RESUMEN

El propósito de este proyecto es crear un simulador de instrucciones del microprocesador MIPS R4000 mediante un programa escrito en lenguaje C. El simulador permitirá ejecutar el repertorio de instrucciones en coma fija implementadas en él, avisando cuando ocurra una excepción. Mediante un menú principal, el usuario podrá moverse por todo el interfaz de usuario del simulador, acceder a la memoria, a los registros, cambiar la configuración y ejecutar instrucciones.

Para todo esto, el simulador contará con un fichero en binario de tamaño 64K, presente en el directorio de trabajo, que se usará para simular la memoria principal. Los registros de propósito general, así como los de propósito especial implementados (HI, LO, PC) serán variables del programa.

INTRODUCCIÓN

El MIPS R4000 es un procesador fabricado por MIPS Tecnologies, Inc. que sigue la filosofía de diseño RISC (*Reduced Instruction Set Computer*). Se trata de un procesador segmentado de 64 bits. El integrado incluye la CPU (que se encarga de ejecutar las instrucciones de enteros), el coprocesador 0 (que se encarga de la gestión de la memoria y el procesamiento de las excepciones) y el coprocesador 1 (encargado de la unidad de coma flotante). Además se hallan en el integrado las dos cachés de primer nivel (instrucciones y datos), así como la lógica de control de la caché de segundo nivel.

El modelo de programación del MIPS R4000 incluye 32 registros de enteros de 64 bits (de los cuales el registro cero es un cero hardware) y 32 de coma flotante de 32 bits, agrupables en 16 de 64 bits.

El resto del libro se estructura de la siguiente manera:

Capítulo 1: Características generales del procesador MIPS R4000. En este capítulo se hace una descripción general de las características del MIPS R4000.

Capítulo 2: Manual de usuario del simulador de instrucciones MIPS R4000. Este capítulo sirve como guía para el usuario del simulador, explicando su manejo y funcionamiento.

Capítulo 3: Descripción del diseño del simulador de instrucciones. En este capítulo se explica como funciona el código del simulador y cómo está estructurado.

Apéndice I: Descripción de las instrucciones implementadas en el simulador MIPS R4000. En este apéndice se incluye el subconjunto de instrucciones implementadas en el simulador, con una descripción detallada de cada una de ellas.

Apéndice II: Código fuente del simulador de instrucciones MIPS R4000.

Bibliografía.

CAPÍTULO 1: CARACTERISTICAS GENERALES DEL PROCESADOR MIPS R-4000

REGISTROS DE LA CPU

La unidad central de proceso (CPU) proporciona los siguientes registros:

- 32 registros de propósito general.
- Un registro contador de programa (PC).
- Dos registros que mantienen los resultados de las operaciones de multiplicación y división entera (HI y LO).

Los registros de la unidad de coma flotante serán descritos más adelante.

Los registros de la CPU pueden ser de 32 o de 64 bits, dependiendo del modo de operación en el que estemos trabajando.

La figura 1 muestra los registros de la CPU.

Figura 1.- Registros de la CPU

Dos de los registros de propósito general de la CPU tienen funciones asignadas:

- R0 está inicializado a cero, y puede ser usado como el registro target para algunas instrucciones cuyo resultado va a ser descartado. R0 puede usarse también como fuente cuando se necesita un valor cero.
- R31 es el registro de enlace usado por instrucciones de salto y enlace. No debería ser usado por otras instrucciones.

La CPU tiene tres registros de proposito especial:

- PC – Registro contador de programa.

- HI – Registro de multiplicación y división (parte alta).
- LO – Registro de multiplicación y división (parte baja).

Los registros de multiplicación y división (HI, LO) almacenan:

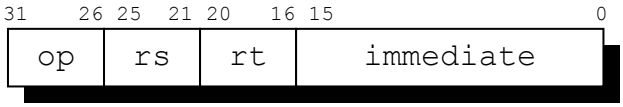
- El producto de las operaciones de multiplicación entera, ó
- El cociente (en LO) y el resto (en HI) de las operaciones de división entera.

SET DE INSTRUCCIONES DE LA CPU

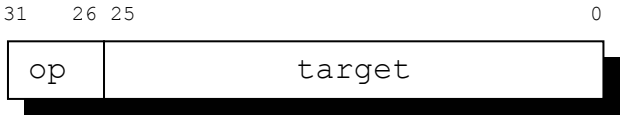
Cada instrucción de la CPU ocupa una palabra (32 bits). Como se ve en la figura 2, hay tres formatos de instrucciones.

- Inmediato (I-type)
- salto (J-type)
- registro (R-type)

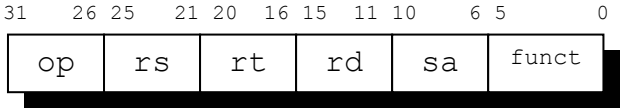
I-Type (Inmediato)



J-Type (Salto)



R-Type (Registro)



op	codigo de operación de 6 bits.
rs	especificador de registro fuente de 5 bits.
rt	registro target (fuente/destino) o condición salto 5 bits.
immediate	valor inmediato de 16 bits, desplazamiento de salto o desplazamiento de dirección.
target	dirección de salto de 26 bits.
rd	especificador de registro destino de 5 bits.
sa	Valor de desplazamiento de 5 bits.
funct	Campo de función de 6 bits.

Figura 2.- Formatos de instrucción de la CPU.

El set de instrucciones puede dividirse en los siguientes grupos:

- **Carga y Almacenamiento:** Mueven datos entre la memoria y los registros de propósito general. Son de tipo inmediato (I-Type), si el modo de direccionamiento que soportan es registro base mas desplazamiento inmediato con signo de 16 bits.
- **Computacionales:** Realizan operaciones aritméticas, lógicas, de desplazamiento, de multiplicación y división con los valores de los registros. Estas operaciones incluyen formatos de instrucciones de Registro (R-Type, en los que ambos operandos y el resultado están almacenados en los registros) e Inmediato (I-Type, en las que un operando es un valor inmediato de 16 bits).
- **Saltos absolutos y relativos:** Estas instrucciones cambian el flujo de control de un programa. En saltos absolutos (Jumps), la dirección se forma poniendo los 26 bits del campo target como los bits de orden alto del Contador de Programa (J-Type) o la dirección de un registro (R-Type). Los saltos relativos (Branches) tienen un offset de 16 bits relativo al contador de programa (I-Type). Las instrucciones de Salto absoluto y enlace guardan su dirección de retorno en el registro 31.
- **Coprocesador:** Estas instrucciones realizan operaciones en los coprocesadores. Las instrucciones de carga y almacenamiento del coprocesador son I-Type.
- **Coprocesador 0:** Estas instrucciones realizan operaciones en los registros del CP0 para controlar el manejo de la memoria y el tratamiento de las excepciones.
- **Especiales:** Estas instrucciones realizan llamadas al sistema y operaciones de punto de parada (breakpoint). Estas instrucciones son siempre R-Type.
- **Excepciones:** Estas instrucciones causan un salto relativo al vector que trata las excepciones basado en el resultado de una comparación.

Estas instrucciones tienen los formatos R-Type (los operandos y el resultado son registros) y I-Type (un operando es un valor inmediato de 16 bits).

A continuación vamos a mostrar la lista de instrucciones de la CPU:

Instrucciones de Carga y Almacenamiento.

OpCode	Descripción
LB	Load Byte
LBU	Load Byte Unsigned
LH	Load Halfword
LHU	Load Halfword Unsigned
LW	Load Word
LWL	Load Word Left
LWR	Load Word Right
SB	Store Byte
SH	Store Halfword
SW	Store Word
SWL	Store Word Left
SWR	Store Word Right

Instrucciones Aritméticas (dato inmediato).

Opcode	Descripción
ADDI	Add Immediate
ADDIU	Add Immediate Unsigned
SLTI	Set on Less Than Immediate
SLTIU	Set on Less Than Immediate Unsigned
ANDI	AND Immediate
ORI	OR Immediate
XORI	Exclusive Or Immediate
LUI	Load Upper Immediate

Instrucciones Aritméticas (3 operandos, R-Type).

OpCode	Descripción
ADD	Add
ADDU	Add Unsigned
SUB	Subtract
SUBU	Subtract Unsigned
SLT	Set on Less Than
SLTU	Set on Less Than Unsigned
AND	AND
OR	OR
XOR	Exclusive OR
NOR	NOR

Instrucciones de Multiplicación y División.

OpCode	Descripción
MULT	Multiply
MULTU	Multiply Unsigned
DIV	Divide
DIVU	Divide Unsigned
MFHI	Move From HI
MTHI	Move To HI
MFLO	Move From LO
MTLO	Move To LO

Instrucciones de Saltos absolutos y relativos.

OpCode	Descripción
J	Jump
JAL	Jump And Link
JR	Jump Register
JALR	Jump And Link Register
BEQ	Branch on Equal
BNE	Branch on Not Equal
BLEZ	Branch on Less Than or Equal to Zero
BGTZ	Branch on Greater Than Zero
BLTZ	Branch on Less Than Zero
BGEZ	Branch on Greater Than or Eq. to Zero
BLTZAL	Branch on Less Than Zero And Link
BGEZAL	Brnch on Less Thn Zro And Link Likely

Instrucciones de desplazamiento de bits.

OpCode	Descripción
SLL	Shift Left Logical
SRL	Shift Right Logical
SRA	Shift Right Arithmetic
SLLV	Shift Left Logical Variable
SRLV	Shift Right Logical Variable
SRAV	Shift Right Arithmetic Variable

Instrucciones del Coprocesador.

OpCode	Descripción
LWCz	Load Word to Coprocessor z
SWCz	Store Word from Coprocessor z
MTCz	Move To Coprocessor z
MFCz	Move From Coprocessor z
CTCz	Move Control to Coprocessor z
CFCz	Move Control From Coprocessor z
COPz	Coprocessor Operation z
BCzT	Branch on Coprocessor z True
BCzF	

Instrucciones especiales.

OpCode	Descripción
SYSCALL	System Call
BREAK	Break

Extensiones a ISA: Instrucciones de carga y almacenamiento.

OpCode	Descripción
LD	Load Doubleword
LDL	Load Doubleword Left
LDR	Load Doubleword Right
LL	Load Linked
LLD	Load Linked Doubleword
LWU	Load Word Unsigned
SC	Store Conditional
SCD	Store Conditional Doubleword
SD	Store Doubleword
SDL	Store Doubleword Left
SDR	Store Doubleword Right
SYNC	Sync

Extensiones a ISA: Instrucciones Aritméticas (dato inmediato).

OpCode	Descripción
DADDI	Doubleword Add Immediate
DADDIU	Doubleword Add Immediate Unsigned

Extensiones a ISA: Instrucciones de Multiplicación y División.

OpCode	Descripción
DMULT	Doubleword Multiply
DMULTU	Doubleword Multiply Unsigned
DDIV	Doubleword Divide
DDIVU	Doubleword Divide Unsigned

Extensiones a ISA: Instrucciones de salto relativo.

OpCode	Descripción
BEQL	Branch on Equal Likely
BNEL	Branch on Not Equal Likely
BLEZL	Branch on Less Than or Equal to Zro Likely
BGTZL	Branch on Greater Than Zero Likely
BLTZL	Branch on Less Than Zero Likely
BGEZL	Brnch on Greatr Thn or Equal to Zro Likely
BLTZALL	Branch on Less Than Zero And Link Likely
BGEZALL	Branch on Greater Than or Equal to Zero And Link Likely
BCzTL	Branch on Coprocessor z True Likely
BCzFL	Branch on Coprocessor z False Likely

Extensiones a ISA: Instrucc. Aritméticas (3 operandos, R-Type).

OpCode	Descripción
DADD	Doubleword Add
DADDU	Doubleword Add Unsigned
DSUB	Doubleword Subtract
DSUBU	Doubleword Subtract Unsigned

Extensiones a ISA: Instrucciones de desplazamiento de bits.

OpCode	Descripción
DSLL	Doubleword Shift Left Logical
DSRL	Doubleword Shift Right Logical
DSRA	Doubleword Shift Right Arithmetic
DSLLV	Doublewrd Shift Left Logical Variable
DSRLV	Doublwrd Shift Right Logical Variable
DSRAV	Doublwrd Shift Right Arithmetic Variable
DSLL32	Doubleword Shift Left Logical + 32
DSRL32	Doubleword Shift Right Logical + 32
DSRA32	Doubleword Shift Right Arithmetic + 32

Estensiones a ISA: Instrucciones de excepciones.

OpCode	Descripción
TGE	Trap if Greater Than or Equal
TGEU	Trap if Greater Than or Equal Unsigned
TLT	Trap if Less Than
TLTU	Trap if Less Than Unsigned
TEQ	Trap if Equal
TNE	Trap if Not Equal
TGEI	Trap if Greater Than or Equal Immediate
TGEIU	Trap if Greater Than or Equal Immediate Unsigned
TLTI	Trap if Less Than Immediate

TLTIU	Trap if Less Than Immediate Unsigned
TEQI	Trap if Equal Immediate
TNEI	Trap if Not Equal Immediate

Estensiones a ISA: Instrucciones del Coprocesador.

OpCode	Descripción
DMFCz	Doubleword Move From Coprocessor z
DMTCz	Doubleword Move To Coprocessor z
LDCz	Load Double Coprocessor z
SDCz	Store Double Coprocessor z

Extensiones a ISA: Instrucciones del CP0.

OpCode	Descripción
DMFC0	Doubleword Move From CP0
DMTC	Doubleword Move To CP0
MTC	Move to CP0
MFC	Move from CP0
TLBR	Read Indexed TLB Entry
TLBWI	Write Indexed TLB Entry
TLBWR	Write Random TLB Entry
TLBP	Probe TLB for Matching Entry
CACHE	Cache Operation
ERET	Exception Return

FORMATOS DE DATOS Y DIRECCIONAMIENTO

El R-4000 usa cuatro formatos de datos: una doble palabra de 64 bits (doubleword), una palabra de 32 bits (word), una media palabra de 16 bits (halfword), y un byte de 8 bits. La ordenación de los bytes en cada uno de los formatos de datos (halfword, word, doubleword) puede configurarse en big-endian o little-endian. Esto se refiere a la localización del byte 0 dentro de una estructura de datos de muchos bytes. Las figuras 3 y 4 muestran la ordenación de los bytes dentro de palabras y la ordenación de las palabras dentro de estructuras de multiples palabras para big-endian y little-endian.

Cuando el R4000 se configura como un sistema big-endian, el byte 0 es el byte más significativo, proporcionando compatibilidad con MC 68000® e IBM 370®. La figura 3 muestra esta configuración:

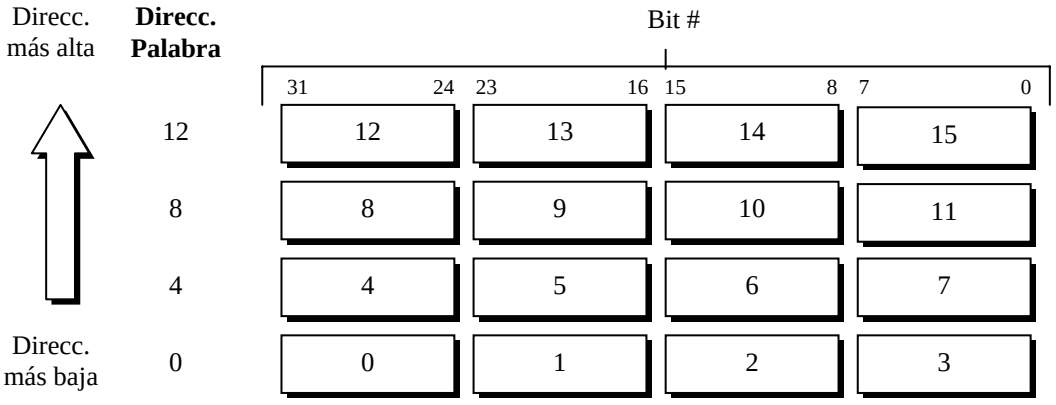


Figura 3.- Ordenación de bytes en big-endian.

Cuando está configurado como un sistema en little-endian, el byte 0 es siempre el byte menos significativo, lo cual es compatible con iAPX® x86 y DEC VAX®. La figura 4 muestra esta configuración:

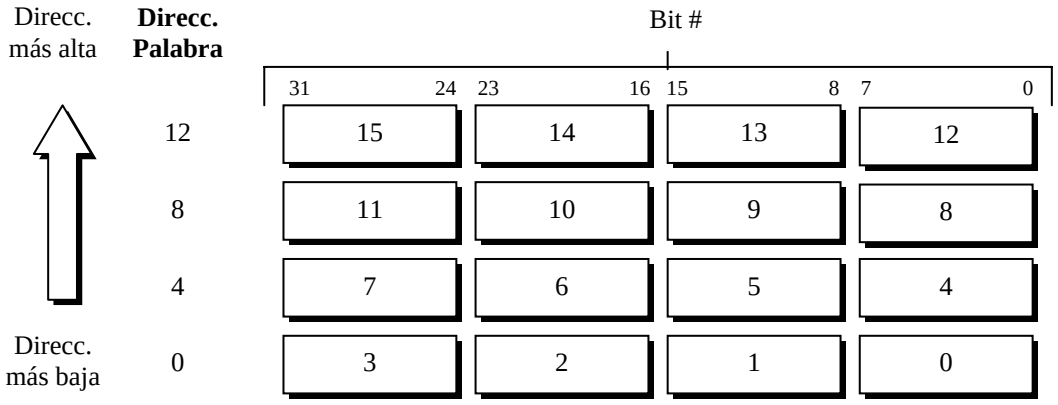


*Figura 4.- Ordenación de bytes en little-*endian*.*

La CPU usa direccionamiento para media palabra (halfword), palabra (word) y doble palabra (doubleword) con las siguientes restricciones de alineación:

- Los accesos a media palabra (halfword) deben estar alineados en un límite de byte divisible por dos (0, 2, 4...).
- Los accesos a palabras (word) deben estar alineados en un límite de byte divisible por cuatro (0, 4, 8...).
- Los accesos a dobles palabras (doubleword) deben alinearse en un límite de byte divisible por ocho (0, 8, 16...).

Las siguientes instrucciones especiales cargan y almacenan palabras que no están alineadas en límites de 4 bytes (word) ó 8 bytes (doubleword):

LWL **LWR** **SWL** **SWR**

LDL **LDR** **SDL** **SDR**

Estas instrucciones se usan en pareja para proporcinar el direccionamiento de palabras no alineadas.

Las figuras 5 y 6 muestran el acceso a una palabra no alineada que tiene dirección de byte 3.

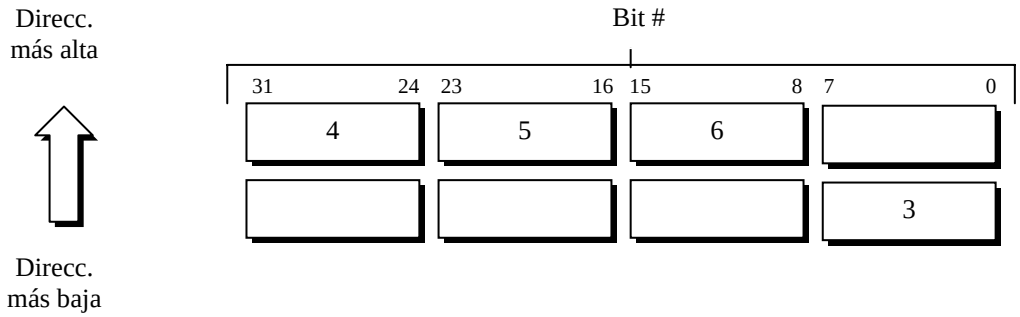


Figura 5.- Direccionamiento de una palabra no alineada Big-endian.

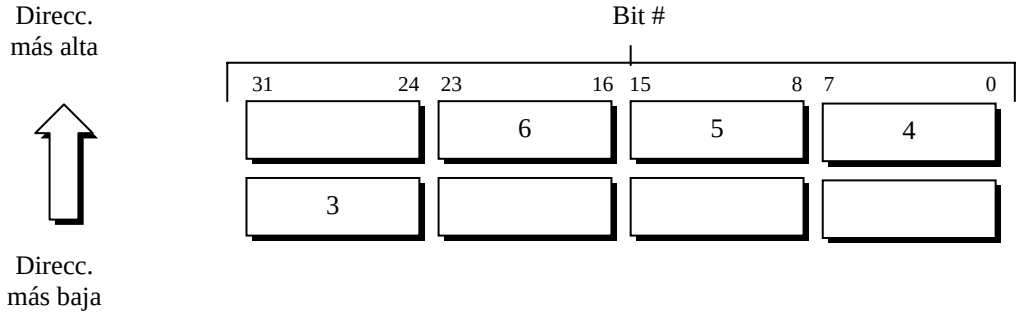


Figura 6.- Direccionamiento de una palabra no alineada Little-endian.

COPROCESADORES (CP0-CP2)

MIPS ISA define tres coprocesadores (De CP0 a CP2):

- El coprocesador 0 **(CP0)** está incorporado en el chip de la CPU y soporta el sistema de memoria virtual y el manejo de las excepciones. CP0 también se llama Coprocesador de control del sistema.
- El coprocesador 1 **(CP1)** está reservado en el chip, para la unidad de coma flotante (FPU).
- El coprocesador 2 (CP2) está reservado para definiciones futuras por MIPS.

Coprocesador de Control del Sistema (CP0)

CP0 traduce las direcciones virtuales a direcciones físicas y maneja excepciones y transiciones entre los estados de kernel, supervisor, y usuario.

CP0 también controla el subsistema de caché, así como también proporciona facilidades para el control de diagnostico y la recuperación de errores.

Los registros del CP0 se muestran en la figura 7. Estos registros son capaces de manipular el manejo de la memoria y el tratamiento de las excepciones de la CPU.

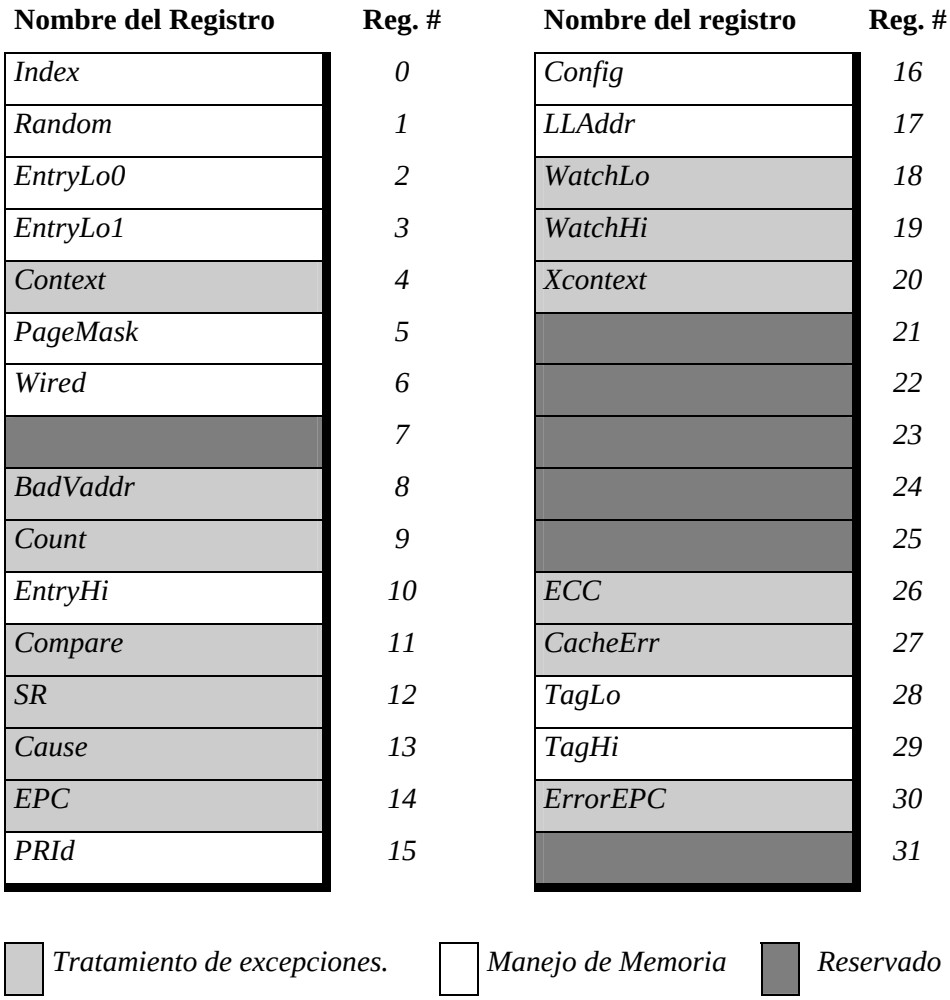


Figura 7.- Registros del CP0.

En la siguiente tabla se describen brevemente los registros del CP0.

Nº	Registro	Descripción
0	Index	Puntero programable dentro del array TLB.
1	Random	Puntero Seudorandom dentro del array TLB (solo lectura).
2	EntryLo0	Parte baja de la entrada del TLB para direcciones virtuales pares (VPN).
3	EntryLo1	Parte alta de la entrada del TLB para direcciones virtuales impares (VPN).
4	Context	Puntero a la entrada de la tabla de página virtual (PTE) en modo de direccionamiento 32 bits.
5	PageMask	Mascara de página del TLB.

6	Wired	Número de entradas del TLB reservadas para uso exclusivo del SSOO.
7	-	Reservado.
8	BadVAddr	Posee dirección virtual mas reciente que causó excepciones determinadas.
9	Count	Contador de tiempo.
10	EntryHi	Parte alta de la entrada del TLB.
11	Compare	Comparador de tiempo.
12	SR	Registro de estado.
13	Cause	Causa de la última excepción.
14	EPC	Contador de programa de excepciones.
15	PRId	Identificador de revisión del procesador.
16	Config	Registro de configuración del procesador.
17	LLAddr	Contiene la direcc. física leida por la instr. de carga y enlace mas reciente.
18	WatchLo	Operac. de carga y almac. especificadas causan una excepc. (parte baja).
19	WatchHi	Parte alta.
20	Xcontext	Puntero a la tabla virtual kernel PTE en modo de direccionam. 64 bits.
21-25	-	Reservado
26	ECC	Chequeo y corrección de errores (ECC) de la cache secundaria y paridad de la primaria.
27	CacheErr	Registro que contiene el indice de cache y bits de estado que indican la fuente y la naturaleza del error.
28	TagLo	Contienen la etiqueta y paridad de la caché primaria, ó la etiqueta y ECC durante procesamiento de inicialización, diagnóstico o error de la caché 2ª.
29	TagHi	Parte alta.
30	ErrorEPC	Igual que EPC pero usado en ECC y excepciones de error de paridad.
31	-	Reservado.

EL TLB (TRANSLATION LOOKASIDE BUFFER)

Las direcciones virtuales mapeadas son traducidas en direcciones físicas usando un TLB en el chip. El TLB es una memoria totalmente asociativa de 48 entradas, que proporciona mapeado a 48 parejas de páginas impares/pares (96 páginas).

Cada entrada de TLB se chequea simultáneamente para un acierto con la dirección virtual que se amplia con un almacenamiento en ASID en el registro EntryHi.

Aciertos y fallos

Si hay un acierto de la dirección virtual en el TLB, el número de página física se extrae del TLB y se concatena con el desplazamiento (offset) para formar la dirección física (ver figura 8).

Si no hay ningún acierto, ocurre una excepción y el software rellena el TLB de la tabla de páginas residente en memoria. El software puede escribir sobre una entrada de TLB seleccionada o usar un mecanismo hardware para escribir en una entrada aleatoria.

Si hay varios aciertos, el TLB se desactiva poniendo a 1 el bit TS del registro de estado.

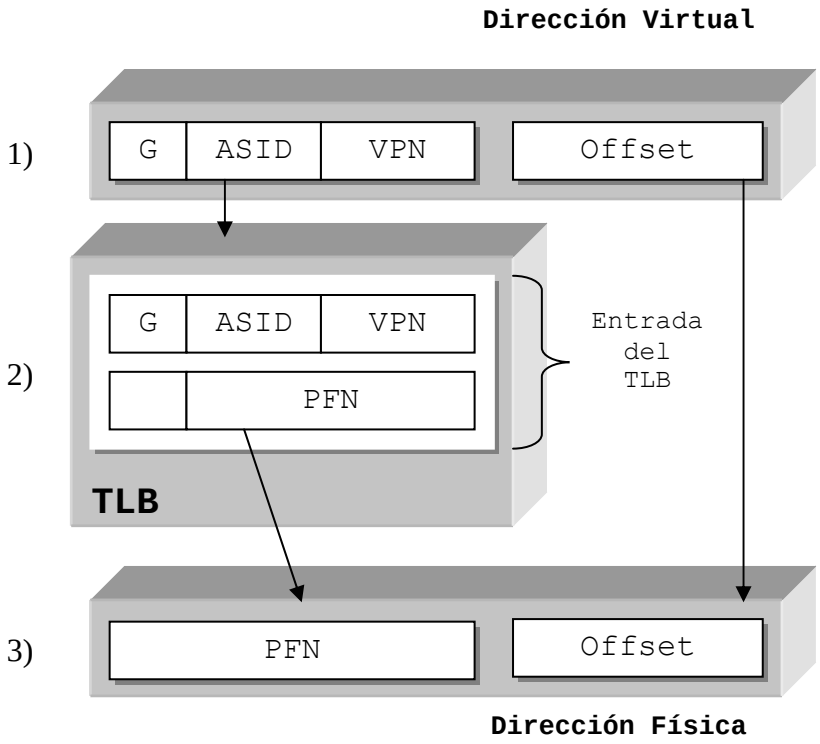


Figura 8.- Traducción de dirección virtual a física

1. La dirección virtual (VA) representada por el número de página virtual (VPN) se compara con la etiqueta en el TLB.
2. Si hay un acierto, el número de marco de página (PFN) representado por los bits más altos de la dirección física (PA) es la salida del TLB.
3. El desplazamiento (offset), que no pasa a través del TLB, se concatena con el PFN.

