por destino de salto fuera de rango de memoria.

Así, se puede usar un código como el siguiente para mostrar la excepción causada:

```
switch (excepcion)
{
case 0:
    printf ("Programa terminado con normalidad. Se encontro STOP en %4.4X\n", pc-1);
    break;
case 1:
    printf ("EXCEPCION. Instruccion (%4.4X) desconocida en %4.4X\n", ins, pc-1);
    break;
case 2:
    printf ("EXCEPCION. La instruccion (%4.4X) en %4.4X intento leer la direccion %4.4X\n", ins, pc-1, mar);
    break;
...
... etc
...
}
```

Este código se añadirá al final de la función *Simulador*; cuando ya ha terminado el bucle de simulación. Se imprime un mensaje indicando si la simulación terminó correctamente o no, y la causa. Después de imprimir, la función retornará con el valor de la variable *excepcion*.

Después de rellenar la tabla, implementa sobre la función *Simulador* todas las instrucciones de la clase 0. Para ello, haz como has hecho en el ejercicio 18, pero para el resto de las instrucciones de esa clase.

Algunas de las instrucciones de esta clase pueden generar una excepción de tipo 1 (lectura de memoria en dirección fuera de rango) o 2 (escritura a memoria en dirección fuera de rango). La condición de "fuera de rango" se testea con el valor de *mar* y la constante *TMEMORIA*. Si *mar* es igual o mayor que *TMEMORIA*, está fuera de rango (mirar ejemplo en la implementación de la instrucción *MOV rd,[ri]* en la tabla anterior).

Para probar tu primera versión del simulador, ya casi completo, escribe algún programa de prueba con el ensamblador suministrado, o usa alguno de los tests preparados a tal efecto.

EJERCICIO 20

Por otra parte, las instrucciones de la clase 1 (aritmético-lógicas) deben actualizar las banderas de la ALU después de la ejecución. La operación de actualización se realiza mediante una función que hemos llamado *ActualizaALU*. Este es su prototipo:

```
void ActualizaALU (unsigned short op1, unsigned short op2, unsigned short res16, int res32);
```

Hemos añadido un ejemplo de instrucción de esa clase: la instrucción de suma. En este caso, el resultado de la operación, es decir, la suma, se guardará temporalmente en una variable entera de 32 bits (un *int*) que hemos llamado *temp*. Acto seguido, ese valor se traslada al registro de destino, que al ser de 16 bits, se quedará sólo con parte del resultado. La actualización de la ALU ocurre en último lugar, llamando a la función *ActualizaALU*.

Normalmente, el resultado almacenado en *temp* coincidirá con el que luego tendrá el registro de destino. Esto ocurrirá siempre que el resultado de la operación no exceda de 16 bits. Pero si ha excedido, tenemos que detectarlo de alguna forma, y ésta es viendo el resultado que debería haber dado la operación, y el resultado que se ha guardado. De ahí que a la función *ActualizaALU* se le entreguen dos parámetros: el valor de 16 bits guardado en el registro de destino, y el valor de hasta 32 bits, resultado original de la operación. Veremos en un momento que esto se usa para averiguar el valor del flag de acarreo (CF).

Esta función debe actualizar convenientemente las banderas de la ALU. Recordemos que éstas son: ZF (flag de cero), SF (flag de signo), CF (flag de acarreo) y OF (flag de *overflow*). Para ello, supondremos que existen, como variables globales, las variables *zf*, *sf*, *cf* y *of*. Estas variables valdrán 0 ó 1 durante la ejecución de un programa. **Al iniciarse la simulación, todas valen 0** (recuerda añadir esto al principio de la función *Simulador*).

Las operaciones aritméticas que soporta esta ALU son las siguientes: ADD, SUB, COMP, INC y DEC. Recuerda que las restas (SUB, COMP y DEC) son en realidad sumas, en las que al segundo operando se le ha cambiado el signo.

Para hallar el valor del bit de acarreo recuerda que en una ALU que opera con N bits, el acarreo es el valor del bit N (contamos las posiciones de los bits comenzando en 0). En nuestro caso, el valor que tenga el bit 16.

El flag de overflow *(OF)*, al igual que el de acarreo, sólo se actualiza para operaciones aritméticas. Para otras operaciones que no sean éstas, valdrá 0.

OF valdrá 1 si los signos de los dos operandos *op1* y *op2* son iguales, pero el signo del resultado de 16 bits no coincide con ellos. Es decir, que vale 1 si *op1* y *op2* son positivos, pero *res16* es negativo, o si *op1* y *op2* son negativos, pero *res16* es positivo. Para el resto de casos vale 0.

El flag de signo *(SF)* se comprueba sobre el valor de 16 bits. En un número de 16 bits en formato de complemento a 2, ¿qué bit guarda el signo y qué valores toma según el signo del resultado? Usa desplazamientos y máscaras para quedarte con el valor de ese bit. El valor del flag de signo es el valor que tenga ese bit.

El flag de cero *(ZF)* es el más sencillo de averiguar. Se comprueba también sobre *res16*.

En total, esta función tendrá que realizar cuatro sentencias condicionales, uno para cada flag, y en función del resultado, poner la variable que representa a ese flag a 1 ó a 0.

Dado que hay instrucciones ALU que no son aritméticas, sino lógicas, y en éstas hemos dicho que tanto CF como OF valdrán 0, simplemente, después de llamar a *ActualizaALU* (que a lo mejor ha cambiado estos flags), las pondremos directamente a 0. Por ejemplo, la parte final de la implementación de la instrucción *XOR rd,rs1,rs2* , que es una instrucción lógica, quedaría así:

```
op1 = R[rs1];
op2 = R[rs2];
temp = op1^op2;  /* el operador ^ es el XOR en lenguaje C */
R[rd] = temp;
ActualizaALU (op1, op2, R[rd], temp); /* aquí es posible que OF y/o CF se pongan a 1 */
of = 0;
cf = 0; /* esta instrucción no es aritmética, así que ponemos estos dos flags a 0 */
```

Escribe el cuerpo de la función *ActualizaALU*. El prototipo lo tienes ya escrito. Una vez escrita, ya se pueden implementar todas las instrucciones de la clase 1 en la función *Simulador*.

Puedes pasar a probar esta nueva versión del simulador escribiendo pequeños programas con el ensamblador, o bien usando alguno de los programa-test suministrados.

EJERCICIO 21

El operando *desplaz* en las instrucciones JMP y BR es un poco especial. En el primero, dicho operando ocupa 14 bits, y en el segundo caso, 11 bits. En C no existen variables que tengan exactamente ese tamaño, así que al usar desplazamientos y máscaras, acabamos guardando este valor en una variable de 16 (*short int*) ó 32 bits (*int*).

El valor, en ambos casos, debe considerarse como en complemento a 2. Es decir, puede ser positivo o negativo. Cuando es positivo no es problema, ya que un número positivo que ocupa 11 ó 14 bits, tiene el mismo valor que si se guarda en una variable que tenga 16 bits, estando los bits no usados a 0.

Pero si es negativo, sí que hay un problema. Pongamos por ejemplo que el valor de *desplaz* es de 14 bits, y es éste (en complemento a 2 de 14 bits): 11111011000000 . Si pasamos este valor a decimal sale -320.

Si durante el proceso de decodificación de este valor lo metemos en una variable de 16 bits (*short int*) tenemos esto: 0011111011000000 . Tomado como un número en complemento a 2, resulta que este valor es positivo (su bit más significativo vale 0). Pasado a decimal vale 16064, que no tiene nada que ver con el -320 que debería salir.

El problema está precisamente al guardar este valor en una variable que tiene espacio para más bits que los que requiere el valor. Para salvaguardar el signo de un número en complemento a 2 que se almacena en una variable de mayor tamaño, se requiere hacer lo que se denomina ***extensión del signo***. Esto consiste en rellenar los bits no usados de la variable con el valor del bit más significativo del valor original. Se realiza de esta forma:

Usando desplazamientos y máscaras, averiguamos el valor del bit 10 (o 13, según el caso que estemos tratando) de *desplaz*. Este será nuestro bit de relleno.

Si el bit es positivo (0) no hacemos nada especial, simplemente dejamos *desplaz* como está.

Si el bit es negativo (1) hacemos: *desplaz = desplaz | 0xC000;* . Esto lo que hace es poner los bits 14 y 15 de *desplaz* a 1. Si *desplaz* no es de 14 bits sino de 11, entonces los bits que tenemos que poner a 1 son desde el bit 11 al 15. Calcula qué valor es el que tendremos que usar en lugar de *0xC000*.

Este valor *desplaz* es al fin, el que sumamos al valor actual de *pc*, que recordemos, apunta siempre a la próxima instrucción a leer de memoria. Antes de realizar la suma, hay que comprobar que el resultado final no excede el valor de TMEMORIA, ya que si es así, debe generarse una *excepción por destino de salto incorrecto* (excepción 4).

Con esta información, ya puedes implementar las instrucciones de la clase 2 y 3, con lo que el simulador está completo. Puedes probarlo con cualquier programa de los suministrados, o bien escribiendo uno propio.