Espacio de direcciones Virtuales

La dirección virtual del procesador puede ser de 32 ó 64 bits de ancho, dependiendo de si el procesador está operando en modo 32 bits o en modo 64 bits.

- En modo 32 bits, las direcciones son de 32 bits de ancho. El tamaño máximo es de 2 gigabytes (2^{31}).
- En modo 64 bits, las direcciones son de 64 bits de ancho. El máximo tamaño es de 1 terabyte (2^{40}).

Espacio de direcciones Físicas

Usando una dirección de 36 bits, el espacio de la dirección física del procesador abarca 64 gigabytes.

MODOS DE OPERACIÓN

El procesador tiene tres modos de operación que funcionan en operaciones de 32 y 64 bits y son:

- Modo usuario.
- Modo supervisor.
- Modo kernel.

Operaciones en modo usuario

En modo usuario, está permitido un espacio de direcciones virtuales uniforme; su tamaño es:

- 2 Gbytes (2^{31} bytes) en modo 32 bits (useg).
- 1 Tbyte (2^{40} bytes) en modo 64 bits (xuseg).

La figura 9 muestra el espacio de direcciones virtuales en modo Usuario:

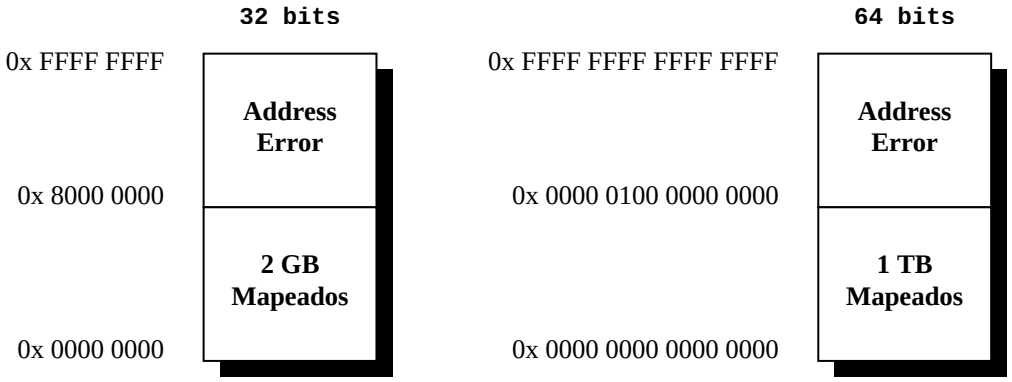


Figura 9.- Espacio de direcciones virtuales en modo usuario.

En la tabla que podemos ver a continuación, observamos las características de los dos segmentos en modo usuario, useg y xuseg.

Valores de los bits de la dirección	Valores de bits del registro de estado				Nombre segmento	Rango de direcciones	Tamaño segmento
	KSU	EXL	ERL	UX			
32 bits A(31)=0	10_2	0	0	0	useg	0x00000000 0x7FFFFFFF	2 Gbyte (2^{31} bytes)
64 bits A(63:40)=0	10_2	0	0	1	xuseg	0x0000000000000000 0x000000FFFFFFFFFF	1 Tbyte (2^{40} bytes)

Operaciones en modo Supervisor

El modo supervisor está designado para sistemas operativos en los que un kernel verdadero corre en modo kernel, y el resto del sistema operativo corre en modo supervisor.

La figura 10 muestra el espacio de direcciones virtuales en modo Supervisor:

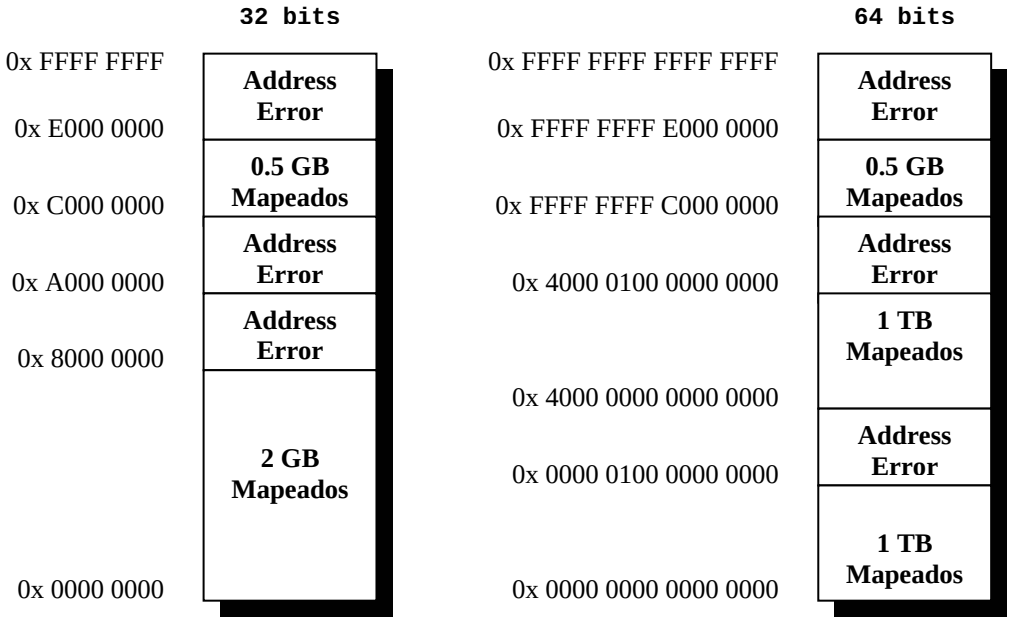


Figura 10.- Espacio de dirs. virtuales modo supervisor.

A continuación vamos a ver la tabla de los segmentos en modo Supervisor.

Valores de los bits de la dirección	**Valores de bits del registro de estado**				**Nombre segmento**	**Rango de direcciones**	**Tamaño segmento**
	KSU	**EXL**	**ERL**	**SX**			
32 bits A(31)=0	01_2	0	0	0	suseg	0x00000000 0x7FFFFFFF	2 Gbyte (2^{31} bytes)
32 bits A(31:29)=110_2	01_2	0	0	0	ssseg	0xC000 0000 0xDFFF FFFF	512 MB (2^{29} bytes)
64 bits A(63:32)=00_2	01_2	0	0	1	xsuseg	0x0000000000000000 0x000000FFFFFFFFFF	1 Tbyte (2^{40} bytes)
64 bits A(63:32)=01_2	01_2	0	0	1	xsseg	0x4000000000000000 0x400000FFFFFFFFFF	1 Tbyte (2^{40} bytes)
64 bits A(63:32)=11_2	01_2	0	0	1	csseg	0xFFFFFFFFC0000000 0xFFFFFFFFDFFFFFFF	512 MB (2^{29} bytes)

Operaciones en modo Kernel

El procesador entra en modo Kernel siempre que se produce una excepción y permanece en modo Kernel hasta que se ejecuta una instrucción de Retorno de Excepción (ERET). La instrucción ERET restaura el procesador al modo que tenía antes de la excepción.

La figura 11 muestra el espacio de direcciones virtuales en modo Kernel:

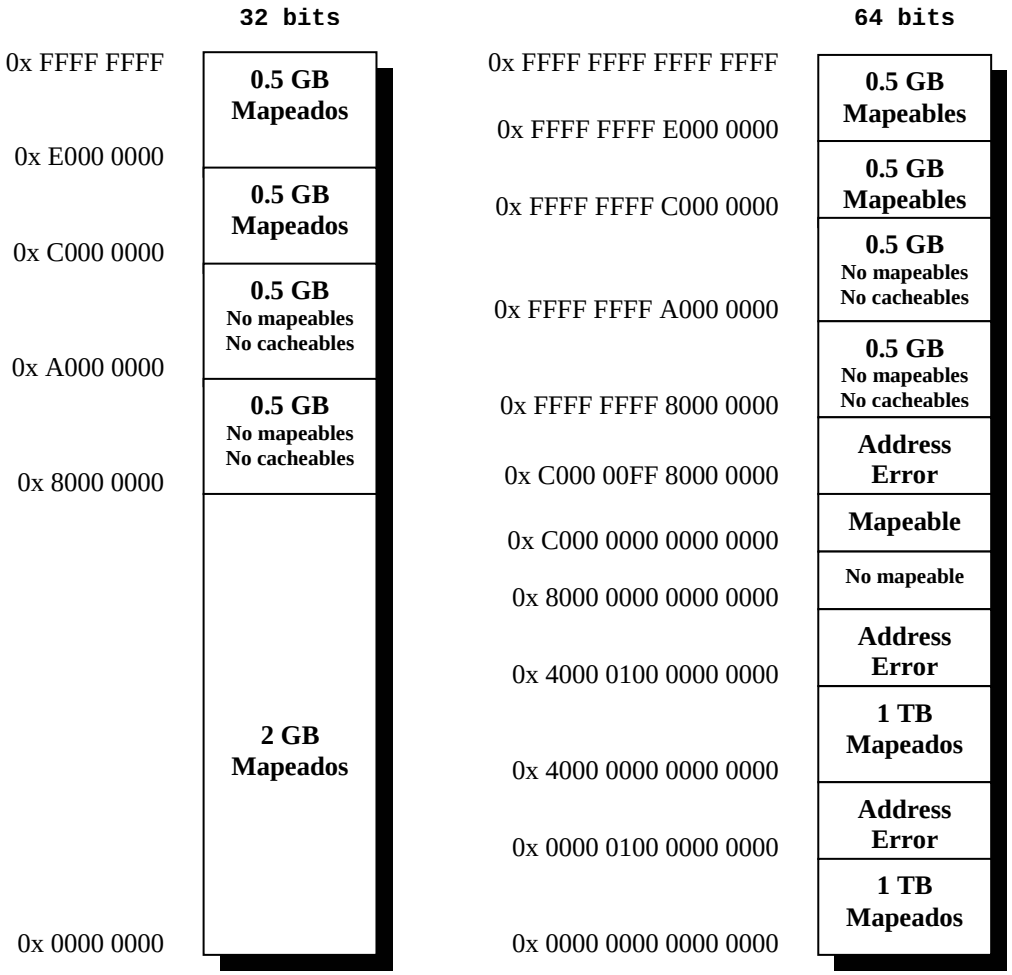


Figura 11.- Espacio de direcciones virtuales en modo Kernel.

A continuación vamos a ver la tabla de segmentos en modo Kernel 32 bits:

Valores de los bits de la dirección	**Valores de bits del registro de estado**				**Nombre segmento**	**Rango de direcciones**	**Tamaño segmento**
	KSU	**EXL**	**ERL**	**KX**			
A(31)=0	KSU=00_2 or EXL=1 or ERL=1			0	kuseg	0x0000 0000 0x7FFF FFFF	2 GB (2^{31} bytes)
A(31:29)=100_2				0	kseg0	0x8000 0000 0x9FFF FFFF	512 MB (2^{29} bytes)
A(31:29)=101_2				0	kseg1	0xA000 0000 0xBFFF FFFF	512 MB (2^{29} bytes)
A(31:29)=110_2				0	ksseg	0xC000 0000 0xDFFF FFFF	512 MB (2^{29} bytes)
A(31:29)=111_2				0	kseg3	0xE000 0000 0xFFFF FFFF	512 MB (2^{29} bytes)

En la siguiente tabla veremos los segmentos en modo Kernel 64 bits:

Valores de los bits de la dirección	**Valores de bits del registro de estado**				**Nombre segmento**	**Rango de direcciones**	**Tamaño segmento**
	KSU	**EXL**	**ERL**	**KX**			
$A(63:62)=00_2$	$KSU=00_2$ Or EXL=1 Or ERL=1			1	xksuseg	0x0000000000000000 0x000000FFFFFFFFFF	1 Tbyte (2^{40} bytes)
$A(63:62)=01_2$				1	xksseg	0x4000000000000000 0x400000FFFFFFFFFF	1 Tbyte (2^{40} bytes)
$A(63:62)=10_2$				1	xkphys	0x8000000000000000 0xBFFFFFFFFFFFFFFF	8 2^{36}-byte espacios
$A(63:62)=11_2$				1	xkseg	0xC000000000000000 0xC00000FF7FFFFFFF	(2^{40}-2^{31}) bytes
$A(63:62)=11_2$ $A(61:31)=-1$				1	ckseg0	0xFFFFFFFF80000000 0xFFFFFFFF9FFFFFFF	512 MB (2^{29} bytes)
$A(63:62)=11_2$ $A(61:31)=-1$				1	ckseg1	0xFFFFFFFFA0000000 0xFFFFFFFFBFFFFFFF	512 MB (2^{29} bytes)
$A(63:62)=11_2$ $A(61:31)=-1$				1	cksseg	0xFFFFFFFFC0000000 0xFFFFFFFFDFFFFFFF	512 MB (2^{29} bytes)
$A(63:62)=11_2$ $A(61:31)=-1$				1	ckseg3	0xFFFFFFFFE0000000 0xFFFFFFFFFFFFFFFF	512 MB (2^{29} bytes)

CÓMO TRABAJA EL PROCESAMIENTO DE EXCEPCIONES

El procesador recibe excepciones de un número de fuentes, incluyendo los fallos del TLB (Translation Lookaside Buffer), desbordamientos aritméticos, interrupciones de Entrada/Salida, y llamadas al sistema. Cuando la CPU detecta una de estas excepciones, la secuencia normal de la ejecución de la instrucción se suspende y el procesador entra en modo Kernel.

Entonces, el procesador desactiva las interrupciones y fuerza la ejecución de una excepción software localizada en una dirección fija. Se salva el contexto del

procesador, incluyendo los contenidos del contador de programa, el modo de operación actual (Usuario o Supervisor), y el estado de las interrupciones (activadas o desactivadas). Este contexto se salva, por lo tanto puede ser restaurado cuando la excepción haya sido atendida.

Cuando ocurre una excepción, la CPU carga el registro Contador de Programa de Excepción (EPC) con una dirección donde la ejecución puede reiniciarse despues de que la excepción haya sido atendida. La dirección de reinicio en el registro EPC es la dirección de la instrucción que causó la excepción ó, si la instrucción se estaba ejecutando en un "branch delay slot", la dirección de la instrucción de salto relativo inmediatamente anterior al "delay slot".

REGISTROS QUE PROCESAN LAS EXCEPCIONES

Esta sección describe los registros del CP0 que se usan en el procesamiento de excepciones. Cada registro tiene un único número de identificación que se refiere a su número de registro. Por ejemplo, el registro ECC es el registro número 26.

El software examina los registros del CP0 durante el procesamiento de una excepción para determinar la causa de la excepción y el estado de la CPU a la vez que está ocurriendo la excepción. En la siguiente tabla podemos ver los registros que se usan en el procesamiento de las excepciones, y son:

Nombre del Registro	Registro Número
Context	4
BadVAddr (Bad Virtual Address)	8
Count	9
Compare register	11
Status	12
Cause	13
EPC (Exception Program Counter)	14

WatchLo (Memory Reference Trap Address Low)	18
WatchHi (Memory Reference Trap Address High)	19
Xcontext	20
ECC	26
CacheErr (Cache Error and Status)	27
Error EPC (Error Exception Program Counter)	30

TIPOS DE EXCEPCIONES

Los tipos de excepciones son los siguientes:

- Reset.
- Soft Reset.
- Nonmaskable Interrupt (NMI).
- Cache Error.
- Resto de excepciones del procesador.

Cuando el bit EXL en el registro de estado es 0, el modo de operación especificado por los bits KSU en el registro de estado puede ser Usuario, Supervisor ó Kernel. Cuando el bit EXL está a 1, el procesador está en modo Kernel.

Cuando en el procesador se produce una excepción, el bit EXL se pone a 1, lo cual significa que el sistema está en modo Kernel. Después de salvar el estado apropiado, se cambian los bits KSU a modo Kernel y pone el bit EXL otra vez a cero. Cuando se restaura el estado y se reinicia la ejecución, se restaura el valor previo del campo KSU y se pone el bit EXL a 1.

Al volver de una excepción (instrucción ERET), el bit EXL se pone a cero.

LOCALIZACIÓN DE LOS VECTORES DE EXCEPCIÓN

Las excepciones Reset, Soft Reset y NMI, tienen el mismo vector de excepción en una dirección no cacheable y no mapeable. Las direcciones para las demás excepciones son una combinación de un vector desplazamiento (offset) y una dirección base.

Cuando el bit BEV=1 en el registro de estado, los vectores están en direcciones no cacheables y no mapeables. Durante una operación normal (BEV=0) las excepciones normales tienen vectores en espacios de dirección cacheables.

La tabla que se muestra a continuación enseña el vector de dirección base para todas las excepciones en modo 64 bits; las direcciones en modo 32 bits son los 32 bits de orden bajo (por ejemplo, la dirección base para NMI en modo 32 bits es 0xBFC0 0000).

Excepción	**BEV**	
	0	**1**
Cache Error	0xFFFF FFFF A000 0000	0xFFFF FFFF BFC0 0200
Otras	0xFFFF FFFF 8000 0000	0xFFFF FFFF BFC0 0200
Reset, NMI, Soft Reset	0xFFFF FFFF BFC0 0000	

En la tabla que se muestra a continuación podemos ver los desplazamientos de los vectores de excepción:

Excepción	**Vector de desplazamiento**
TLB refill, EXL=0	0x000
XTLB refill, EXL=0 (X=TLB de 64 bits)	0x080
Cache Error	0x100
Otras	0x180
Reset, Soft Reset, NMI	Ninguno.

PRIORIDAD DE LAS EXCEPCIONES

Cuando en la ejecución de una instrucción simple puede ocurrir más de una excepción, solo se atiende la excepción que tiene mayor prioridad. En la siguiente tabla se muestra la prioridad de las excepciones, además de enumerarlas.

Reset (prioridad más alta)
Soft Reset causada por la señal Reset* (* indica señal activa a nivel bajo)
Interrupción No enmascarable (NMI) (Soft reset causada por NMI)
Address error – Búsqueda de la instrucción
TLB refill – Búsqueda de la instrucción
TLB invalid – Búsqueda de la instrucción
Cache error – Búsqueda de la instrucción
Virtual Coherency – Búsqueda de la instrucción
Bus error – Búsqueda de la instrucción
Integer Overflow, Trap, System Call, Breakpoint, Reserved Instruction, Coprocessor Unusable, ó Floating-Point Excepcion.
Address error – Acceso a Datos
TLB refill – Acceso a Datos
TLB invalid – Acceso a Datos
TLB modified – Escritura de Datos
Cache error – Acceso a Datos
Watch
Virtual Coherency – Acceso a Datos
Bus error – Acceso a Datos
Interrupt (prioridad más baja)

UNIDAD DE COMA FLOTANTE

La Unidad de Coma Flotante (FPU) opera como un coprocesador para la CPU (llamado coprocesador CP1), y extiende el set de instrucciones de la CPU para realizar operaciones aritméticas con valores en coma flotante.

La figura 12 ilustra la organización funcional de la FPU.

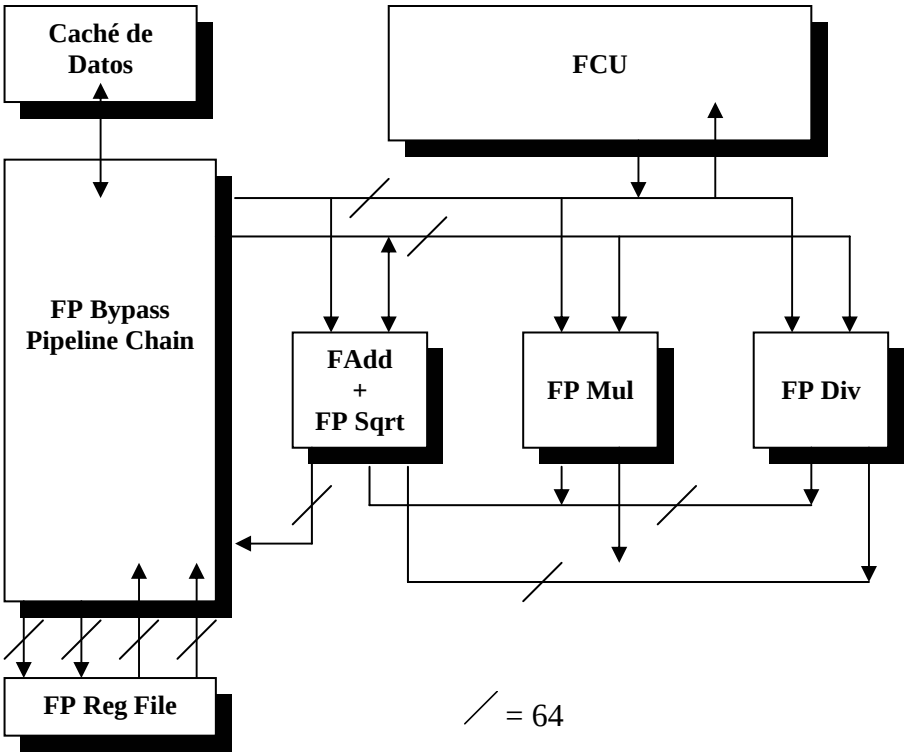


Figura 12.- Diagrama de bloques funcional de la FPU

CARACTERÍSTICAS DE LA FPU

Esta sección describe brevemente el modelo de operación, el set de instrucciones de carga y almacenamiento, y el interface del coprocesador en la FPU.

- **Operaciones de 64 bits.** Cuando el bit FR en el registro de estado de la CPU es igual a cero, la FPU está en modo 32 bits y contiene 32 registros de 32 bits que tienen valores de simple- ó, cuando se usan

en pareja, doble-precisión. Cuando el bit FR en el registro de estado de la CPU es igual a uno, La FPU está en modo 64 bits y los registros se expanden a 64 bits de ancho. Cada registro tiene valores de simple- ó doble-precisión. La FPU también incluye un registro de Control/Estado de 32 bits que proporciona acceso a todas las capacidades de manejo de excepciones Estandar-IEEE.

- **Set de Instrucciones de Carga y Almacenamiento.** Como la CPU, la FPU usa un set de instrucciones orientado a la carga y almacenamiento. Las operaciones en coma flotante comienzan en un ciclo sencillo su ejecución solapa otras operaciones en coma fija ó en coma flotante.
- La FPU y la CPU son un conjunto de procesadores fuertemente acoplados para proporcionar una integración sin fisuras entre los repertorios de instrucciones de enteros y de coma flotante. Puesto que cada unidad recibe y ejecuta instrucciones en paralelo, algunas instrucciones en coma flotante pueden ejecutarse en un ciclo del mismo modo que las instrucciones de enteros.

REGISTROS DE PROPOSITO GENERAL DE LA FPU (FGRs)

La FPU tiene un conjunto de registros de proposito general en coma flotante (FGRs) a los que se puede acceder de las siguientes maneras:

- Como 32 registros de proposito general (32 FGRs), cada uno de ellos es de 32 bits de ancho cuando el bit FR en el registro de estado es igual a cero; ó como 32 registros de proposito general (32 FGRs), cada uno de ellos es de 64 bits de ancho cuando FR es igual a uno. Los accesos de la CPU a estos registros se hace a traves de instrucciones de movimiento, carga y almacenamiento.
- Como 16 registros de coma flotante, cada uno de ellos es de 64 bits de ancho (FPRs), cuando el bit FR en el registro de estado de la CPU

es igual a cero. Los FPRs mantienen valores en formato de simple- ó doble-precisión.

- Como 32 registros de coma flotante, cada uno de ellos es de 64 bits de ancho, cuando el bit FR en el registro de estado de la CPU es igual a uno. Los FPRs mantienen valores en formato coma flotante de simple- ó doble-precisión.

En la figura 13 se muestran los registros de la unidad de coma flotante (FPU).

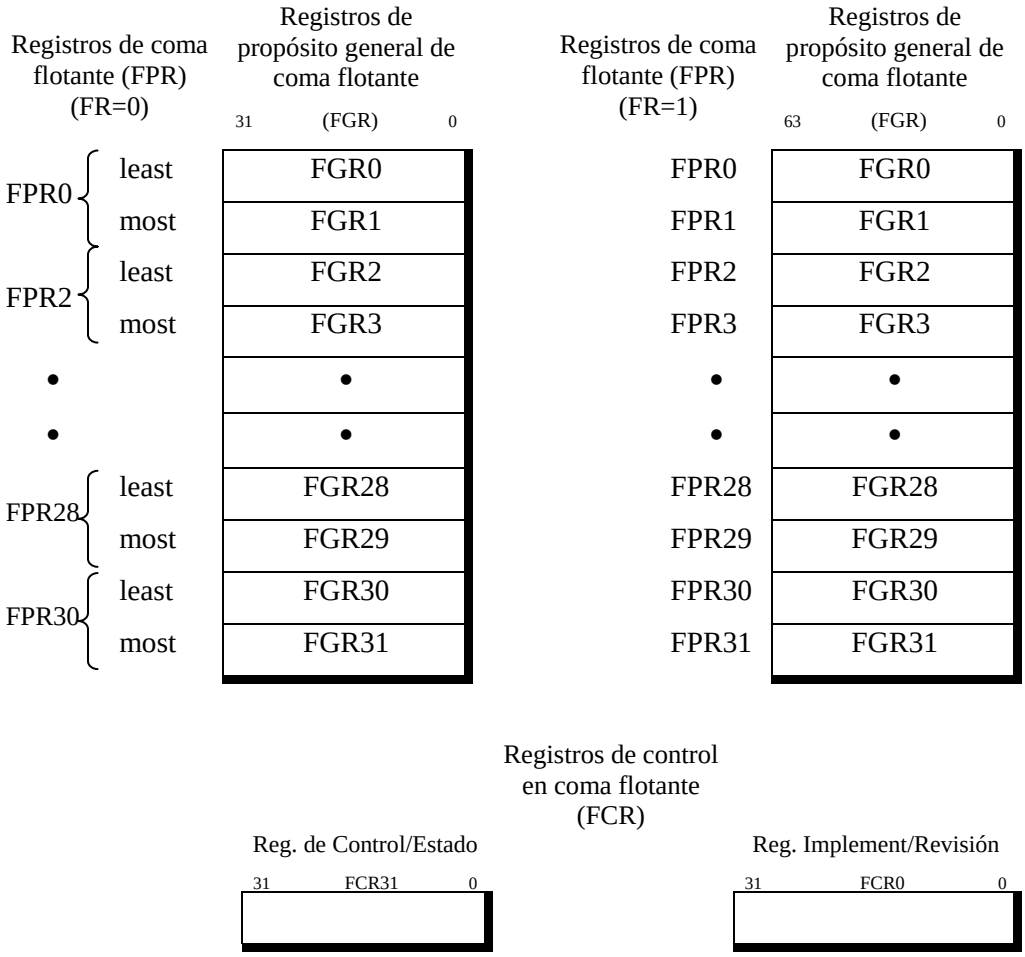


Figura 13.- Registros de la FPU

REGISTROS DE CONTROL EN COMA FLOTANTE

La FPU tiene 32 registros de control (FCRs) a los que solo se puede acceder por operaciones de movimiento.

- El registro de Implementación/Revisión (FCR0) contiene información de revisión sobre la CPU.
- El registro de Control/Estado (FCR31) controla las excepciones y mantiene el resultado de operaciones de comparación.
- De FCR1 hasta FCR30 están reservados.

FORMATOS DE COMA FLOTANTE

La FPU realiza operaciones de 32 bits (simple-precisión) y de 64 bits (doble-precisión). El formato de simple precisión (32 bits) tiene un campo de fracción en signo magnitud de 24 bits (f+s) y un exponente (e) de 8 bits, como vemos en la figura 14.

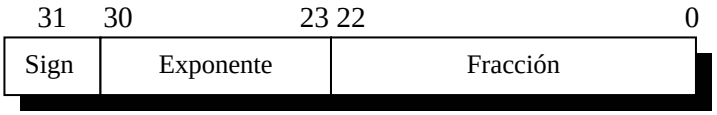


Figura 14.- Formato en coma flotante de Simple-precisión

El formato de doble-precisión (64 bits) tiene un campo de fracción de 53 bits en signo magnitud (f+s) y un exponente de 11 bits, como vemos en la figura 15.

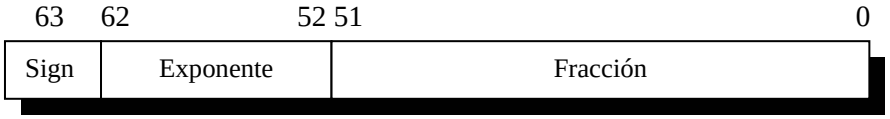


Figura 15.- Formato en coma flotante de Doble-precisión

FORMATO BINARIO EN COMA FIJA

Los valores binarios en coma fija están en formato de complemento a dos. Los valores en coma fija no son directamente proporcionados por el set de instrucciones en coma flotante. La figura 16 ilustra el formato binario en coma fija:

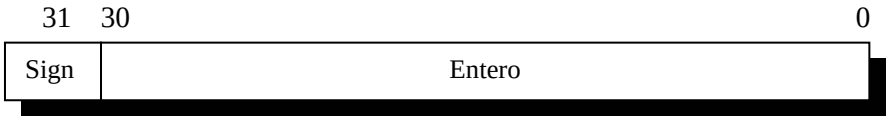


Figura 16.- Formato binario en coma fija

SET DE INSTRUCCIONES EN COMA FLOTANTE

Todas las instrucciones de la FPU son de 32 bits de longitud, alineadas en el límite de una palabra. Pueden ser divididas en los siguientes grupos:

- **Carga, almacenamiento y movimiento:** Estas instrucciones mueven datos entre memoria, el procesador principal y los registros de propósito general de la FPU.
- **Conversión:** Estas instrucciones realizan operaciones de conversión entre varios formatos de datos.
- **Computacionales:** Estas instrucciones realizan operaciones aritméticas de valores en coma flotante en los registros de la FPU.
- **Comparación:** Estas instrucciones realizan comparaciones de los contenidos de los registros y actiúan sobre un bit condicional basado en los resultados.
- **Salto relativo si se cumple la condición:** Estas instrucciones realizan un salto a la dirección especificada si la condición se cumple.

En los formatos de instrucción que vemos en las tablas que se muestran a continuación, el *fmt* adjunto al código de la operación de la instrucción especifica el formato de dato: S especifica coma flotante en binario simple-

precisión, D especifica coma flotante en binario doble-precisión, W especifica coma-fija en binario de 32 bits, y L especifica coma-fija en binario de 64 bits.
En las siguientes tablas muestran el set de instrucciones de coma flotante:

Instrucciones de Carga, Movimiento y Almacenamiento de datos

OpCode	Descripción
LWC1	Load Word to FPU
SWC1	Store Word from FPU
LDC1	Load Doubleword to FPU
SDC1	Store Doubleword From FPU
MTC1	Move Word To FPU
MFC1	Move Word From FPU
CTC1	Move Control Word To FPU
CFC1	Move Control Word From FPU
DMTC1	Doubleword Move To FPU
DMFC1	Doubleword Move From FPU

Instrucciones de Conversión de datos

OpCode	Descripción
CVT.S.fmt	Floating-point Convert to Single FP
CVT.D.fmt	Floating-point Convert to Double FP
CVT.W.fmt	Floating-point Convert to 32-bit Fixed Point
CVT.L.fmt	Floating-point Convert to 64-bit Fixed Point
ROUND.W.fmt	Floating-point Round to 32-bit Fixed Point
ROUND.L.fmt	Floating-point Round to 64-bit Fixed Point
TRUNC.W.fmt	Floating-point Truncate to 32-bit Fixed Point
TRUNC.L.fmt	Floating-point Truncate to 64-bit Fixed Point
CEIL.W.fmt	Floating-point Ceiling to 32-bit Fixed Point
CEIL.L.fmt	Floating-point Ceiling to 64-bit Fixed Point
FLOOR.W.fmt	Floating-point Floor to 32-bit Fixed Point

FLOOR.L.fmt	Floating-point Floor to 64-bit Fixed Point

Instrucciones Computacionales

OpCode	Descripción
ADD.fmt	Floating-point Add
SUB.fmt	Floating-point Subtract
MUL.fmt	Floating-point Multiply
DIV.fmt	Floating-point Divide
ABS.fmt	Floating-point Absolute Value
MOV.fmt	Floating-point Move
NEG.fmt	Floating-point Negate
SQRT.fmt	Floating-point Square Root

Instrucciones de Salto relativo y Comparación

OpCode	Descripción
C.cond.fmt	Floating-point Compare
BC1T	Branch on FPU True
BC1F	Branch on FPU False
BC1TL	Branch on FPU True Likely
BC1FL	Branch on FPU False Likely

EXCEPCIONES EN COMA FLOTANTE

El registro de Control/Estado de la FP contiene un bit Enable para cada tipo de excepción.

ORGANIZACIÓN DE LA CACHÉ Y OPERACIÓNES

Vamos a usar la siguiente terminología:

- La caché primaria será referida como la P-caché.
- La caché secundaria será referida como la S-caché.

- La caché de datos primaria será referida como la D-caché.
- La caché de instrucciones primaria será referida como la I-caché.

ORGANIZACIÓN DE LA MEMORIA

La figura 17 muestra la jerarquia del sistema de memoria del R-4000. Las cachés están situadas entre la CPU y la memoria principal. Están diseñadas para acelerar los accesos a memoria y son transparentes al usuario. Cada bloque funcional en la figura 17 tiene la capacidad de almacenar más datos que el bloque situado sobre él. Por ejemplo, la memoria principal tiene una mayor capacidad que la caché secundaria. Al mismo tiempo, cada bloque funcional tarda más en acceder a los datos que el bloque situado sobre él. Por ejemplo, se tarda más en acceder a los datos en la memoria principal que en los registros de la CPU.

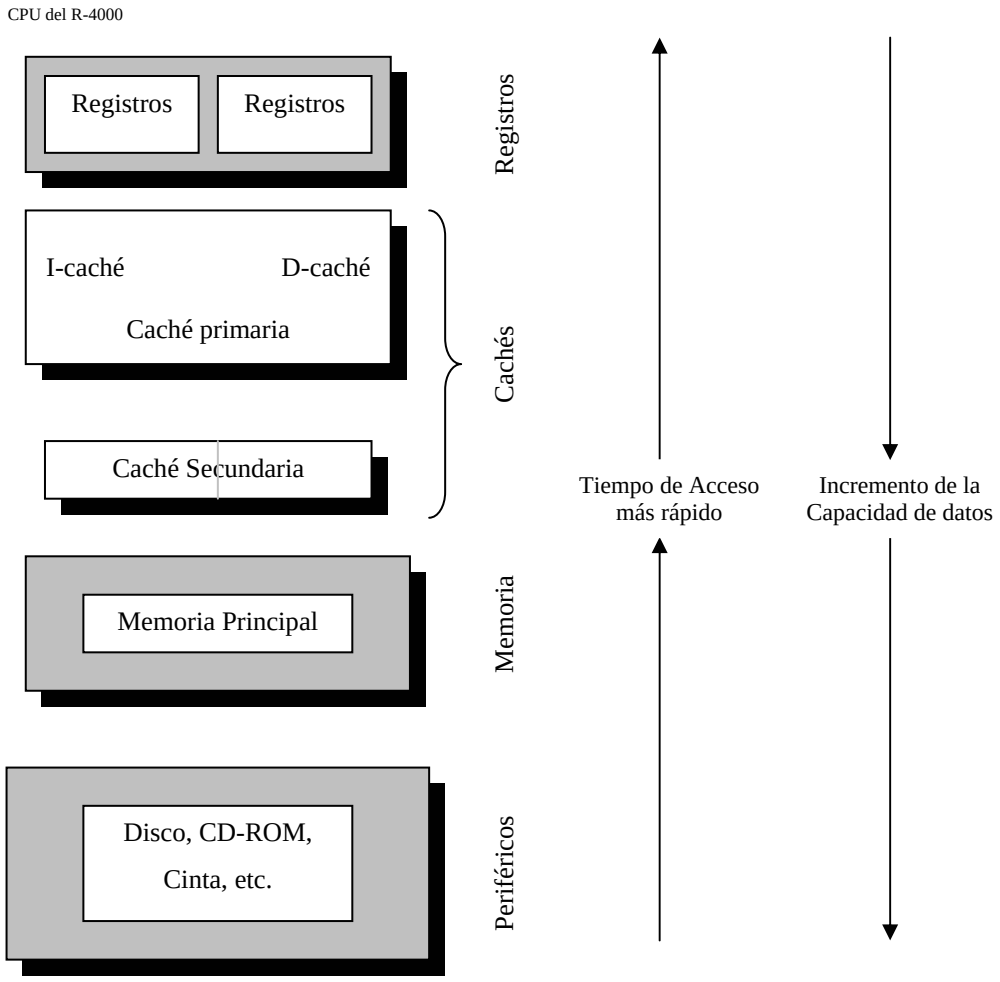


Figura 17.- Jerarquía de Memoria

El R-4000 tiene dos cachés primarias en el chip; una contiene instrucciones (la caché de instrucciones) y la otra contiene datos (la caché de datos). Fuera del chip, el R-4000 tiene una caché secundaria, pero solo en los modelos R4000SC y R4000MC.

OPERACIONES DE LA CACHÉ

Las cachés proporcionan un rápido almacenamiento temporal de datos. En general, el procesador accede a instrucciones ó datos a través de los siguientes procedimientos:

1. El procesador, a través de un controlador de caché en el chip, intenta acceder a la siguiente instrucción ó dato en la caché primaria.
2. El controlador de caché hace un chequeo para ver si esta instrucción o dato está presente en la caché primaria.
   - Si la instrucción/dato está presente, el procesador la coge. Esto es lo que se llama un acierto de caché primaria.
   - Si la instrucción/dato no está presente en la caché primaria, el controlador de caché debe cogerla de la caché secundaria o de memoria. Esto es lo que se llama un fallo de caché primaria.
3. Si ocurre un fallo de caché primaria, el controlador de caché hace un chequeo para ver si la instrucción/dato está en la caché secundaria.
   - Si la instrucción/dato está presente en la caché secundaria, se coge y se escribe en la caché primaria.
   - Si la instrucción/dato no está presente en la caché secundaria, el dato se coge de la memoria como si fuera una linea de caché y se escribe en la caché secundaria y en la caché primaria.
4. El procesador coge la instrucción/dato de la caché primaria y la operación continúa.

Es posible que el mismo dato esté en tres lugares simultáneamente: la memoria principal, la caché secundaria y la caché primaria. Utiliza la política de post-escritura (write-back) que consiste en que el dato modificado en la caché primaria no se escribe en memoria hasta que la línea de caché sea reemplazada.

DESCRIPCIÓN DE LA CACHÉ DEL R4000

Como se mostraba en la figura 17, el R4000 contiene caches de instrucciones y datos separadas. La figura 17 también mostraba que el R4000 soporta una caché secundaria que puede estar dividida en porciones separadas, una porción contene datos y la otra porción contiene instrucciones, o puede ser una caché unida, conteniendo instrucciones y datos combinados.

La tabla que se muestra a continuación lista la caché y la coherencia de caché soportada por los tres modelos del R4000.

Modelo R4000	**¿Soporta caché primaria?**	**¿Soporta caché secundaria?**	**¿Soporta coherencia de caché?**
R4000PC	Si	No	No
R4000SC	Si	Si	No
R4000MC	Si	Si	Si

La figura 18 proporciona diagramas de bloques para los tres modelos del R4000:

- R4000PC que soporta solo la caché primaria.
- R4000SC y R4000MC, que soportan cachés primaria y secundaria.

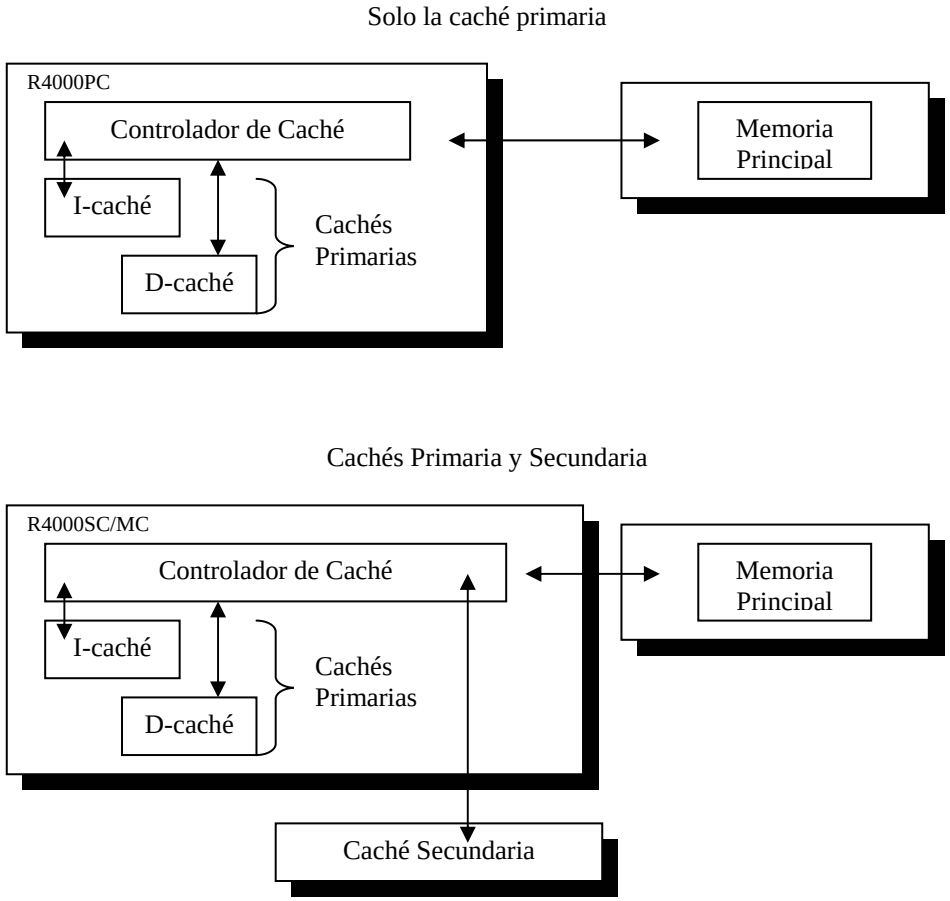


Figura 18.- Caché soportada en el R4000PC, R4000SC, Y R4000MC

Tamaño de la caché secundaria

La tabla que se muestra a continuación lista el rango de tamaños de la caché secundaria. La caché secundaria puede ser una caché unida, conteniendo datos e instrucciones en una simple caché, o puede estar dividida en cachés separadas de datos e instrucciones.

Caché	Tamaño Mínimo	Tamaño Máximo
Caché secundaria unida	128 Kbytes	4 Mbytes
I-Caché secundaria dividida	128 Kbytes	2 Mbytes
D-Caché secundaria dividida	128 Kbytes	2 Mbytes

Lineas de Caché de longitud variable

Una linea de caché es la unidad de información más pequeña que puede ser buscada de la caché, y está representada por una simple etiqueta. Una línea de caché primaria puede tener 4 ó 8 palabras de longitud; una línea de caché secundaria puede tener 4,8,16 ó 32 palabras de longitud.

La longitud de la línea de caché primaria nunca puede ser más larga que la de la caché secundaria; debe ser siempre menor o igual que la longitud de la línea de caché secundaria. Esto significa que la caché secundaria no puede tener una longitud de línea de 4 palabras mientras la caché primaria tenga una longitud de línea de 8 palabras.

Organización y accesibilidad de la Caché

Las cachés primarias de instrucción y datos son indexadas con una dirección virtual (VA), mientras que la caché secundaria es indexada con una dirección física (PA).

Organización de la Caché de Instrucciónes primaria (I-Caché)

Cada línea de datos (que son instrucciones) de la caché de instrucciones primaria tiene una etiqueta asociada de 26 bits que contiene una dirección física de 24 bits, un bit de valido, y un bit de paridad.

La caché de instrucciones primaria del R4000 tiene las siguientes características:

- Mapeado directo.
- Indexado con una dirección virtual.
- Chequeado con una etiqueta física
- Organizado con una línea de caché de 4 ó 8 palabras.

La figura 19 muestra el formato de una línea de caché primaria de 8 palabras (32 bytes).

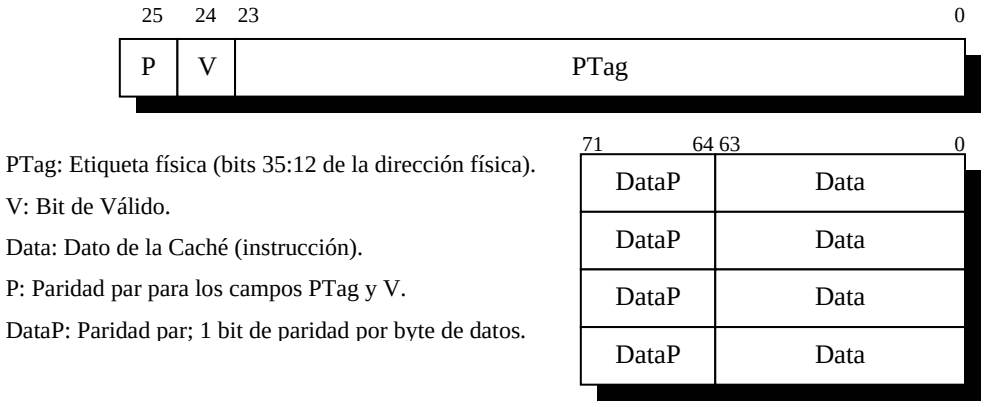


Figura 19.- Formato de Línea de I-caché Primaria de 8 palabras

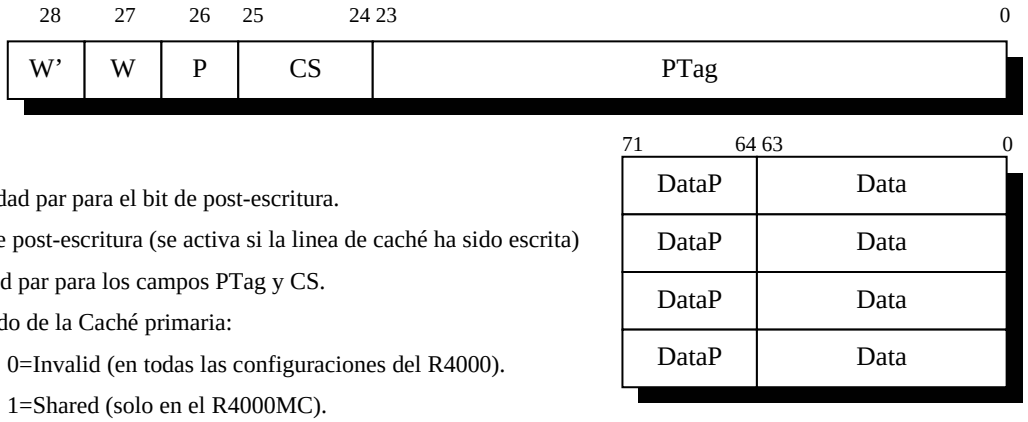
Organización de la Caché de datos Primaria (D-caché)

Cada línea de datos (que son datos) de la caché de datos primaria tiene una etiqueta asociada de 29 bits que contiene una dirección física de 24 bits, 2 bits del estado de línea de caché, un bit de post-escritura (write-back), un bit de paridad para la dirección física y campos de estado de la caché, y un bit de paridad para el bit de post-escritura.

La caché primaria de datos tiene las siguientes características:

- Post-escritura.
- Mapeado directo.
- Indexado con una dirección virtual.
- Chequeado con una etiqueta física.
- Organizado con una línea de caché de 4 ó 8 palabras.

La figura 20 muestra el formato de una línea de caché de datos primaria de 8 palabras (32 bytes).



W': Paridad par para el bit de post-escritura.

W: Bit de post-escritura (se activa si la linea de caché ha sido escrita)

P: Paridad par para los campos PTag y CS.

CS: Estado de la Caché primaria:

0=Invalid (en todas las configuraciones del R4000).

1=Shared (solo en el R4000MC).

2=Clean Exclusive (solo en R4000SC y MC).

3=Dirty Exclusive (en todas las configuraciones del R4000)

PTag: Etiqueta física (bits 35:12 de la dirección física).

DataP: Paridad par para el dato.

Data: Dato de la caché.

Figura 20.- Formato de Línea de D-caché Primaria de 8 palabras

En todos los procesadores R4000, el bit W (write-back), no el estado de caché, indica si la caché primaria contiene datos modificados que deben escribirse en memoria o en la caché secundaria (usando la política de post-escritura), o no los tiene.

Acceso a las cachés primarias:

La figura 21 muestra el índice de la dirección virtual (VA) en las cachés primarias. Cada caché (instrucción y datos) tiene un rango de tamaño de 8 Kbytes hasta 32 Kbytes; por lo tanto, el número de bits de dirección virtual usado para indexar la caché depende del tamaño de la caché.

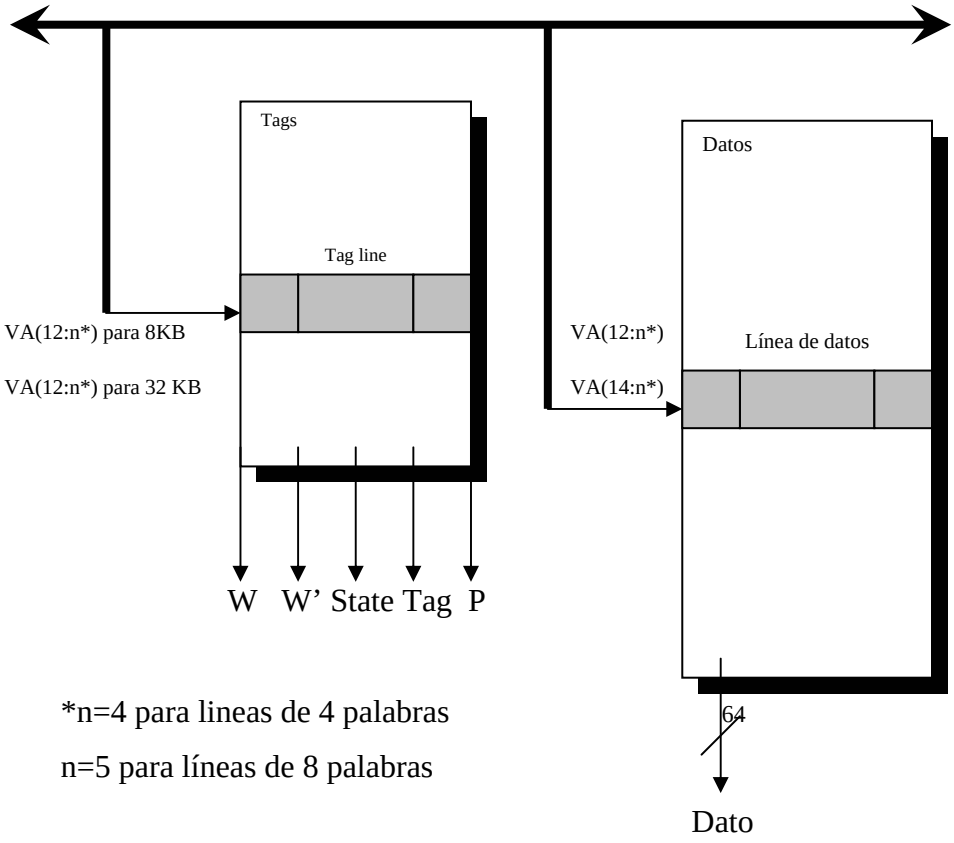


Figura 21.- Organización de los datos y etiquetas de la caché primaria

Organización de la Caché Secundaria

Cada línea de caché secundaria tiene asociada una etiqueta de 19 bits que contiene los bits 35:17 de la dirección física, un índice de 3 bits para la caché primaria, VA(14:12), y 3 bits para el estado de línea de la caché. Estos 25 bits por un código de detección de errores (ECC) de 7 bits.

La figura 22 muestra el formato de la línea de caché secundaria del procesador R4000. El tamaño de la línea de caché secundaria está en el campo SB del registro de configuración (Config register).

ECC: ECC para la etiqueta secundaria
CS: Estado de la caché secundaria
0=Invalid
1=reserved
2=reserved
3=reserved
4=Clean Exclusive
5=Dirty Exclusive
6=Shared
7=Dirty Shared
Pidx: Indice de la caché primaria (bits 14:12 de la dirección virtual)
Stag: Etiqueta física (bits 35:17 de la dirección física)

Figura 22.- Formato de línea de la caché secundaria

La caché secundaria del R4000 tiene las siguientes características:

- Post-escritura.
- Mapeado directo.
- Indexada con una dirección física
- Chequeada con una etiqueta física
- Organizada con una línea de caché de 4, 8, 16 ó 32 palabras.

Los bits del estado de la caché secundaria (CS) indican si:

- El dato de la línea de caché y etiqueta son validos.
- El dato está potencialmente presente en las cachés de otros procesadores (shared contra exclusive).
- El procesador es responsable de actualizar la memoria principal (clean contra dirty).

Acceso a la caché secundaria

La figura 23 muestra el índice de la dirección virtual (VA) en las caché secundaria. La caché secundaria tiene un rango de tamaño de 128 Kbytes hasta 4 Mbytes, y el número de bits de dirección física usados para indexar la caché depende del tamaño de la caché. Por ejemplo, PA(16:4)

accede a las etiquetas en una caché secundaria de 128 Kbytes con líneas de 4 palabras; PA(21:5) accede a las etiquetas en una caché secundaria de 4Mbytes con líneas de 8 palabras.

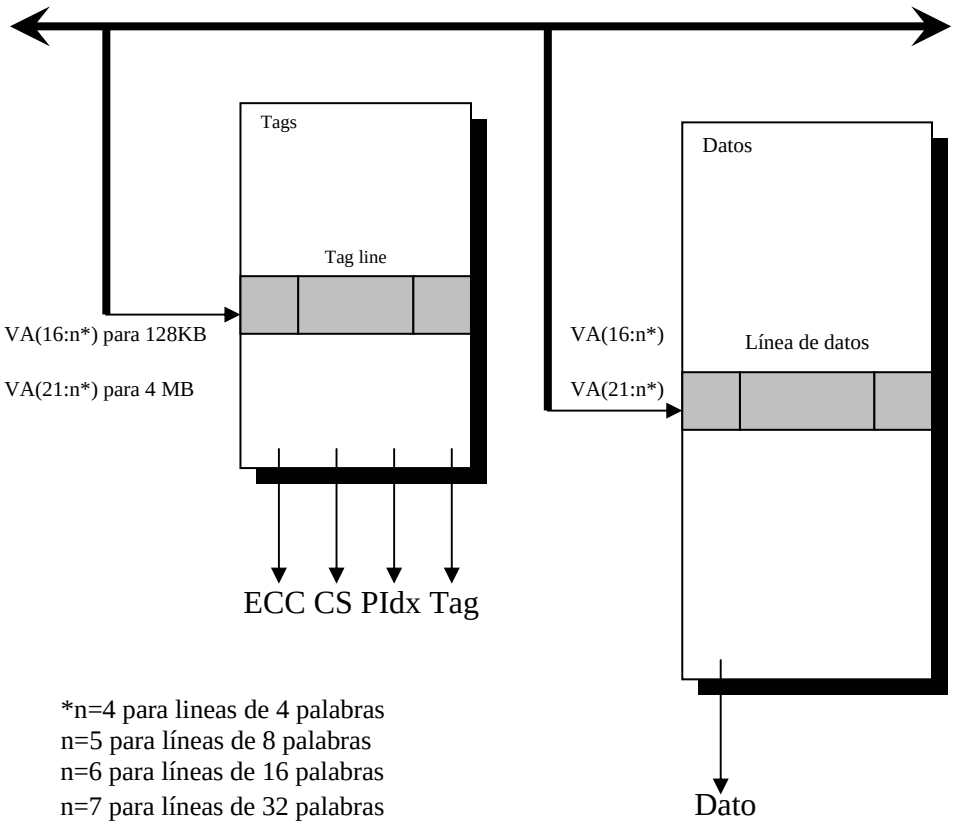


Figura 23.- Organización de los datos y etiquetas de la caché secundaria

Estados de la Caché

Estos cuatro términos se usan para describir el estado de una línea de caché:

- **Exclusive:** Una línea de caché que está presente en exactamente una caché en el sistema es exclusive, y puede estar en uno de los estados exclusive.
- **Dirty:** Una línea de caché que contiene datos que han cambiado desde que fueron cargados de memoria está en estado Dirty, y debe estar en uno de los estados dirty ó shared.

- **Clean:** Una línea de caché que contiene datos que no han sido cambiados desde que fueron cargados de memoria está en estado clean, y puede estar en uno de los estados clean.
- **Shared:** Una línea de caché que está presente en mas de una caché en el sistema.

Cada línea de caché primaria y secundaria en los sistemas R4000 está en uno de los estados descritos en la tabla que se muestra a continuación y que lista con los tipos de caché y los modelos del R4000 en cuáles de los estados pueden ser encontrados.

Estado de la línea de Caché	**Descripción**	**Dónde se usa ese estado**	**Permitido en los modelos del R4000**
Invalid	Una línea de caché que no contiene información valida debe ser marcada invalid, y no puede ser usada. Por ejemplo, una línea de caché se marca invalid si la misma información localizada en otra caché, es modificada. Una línea de caché en cualquier otro estado que no sea invalid se asume que contiene información válida.	Caché Primaria ó Secundaria	R4000PC R4000SC R4000MC
Shared	Una línea de caché que está presente en más de una caché en el sistema es shared.	Caché Primaria ó Secundaria	Solo R4000MC
Dirty Shared	Una línea de caché dirty shared contiene información válida y puede ser presentada en otra caché. Esta línea de caché es inconsistente con la memoria.	Solo Caché Secundaria.	Solo R4000MC

Clean Exclusive	Una línea de caché clean exclusive contiene información valida y esta línea de caché no está presente en cualquier otra caché. La línea de caché es consistente con la memoria.	Caché Primaria ó Secundaria	R4000SC R4000MC
Dirty Exclusive	Una línea de caché dirty exclusive contiene información válida y no está presente en cualquier otra caché. La línea de caché es inconsistente con la memoria.	Caché Primaria ó Secundaria	R4000PC R4000SC R4000MC

Estados de la caché Primaria

Cada línea de caché de datos primaria está en uno de los siguientes estados:

- Invalid
- Shared
- Clean exclusive
- Dirty exclusive

Cada línea de la caché de instrucciones primaria está en uno de los siguientes estados:

- Invalid
- Valid

Estados de la Caché Secundaria

Cada línea de caché secundaria está en uno de los siguientes estados:

- Invalid
- Shared
- Dirty Shared
- Clean Exclusive
- Dirty Exclusive

EL PIPELINE DE LA CPU

La CPU tiene un pipeline de instrucciones de ocho estados; cada estado ocupa un Pcycle (un ciclo de Pclock, el cual tiene dos veces la frecuencia del MasterClock). Así, la ejecución de cada instrucción ocupa al menos ocho Pcycles (cuatro ciclos de MasterClock). Una instrucción puede ocupar más de ocho ciclos, por ejemplo, si el dato requerido no está en la caché, el dato debe ser cogido de la memoria principal.

Una vez que el pipeline se ha llenado, se están ejecutando ocho instrucciones simultáneamente. La figura 24 muestra los ocho estados del pipeline o segmentación de una instrucción; la siguiente sección describe los estados del pipeline.

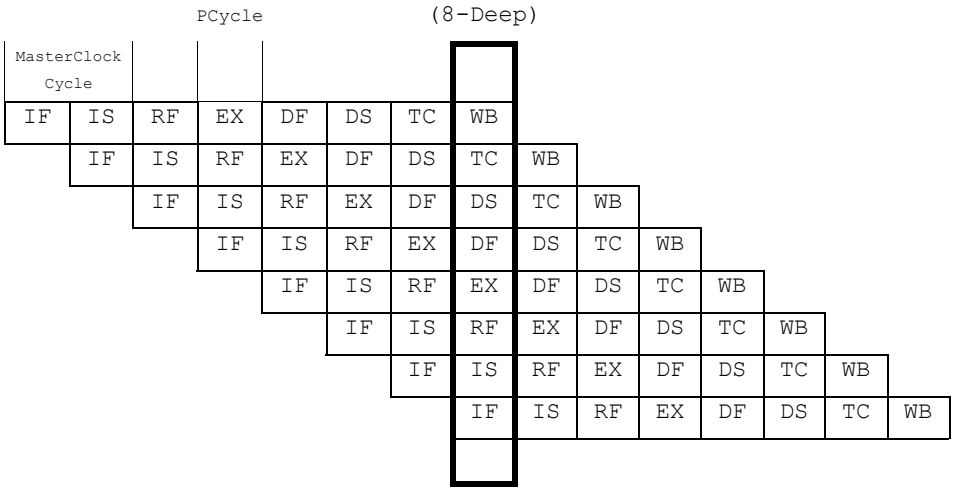


Figura 24 .- Estados del Pipeline de Instrucciones

ESTADOS DEL PIPELINE DE LA CPU

Esta sección describe cada uno de los ocho estados del pipeline:

- IF – Búsqueda de instrucción, Primera Mitad

- IS – Búsqueda de la Instrucción, Segunda mitad
- RF – Búsqueda de registros
- EX – Ejecución
- DF – Búsqueda de datos, Primera mitad
- DS – Búsqueda de datos, Segunda mitad
- TC – Chequeo de etiqueta
- WB – Post-escritura (Write-back)

IF – Búsqueda de la instrucción, Primera mitad

Durante el estado IF, ocurre lo siguiente:

- La lógica de salto selecciona una dirección de instrucción y la búsqueda en la caché de instrucciones comienza.
- El TLB de instrucciones (ITLB) comienza la traducción de dirección virtual a física.

IS – Búsqueda de la instrucción, Segunda mitad

Durante es estado IS, la búsqueda en la caché de instrucciones y la traducción de dirección virtual a física se completan.

RF – Búsqueda de registros

Durante el estado RF, ocurre lo siguiente:

- El decodificador de instrucciones (IDEC) decodifica la instrucción.
- La etiqueta de la caché de instrucciones se compara con el número de marco de página obtenido del ITLB.
- Algunos operandos requeridos se cogen de los registros.

EX – Ejecución

Durante el estado EX, ocurre un de los siguientes puntos:

- La ALU (Unidad aritmético lógica) realiza la operación aritmética o lógica para instrucciones de registro a registro.

- La ALU calcula la dirección virtual para instrucciones de carga y almacenamiento.
- La ALU determina si la condición de salto es verdad y calcula el salto relativo para las instrucciones de salto relativo.

DF – Búsqueda de datos, Primera mitad

Durante el estado DF, ocurre uno de los siguientes puntos:

- La búsqueda en la caché de datos y la traducción de datos comienza para instrucciones de carga y almacenamiento
- La traducción de direcciones de instrucciones de salto relativo y la actualización del TLB comienza para las instrucciones de salto relativo.
- No se realizan operaciones durante los estados DF, DS, Y TC para las instrucciones de registro a registro.

DS – Búsqueda de datos, Segunda mitad

Durante el estado DS, ocurre uno de los siguientes puntos:

- La búsqueda en la caché de datos y la traducción de datos de virtual a física se completan para las instrucciones de carga y almacenamiento. El desplazador alinea datos a su límite de palabra ó doble palabra.
- La traducción de direcciones de instrucciones de salto relativo y la actualización del TLB se completan para las instrucciones de salto relativo.

TC – Chequeo de Etiquetas

Para las instrucciones de carga y almacenamiento, la caché realiza el chequeo de etiquetas durante el estado TC. La dirección física del TLB se compara con la etiqueta de caché para determinar si hay un acierto o un fallo.

WB – Write Back

Para las instrucciones de registro a registro, el resultado de la instrucción se escribe en el registro durante el estado WB. Las instrucciones de salto relativo no realizan operaciones durante este estado.

RETARDO DE SALTO (Branch Delay)

El pipeline de la CPU tiene un retardo de salto de tres ciclos y un retardo de carga (load delay) de dos ciclos. El retardo de salto de tres ciclos es el resultado de la comparación de salto durante el estado EX de el salto, produciendo una dirección de instrucción que está permitida en el estado IF, cuatro ciclos más tarde. La figura 25 ilustra el retardo de salto.

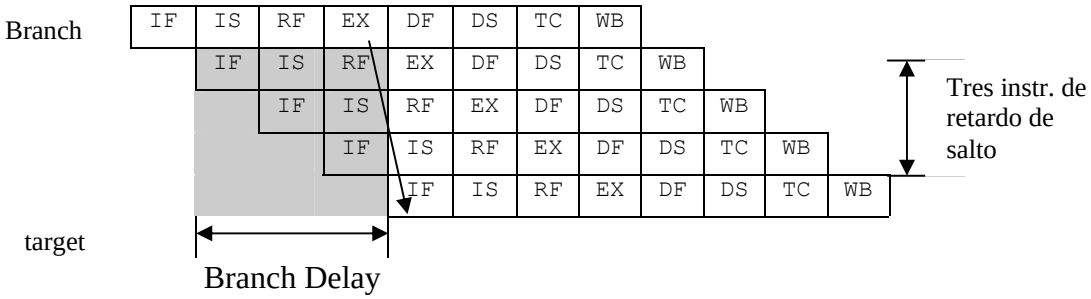


Figura 25 .- Retardo de Salto del Pipeline de la CPU

RETARDO DE CARGA (Load Delay)

La termianación de una carga al final del estado DS produce un operando que está permitido para el estado EX de la tercera instrucción subsecuente.

La figura 26 muestra retardo de carga.

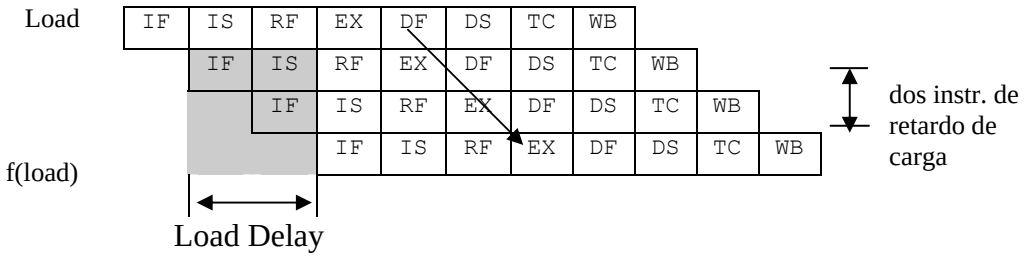
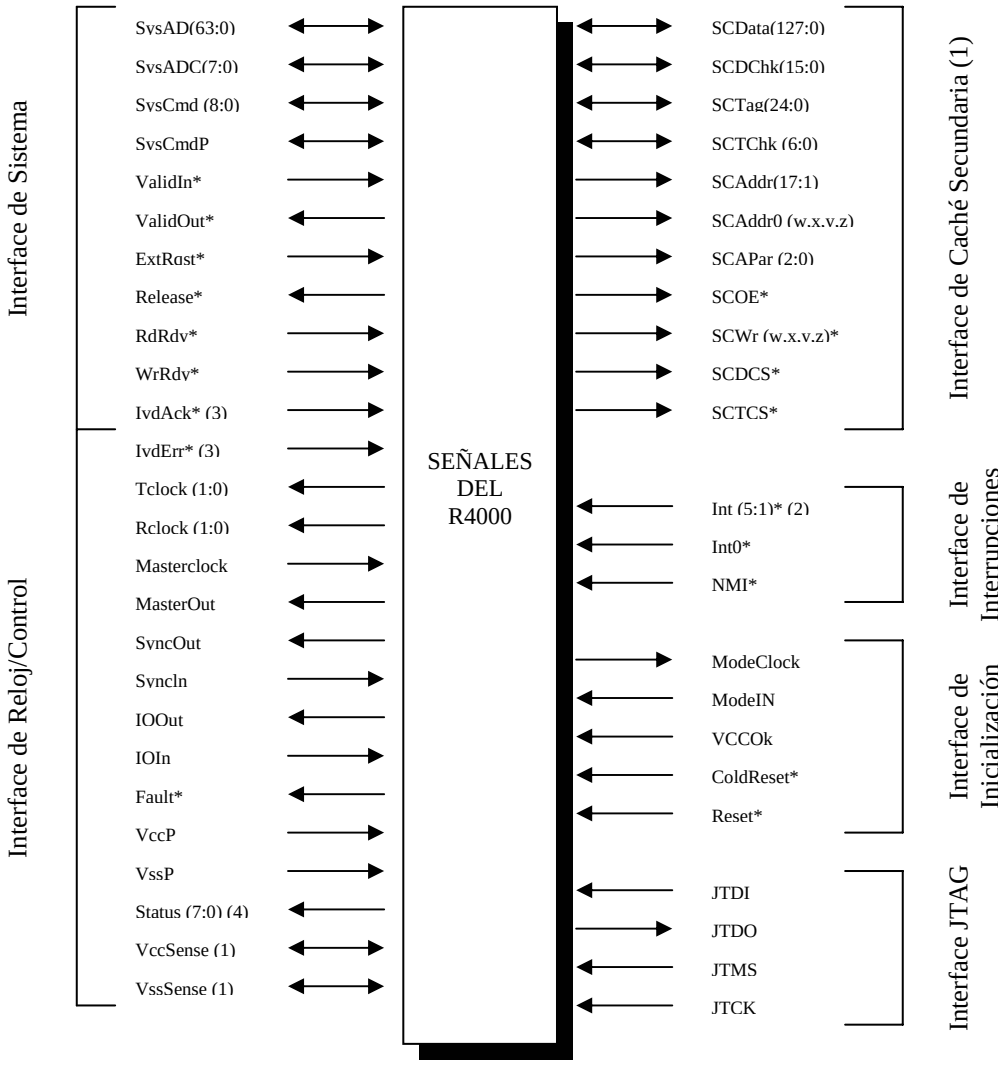


Figura 26.- Retardo de Carga del Pipeline de la CPU

DIBUJO DE LAS SEÑALES DEL MICRO

En este apartado solo se va a mostrar en la figura 27 las señales del microprocesador.



(1)=Solo R4000SC y R4000MC (2)=Solo R4000PC
(3)=Solo R4000MC (4)=Solo R4400

Figura 27 .- Señales del Procesador R4000

DIAGRAMA DE BLOQUES INTERNO DEL R4000

Por último, en la figura 28 se muestra el diagrama de bloques interno del MIPS R-4000, para tener una idea general de cómo está organizado por dentro.

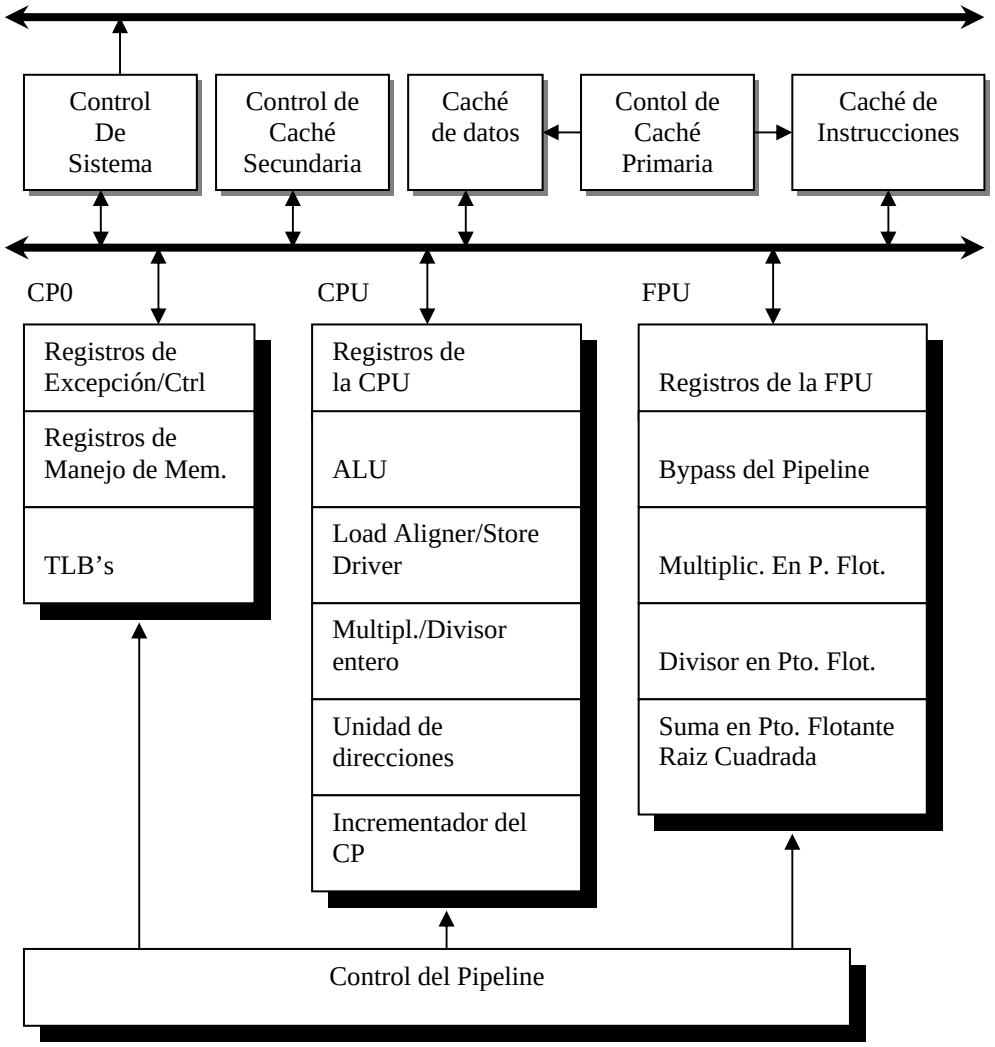


Figura 28 .- Diagrama de bloques interno del R4000

CAPÍTULO 2:

MANUAL DE USUARIO DEL SIMULADOR DE INSTRUCCIONES MIPS R-4000

Antes de comenzar a explicar como funciona el programa, vamos a enumerar las instrucciones que están implementadas en el simulador.

INSTRUCCIONES IMPLEMENTADAS

Instrucciones de Carga y Almacenamiento.

OpCode	Descripción
LB	Load Byte
LBU	Load Byte Unsigned
LH	Load Halfword
LHU	Load Halfword Unsigned
LW	Load Word
LWL	Load Word Left
LWR	Load Word Right
SB	Store Byte
SH	Store Halfword
SW	Store Word
SWL	Store Word Left
SWR	Store Word Right

Instrucciones Aritméticas (dato inmediato).

Opcode	Descripción
ADDI	Add Immediate
ADDIU	Add Immediate Unsigned
SLTI	Set on Less Than Immediate
SLTIU	Set on Less Than Immediate Unsigned
ANDI	AND Immediate
ORI	OR Immediate
XORI	Exclusive Or Immediate
LUI	Load Upper Immediate

Instrucciones Aritméticas (3 operandos, R-Type).

OpCode	Descripción
ADD	Add
ADDU	Add Unsigned
SUB	Subtract
SUBU	Subtract Unsigned
SLT	Set on Less Than
SLTU	Set on Less Than Unsigned
AND	AND
OR	OR
XOR	Exclusive OR
NOR	NOR

Instrucciones de Multiplicación y División.

OpCode	Descripción
MULT	Multiply
MULTU	Multiply Unsigned
DIV	Divide
DIVU	Divide Unsigned
MFHI	Move From HI
MTHI	Move To HI
MFLO	Move From LO
MTLO	Move To LO

Instrucciones de Saltos absolutos y relativos.

OpCode	Descripción
J	Jump
JAL	Jump And Link
JR	Jump Register
JALR	Jump And Link Register
BEQ	Branch on Equal
BNE	Branch on Not Equal
BLEZ	Branch on Less Than or Equal to Zero
BGTZ	Branch on Greater Than Zero
BLTZ	Branch on Less Than Zero
BGEZ	Branch on Greater Than or Eq. to Zero
BLTZAL	Branch on Less Than Zero And Link
BGEZAL	Brnch on Less Thn Zro And Link Likely

Instrucciones de desplazamiento de bits.

OpCode	Descripción
SLL	Shift Left Logical
SRL	Shift Right Logical
SRA	Shift Right Arithmetic
SLLV	Shift Left Logical Variable
SRLV	Shift Right Logical Variable
SRAV	Shift Right Arithmetic Variable

Extensiones a ISA: Instrucciones de carga y almacenamiento.

OpCode	Descripción
LD	Load Doubleword
LDL	Load Doubleword Left
LDR	Load Doubleword Right
LWU	Load Word Unsigned
SD	Store Doubleword
SDL	Store Doubleword Left
SDR	Store Doubleword Right

Extensiones a ISA: Instrucciones Aritméticas (dato inmediato).

OpCode	Descripción
DADDI	Doubleword Add Immediate
DADDIU	Doubleword Add Immediate Unsigned

Extensiones a ISA: Instrucciones de Multiplicación y División.

OpCode	Descripción
DMULT	Doubleword Multiply
DMULTU	Doubleword Multiply Unsigned
DDIV	Doubleword Divide
DDIVU	Doubleword Divide Unsigned

Extensiones a ISA: Instrucciones de salto relativo.

OpCode	Descripción
BEQL	Branch on Equal Likely
BNEL	Branch on Not Equal Likely
BLEZL	Branch on Less Than or Equal to Zro Likely
BGTZL	Branch on Greater Than Zero Likely
BLTZL	Branch on Less Than Zero Likely
BGEZL	Brnch on Greatr Thn or Equal to Zro Likely
BLTZALL	Branch on Less Than Zero And Link Likely
BGEZALL	Branch on Greater Than or Equal to Zero And Link Likely

Extensiones a ISA: Instrucc. Aritméticas (3 operandos, R-Type).

OpCode	Descripción
DADD	Doubleword Add
DADDU	Doubleword Add Unsigned
DSUB	Doubleword Subtract
DSUBU	Doubleword Subtract Unsigned

Extensiones a ISA: Instrucciones de desplazamiento de bits.

OpCode	Descripción
DSLL	Doubleword Shift Left Logical
DSRL	Doubleword Shift Right Logical
DSRA	Doubleword Shift Right Arithmetic
DSLLV	Doublewrd Shift Left Logical Variable
DSRLV	Doublwrd Shift Right Logical Variable
DSRAV	Doublwrd Shift Right Arithmetic Variable
DSLL32	Doubleword Shift Left Logical + 32
DSRL32	Doubleword Shift Right Logical + 32
DSRA32	Doubleword Shift Right Arithmetic + 32

Estensiones a ISA: Instrucciones de excepciones.

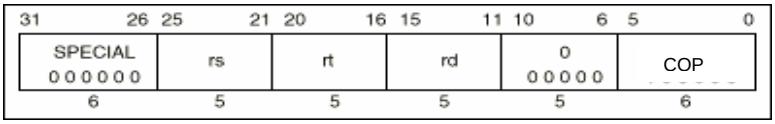
OpCode	Descripción
TGE	Trap if Greater Than or Equal
TGEU	Trap if Greater Than or Equal Unsigned
TLT	Trap if Less Than
TLTU	Trap if Less Than Unsigned
TEQ	Trap if Equal
TNE	Trap if Not Equal
TGEI	Trap if Greater Than or Equal Immediate
TGEIU	Trap if Greater Than or Equal Immediate Unsigned
TLTI	Trap if Less Than Immediate
TLTIU	Trap if Less Than Immediate Unsigned
TEQI	Trap if Equal Immediate
TNEI	Trap if Not Equal Immediate

Extensiones a ISA: Instrucciones del CP0.

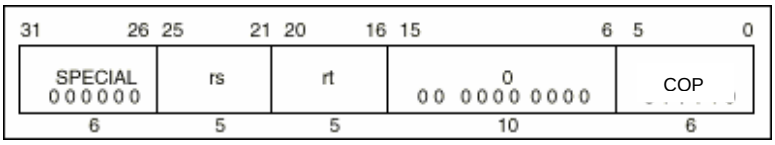
OpCode	Descripción
ERET	Exception Return

La página que viene a continuación contiene una serie de tablas que serán muy útiles a la hora de que el usuario del simulador tenga que hacer un programa en el mismo. En dichas tablas se puede ver que campos utiliza cada instrucción y el código de operación de cada una de ellas, de este modo el usuario podrá codificar fácilmente cualquier instrucción:

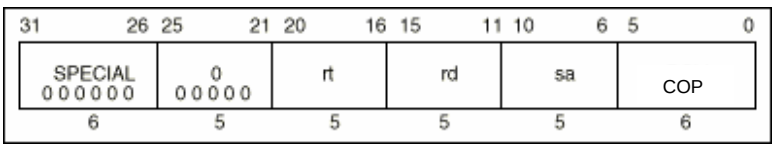
FORMATOS Y CÓDIGOS DE OPERACIÓN DE LAS INSTRUCCIONES DEL R-4000



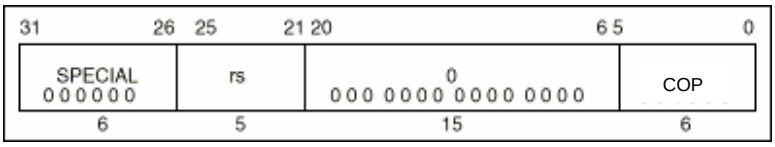
FORMATO INSTRUCCION	CÓDIGO DE OPERACION (COP)
ADD RD,RS,RT	100000
ADDU RD,RS,RT	100001
AND RD,RS,RT	100100
DADD RD,RS,RT	101100
DADDU RD,RS,RT	101101
DSLLV RD,RT,RS	010100
DSRAV RD,RT,RS	010111
DSRLV RD,RT,RS	010110
DSUB RD,RS,RT	101110
DSUBU RD,RS,RT	101111
NOR RD,RS,RT	100111
OR RD,RS,RT	100101
SLLV RD,RT,RS	000100
SLT RD,RS,RT	101010
SLTU RD,RS,RT	101011
SRAV RD,RT,RS	000111
SRLV RD,RT,RS	000110
SUB RD,RS,RT	100010
SUBU RD,RS,RT	100011
XOR RD,RS,RT	100110



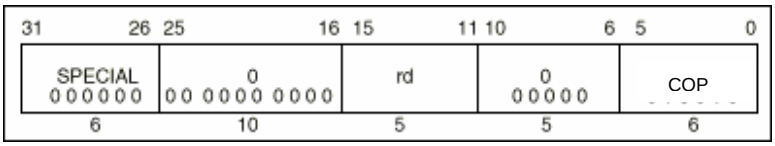
INSTRUCCION	CÓDIGO DE OPERACION (COP)
DDIV RS,RT	011110
DDIVU RS,RT	011111
DIV RS,RT	011010
DIVU RS,RT	011011
DMULT RS,RT	011100
DMULTU RS,RT	011101
MULT RS,RT	011000
MULTU RS,RT	011001
TEQ RS,RT	110100
TGE RS,RT	110000
TGEU RS,RT	110001
TLT RS,RT	110010
TLTU RS,RT	110011
TNE RS,RT	110110



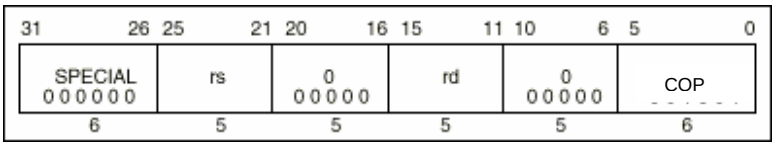
INSTRUCCION	CÓDIGO DE OPERACION (COP)
DSLL RD,RT,SA	111000
DSLL32 RD,RT,SA	111100
DSRA RD,RT,SA	111011
DSRA32 RD,RT,SA	111111
DSRL RD,RT,SA	111010
DSRL32 RD,RT,SA	111110
SLL RD,RT,SA	000000
SRA RD,RT,SA	000011
SRL RD,RT,SA	000010



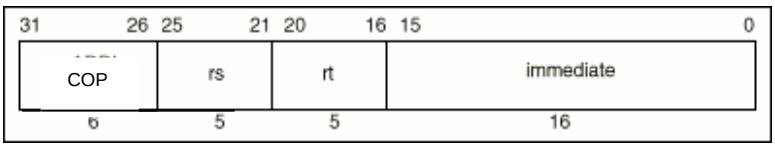
INSTRUCCION	CÓDIGO DE OPERACION (COP)
JR RS	001000
MTHI RS	010001
MTLO RS	010011



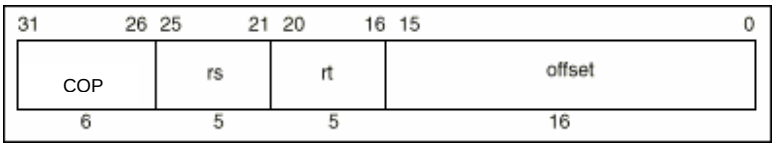
INSTRUCCION	CÓDIGO DE OPERACION (COP)
MFHI RD	010000
MFLO RD	010010



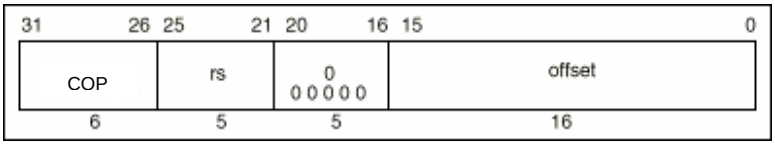
INSTRUCCION	CÓDIGO DE OPERACION (COP)
JALR RS Ó JALR RD,RS	001001



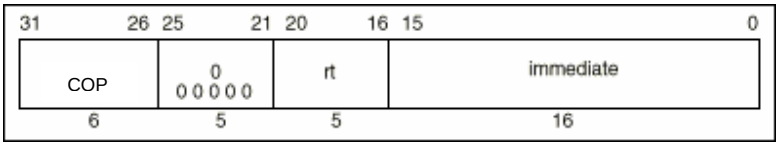
INSTRUCCION	CÓDIGO DE OPERACION (COP)
ADDI RT,RS,INMEDIATO	001000
ADDIU RT,RS,INMEDIATO	001001
ANDI RT,RS,INMEDIATO	001100
DADDI RT,RS,INMEDIATO	011000
DADDIU RT,RS,INMEDIATO	011001
ORI RT,RS,INMEDIATO	001101
SLTI RT,RS,INMEDIATO	001010
SLTIU RT,RS,INMEDIATO	001011
XORI RT,RS,INMEDIATO	001110



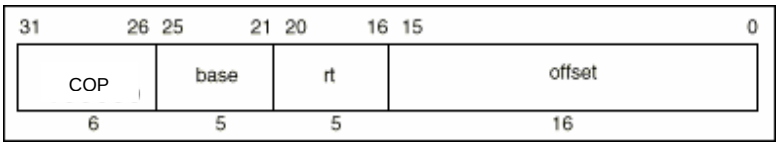
INSTRUCCION	CÓDIGO DE OPERACION (COP)
BEQ RS,RT,OFFSET	000100
BEQL RS,RT,OFFSET	010100
BNE RS,RT,OFFSET	000101
BNEL RS,RT,OFFSET	010101



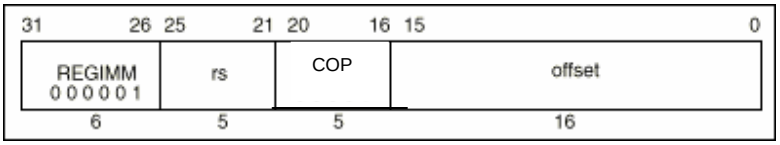
INSTRUCCION	CÓDIGO DE OPERACION (COP)
BGTZ RS,OFFSET	000111
BGTZL RS,OFFSET	010111
BLEZ RS,OFFSET	000110
BLEZL RS,OFFSET	010110



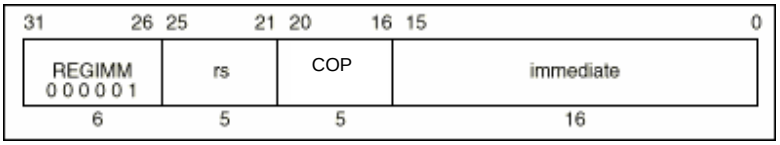
INSTRUCCION	CÓDIGO DE OPERACION (COP)
LUI RT,INMEDIATO	001111



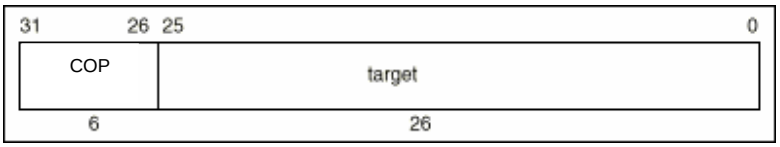
INSTRUCCION	CÓDIGO DE OPERACION (COP)
LB RT,OFFSET(BASE)	100000
LBU RT,OFFSET(BASE)	100100
LD RT,OFFSET(BASE)	110111
LDL RT,OFFSET(BASE)	011010
LDR RT,OFFSET(BASE)	011011
LH RT,OFFSET(BASE)	100001
LHU RT,OFFSET(BASE)	100101
LW RT,OFFSET(BASE)	100011
LWL RT,OFFSET(BASE)	100010
LWR RT,OFFSET(BASE)	100110
LWU RT,OFFSET(BASE)	100111
SB RT,OFFSET(BASE)	101000
SD RT,OFFSET(BASE)	111111
SDL RT,OFFSET(BASE)	101100
SDR RT,OFFSET(BASE)	101101
SH RT,OFFSET(BASE)	101001
SW RT,OFFSET(BASE)	101011
SWL RT,OFFSET(BASE)	101010
SWR RT,OFFSET(BASE)	101110



INSTRUCCION	CÓDIGO DE OPERACION (COP)
BGEZ RS,OFFSET	00001
BGEZAL RS,OFFSET	10001
BGEZALL RS,OFFSET	10011
BGEZL RS,OFFSET	00011
BLTZ RS,OFFSET	00000
BLTZAL RS,OFFSET	10000
BLTZALL RS,OFFSET	10010
BLTZL RS,OFFSET	00010



INSTRUCCION	CÓDIGO DE OPERACION (COP)
TEQI RS,INMEDIATO	01100
TGEI RS,INMEDIATO	01000
TGEIU RS,INMEDIATO	01001
TLTI RS,INMEDIATO	01010
TLTIU RS,INMEDIATO	01011
TNEI RS,INMEDIATO	01110



INSTRUCCION	CÓDIGO DE OPERACION (COP)
J TARGET	000010
JAL TARGET	000011

ERET → Código de la instrucción : 42000018 (código a introducir en el simulador).

EXPLICACIÓN DE LAS OPCIONES DEL MENÚ PRINCIPAL

A partir de aquí se van a explicar todas y cada una de las opciones del simulador, de manera que el usuario, al leerlo, pueda manejar el programa y moverse por el interfaz de usuario con más soltura.

El simulador de instrucciones del R-4000 sirve para ejecutar programas que utilizan el repertorio de instrucciones del propio microprocesador.

El menú principal se compone de 10 opciones que se muestran en la figura 1. Vamos a explicar que hace cada una de estas opciones, también habrá ejemplos gráficos:

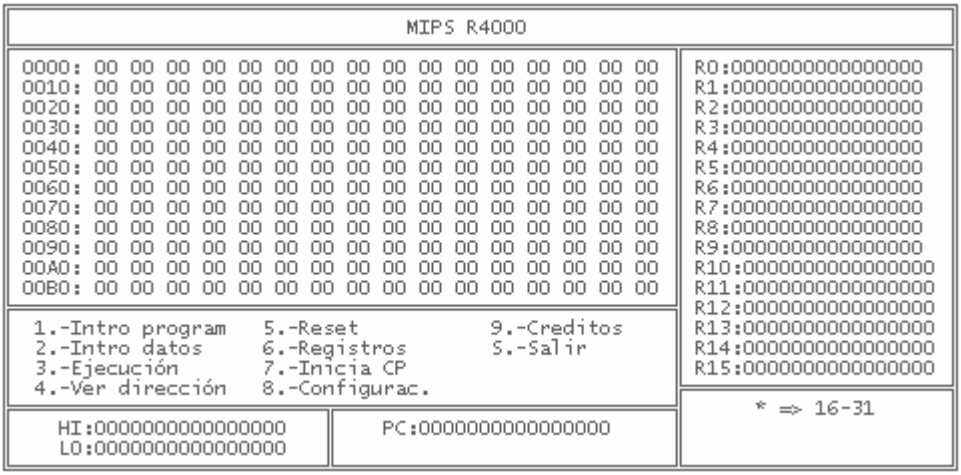


Figura 1

1.-Introducir Programa:

Esta opción nos permite introducir nuestro programa en memoria principal. Cuando pulsamos esta opción, el simulador pide una dirección de comienzo. A continuación nos pide que metamos una instrucción completa. Cuando hemos introducido la instrucción, nos pide la siguiente, y así sucesivamente hasta que metamos todas las instrucciones que forman nuestro programa. Para terminar de meter instrucciones, pulsaremos la tecla escape.

Para verlo mejor, veremos ejemplos con imágenes del propio simulador ejecutando esta opción.

Si pulsamos la opción 1 desde el menú principal, nos pide la dirección de comienzo. Esta dirección debe estar alineada en el límite de una palabra, es decir, múltiplo de 4 porque las instrucciones son de 32 bits (una palabra). Si la dirección no es múltiplo de cuatro, nos la vuelve a pedir. Introducimos la dirección 0010 y, como veremos en la figura 3, las direcciones de la pantalla se comienzan a visualizar a partir de la dirección 0010. Ya podemos comenzar a introducir instrucciones en memoria.

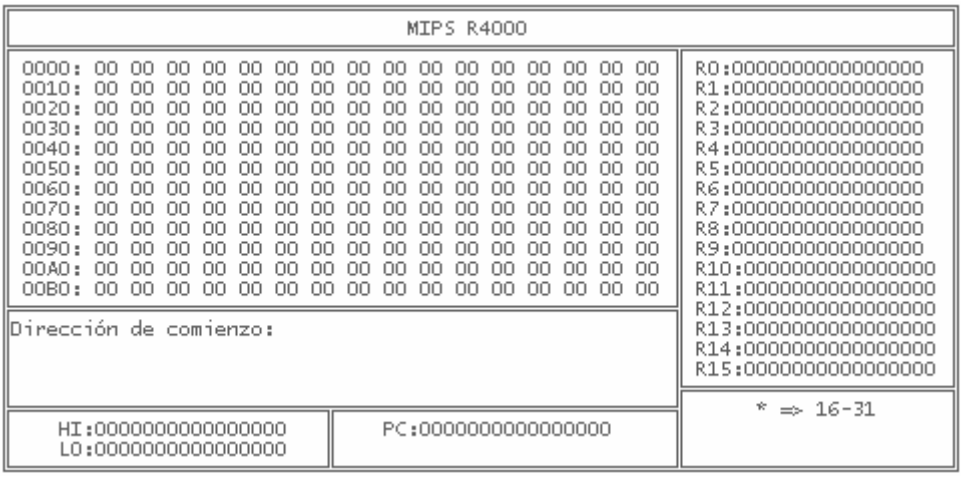


Figura 2

En la figura 3 vemos que, después de haber introducido la dirección de comienzo, las direcciones de pantalla comienzan a visualizarse a partir de la dirección 0010, entonces nos pide que introduzcamos una instrucción completa, en este caso, por hacer una prueba, vamos a meter la instrucción OR r7, r5, r3 que hace la OR lógica de los registros r5 y r3 y el resultado se guarda en el registro r7. El código que corresponde a esta instrucción concreta es 00A33825 que es el que tenemos que introducir en memoria.

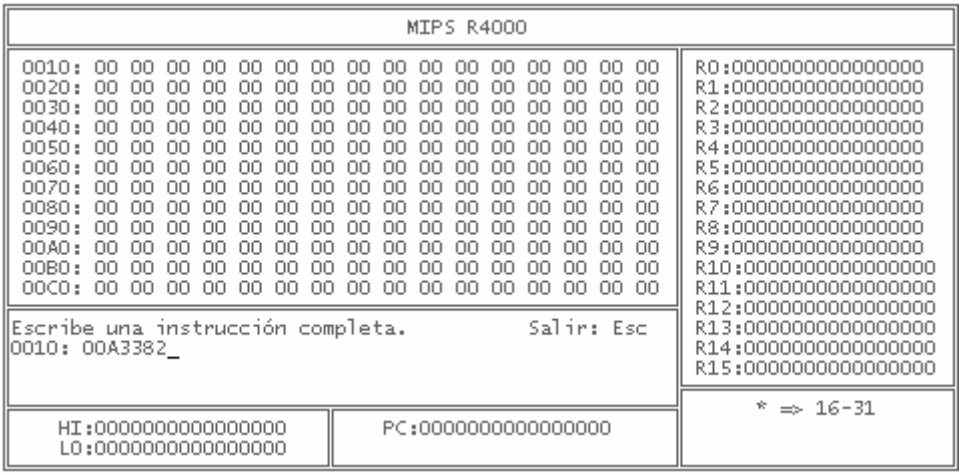


Figura 3

En la figura 4 vemos que después de escribir la instrucción, ésta se mete en la memoria y el simulador nos pide la siguiente instrucción. Las instrucciones se pueden meter en memoria de dos formas: little-endian y big-endian. La diferencia entre las dos es la ordenación de los bytes en memoria. En este ejemplo, el simulador está configurado en little-endian. Como no vamos a meter ninguna otra instrucción, pulsaremos la tecla escape para salir de la opción 1.

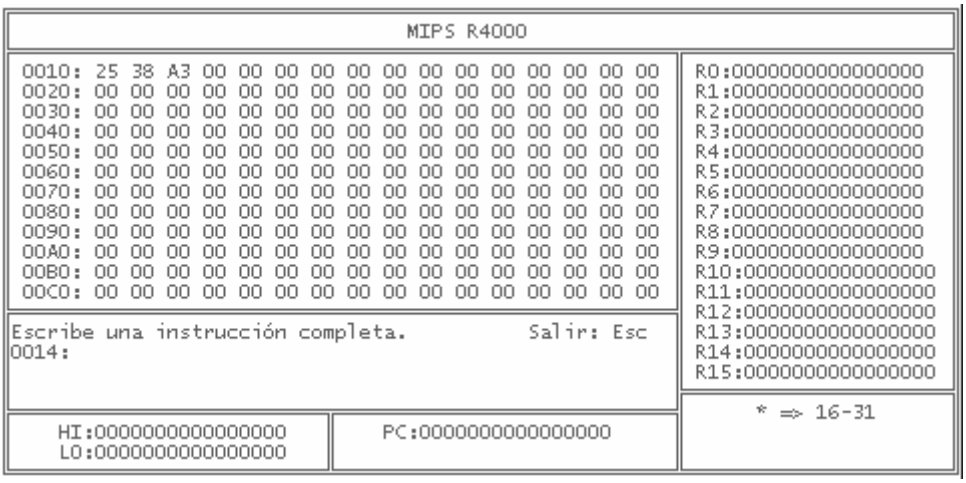


Figura 4

2.-Introducir datos

Esta opción sirve para introducir datos en memoria byte a byte. Cuando pulsamos esta opción nos piden la dirección a modificar según se muestra en la figura 5.

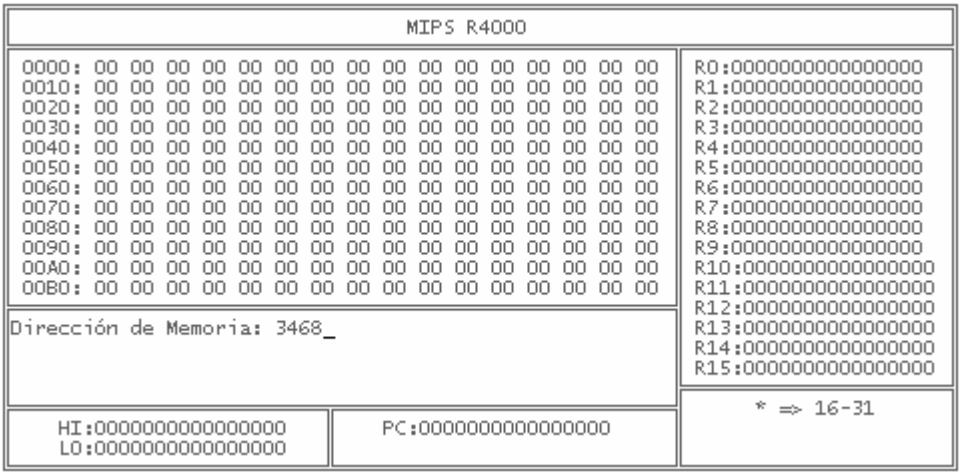


Figura 5

Cuando introducimos esta dirección, a continuación se muestra por pantalla la memoria a partir de esa dirección y pide el byte a introducir (figura 6).

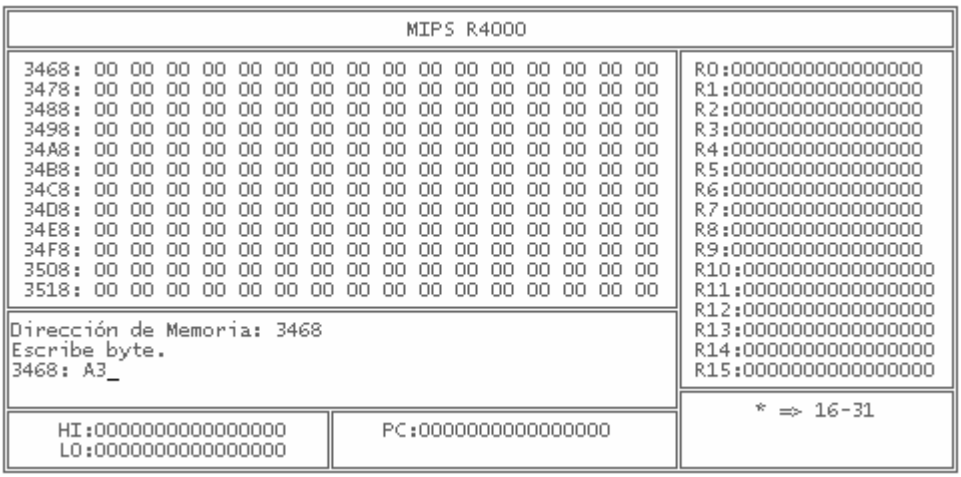


Figura 6

Metemos el byte A3 (por ejemplo). Mete el dato en la dirección especificada y nos pregunta si queremos meter otro dato en la siguiente dirección, en otra dirección distinta o si queremos terminar. Solo tenemos que pulsar una de las tres opciones. Si pulsamos la primera solo tenemos que introducir el byte que nos piden que es el de la siguiente

dirección. Si pulsamos la segunda tenemos que introducir la dirección y luego el byte y si pulsamos la tercera, volvemos al menú principal. La figura 7 muestra lo explicado hasta este punto.

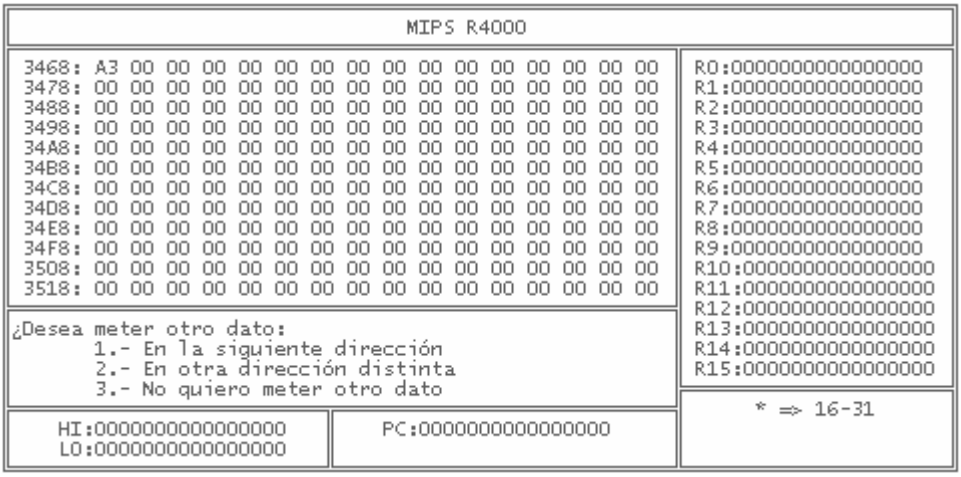


Figura 7

3.-Ejecución

Esta opción sirve para ejecutar nuestro programa a partir de la dirección a la que apunta el CP. Esta dirección se puede modificar con la opción 7 del menú principal.

Si en el menú principal pulsamos la opción 3, aparece un submenú en el que muestra al usuario las tres opciones que tiene para ejecutar un programa: Secuencialmente, en paralelo y poniendo un punto de parada (breakpoint). En la figura 8 se puede ver el submenú con las tres opciones de ejecución:

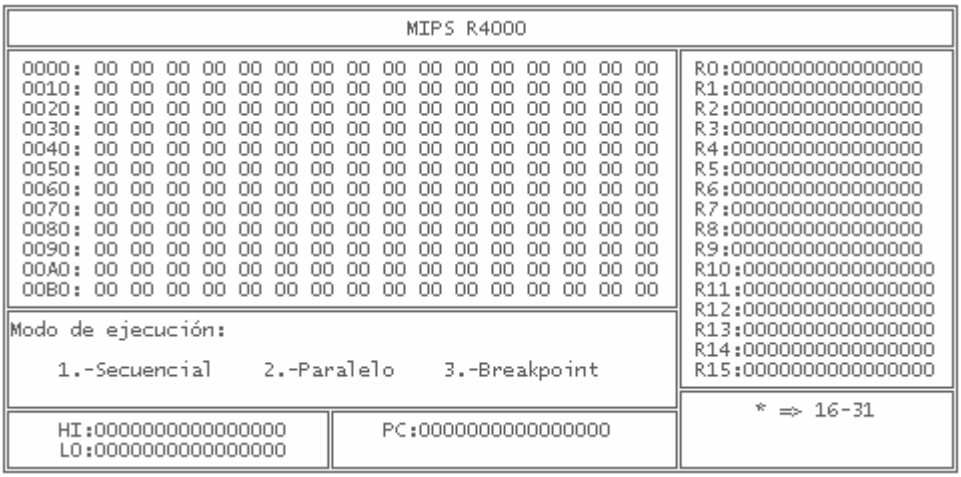


Figura 8

Si pulsamos Secuencial (opción 1), las instrucciones se ejecutarán de una en una, es decir, sale el mnemónico por pantalla; si se pulsa 'E', la instrucción será ejecutada y pasará a la siguiente. Si se pulsa 'P' la instrucción no se ejecutará y se sale al menú principal. También se puede pulsar '*' para ver el contenido de todos los registros.

Si pulsamos Paralelo (opción 2), las instrucciones se ejecutarán de dos en dos, es decir, salen por pantalla los dos mnemónicos de las dos instrucciones que se van a ejecutar; si se pulsa 'E', esas dos instrucciones se ejecutarán y pasará a las dos siguientes. Al igual que en secuencial, si se pulsa 'P' las instrucciones no se ejecutan y después de pulsar una tecla se sale al menú principal.

Por último, si pulsamos Breakpoint (opción 3), el simulador pide una dirección que será la dirección de punto de parada. Al meter la dirección, inmediatamente comienza a ejecutar instrucciones una a una hasta que llega a ese punto de parada. Después sale al menú principal.

En la figura 9 se puede ver un ejemplo de un tipo de ejecución (en este caso paralelo). Previamente hemos introducido las instrucciones ORI R3,R0,00FFh y ADDIU R4,R4,0001h. Sus códigos son respectivamente 340300FF y 24840001.

En la parte izquierda del cuadro de ejecución se puede ver cómo está configurado el simulador. Ejecución paralela (también puede ser secuencial ó breakpoint), ordenación de bytes little-endian (también puede ser big-endian), modo Usuario (también puede ser Supervisor y Kernel) y modo de ejecución 32 bits (también puede ser 64 bits).

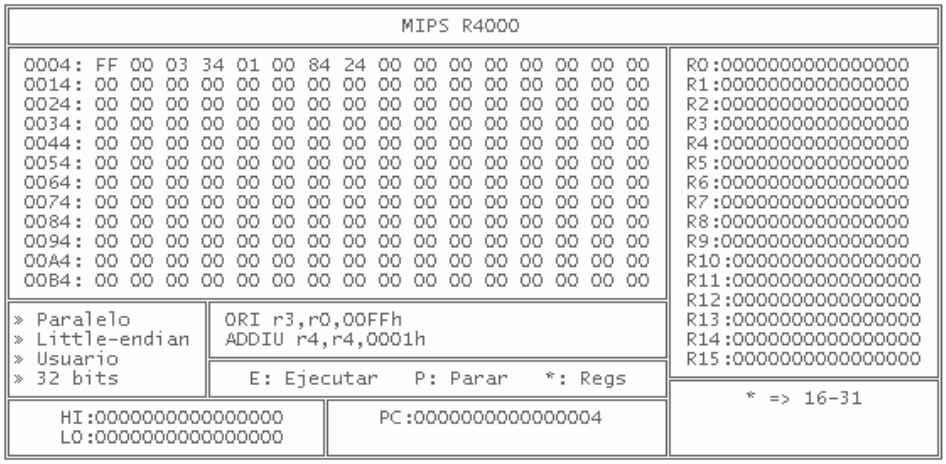


Figura 9

4.- Ver dirección

Esta opción sirve solo para visualizar la memoria a partir de una dirección dada. Al pulsar esta opción, el ordenador nos pide la dirección a partir de la cual queremos visualizar la memoria; cuando introducimos la dirección, en el cuadro donde se muestran las direcciones comienza a visualizar los bytes a partir de la dirección de memoria introducida (Ver figuras 10 y 11).

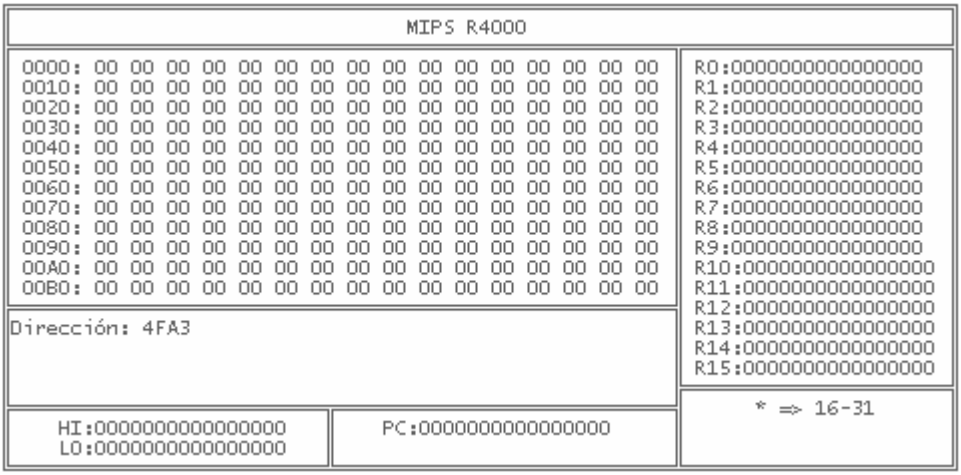


Figura 10

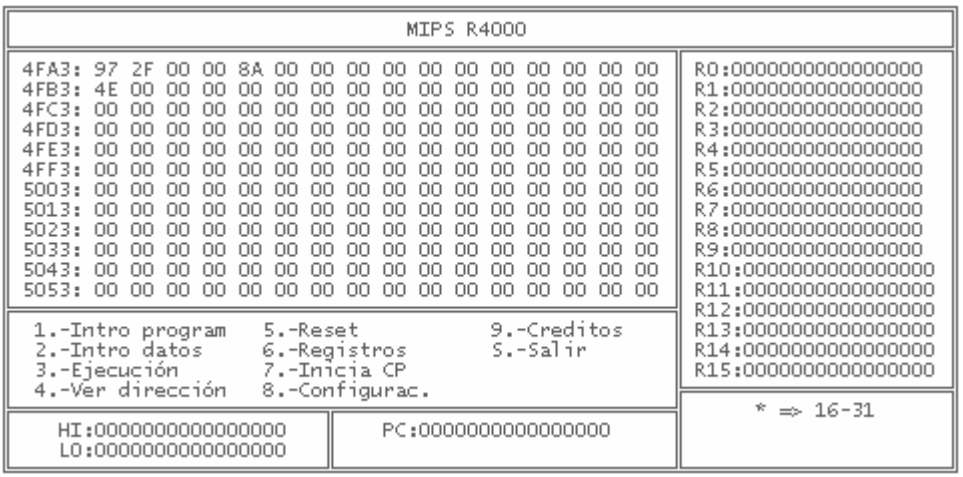


Figura 11

5.-Reset

Esta opción inicializa toda la memoria, los registros de proposito general 0..31, los de propósito especial HI, LO y el Contador de Programa (CP) a cero.

Al pulsar esta opción, el simulador pide confirmación antes de resetear el simulador según podemos ver en la figura 12. Si la respuesta es ‘S’ el simulador se resetea, y si es ‘N’ se queda tal y como está.

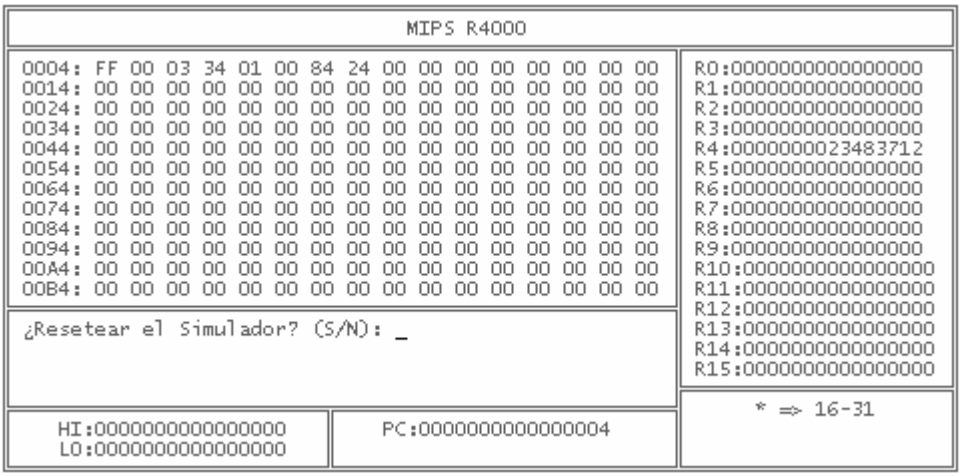


Figura 12

6.-Registros

Esta opción es para meter datos de 32 ó 64 bits en los registos de propósito general.

Como muestra la figura 13, al pulsar esta opción lo primero que nos piden es el número de registro a modificar. Solo podemos introducir un número comprendido entre 1 y 31 inclusive (R0 siempre vale 0). De no ser así, nos lo vuelve a pedir.

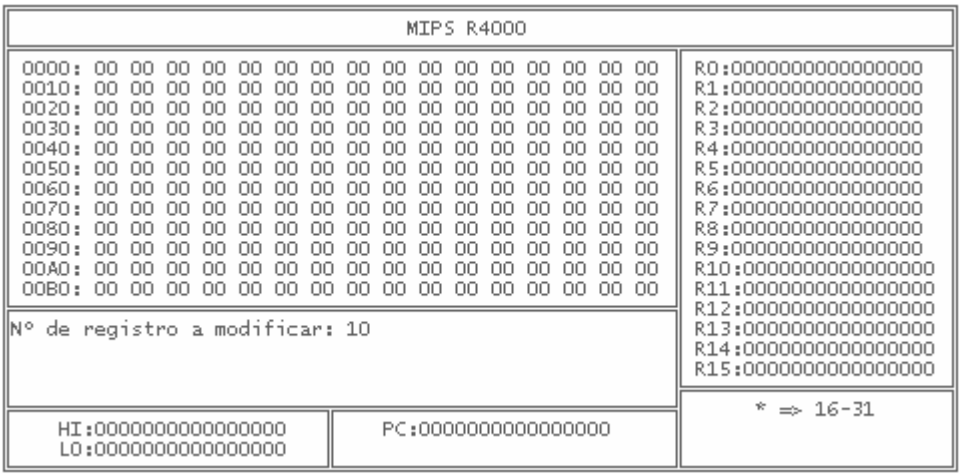


Figura 13

Cuando lo hemos introducido (en este ejemplo, registro 10), nos piden si la longitud del dato que vamos a introducir en el registro es de 32 ó de 64 bits. En el ejemplo de la figura 14 le diremos al simulador que el dato va a ser de 64 bits.

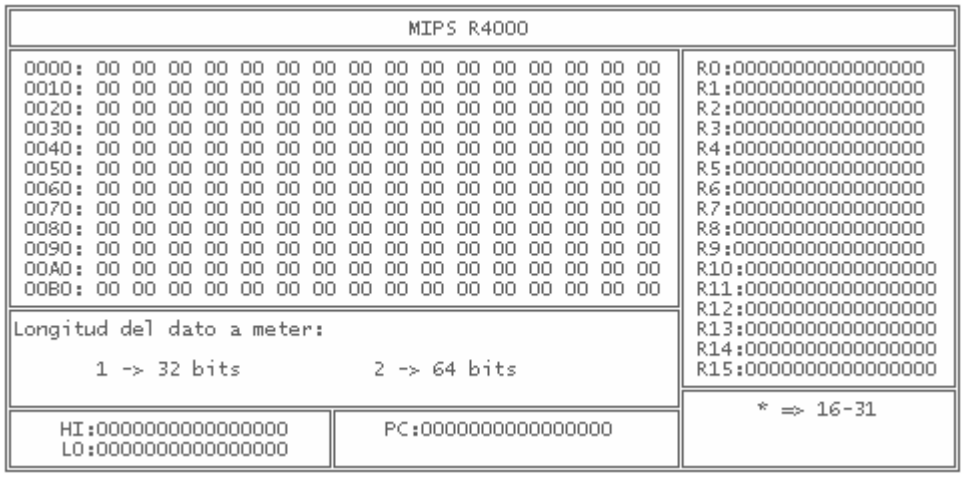


Figura 14

A continuación nos va a pedir los 32 bits de mayor peso del dato de 64 bits. Una vez metido, nos pedirá los siguientes 32 bits, que serán los de menor peso (Ver figura 15). Después de introducir todo el dato, este se guarda en el registro indicado (en este caso

10) y pregunta si se desea modificar otro registro. Si se hubiese elegido introducir un dato de 32 bits, únicamente hubiesemos tenido que introducir un dato de 32 bits. Los registros de propósito especial HI y LO, y el contador de programa (CP) no se pueden modificar desde aquí. El único modo de modificar los registros HI y LO es a través de las instrucciones de multiplicación, división y movimiento de datos.

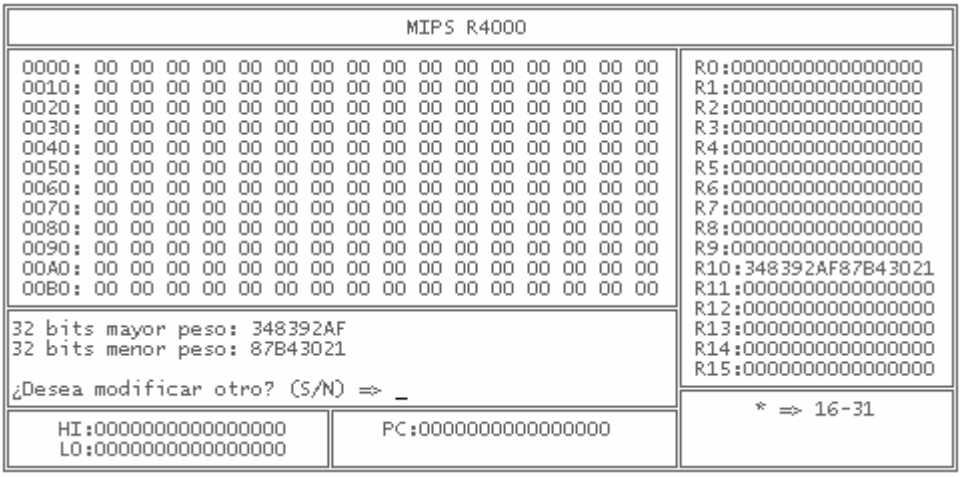


Figura 15

7.- Inicia_CP

Esta opción se suele utilizar cuando se desea que el registro CP apunte a otra dirección distinta (siempre alineada como palabra). De este modo, podemos comenzar a ejecutar un programa a partir de la dirección que nosotros queramos.

Si el valor de la dirección introducida no es múltiplo de cuatro (no está alineada como una palabra), nos volverá a pedir la dirección hasta que sea múltiplo de cuatro

8.- Configuración

Esta opción sirve para cambiar la configuración del simulador. Cuando se pulsa esta opción, el simulador muestra la configuración actual y según nos muestra la figura 16, nos pide si queremos cambiar la configuración.

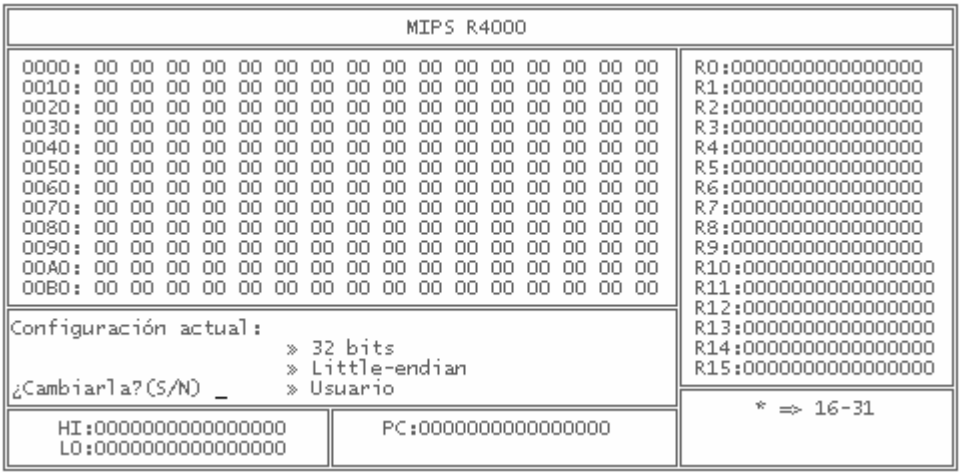


Figura 16

Si la opción pulsada es 'N', la configuración se queda tal y como está. Por el contrario, si la opción pulsada es 'S', nos da a elegir entre modo de ejecución 32 bit y modo de ejecución en 64 bits. Después da a elegir la ordenación de los bytes en memoria (little-endian ó big-endian). Ya se ha cambiado la configuración. Muestra la actual y pulsando una tecla sale al menú principal. La figura 17 muestra cómo pide el modo de operación (32 ó 64 bits).

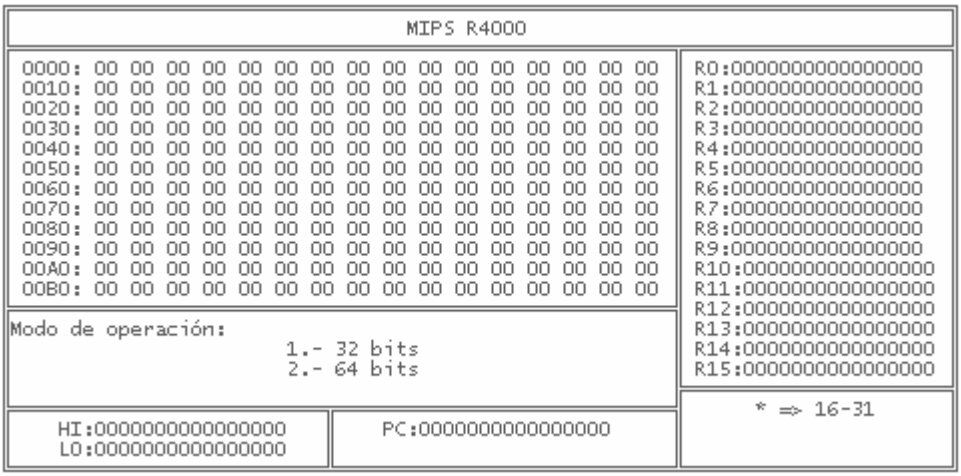


Figura 17

9.- Créditos

Como se ve en la figura 18, la única función que realiza esta opción es mostrar los datos del creador de este proyecto. Los datos que figuran son nombre, apellidos, carrera y año de finalización de este proyecto.

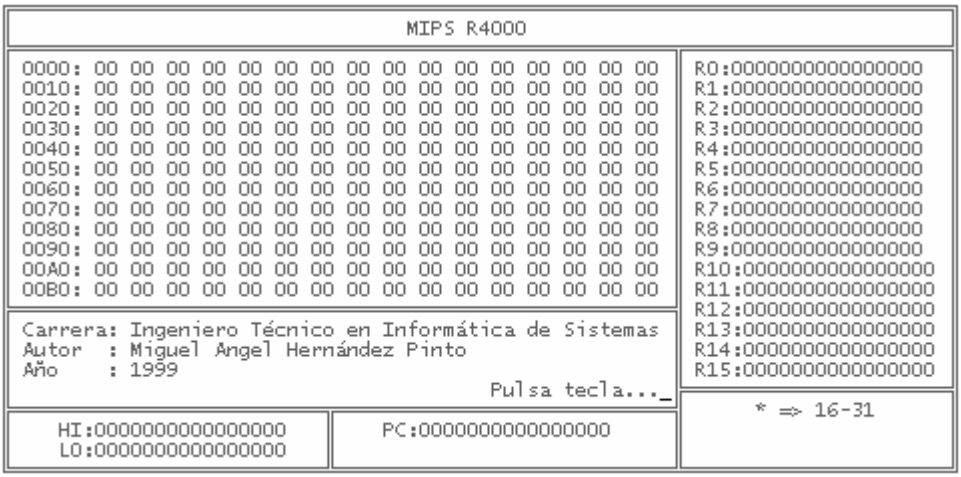


Figura 18

S.- Salir

Se sale del simulador de instrucciones del MIPS R-4000.

CODIFICAR UNA INSTRUCCIÓN

Para introducir cualquiera de estas instrucciones en el simulador, primeramente se debe hacer una codificación de la instrucción. Cada instrucción ocupa 32 bits, y dependiendo de la que sea, se necesitarán unos operandos u otros. Del mismo modo, cada instrucción tiene un código de operación; en algunas ocupa seis bits y en otras cinco bits. Si por ejemplo tenemos la instrucción **ADD r4, r5, r6** (suma con signo los contenidos de los registros r5 y r6 y el resultado lo guarda en el registro r4), y queremos obtener el código de esta instrucción (que será el que meteremos en el simulador), lo primero que debemos hacer es acudir a las tablas proporcionadas en el manual de usuario, donde está el formato, los campos y el código de operación de cada instrucción. Con esta información podremos construir el código de la instrucción arriba indicada. Si accedemos a estas tablas y buscamos la instrucción ADD, vemos que el formato es ADD rd, rs, rt. Tenemos seis campos que debemos rellenar: especial, rs, rt, rd, 0, cop. En el campo especial que ocupa 6 bits metemos seis ceros. En el campo rs queremos un cinco, por tanto metemos un cinco en binario. En el campo rt queremos un seis, por tanto meteremos un seis en binario; y en el campo rd queremos un cuatro, por tanto meteremos un cuatro en binario. El campo 0 son cinco bits a cero y en el campo COP metemos el código de operación de la instrucción ADD que es 100000.

Special	**rs**	**rt**	**Rd**	**0**	**ADD**
000000	00101	00110	00100	00000	100000

Por tanto, si estos 32 bits los separamos de 4 en cuatro bits (para poderlo pasar a hexadecimal) nos queda lo siguiente.

0000	0000	1010	0110	0010	0000	0010	0000
0	**0**	**A**	**6**	**2**	**0**	**2**	**0**

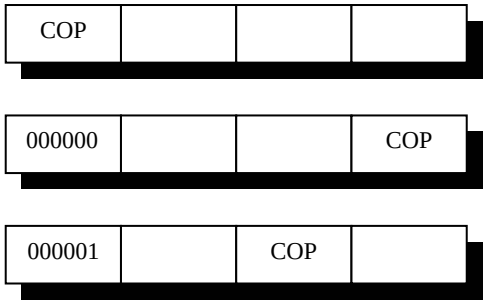
El código de la instrucción ADD r4, r5, r6 es 00A62020 y es el que debemos de introducir en el simulador para que éste lo entienda.

CAPÍTULO 3: DESCRIPCIÓN DEL DISEÑO DEL SIMULADOR DE INSTRUCCIONES

CÓMO SE DECODIFICA Y SE EJECUTA UNA INSTR. EN EL SIMULADOR

En este apartado se va a explicar cómo el simulador a partir del código de la instrucción introducido en memoria, decodifica dicha instrucción para después poder ejecutarla. Por lo tanto, primero vamos a explicar el proceso de decodificación y seguidamente el proceso de la ejecución de la instrucción, pero antes debemos saber lo siguiente:

El código de operación de algunas instrucciones está localizado en los 6 bits de orden más alto, pero hay otros dos tipos de instrucciones, en el primero de los cuales los seis bits de orden más alto son 000000 y el código de operación está situado en los seis bits de orden más bajo; y en el segundo, los seis bits de orden más alto contienen 000001 y el código de operación es de cinco bits desde el bit 16 hasta el 20 inclusive. La siguiente figura ilustra lo explicado en este apartado.



Proceso de decodificación de una instrucción

Este proceso de decodificación se hace para que antes de ejecutar una instrucción aparezca el mnemónico de la misma en la pantalla del simulador. De este modo el usuario del simulador podrá saber en todo momento que instrucción va a ejecutar.

Estructuras de datos utilizadas:

En este apartado voy a explicar las estructuras de datos que he utilizado para decodificar una instrucción. He utilizado un array de estructuras llamado ***formatos*** de 256 posiciones y la estructura que hay en cada posición del array está formada por los campos *formato* y *nombre*. Las instrucciones están divididas en 16 formatos. Cada una de ellas pertenece a un formato. El campo

formato de la estructura es de tipo char y va a contener el número de formato al que pertenece la instrucción (0..15), y el campo *nombre* que es un puntero de tipo char contiene el nombre del mnemónico de la instrucción.

En el array ***formatos*** están ordenadas todas las instrucciones del repertorio del simulador siguiendo un criterio. El criterio de ordenación es el siguiente:

No podemos ordenar las instrucciones por el código de operación, debido a que al tener tres tipos de instrucciones, puede que algún código de operación se repita. Para poder ordenarlas en el array primero declaramos una variable de tipo char. Como el tamaño de dicha variable va a ser de 8 bits, para las instrucciones del primer tipo, ponemos los dos bits de mayor peso de la variable a 10 y se concatenan con los 6 bits del código de operación. Para las del segundo tipo, ponemos los dos bits de mayor peso de la variable a 00 y se concatenan con los 6 bits del código de operación y para el tercer tipo, los tres bits de mayor peso de la variable se ponen a 010 y se concatenan con los 5 bits del código de operación. De esta forma todos los números son diferentes y se pueden ordenar en el array. Para posicionar la instrucción ADD en el array, vemos que pertenece al segundo tipo, por tanto en la variable guardaremos el siguiente valor en binario: 0010 0000 que en decimal corresponde al número 32. Entonces, en la posición 32 del array ***formatos*** estará la estructura formada por el número de formato que corresponde a la instrucción ADD y el nombre del mnemónico de la instrucción.

Otra estructura de datos implementada para la decodificación es un array de punteros a funciones llamado ***tablaf***, en cuyas posiciones hay un puntero que apunta a una función determinada. La posición cero de este array, apuntará a la función Formato_cero, la uno a Formato_uno, y así sucesivamente. Lo que hacen estas funciones es sacar por la pantalla del simulador el mnemónico de la instrucción a ejecutar, junto con sus registros o datos inmediatos dependiendo del modo de direccionamiento que utilice ese formato.

Para que el simulador saque el mnemónico de la instrucción por pantalla (decodificar instrucción), lo primero que hay que hacer es la concatenación de bits explicada

anteriormente, formando así el número de la posición del array ***formato*** a la que hay que acceder. En esa posición del array nos encontramos, como hemos dicho antes, con una estructura de dos campos. De estos dos campos nos interesa coger el dato que hay en el campo *formato*. Con este número podemos acceder a la posición *formato* en el array ***tablaf*** mencionado en las estructuras de datos anteriormente. Como dicho array es de punteros a funciones, si por ejemplo el campo *formato* valiese 3, accederíamos a la posición 3 del array ***tablaf*** que apuntaría a la función Formato_tres que está encargada de sacar por pantalla cualquier instrucción perteneciente al formato tres.

A estas funciones (Formato_cero, Formato_uno, etc...) se les pasa como parámetro el campo nombre de la misma estructura de la cual hemos obtenido el campo formato. Esto se hace para que la función pueda sacar el mnemónico de la instrucción a ejecutar.

Proceso de ejecución de una instrucción

Después del proceso de decodificación, el usuario del simulador ya sabe la instrucción que va a ser ejecutada. Cuando el usuario pulse la tecla 'E', la instrucción que está en la pantalla del simulador se ejecutará.

Estructuras de datos utilizadas:

En este apartado voy a explicar las estructuras de datos que he utilizado para ejecutar una instrucción.

Se han declarado tres arrays de punteros a funciones que se llaman *tabla, tabla0, y tabla1*, cada una de las cuales contiene las instrucciones que le corresponde. Las instrucciones pertenecientes al primer tipo están en tabla, las del segundo tipo en tabla0 y las del tercer tipo en tabla1. En las tres tablas están ordenadas cada una de ellas por número de código de operación. No hay posibilidad de repetirse códigos de operación porque las instrucciones están distribuidas en tres tablas.

Antes de hacer la llamada a la función que ejecuta la instrucción correspondiente, la función encargada de ejecutar las instrucciones mira los seis bits más altos del código de la instrucción que se quiere ejecutar. Si estos bits son distintos de cero o uno, accede a tabla y con el código de operación, se accede al puntero de *tabla* que hace la llamada a

la función que ejecuta la instrucción. Si los seis bits más altos son igual a cero, se miran los seis bits de menor peso (que corresponden al código de operación) y en *tabla0* se accede a la posición del array que hace la llamada a la función; y por último, si los seis bits más altos son igual a uno, se miran los bits 16 al 20, se coge el codigo de operación y se llama a la instrucción mediante *tabla1*.

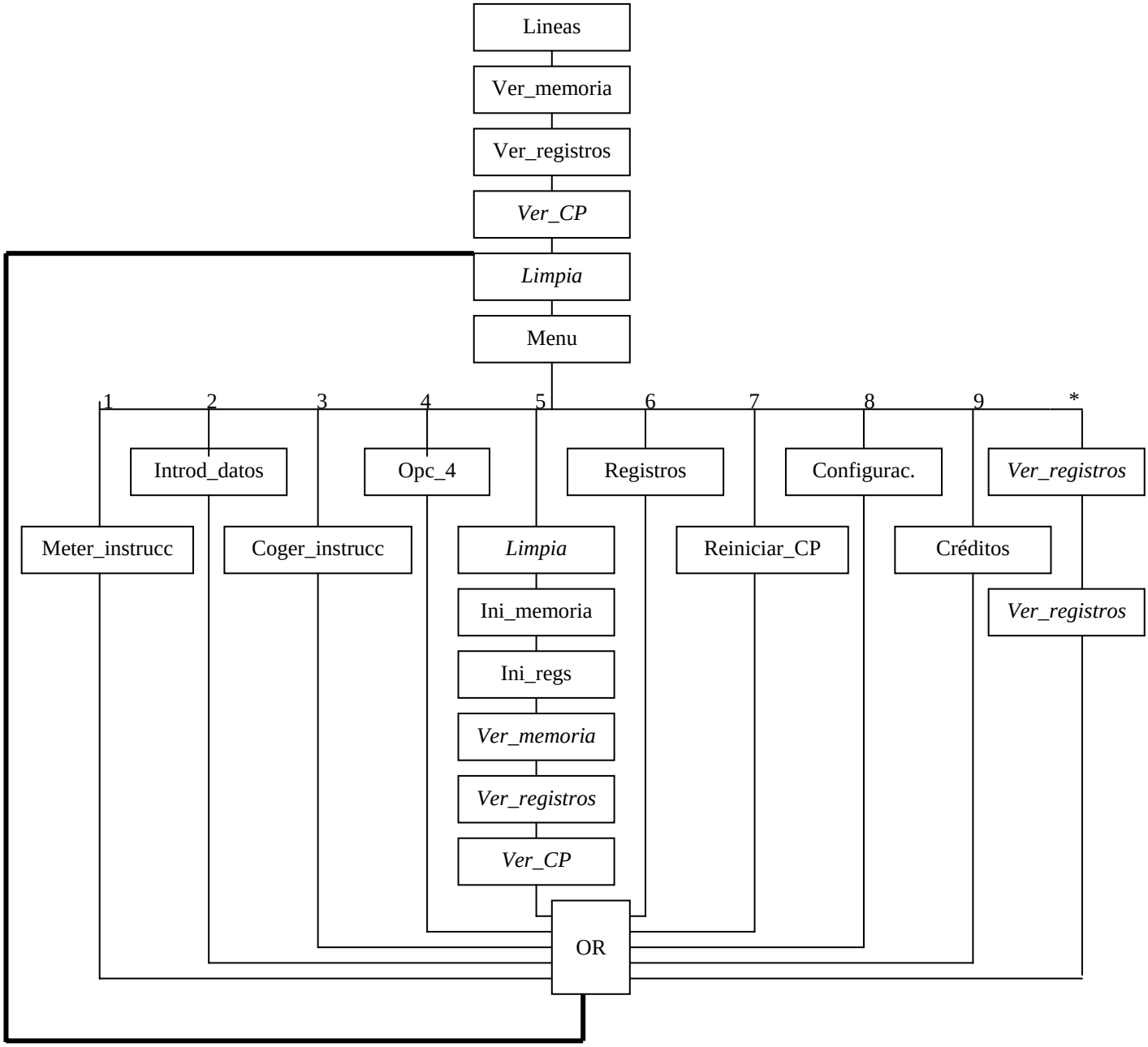
Este ha sido todo el proceso para decodificar y ejecutar una instrucción. Si en algún caso la instrucción no se desea ejecutar y ya se ha decodificado, el usuario debe pulsar la tecla 'P' (parar).

DIAGRAMA DE BLOQUES DEL PROGRAMA

En la siguiente página, podremos ver, mediante un diagrama de cajas, como funciona el programa, los procedimientos a los que se llaman en el programa principal, y las funciones o procedimientos que se llaman dentro de otros.

Primeramente dibuja las líneas del interfaz de usuario. A continuación visualiza la memoria a partir de una dirección dada y visualiza el contenido de los registros de propósito general y los registros HI y LO. Por útimo visualiza por pantalla el contenido del registro Contador de Programa.

A continuación comienza un bucle infinito **do-while(1)**. Limpia una zona de la pantalla y muestra el menú principal. En un **switch** podemos elegir entre once opciones incluyendo la de salir que no se muestra en este diagrama. Con esta opción se sale del bucle infinito y termina el programa.



Cada opción del switch contiene un procedimiento que ejecuta la opción, exceptuando las opciones 5 y *. Cada uno de estos procedimientos hace llamadas a otros procedimientos. En los cuadros siguientes vamos a ver las llamadas que se hacen dentro de cada procedimiento.

METER_INSTRUCC

Limpia ;Limpia una zona de la pantalla

Leedir ;Lee dirección desde teclado

Ver_memoria ;Visualiza la memoria por pantalla

Leeinstr ;Lee instrucción desde teclado

Lontocar ;pasa un dato de 32 bits, a 4 datos de 8 bits

INTROD_DATOS

Limpia ;Limpia una zona de la pantalla

Leedir ;Lee dirección desde teclado

Ver_memoria ;Visualiza la memoria por pantalla

Leebyte ;Lee byte desde teclado

COGER_INSTR

Limpia ;Limpia una zona de la pantalla

Leedir ;Lee dirección desde teclado

Pon_Estado ;Pone en pantalla el estado del micro.

Chartolong ;Pasa 4 datos de 8 bits a uno de 32 bits

Cod_op ;Coge el cód. de operación de la instrucción

Decodificación

INSTRUCCIÓN(Ejecución. De instr.)

Ver_Registros ;Visualiza los registros

Ver_Cp ;Visualiza el contador de Programa

INSTRUCCIÓN

Campo ;Coge de la instr. el campo indicado

Estado ;Devuelve estado de micro (32 o 64 bits)

Acarreo ;Mira si hay acarreo entre bits 31 y 32

Multiplicación ;Hace multiplicación

Dividir ;Hace la división

Lontocar ;Pasa dato de 32 bits a 4 de 8 bits

Chartolong ;Pasa 4 datos de 8 bits a 1 de 32

OPC_4

Limpia ;Limpia una zona de la pantalla

Leedir ;Lee dirección de teclado

Ver_memoria ;Visualiza la memoria por pantalla

REGISTROS

Limpia ;Limpia una zona de la pantalla

Leebytedecimal ;Lee byte en decimal de teclado

Leeinstr ;Lee instrucción de teclado

Ver_Registros ;Visualiza los registros

REINICIAR_CP

Limpia ;Limpia una zona de la pantalla

Leedir ;Lee dirección de teclado

Ver_memoria ;Visualiza la memoria por pantalla

Ver_CP ;Visualiza el contenido del CP

CONFIGURACIÓN

Limpia ;Limpia una zona de la pantalla

Estado ;Devuelve estado de un campo de reg. Estado

CRÉDITOS

Este procedimiento no hace ninguna llamada a otro procedimiento o función dentro de él.

APÉNDICE I: DESCRIPCIÓN DE LAS INSTRUCCIONES IMPLEMENTADAS EN EL SIMULADOR MIPS R4000

ADD (Add)

El **formato** de esta instrucción es:

ADD rd,rs,rt

Descipción de la instrucción:

El contenido del registro rs y el contenido del registro rt se suman y el resultado se guarda en el registro rd. En modo 64 bits, el bit 31 del resultado se extiende.

Ocurre una excepción de desbordamiento si ocurre un "overflow" en complemento a 2. El registro destino (rd) no se modifica cuando ocurre una excepción de desbordamiento.

Operación:

```
32:  GPR[rd] ← GPR[rs] + GPR[rt]

64:  temp ← GPR[rs] + GPR[rt]
     GPR[rd] ← (temp31)32 || temp31..0
```

Excepciones:

Excepción de desbordamiento en complemento a dos. (Integer overflow exception).

ADDI (Add Immediate)

El **formato** de esta instrucción es:

ADDI rt,rs,immediate

Descripción de la instrucción:

Con el dato inmediato de 16 bits se hace una extensión de signo a 32 ó 64 bits(dependiendo del modo de operación) y ese dato se suma al dato que hay en el registro rs. El resultado se guarda en el registro rt. En modo 64 bits, el bit 31 del resultado se extiende.

Ocurre una excepción de desbordamiento si ocurre un "overflow" en complemento a 2. El registro destino (rt) no se modifica cuando ocurre una excepción de desbordamiento.

Operación:

32: $GPR[rt] \leftarrow GPR[rs] + (immediate_{15})^{16} || immediate_{15..0}$

64: $temp \leftarrow GPR[rs] + (immediate_{15})^{48} || immediate_{15..0}$

$GPR[rt] \leftarrow (temp_{31})^{32} || temp_{31..0}$

Excepciones:

Excepción de desbordamiento en complemento a dos. (Integer overflow exception).

ADDIU (Add Immediate Unsigned)

El **formato** de esta instrucción es:

ADDIU rt,rs,immediate

Descripción de la instrucción:

Con el dato inmediato de 16 bits se hace una extensión de signo a 32 ó 64 bits (dependiendo del modo de operación) y ese dato se suma al dato que hay en el registro rs. El resultado se guarda en el registro rt. En modo 64 bits, el bit 31 del resultado se extiende.

La única diferencia entre esta instrucción y la instrucción ADDI es que ADDIU nunca causa una excepción de desbordamiento (Integer Overflow Exception).

Operación:

```
32:  GPR[rt] ← GPR[rs] + (immediate15)16||immediate15..0

64:  temp ← GPR[rs] + (immediate15)48||immediate15..0
     GPR[rt] ← (temp31)32 || temp31..0
```

Excepciones:

Ninguna.

ADDU (Add Unsigned)

El **formato** de la instrucción es:

ADDU rd,rs,rt

Descripción de la instrucción:

El contenido del registro rs y el contenido del registro rt se suman y el resultado se guarda en el registro rd. En modo 64 bits, el bit 31 del resultado se extiende.

La única diferencia entre esta instrucción y la instrucción ADD es que ADDU nunca causa una excepción de desbordamiento (Integer Overflow Exception).

Operación:

```
32:  GPR[rd] ← GPR[rs] + GPR[rt]

64:  temp ← GPR[rs] + GPR[rt]
     GPR[rd] ← (temp31)32 || temp31..0
```

Excepciones:

Ninguna.

AND (And)

El **formato** de la instrucción es:

AND rd,rs,rt

Descripción de la instrucción:

El contenido del registro rs hace la operación lógica AND con el contenido del registro rt, y el resultado es guardado en el registro rd.

Operación:

```
32:  GPR[rd] ← GPR[rs] and GPR[rt]

64:  GPR[rd] ← GPR[rs] and GPR[rt]
```

Excepciones:

Ninguna.

ANDI (And Immediate)

El **formato** de la instrucción es:

ANDI rt,rs,immediate

Descripción de la instrucción:

Al dato inmediato de 16 bits se le hace una extensión a 32 bits poniendo los bits 16 al 32 a cero o a 64 bits poniendo los bits 16 a 63 a cero (depende del modo de

operación) y se hace la operación lógica AND con el registro rs. El resultado se guarda en el registro rt.

Operación:

```
32:  GPR[rt] ← 0^16 || (immediate and GPR[rs]15..0)
64:  GPR[rt] ← 0^48 || (immediate and GPR[rs]15..0)
```

Excepciones:

Ninguna.

BEQ (Branch On Equal)

El **formato** de la instrucción es:

BEQ rs,rt,offset

Descripción de la instrucción:

Se forma una dirección de salto relativo de la suma de la dirección de la siguiente instrucción al salto y el offset de 16 bits con extensión de signo desplazado a la izquierda dos bits. El contenido del registro rs y el contenido del registro rt se comparan. Si los dos registros son iguales, el programa salta a la dirección que se ha formado de la suma anterior.

Operación:

```
32:  target ← (offset15)^14 || offset || 0^2
     condicion ← (GPR[rs]=GPR[rt])
     if condicion then
          PC ← PC + target
     endif
```

```
64:  target ← (offset15)46 || offset || 02
     condicion ← (GPR[rs]=GPR[rt])
     if condicion then
          PC ← PC + target
     endif
```

Excepciones:

Ninguna.

BEQL (Branch On Equal Likely)

El **formato** de la instrucción es:

BEQL rs,rt,offset

Descripción de la instrucción:

Se forma una dirección de salto relativo de la suma de la dirección de la siguiente instrucción al salto y el offset de 16 bits con extensión de signo desplazado a la izquierda dos bits. El contenido del registro rs y el contenido del registro rt se comparan. Si los dos registros son iguales, el programa salta a la dirección que se ha formado de la suma anterior. Si no se cumple la condición, la instrucción siguiente a la instrucción de salto se anula y no se ejecuta.

Operación:

```
32:  target ← (offset15)14 || offset || 02
     condicion ← (GPR[rs]=GPR[rt])
     if condicion then
          PC ← PC + target
     else
          Se anula la siguiente instr.
     endif
```

```
64:  target ← (offset15)46 || offset || 02
     condicion ← (GPR[rs]=GPR[rt])
     if condicion then
          PC ← PC + target
     else
          se anula la siguiente instr.
     endif
```

Excepciones:

Ninguna.

BNE (Branch On Not Equal)

El **formato** de la instrucción es:

BNE rs,rt,offset

Descripción de la instrucción:

Se forma una dirección de salto relativo de la suma de la dirección de la siguiente instrucción al salto y el offset de 16 bits con extensión de signo desplazado a la izquierda dos bits. El contenido del registro rs y el contenido del registro rt se comparan. Si los dos registros son distintos, el programa salta a la dirección que se ha formado de la suma anterior.

Operación:

```
32:  target ← (offset15)14 || offset || 02
     condicion ← (GPR[rs]≠GPR[rt])
     if condicion then
          PC ← PC + target
```

```
        endif

64:  target ← (offset15)46 || offset || 02
     condicion ← (GPR[rs]≠GPR[rt])
     if condicion then
          PC ← PC + target
     endif
```

Excepciones:

Ninguna.

BNEL (Branch On Not Equal Likely)

El **formato** de la instrucción es:

BNEL rs,rt,offset

Descripción de la instrucción:

Se forma una dirección de salto relativo de la suma de la dirección de la siguiente instrucción al salto y el offset de 16 bits con extensión de signo desplazado a la izquierda dos bits. El contenido del registro rs y el contenido del registro rt se comparan. Si los dos registros son distintos, el programa salta a la dirección que se ha formado de la suma anterior. Si no se cumple la condición, la instrucción siguiente a la instrucción de salto se anula y no se ejecuta.

Operación:

```
32:  target ← (offset15)14 || offset || 02
     condicion ← (GPR[rs]≠GPR[rt])
     if condicion then
          PC ← PC + target
```

```
        else
             se anula la siguiente instr.
        endif

64:  target ← (offset15)46 || offset || 02
        condicion ← (GPR[rs]≠GPR[rt])
        if condicion then
             PC ← PC + target
        else
             se anula la siguiente instr.
        endif
```

Excepciones:

Ninguna.

BGTZ (Branch On Greater Than Zero)

El **formato** de la instrucción es:

BGTZ rs,offset

Descripción de la instrucción:

Se forma una dirección de salto relativo de la suma de la dirección de la siguiente instrucción al salto y el offset de 16 bits con extensión de signo desplazado a la izquierda dos bits. Si el contenido del registro rs es mayor que cero, el programa salta a la dirección que se ha formado de la suma anterior.

Operación:

```
32:  target ← (offset15)14 || offset || 02
        condicion ← (GPR[rs] > 0)
```

```
        if condicion then
             PC ← PC + target
        endif

64:     target ← (offset15)46 || offset || 02
        condicion ← (GPR[rs] > 0)
        if condicion then
             PC ← PC + target
        endif
```

Excepciones:

Ninguna.

BGTZL (Branch On Greater Than Zero Likely)

El **formato** de la instrucción es:

BGTZL rs,offset

Descripción de la instrucción:

Se forma una dirección de salto relativo de la suma de la dirección de la siguiente instrucción al salto y el offset de 16 bits con extensión de signo desplazado a la izquierda dos bits. Si el contenido del registro rs es mayor que cero, el programa salta a la dirección que se ha formado de la suma anterior. Si no se cumple la condición, la instrucción siguiente a la instrucción de salto se anula y no se ejecuta.

Operación:

```
32:     target ← (offset15)14 || offset || 02
        condicion ← (GPR[rs] > 0)
```

```
        if condicion then
             PC ← PC + target
        else
             se anula la siguiente instr.
        endif

64:     target ← (offset15)46 || offset || 02
        condicion ← (GPR[rs] > 0)
        if condicion then
             PC ← PC + target
        else
             se anula la siguiente instr.
        endif
```

Excepciones:

Ninguna.

BLEZ (Branch on Less Than Or Equal To Zero)

El **formato** de la instrucción es:

BLEZ rs,offset

Descripción de la instrucción:

Se forma una dirección de salto relativo de la suma de la dirección de la siguiente instrucción al salto y el offset de 16 bits con extensión de signo desplazado a la izquierda dos bits. Si el contenido del registro rs es menor o igual que cero, el programa salta a la dirección que se ha formado de la suma anterior.

Operación:

```
32:  target ← (offset15)14 || offset || 02
     condicion ← (GPR[rs] ≤ 0)
     if condicion then
          PC ← PC + target
     endif

64:  target ← (offset15)46 || offset || 02
     condicion ← (GPR[rs] ≤ 0)
     if condicion then
          PC ← PC + target
     endif
```

Excepciones:

Ninguna.

BLEZL (Branch On Less Than Or Equal To Zero Likely)

El **formato** de la instrucción es:

BLEZL rs,offset

Descripción de la instrucción:

Se forma una dirección de salto relativo de la suma de la dirección de la siguiente instrucción al salto y el offset de 16 bits con extensión de signo desplazado a la izquierda dos bits. Si el contenido del registro rs es menor o igual que cero, el programa salta a la dirección que se ha formado de la suma anterior. Si no se cumple la condición, la instrucción siguiente a la instrucción de salto se anula y no se ejecuta.

Operación:

```
32:  target ← (offset15)^14 || offset || 0^2
     condicion ← (GPR[rs] ≤ 0)
     if condicion then
          PC ← PC + target
     else
          se anula la siguiente instr.
     endif

64:  target ← (offset15)^46 || offset || 0^2
     condicion ← (GPR[rs] ≤ 0)
     if condicion then
          PC ← PC + target
     else
          se anula la siguiente instr.
     endif
```

Excepciones:

Ninguna.

BGEZ (Branch On Greater Than Or Equal To Zero)

El **formato** de la instrucción es:

BGEZ rs,offset

Descripción de la instrucción:

Se forma una dirección de salto relativo de la suma de la dirección de la siguiente instrucción al salto y el offset de 16 bits con extensión de signo desplazado a la izquierda dos bits. Si el contenido del registro rs es mayor o

igual que cero, el programa salta a la dirección que se ha formado de la suma anterior.

Operación:

```
32:  target ← (offset15)^14 || offset || 0^2
     condicion ← (GPR[rs] ≥ 0)
     if condicion then
          PC ← PC + target
     endif

64:  target ← (offset15)^46 || offset || 0^2
     condicion ← (GPR[rs] ≥ 0)
     if condicion then
          PC ← PC + target
     endif
```

Excepciones:

Ninguna.

BGEZAL (Branch On Greater Than Or Equal To Zero And Link)

El **formato** de la instrucción es:

BGEZAL rs,offset

Descripción de la instrucción:

Se forma una dirección de salto relativo de la suma de la dirección de la siguiente instrucción al salto y el offset de 16 bits con extensión de signo desplazado a la izquierda dos bits. La dirección de la instrucción siguiente a la instrucción siguiente a la instrucción de salto se guarda en el registro 31. Si el

contenido del registro rs es mayor o igual que cero, el programa salta a la dirección que se ha formado de la suma anterior.

Operación:

```
32:  target ← (offset15)^14 || offset || 0^2
     condicion ← (GPR[rs] ≥ 0)
     GPR[31] ← PC + 8
     if condicion then
          PC ← PC + target
     endif

64:  target ← (offset15)^46 || offset || 0^2
     condicion ← (GPR[rs] ≥ 0)
     GPR[31] ← PC + 8
     if condicion then
          PC ← PC + target
     endif
```

Excepciones:

Ninguna.

BGEZALL (Branch On Greater Than Or Equal To Zero And Link Likely)

El **formato** de la instrucción es:

BGEZALL rs,offset

Descripción de la instrucción:

Se forma una dirección de salto relativo de la suma de la dirección de la siguiente instrucción al salto y el offset de 16 bits con extensión de signo desplazado a la izquierda dos bits. La dirección de la instrucción siguiente a la instrucción siguiente a la instrucción de salto se guarda en el registro 31. Si el contenido del registro rs es mayor o igual que cero, el programa salta a la dirección que se ha formado de la suma anterior. Si la condición no se cumple, la instrucción siguiente a la instrucción de salto se anula y no se ejecuta.

Operación:

```
32:  target ← (offset15)^14 || offset || 0^2
     condicion ← (GPR[rs] ≥ 0)
     GPR[31] ← PC + 8
     if condicion then
          PC ← PC + target
     else
          se anula la siguiente instr.
     endif

64:  target ← (offset15)^46 || offset || 0^2
     condicion ← (GPR[rs] ≥ 0)
     GPR[31] ← PC + 8
     if condicion then
          PC ← PC + target
     else
          se anula la siguiente instr.
     endif
```

Excepciones:

Ninguna.

BGEZL (Branch On Greater Than Or Equal To Zero Likely)

El **formato** de la instrucción es:

BGEZL rs,offset

Descripción de la instrucción:

Se forma una dirección de salto relativo de la suma de la dirección de la siguiente instrucción al salto y el offset de 16 bits con extensión de signo desplazado a la izquierda dos bits. Si el contenido del registro rs es mayor o igual que cero, el programa salta a la dirección que se ha formado de la suma anterior. Si la condición no se cumple, la instrucción siguiente a la instrucción de salto se anula y no se ejecuta.

Operación:

```
32:  target ← (offset15)^14 || offset || 0^2
     condicion ← (GPR[rs] ≥ 0)
     if condicion then
          PC ← PC + target
     else
          se anula la siguiente instr.
     endif

64:  target ← (offset15)^46 || offset || 0^2
     condicion ← (GPR[rs] ≥ 0)
     if condicion then
          PC ← PC + target
     else
          se anula la siguiente instr.
     endif
```

Excepciones:

Ninguna.

BLTZ (Branch On Less Than Zero)

El **formato** de la instrucción es:

BLTZ rs,offset

Descripción de la instrucción:

Se forma una dirección de salto relativo de la suma de la dirección de la siguiente instrucción al salto y el offset de 16 bits con extensión de signo desplazado a la izquierda dos bits. Si el contenido del registro rs es menor que cero, el programa salta a la dirección que se ha formado de la suma anterior.

Operación:

```
32:  target ← (offset15)^14 || offset || 0^2
     condicion ← (GPR[rs] < 0)
     if condicion then
          PC ← PC + target
     endif

64:  target ← (offset15)^46 || offset || 0^2
     condicion ← (GPR[rs] < 0)
     if condicion then
          PC ← PC + target
     endif
```

Excepciones:

Ninguna.

BLTZAL (Branch On Less Than Zero And Link)

El **formato** de la instrucción es:

BLTZAL rs,offset

Descripción de la instrucción:

Se forma una dirección de salto relativo de la suma de la dirección de la siguiente instrucción al salto y el offset de 16 bits con extensión de signo desplazado a la izquierda dos bits. La dirección de la instrucción siguiente a la instrucción siguiente a la instrucción de salto se guarda en el registro 31. Si el contenido del registro rs es menor que cero, el programa salta a la dirección que se ha formado de la suma anterior.

Operación:

```
32:  target ← (offset15)^14 || offset || 0^2
     condicion ← (GPR[rs] < 0)
     GPR[31] ← PC + 8
     if condicion then
          PC ← PC + target
     endif

64:  target ← (offset15)^46 || offset || 0^2
     condicion ← (GPR[rs] < 0)
     GPR[31] ← PC + 8
     if condicion then
```

```
            PC ← PC + target
      endif
```

Excepciones:

Ninguna.

BLTZALL (Branch On Less Than Zero And Link Likely)

El **formato** de la instrucción es:

BLTZALL rs,offset

Descripción de la instrucción:

Se forma una dirección de salto relativo de la suma de la dirección de la siguiente instrucción al salto y el offset de 16 bits con extensión de signo desplazado a la izquierda dos bits. La dirección de la instrucción siguiente a la instrucción siguiente a la instrucción de salto se guarda en el registro 31. Si el contenido del registro rs es menor que cero, el programa salta a la dirección que se ha formado de la suma anterior. Si la condición de salto no se cumple, la instrucción siguiente a la instrucción de salto se anula y no se ejecuta.

Operación:

```
32:  target ← (offset15)^14 || offset || 0^2
     condicion ← (GPR[rs] < 0)
     GPR[31] ← PC + 8
     if condicion then
          PC ← PC + target
     else
          se anula la siguiente instr.
     endif
```

```
64:  target ← (offset15)46 || offset || 02
     condicion ← (GPR[rs] < 0)
     GPR[31] ← PC + 8
     if condicion then
          PC ← PC + target
     else
          se anula la siguiente instr.
     endif
```

Excepciones:

Ninguna.

BLTZL (Branch On Less Than Zero Likely)

El **formato** de la instrucción es:

BLTZL rs,offset

Descripción de la instrucción:

Se forma una dirección de salto relativo de la suma de la dirección de la siguiente instrucción al salto y el offset de 16 bits con extensión de signo desplazado a la izquierda dos bits. Si el contenido del registro rs es menor que cero, el programa salta a la dirección que se ha formado de la suma anterior. Si la condición de salto no se cumple, la instrucción siguiente a la instrucción de salto se anula y no se ejecuta.

Operación:

```
32:  target ← (offset15)14 || offset || 02
     condicion ← (GPR[rs] < 0)
```

```
        if condicion then
             PC ← PC + target
        else
             se anula la siguiente instr.
        endif

64:     target ← (offset15)46 || offset || 02
        condicion ← (GPR[rs] < 0)
        if condicion then
             PC ← PC + target
        else
             se anula la siguiente instr.
        endif
```

Excepciones:

Ninguna.

DADD (Doubleword Add)

El **formato** de la instrucción es:

DADD rd,rs,rt

Descripción de la instrucción:

El contenido del registro rs y el contenido del registro rt se suma para formar el resultado, que se guarda en el registro rd.

Ocurre una “Integer overflow exception” si hay un desbordamiento en complemento a dos. El registro destino no se modifica cuando ocurre una “Integer overflow exception”.

Esta operación está definida solo para modo de operación en 64 bits. La ejecución de esta instrucción en modo 32 bits causa una "Reserved instruction exception".

Operación:

```
64:  GPR[rd] ← GPR[rs] + GPR[rt]
```

Excepciones:

Integer overflow exception.

Reserved instruction exception (En modo 32 bits).

DADDI (Doubleword Add Immediate)

El **formato** de la instrucción es:

DADDI rt,rs,immediate

Descripción de la instrucción:

En el dato inmediato de 16 bits se hace una extensión de signo y se suma con el contenido del registro rs para formar el resultado que se guarda en el registro rt.

Ocurre una "Integer overflow exception" si se produce un desbordamiento en complemento a dos. El registro destino (rt) no se modifica cuando ocurre una excepción de este tipo.

Esta operación se define solo para modo de operación de 64 bits. La ejecución de esta instrucción en modo 32 bits causa una "Reserved instruction exception".

Operación:

$64:\ GPR[rt] \leftarrow GPR[rs] + (immediate_{15})^{48} || immediate_{15..0}$

Excepciones:

Integer overflow exception.

Reserved instruction exception (En modo 32 bits).

DADDIU (Doubleword Add Immediate Unsigned)

El **formato** de la instrucción es:

DADDIU rt,rs,immediate

Descripción de la instrucción:

En el dato inmediato de 16 bits se hace una extensión de signo y se suma con el contenido del registro rs para formar el resultado que se guarda en el registro rt.

No ocurre “Integer overflow exception” bajo ninguna circunstancia.

La única diferencia entre esta instrucción y la instrucción DADDI es que DADDIU nunca causa una “Integer overflow exception”.

Esta operación está definida para modo de operación 64 bits. La ejecución de esta instrucción en modo 32 bits causa una “Reserved instruction exception”.

Operación:

```
64:  GPR[rt] ← GPR[rs] + (immediate15)48||immediate15..0
```

Excepciones:

Reserved instruction exception (En modo 32 bits).

DADDU (Doubleword Add Unsigned)

El **formato** de la instrucción es:

DADDU rd,rs,rt

Descripción de la instrucción:

El contenido del registro rs y el contenido del registro rt se suma para formar el resultado, que se guarda en el registro rd.

No ocurre "Integer overflow exteption" bajo ninguna circunstancia.

La única diferencia entre esta instrucción y la instrucción DADD es que DADDU nunca causa una "Integer overflow exception".

Esta operación está definida solo para modo de operación en 64 bits. La ejecución de esta instrucción en modo 32 bits causa una "Reserved instruction exception".

Operación:

```
64:  GPR[rd] ← GPR[rs] + GPR[rt]
```

Excepciones:

Reserved instruction exception (En modo 32 bits).

DDIV (Doubleword Divide)

El **formato** de la instrucción es:

DDIV rs,rt

Descripción de la instrucción:

El contenido del registro rs se divide por el contenido del registro rt, tratando ambos operandos como valores en complemento a dos. No ocurre ningúna "Integer overflow excepción", y el resultado de esta operación es indefinido cuando el divisor es cero.

Cuando la operación se completa, el cociente se guarda en el registro de proposito especial LO, y el resto se guarda en el registro de proposito especial HI.

Esta operación se define solo para modo de operación en 64 bits. La ejecución de esta instrucción en modo 32 bits causa una "Reserved instruction exception".

Operación:

```
64:  LO ← GPR[rs] div GPR[rt]
     HI ← GPR[rs] mod GPR[rt]
```

Excepciones:

Reserved instruction exception (En modo 32 bits).

DDIVU (Doubleword Divide Unsigned)

El **formato** de la instrucción es:

DDIVU rs,rt

Descripción de la instrucción:

El contenido del registro rs se divide por el contenido del registro rt, tratando ambos operandos como valores sin signo. No ocurre ningúna "Integer overflow excepción", y el resultado de esta operación es indefinido cuando el divisor es cero.

Cuando la operación se completa, el cociente se guarda en el registro de proposito especial LO, y el resto se guarda en el registro de proposito especial HI.

Esta operación se define solo para modo de operación en 64 bits. La ejecución de esta instrucción en modo 32 bits causa una "Reserved instruction exception".

Operación:

```
64:  LO ← (0 || GPR[rs]) div (0 || GPR[rt])
     HI ← (0 || GPR[rs]) mod (0 || GPR[rt])
```

Excepciones:

Reserved instruction exception (En modo 32 bits).

DIV (Divide)

El **formato** de la instrucción es:

DIV rs,rt

Descripción de la instrucción:

El contenido del registro rs se divide por el contenido del registro rt, tratando ambos operandos como valores en complemento a dos. No ocurre ninguna excepción de desbordamiento, y el resultado de esta operación es indefinido cuando el divisor es cero.

En modo 64 bits, los operandos deben ser valores de 32 bits con extensión de signo.

Cuando la operación se completa, el cociente se carga en el registro LO, y el resto se carga en el registro HI.

Operación:

```
32:  LO ← GPR[rs] div GPR[rt]
     HI ← GPR[rs] mod GPR[rt]

64:  q  ← GPR[rs]31..0 div GPR[rt]31..0
     r  ← GPR[rs]31..0 div GPR[rt]31..0
```

```
LO ← (q31)32 || q31..0
HI ← (r31)32 || r31..0
```

Excepciones:

Ninguna.

DIVU (Divide Unsigned)

El **formato** de la instrucción es:

DIVU rs,rt

Descripción de la instrucción:

El contenido del registro rs se divide por el contenido del registro rt, tratando ambos operandos como valores sin signo. No ocurre ninguna excepción de desbordamiento, y el resultado de esta operación es indefinido cuando el divisor es cero.

En modo 64 bits, los operandos deben ser valores de 32 bits con extensión de signo.

Cuando la operación se completa, el cociente se carga en el registro LO, y el resto se carga en el registro HI.

Operación:

```
32:  LO ← (0 || GPR[rs]) div (0 || GPR[rt])
     HI ← (0 || GPR[rs]) mod (0 || GPR[rt])

64:  q  ← (0 || GPR[rs]31..0) div (0 || GPR[rt]31..0)
     r  ← (0 || GPR[rs]31..0) div (0 || GPR[rt]31..0)
     LO ← (q31)32 || q31..0
```

```
HI ← (r31)32 || r31..0
```

Excepciones:

Ninguna.

DMULT (Doubleword Multiply)

El **formato** de la instrucción es:

DMULT rs,rt

Descripción de la instrucción:

El contenido del registro rs y rt se multiplican, tratando ambos operandos como valores en complemento a dos. No ocurre una "Integer overflow exception".

Cuando la operación se completa, la parte baja del resultado se guarda en el registro de proposito especial LO, y la parte alta se guarda en el registro de proposito especial HI.

Esta operación se define solo para modo de operación en 64 bits. La ejecución de esta instrucción en modo 32 bits, causa una "Reserved instruction exception).

Operación:

```
64:  t  ← GPR[rs] * GPR[rt]
     LO ← t63..0
     HI ← t127..64
```

Excepciones:

Reserved instruction exception (En modo 32 bits).

DMULTU (Doubleword Multiply Unsigned)

El **formato** de la instrucción es:

DMULTU rs,rt

Descripción de la instrucción:

El contenido del registro rs y el contenido del registro rt se multiplican, tratando ambos operandos como valores sin signo. No ocurre ninguna "Integer overflow exception".

Cuando la operación se completa, la parte baja del resultado se guarda en el registro de proposito especial LO, y la parte alta del resultado se guarda en el regitro de proposito especial HI.

Esta operación se define solo para modo de operación en 64 bits. La ejecución de esta instrucción en modo 32 bits causa una "Reserved instruction exception".

Operación:

```
64:  t  ← (0 || GPR[rs]) * (0 || GPR[rt])
     LO ← t63..0
     HI ← t127..64
```

Excepciones:

Reserved instruction exception (En modo 32 bits).

DSLL (Doubleword Shift Left Logical)

El **formato** de la instrucción es:

DSLL rd,rt,sa

Descripción de la instrucción:

El contenido del registro rt es desplazado a la izquierda sa bits, insertando ceros en los bits de orden bajo. El resultado se guarda en el registro rd.

Operación:

```
64:  s ← 0 || sa
     GPR[rd] ← GPR[rt](63-s)..0 || 0^s
```

Excepciones:

Reserved Instruction exception (En modo 32-bit, usuario o supervisor). En Kernel o 64 bits no se produce.

DSLLV (Doubleword Shift Left Logical Variable)

El **formato** de la instrucción es:

DSLLV rd,rt,rs

Descripción de la instrucción:

El contenido del registro rt es desplazado a la izquierda el número de bits especificado por los seis bits de menor peso que contiene el registro rs, insertando ceros en los bits de menor peso. El resultado se guarda en el registro rd.

Esta operación se define solo para modo de operación de 64 bits. La ejecución de esta instrucción en modo 32 bits causa una “reserved instruction exception”.

Operación:

```
64:  s ← GPR[rs]5..0
     GPR[rd] ← GPR[rt](63-s)..0 || 0^s
```

Excepciones:

Reserved Instruction exception (En modo 32-bit).

DSLL32 (Doubleword Shift Left Logical + 32)

El **formato** de la instrucción es:

DSLL32 rd,rt,sa

Descripción de la instrucción:

El contenido del registro rt es desplazado a la izquierda 32+sa bits, instertando ceros in los bits de orden bajo. El resultado se guarda en el registro rd.

Esta operación se define solo para modo de operación de 64 bits. La ejecución de esta instrucción en modo 32 bits causa una "reserved instruction exception".

Operación:

```
64:  s ← 1 || sa
     GPR[rd] ← GPR[rt](63-s)..0 || 0^s
```

Excepciones:

Reserved instruction exception (En modo 32 bits).

DSRA (Doubleword Shift Right Arithmetic)

El **formato** de la instrucción es:

DSRA rd,rt,sa

Descripción de la instrucción:

El contenido del registro rt es desplazado a la derecha sa bits, haciendo una extensión de signo en los bits de mayor peso. El resultado se guarda en el registro rd.

Esta operación se define solo para modo de operación de 64 bits. La ejecución de esta instrucción en modo 32 bits causa una "reserved instruction exception".

Operación:

```
64:  s ← 0 || sa
     GPR[rd] ← (GPR[rt]63)^s || GPR[rt]63..s
```

Excepciones:

Reserved instruction exception (En modo 32 bits).

DSRAV (Doubleword Shift Right Arithmetic Variable)

El **formato** de la instrucción es:

DSRAV rd,rt,rs

Descripción de la instrucción:

El contenido del registro rt se desplaza a la derecha el número de bits especificado por los seis bits de orden bajo del registro rs, haciendo una extensión de signo en los bits de mayor peso. El resultado se guarda en el registro rd.

Esta operación se define solo para modo de operación de 64 bits. La ejecución de esta instrucción en modo 32 bits causa una "reserved instruction exception".

Operación:

```
64:  s ← GPR[rs]5..0
     GPR[rd] ← (GPR[rt]63)^s || GPR[rt]63..s
```

Excepciones:

Reserved instruction exception (En modo 32 bits).

DSRA32 (Doubleword Shift Right Arithmetic + 32)

El **formato** de la instrucción es:

DSRA32 rd,rt,sa

Descripción de la instrucción:

El contenido del registro rt se desplaza a la derecha 32+sa bits, haciendo una extensión de signo en los bits de mayor peso. El resultado se guarda en el registro rd.

Esta operación se define solo para modo de operación de 64 bits. La ejecución de esta instrucción en modo 32 bits causa una "reserved instruction exception".

Operación:

```
64:  s ← 1 || sa
     GPR[rd] ← (GPR[rt]63)^s || GPR[rt]63..s
```

Excepciones:

Reserved instruction exception (En modo 32 bits).

DSRL (Doubleword Shift Right Logical)

El **formato** de la instrucción es:

DSRL rd,rt,sa

Descripción de la instrucción:

El contenido del registro rt se desplaza a la derecha sa bits, insertando ceros en los bits de mayor peso. El resultado se guarda en el registro rd.

Esta operación se define solo para modo de operación de 64 bits. La ejecución de esta instrucción en modo 32 bits causa una "reserved instruction exception".

Operación:

```
64:  s ← 0 || sa
     GPR[rd] ← 0^s || GPR[rt]63..s
```

Excepciones:

Reserved instruction exception (En modo 32 bits).

DSRLV (Doubleword Shift Right Logical Variable)

El **formato** de la instrucción es:

DSRLV rd,rt,rs

Descripción de la instrucción:

El contenido del registro rt se desplaza a la derecha el número de bits especificado por los seis bits de menor peso del registro rs, insertando ceros en los bits de mayor peso. El resultado se guarda en el registro rd.

Esta operación se define solo para modo de operación de 64 bits. La ejecución de esta instrucción en modo 32 bits causa una "reserved instruction exception".

Operación:

```
64:  s ← GPR[rs]5..0
     GPR[rd] ← 0^s || GPR[rt]63..s
```

Excepciones:

Reserved instruction exception (En modo 32 bits).

DSRL32 (Doubleword Shift Right Logical + 32)

El **formato** de la instrucción es:

DSRL32 rd,rt,sa

Descripción de la instrucción:

El contenido del registro rt se desplaza a la derecha 32+sa bits, insertando ceros en los bits de mayor peso. El resultado se guarda en el registro rd.

Esta operación se define solo para modo de operación de 64 bits. La ejecución de esta instrucción en modo 32 bits causa una "reserved instruction exception".

Operación:

```
64:  s ← 1 || sa
     GPR[rd] ← 0^s || GPR[rt]63..s
```

Excepciones:

Reserved instruction exception (En modo 32 bits).

DSUB (Doubleword Subtract)

El **formato** de la instrucción es:

DSUB rd,rs,rt

Descripción de la instrucción:

El contenido del registro rt se resta del contenido del registro rs para formar un resultado que se guarda en el registro rd.

Ocurre una "Integer overflow exception" si se produce un desbordamiento en complemento a dos. El registro destino rd no se modifica cuando ocurre esta excepción.

Esta operación está definida solo para modo de operación en 64 bits. La ejecución de esta instrucción en modo 32 bits causa una "Reserved instruction exception".

Operación:

```
64:  GPR[rd] ← GPR[rs] - GPR[rt]
```

Excepciones:

Integer overflow exception.

Reserved instruction exception (En modo 32 bits).

DSUBU (Doubleword Subtract Unsigned)

El **formato** de la instrucción es:

DSUBU rd,rs,rt

Descripción de la instrucción:

El contenido del registro rt se resta del contenido del registro rs para formar un resultado que se guarda en el registro rd.

La única diferencia entre esta instrucción y la instrucción DSUB es que DSUBU nunca causa desbordamiento. No ocurre nunca una "Integer overflow exception".

Esta operación solo está definida para modo de operación en 64 bits. La ejecución de esta instrucción en modo 32 bits causa una "Reserved instruction exception".

Operación:

```
64:  GPR[rd] ← GPR[rs] – GPR[rt]
```

Excepciones:

Reserved instruction exception (En modo 32 bits).

J (Jump)

El **formato** de la instrucción es:

J target

Descripción de la instrucción:

La dirección de 26 bits del campo target se desplaza hacia la izquierda dos bits y se combina con los bits de orden alto de la dirección del CP. El programa salta a esta dirección.

Operación:

```
32:  temp ← target
     PC ← PC31..28 || temp || 0^2

64:  temp ← target
     PC ← PC63..28 || temp || 0^2
```

Excepciones:

Ninguna.

JAL (Jump And Link)

El **formato** de la instrucción es:

JAL target

Descripción de la instrucción:

La dirección de 26 bits del campo target se desplaza hacia la izquierda dos bits y se combina con los bits de orden alto de la dirección del CP. El programa salta a esta dirección. La dirección de la instrucción siguiente a la instrucción siguiente a la instrucción de salto se guarda en el registro 31.

Operación:

```
32:  temp ← target
     GPR[31] ← PC + 8
     PC ← PC31..28 || temp || 0^2

64:  temp ← target
     GPR[31] ← PC + 8
     PC ← PC63..28 || temp || 0^2
```

Excepciones:

Ninguna.

JALR (Jump Register)

El **formato** de la instrucción es:

JALR rs

JALR rd,rs

Descripción de la instrucción:

El programa salta a la dirección contenida en el registro rs.

La dirección de la instrucción siguiente a la instrucción siguiente a la instrucción de salto se guarda en el registro rd. El valor por defecto del registro rd es 31 si el valor del campo rd es cero.

La dirección contenida en el registro rs debe ser múltiplo de cuatro debido a que las instrucciones están alineadas como palabras. Si la dirección no es múltiplo de cuatro, ocurre una excepción de error de direccionamiento.

Operación:

```
32,64:    temp ← GPR[rs]
          GPR[rd] ← PC + 8
          PC ← temp
```

Excepciones:

Address Error Exception (Error de direccionamiento).

JR (Jump Register)

El **formato** de la instrucción es:

JR rs

Descripción de la instrucción:

El programa salta a la dirección contenida en el registro rs.

La dirección contenida en el registro rs debe ser múltiplo de cuatro debido a que las instrucciones están alineadas como palabras. Si la dirección no es múltiplo de cuatro, ocurre una excepción de error de direccionamiento.

Operación:

```
32,64:    temp ← GPR[rs]
          PC ← temp
```

Excepciones:

Address Error Exception (Error de direccionamiento).

LB (Load Byte)

El **formato** de la instrucción es:

LB rt,offset(base)

Descripción de la instrucción:

En el desplazamiento de 16 bits se hace una extensión de signo y se suma a el contenido del registro base para formar la dirección a la que vamos a acceder. Al byte localizado en la dirección de memoria especificada, se le hace una extensión de signo y se carga en el registro rt.

Operación:

```
32:  direccion ← ((offset15)16||offset15..0)+GPR[base]
     temp ← byte (de la dirección especificada)
     GPR[rt] ← (temp7)24 || temp

64:  direccion ← ((offset15)48||offset15..0)+GPR[base]
     temp ← byte (de la dirección especificada)
     GPR[rt] ← (temp7)56 || temp
```

Excepciones:

Ninguna.

LBU (Load Byte)

El **formato** de la instrucción es:

LBU rt,offset(base)

Descripción de la instrucción:

En el desplazamiento de 16 bits se hace una extensión de signo y se suma a el contenido del registro base para formar la dirección a la que vamos a acceder. Al byte localizado en la dirección de memoria especificada, se le hace una extensión con ceros y se carga en el registro rt.

Operación:

```
32:  direccion ← ((offset15)^16||offset15..0)+GPR[base]
     temp ← byte (de la dirección especificada)
     GPR[rt] ← 0^24 || temp

64:  direccion ← ((offset15)^48||offset15..0)+GPR[base]
     temp ← byte (de la dirección especificada)
     GPR[rt] ← 0^56 || temp
```

Excepciones:

Ninguna.

LD (Load Doubleword)

El **formato** de la instrucción es:

LD rt,offset(base)

Descripción de la instrucción:

Al desplazamiento de 16 bits se le hace una extensión de signo y se suma a el contenido del registro base para formar la dirección a la que vamos a acceder. Los 64 bits de la doble palabra localizada a partir de la dirección de memoria especificada, se guardan en el registro rt.

Si alguno de los tres bits menos significativos de la dirección a la que se va a acceder son distintos de cero, ocurre una "Address error exception" (excepción de error de direccionamiento).

Esta operación solo está definida en modo de operación de 64 bits. La ejecución de esta instrucción en modo 32 bits causa una "Reserved instruction exception".

Operación:

```
64:  direccion ← ((offset15)48||offset15..0)+GPR[base]
     temp ← Doubleword (a partir de la dirección)
     GPR[rt] ← temp
```

Excepciones:

Address error exception.

Reserved instruction exception (En modo usuario 32 bits o supervisor 32 bits).

LDL (Load Doubleword Left)

El **formato** de la instrucción es:

LDL rt,offset(base)

Descripción de la instrucción:

Esta instrucción puede usarse en combinación con la instrucción LDR para cargar un registro con ocho bytes consecutivos desde memoria, cuando los bytes cruzan el límite de una doble palabra.

Al desplazamiento de 16 bits se le hace una extensión de signo y se suma al contenido del registro base para formar una dirección la cual especifica un byte arbitrario. La instrucción lee bytes de la doble palabra en memoria que contiene el byte de comienzo especificado. Se cargarán de uno a ocho bytes, dependiendo del byte de comienzo. Esta operación solo está definida en modo de operación de 64 bits. La ejecución de esta instrucción en modo 32 bits causa una "Reserved instruction exception".

Operación:

Tenemos una doble palabra (doubleword) que es: 394573587631292A

Para Big Endian:

Memoria:

0000: 39 45 73 58 76 31 29 2A

Registro:

Desplazam.	Registro rt							
0 →	39	45	73	58	76	31	29	2A
1 →	45	73	58	76	31	29	2A	X
2 →	73	58	76	31	29	2A	X	X
3 →	58	76	31	29	2A	X	X	X
4 →	76	31	29	2A	X	X	X	X
5 →	31	29	2A	X	X	X	X	X
6 →	29	2A	X	X	X	X	X	X
7 →	2A	X	X	X	X	X	X	X

Para Little Endian:

Memoria:

0000: 2A 29 31 76 58 73 45 39

Registro:

Desplazam.	Registro rt							
0 →	2A	X	X	X	X	X	X	X
1 →	29	2A	X	X	X	X	X	X
2 →	31	29	2A	X	X	X	X	X
3 →	76	31	29	2A	X	X	X	X
4 →	58	76	31	29	2A	X	X	X
5 →	73	58	76	31	29	2A	X	X
6 →	45	73	58	76	31	29	2A	X
7 →	39	45	73	58	76	31	29	2A

X: Lo que había en el registro antes de LDL

Excepciones:

Reserved instruction exception (En modo 32 bits).

LDR (Load Doubleword Right)

El **formato** de la instrucción es:

LDR rt,offset(base)

Descripción de la instrucción:

Esta instrucción puede usarse en combinación con la instrucción LDL para cargar un registro con ocho bytes consecutivos desde memoria, cuando los bytes cruzan el límite de una doble palabra.

Al desplazamiento de 16 bits se le hace una extensión de signo y se suma al contenido del registro base para formar una dirección la cual especifica un byte arbitrario. La instrucción lee bytes de la doble palabra en memoria que contiene el byte de comienzo especificado. Se cargarán de uno a ocho bytes, dependiendo del byte de comienzo. Esta operación solo está definida en modo de operación de 64 bits. La ejecución de esta instrucción en modo 32 bits causa una "Reserved instruction exception".

Operación:

Tenemos una doble palabra (doubleword) que es: 394573587631292A

Para Big Endian:

Memoria:

0000: 39 45 73 58 76 31 29 2A

Registro:

Desplazam.	Registro rt							
0 →	X	X	X	X	X	X	X	39
1 →	X	X	X	X	X	X	39	45
2 →	X	X	X	X	X	39	45	73
3 →	X	X	X	X	39	45	73	58
4 →	X	X	X	39	45	73	58	76
5 →	X	X	39	45	73	58	76	31
6 →	X	39	45	73	58	76	31	29
7 →	39	45	73	58	76	31	29	2A

Para Little Endian:

Memoria:

0000: 2A 29 31 76 58 73 45 39

Registro:

Desplazam.	Registro rt							
0 →	39	45	73	58	76	31	29	2A
1 →	X	39	45	73	58	76	31	29
2 →	X	X	39	45	73	58	76	31
3 →	X	X	X	39	45	73	58	76
4 →	X	X	X	X	39	45	73	58
5 →	X	X	X	X	X	39	45	73
6 →	X	X	X	X	X	X	39	45
7 →	X	X	X	X	X	X	X	39

X: Lo que había en el registro antes de LDR

Excepciones:

Reserved instruction exception (En modo 32 bits).

LH (Load Halfword)

El **formato** de la instrucción es:

LH rt,offset(base)

Descripción de la instrucción:

Al desplazamiento de 16 bits se le hace una extensión de signo y se suma a el contenido del registro base para formar la dirección a la que vamos a acceder. A la media palabra (Halfword) localizada a partir de la dirección de memoria especificada, se le hace una extensión de signo y se guarda en el registro rt.

Si el bit menos significativo de la dirección a la que se va a acceder es distinto de cero, ocurre una "Address error exception".

Operación:

```
32:  direccion ← ((offset15)16||offset15..0)+GPR[base]
     temp ← Halfword (A partir de la dirección)
     GPR[rt] ← (temp15)16 || temp

64:  direccion ← ((offset15)48||offset15..0)+GPR[base]
     temp ← Halfword (A partir de la dirección)
     GPR[rt] ← (temp15)48 || temp
```

Excepciones:

Address error exception.

LHU (Load Halfword Unsigned)

El **formato** de la instrucción es:

LHU rt,offset(base)

Descripción de la instrucción:

Al desplazamiento de 16 bits se le hace una extensión de signo y se suma a el contenido del registro base para formar la dirección a la que vamos a acceder. A la media palabra (Halfword) localizada a partir de la dirección de memoria especificada, se le hace una extensión con ceros y se guarda en el registro rt.

Si el bit menos significativo de la dirección a la que se va a acceder es distinto de cero, ocurre una "Address error exception".

Operación:

```
32:  direccion ← ((offset15)16||offset15..0)+GPR[base]
     temp ← Halfword (A partir de la dirección)
     GPR[rt] ← 016 || temp
```

64: direccion ← ((offset$_{15}$)^{48}||offset$_{15..0}$)+GPR[base]
temp ← Halfword (A partir de la dirección)
GPR[rt] ← 0^{48} || temp

Excepciones:

Address error exception.

LUI (Load Upper Immediate)

El **formato** de la instrucción es:

LUI rt,immediate

Descripción de la instrucción:

El dato inmediato de 16 bits se desplaza hacia la izquierda 16 bits y se concatena con 16 bits de ceros. El resultado se guarda en el registro rt. En modo 64 bits, a la palabra que se guarda en el registro se le hace una extensión de signo.

Operación:

32: GPR[rt] ← immediate || 0^{16}

64: GPR[rt] ← (immediate$_{15}$)^{32} || immediate || 0^{16}

Excepciones:

Ninguna.

LW (Load Word)

El **formato** de la instrucción es:

LW rt,offset(base)

Descripción de la instrucción:

Al desplazamiento de 16 bits se le hace una extensión de signo y se suma a el contenido del registro base para formar la dirección a la que vamos a acceder. La palabra localizada a partir de la dirección de memoria especificada, se guarda en el registro rt. En modo 64 bits, a esta palabra se le hace una extensión de signo.

Si alguno de los dos bits menos significativos de la dirección a la que vamos a acceder es distinto de cero, ocurre una "Address error exception".

Operación:

```
32:  direccion ← ((offset15)16||offset15..0)+GPR[base]
     temp ← Word (A partir de la dirección)
     GPR[rt] ← temp

64:  direccion ← ((offset15)48||offset15..0)+GPR[base]
     temp ← Word (A partir de la dirección)
     GPR[rt] ← (temp31)32 || temp
```

Excepciones:

Address error exception.

LWL (Load Word Left)

El **formato** de la instrucción es:

LWL rt,offset(base)

Descripción de la instrucción:

Esta instrucción puede usarse en combinación con la instrucción LWR para cargar un registro con cuatro bytes consecutivos desde memoria, cuando los bytes cruzan el límite de una palabra.

Al desplazamiento de 16 bits se le hace una extensión de signo y se suma al contenido del registro base para formar una dirección la cual especifica un byte arbitrario. La instrucción lee bytes de la palabra en memoria que contiene el byte de comienzo especificado. Se cargarán de uno a cuatro bytes, dependiendo del byte de comienzo. En modo de operación 64 bits, se hace una extensión de signo de la palabra cargada.

Operación:

Tenemos una palabra (word) que es:

39457358

Para Big Endian:

Memoria:

0000: 39 45 73 58

Registro:

Desplazam.	Registro rt							
0 →	S	S	S	S	39	45	73	58
1 →	S	S	S	S	45	73	58	X
2 →	S	S	S	S	73	58	X	X
3 →	S	S	S	S	58	X	X	X

Para Little Endian:

Memoria:

0000: 58 73 45 39

Registro:

Desplazam.	Registro rt							
0 →	S	S	S	S	58	X	X	X
1 →	S	S	S	S	73	58	X	X
2 →	S	S	S	S	45	73	58	X
3 →	S	S	S	S	39	45	73	58

X: Lo que había en el registro antes de LWL

S: Extensión de signo

Excepciones:

Ninguna.

LWR (Load Word Right)

El **formato** de la instrucción es:

LWR rt,offset(base)

Descripción de la instrucción:

Esta instrucción puede usarse en combinación con la instrucción LWL para cargar un registro con cuatro bytes consecutivos desde memoria, cuando los bytes cruzan el límite de una palabra.

Al desplazamiento de 16 bits se le hace una extensión de signo y se suma al contenido del registro base para formar una dirección la cual especifica un byte arbitrario. La instrucción lee bytes de la palabra en memoria que contiene el byte de comienzo especificado. Se cargarán de uno a cuatro bytes, dependiendo del byte de comienzo. En modo de operación 64 bits, se hace una extensión de signo del bit 31 del registro.

Operación:

Tenemos una palabra (word) que es:

39457358

Para Big Endian:

Memoria:

0000: 39 45 73 58

Registro:

Desplazam.	Registro rt							
0 →	S	S	S	S	X	X	X	39
1 →	S	S	S	S	X	X	39	45
2 →	S	S	S	S	X	39	45	73
3 →	S	S	S	S	39	45	73	58

Para Little Endian:

Memoria:

0000: 58 73 45 39

Registro:

Desplazam.	Registro rt							
0 →	S	S	S	S	39	45	73	58
1 →	S	S	S	S	X	39	45	73
2 →	S	S	S	S	X	X	39	45
3 →	S	S	S	S	X	X	X	39

X: Lo que había en el registro antes de LWR

S: Extensión de signo

Excepciones:

Ninguna.

LWU (Load Word Unsigned)

El **formato** de la instrucción es:

LWU rt,offset(base)

Descripción de la instrucción:

Al desplazamiento de 16 bits se le hace una extensión de signo y se suma al contenido del registro base para formar la dirección a la que vamos a acceder. La palabra localizada a partir de la dirección de memoria especificada, se guarda en el registro rt. A esta palabra que hemos guardado se le hace una extensión con ceros.

Si alguno de los dos bits menos significativos de la dirección a la que vamos a acceder es distinto de cero, ocurre una "Address error exception".

Esta operación está definida solo para modo de operación en 64 bits. La ejecución de esta instrucción en modo 32 bits causa una "Reserved instruction exception).

Operación:

```
64:  direccion ← ((offset15)48||offset15..0)+GPR[base]
     temp ← Word (A partir de la dirección)
     GPR[rt] ← 032 || temp
```

Excepciones:

Address error exception.

Reserved instruction exception (En modo 32 bits).

MFHI (Move From Hi)

El **formato** de la instrucción es:

MFHI rd

Descripción de la instrucción:

El contenido del registro de proposito especial HI es cargado en el registro de proposito general rd.

Operación:

```
32,64:    GPR[rd] ← HI
```

Excepciones:

Ninguna.

MFLO (Move From Lo)

El **formato** de la instrucción es:

MFLO rd

Descripción de la instrucción:

El contenido del registro de proposito especial LO es cargado en el registro de proposito general rd.

Operación:

```
32,64:    GPR[rd] ← LO
```

Excepciones:

Ninguna.

MTHI (Move To Hi)

El **formato** de la instrucción es:

MTHI rs

Descripción de la instrucción:

El contenido del registro de proposito general rs se carga en el registro de proposito especial HI.

Operación:

```
32,64:    HI ← GPR[rs]
```

Excepciones:

Ninguna.

MTLO (Move To Lo)

El **formato** de la instrucción es:

MTLO rs

Descripción de la instrucción:

El contenido del registro de proposito general rs se carga en el registro de proposito especial LO.

Operación:

```
32,64:    LO ← GPR[rs]
```

Excepciones:

Ninguna.

MULT (Multiply)

El **formato** de la instrucción es:

MULT rs,rt

Descripción de la instrucción:

El contenido del registro rs y rt se multiplican, tratando ambos operandos como valores de 32 bits en complemento a dos. No ocurre ninguna "Integer overflow exception". En modo 64 bits, los operandos deben ser valores de 32 bits con extensión de signo.

Cuando la operación se completa, la parte baja del resultado se guarda en el registro LO, y la parte alta del resultado se guarda en el registro HI.

Operación:

```
32:  t  ← GPR[rs] * GPR[rt]
     LO ← t31..0
     HI ← t63..32

64:  t  ← GPR[rs]31..0 * GPR[rt]31..0
     LO ← (t31)32 || t31..0
     HI ← (t63)32 || t63..32
```

Excepciones:

Ninguna.

MULTU (Multiply Unsigned)

El **formato** de la instrucción es:

MULTU rs,rt

Descripción de la instrucción:

El contenido del registro rs y el contenido del registro rt se multiplican, tratando ambos operandos como valores sin signo. No ocurre "Integer overflow exception". En modo 64 bits, los operandos deben ser valores de 32 bits con extensión de signo.

Cuando la operación se completa, la parte baja del resultado se guarda en el registro de proposito especial LO, y la parte alta del resultado se guarda en el registro de proposito especial HI.

Operación:

```
32:  t  ← (0 || GPR[rs]) * (0 || GPR[rt])
     LO ← t31..0
     HI ← t63..32

64:  t  ← (0 || GPR[rs]31..0) * (0 || GPR[rt]31..0)
     LO ← (t31)32 || t31..0
     HI ← (t63)32 || t63..32
```

Excepciones:

Ninguna.

NOR (Nor)

El **formato** de la instrucción es:

NOR rd,rs,rt

Descripción de la instrucción:

El contenido del registro rs se combina con el contenido del registro rt en una operación lógica NOR. El resultado se guarda en el registro rd.

Operación:

```
32,64:    GPR[rd] ← GPR[rs] nor GPR[rt]
```

Excepciones:

Ninguna.

OR (Or)

El **formato** de la instrucción es:

OR rd,rs,rt

Descripción de la instrucción:

El contenido del registro rs se combina con el contenido del registro rt en una operación lógica OR. El resultado se guarda en el registro rd.

Operación:

```
32,64:    GPR[rd] ← GPR[rs] or GPR[rt]
```

Excepciones:

Ninguna.

ORI (Or Immediate)

El **formato** de la instrucción es:

ORI rt,rs,immediate

Descripción de la instrucción:

En el dato inmediato de 16 bits se hace una extensión con ceros y se combina con el contenido del registro rs in una operación lógica OR. El resultado se guarda en el registro rt.

Operación:

32: $GPR[rt] \leftarrow GPR[rs]_{31..16} || (immediate \text{ or } GPR[rs]_{15..0})$

64: $GPR[rt] \leftarrow GPR[rs]_{63..16} || (immediate \text{ or } GPR[rs]_{15..0})$

Excepciones:

Ninguna.

SB (Store Byte)

El **formato** de la instrucción es:

SB rt,offset(base)

Descripción de la instrucción:

Al desplazamiento de 16 bits se le hace una extensión de signo y se suma al contenido del registro base para formar la dirección a la que vamos a acceder. El

byte menos significativo del registro rt es almacenado en la dirección de memoria especificada.

Operación:

```
32:  direccion ← ((offset15)^16||offset15..0)+GPR[base]
     temp ← GPR[rt]7..0
     dirección (de memoria) ← temp

64:  direccion ← ((offset15)^48||offset15..0)+GPR[base]
     temp ← GPR[rt]7..0
     dirección (de memoria) ← temp
```

Excepciones:

Ninguna.

SD (Store Doubleword)

El **formato** de la instrucción es:

SD rt,offset(base)

Descripción de la instrucción:

Al desplazamiento de 16 bits se le hace una extensión de signo y se suma al contenido del registro base para formar la dirección a la que vamos a acceder. El contenido del registro rt se almacena a partir de la dirección de memoria especificada.

Si alguno de los tres bits menos significativos de la dirección a la que vamos a acceder es distinto de cero, ocurre una “Address error exception”.

Esta operación está definida solo para modo 64 bits. La ejecución de esta instrucción en modo 32 bits causa una “Reserved instruction exception”.

Operación:

```
64:  direccion ← ((offset15)48||offset15..0)+GPR[base]
     temp ← GPR[rt]
     dirección (de memoria) ← temp
```

Excepciones:

Address error exception.

Reserved instruction exception (En modo usuario 32 bits o supervisor 32 bits).

SDL (Store Doubleword Left)

El **formato** de la instrucción es:

SDL rt,offset(base)

Descripción de la instrucción:

Esta instrucción puede usarse con la instrucción SDR para almacenar el contenido de un registro en ocho bytes de memoria consecutivos, cuando los bytes cruzan el límite de una doble palabra.

Al desplazamiento de 16 bits se le hace una extensión de signo y se suma al contenido del registro base para formar una dirección la cual especifica un byte arbitrario. Se almacenarán de uno a ocho bytes, dependiendo del byte de comienzo especificado. Si la instrucción de ejecuta en modo de operación 32 bits se produce una Reserved instruction Exception.

Operación:

Tenemos una doble palabra (doubleword) que es: 394573587631292A

Para Big Endian:

Registro:

rt : 39 45 73 58 76 31 29 2A

Memoria

Desplaz	Dir.	Direcciones de memoria							
0 →	0000:	39	45	73	58	76	31	29	2A
1 →	0000:	X	39	45	73	58	76	31	29
2 →	0000:	X	X	39	45	73	58	76	31
3 →	0000:	X	X	X	39	45	73	58	76
4 →	0000:	X	X	X	X	39	45	73	58
5 →	0000:	X	X	X	X	X	39	45	73
6 →	0000:	X	X	X	X	X	X	39	45
7 →	0000:	X	X	X	X	X	X	X	39

Para Little Endian:

Registro:

rt : 39 45 73 58 76 31 29 2A

Memoria:

Desplaz	Dir.	Direcciones de memoria							
0 →	0000:	39	X	X	X	X	X	X	X
1 →	0000:	45	39	X	X	X	X	X	X
2 →	0000:	73	45	39	X	X	X	X	X
3 →	0000:	58	73	45	39	X	X	X	X
4 →	0000:	76	58	73	45	39	X	X	X
5 →	0000:	31	76	58	73	45	39	X	X
6 →	0000:	29	31	76	58	73	45	39	X
7 →	0000:	2A	29	31	76	58	73	45	39

X: Lo que había en memoria antes de SDL

Excepciones:

Reserved instruction exception (En modo 32 bits).

SDR (Store Doubleword Right)

El **formato** de la instrucción es:

SDR rt,offset(base)

Descripción de la instrucción:

Esta instrucción puede usarse con la instrucción SDL para almacenar el contenido de un registro en ocho bytes de memoria consecutivos, cuando los bytes cruzan el límite de una doble palabra.

Al desplazamiento de 16 bits se le hace una extensión de signo y se suma al contenido del registro base para formar una dirección la cual especifica un byte arbitrario. Se almacenarán de uno a ocho bytes, dependiendo del byte de comienzo especificado. Si la instrucción de ejecuta en modo de operación 32 bits se produce una Reserved instruction Exception.

Operación:

Tenemos una doble palabra (doubleword) que es: 394573587631292A

Para Big Endian:

Registro:

rt : 39 45 73 58 76 31 29 2A

Memoria

Desplaz	Dir.	Direcciones de memoria							
0 →	0000:	2A	X	X	X	X	X	X	X
1 →	0000:	29	2A	X	X	X	X	X	X
2 →	0000:	31	29	2A	X	X	X	X	X

3 →	0000:	76	31	29	2A	X	X	X	X
4 →	0000:	58	76	31	29	2A	X	X	X
5 →	0000:	73	58	76	31	29	2A	X	X
6 →	0000:	45	73	58	76	31	29	2A	X
7 →	0000:	39	45	73	58	76	31	29	2A

Para Little Endian:

Registro:

rt : 39 45 73 58 76 31 29 2A

Memoria:

Desplaz	Dir.	Direcciones de memoria							
0 →	0000:	2A	29	31	76	58	73	45	39
1 →	0000:	X	2A	29	31	76	58	73	45
2 →	0000:	X	X	2A	29	31	76	58	73
3 →	0000:	X	X	X	2A	29	31	76	58
4 →	0000:	X	X	X	X	2A	29	31	76
5 →	0000:	X	X	X	X	X	2A	29	31
6 →	0000:	X	X	X	X	X	X	2A	29
7 →	0000:	X	X	X	X	X	X	X	2A

X: Lo que había en memoria antes de SDR

Excepciones:

Reserved instruction exception (En modo 32 bits).

SH (Store Halfword)

El **formato** de la instrucción es:

SH rt,offset(base)

Descripción de la instrucción:

Al desplazamiento de 16 bits se le hace una extensión de signo y se suma al contenido del registro base para formar la dirección a la que vamos a acceder. La media palabra (halfword) de orden bajo del registro rt se almacena a partir de la dirección de memoria especificada.

Si el bit menos significativo de la dirección a la que vamos a acceder es distinto de cero, ocurre una "Address error exception".

Operación:

```
32:  direccion ← ((offset15)16||offset15..0)+GPR[base]
     temp ← GPR[rt]15..0
     dirección (de memoria) ← temp

64:  direccion ← ((offset15)48||offset15..0)+GPR[base]
     temp ← GPR[rt]15..0
     dirección (de memoria) ← temp
```

Excepciones:

Address error exception.

SW (Store Word)

El **formato** de la instrucción es:

SW rt,offset(base)

Descripción de la instrucción:

Al desplazamiento de 16 bits se le hace una extensión de signo y se suma al contenido del registro base para formar la dirección a la que vamos a acceder. El

contenido del registro rt se almacena a partir de la dirección de memoria especificada.

Si alguno de los dos bits menos significativos de la dirección a la que vamos a acceder es distinto de cero, ocurre una "Address error exception".

Operación:

```
32:  direccion ← ((offset15)16||offset15..0)+GPR[base]
     temp ← GPR[rt]31..0
     dirección (de memoria) ← temp

64:  direccion ← ((offset15)48||offset15..0)+GPR[base]
     temp ← GPR[rt]31..0
     dirección (de memoria) ← temp
```

Excepciones:

Address error exception.

SLL (Shift Left Logical)

El **formato** de la instrucción es:

SLL rd,rt,sa

Descripción de la instrucción:

El contenido del registro rt se desplaza hacia la izquierda sa bits, insertando ceros en los bits de orden bajo. El resultado se guarda en el registro rd.

En modo 64 bits, el resultado de 32 bits hace una extensión de signo cuando se mete en el registro destino.

Operación:

```
32:  GPR[rd] ← GPR[rt]31-sa..0 || 0sa

64:  s ← 0 || sa
     temp ← GPR[rt]31-s..0 || 0s
     GPR[rd] ← (temp31)32 || temp
```

Excepciones:

Ninguna.

SLLV (Shift Left Logical Variable)

El **formato** de la instrucción es:

SLLV rd,rt,rs

Descripción de la instrucción:

El contenido del registro rt se desplaza a la izquierda el número de bits especificados por los cinco bits de orden bajo que contiene el registro rs. El resultado se guarda en el registro rd.

En modo 64 bits, el resultado de 32 bits hace una extensión de signo cuando se guarda en el registro destino.

Operación:

```
32:  s ← GPR[rs]4..0
     GPR[rd] ← GPR[rt](31-s)..0 || 0s

64:  s ← 0 || GPR[rs]4..0
     temp ← GPR[rt](31-s)..0 || 0s
     GPR[rd] ← (temp31)32 || temp
```

Excepciones:

Ninguna.

SLT (Set On Less Than)

El **formato** de la instrucción es:

SLT rd,rs,rt

Descripción de la instrucción:

El contenido del registro rt se compara con el contenido del registro rs. Considerando ambas cantidades como enteros con signo, si el contenido del registro rs es menor que el contenido del registro rt, el resultado se pone a uno; en caso contrario el resultado se pone a cero. El resultado se guarda en el registro rd.

Operación:

```
32:  if GPR[rs] < GPR[rt] then
          GPR[rd] ← 0^31 || 1
     else
          GPR[rd] ← 0^32
     endif

64:  if GPR[rs] < GPR[rt] then
          GPR[rd] ← 0^63 || 1
     else
          GPR[rd] ← 0^64
     endif
```

Excepciones:

Ninguna.

SLTI (Set On Less Than Immediate)

El **formato** de la instrucción es:

SLTI rt,rs,immediate

Descripción de la instrucción:

El dato inmediato de 16 bits hace una extensión de signo y se compara con el contenido del registro rs. Considerando ambas cantidades como enteros con signo, si el registro rs es menor que el dato inmediato con su extensión de signo, el resultado se pone a uno; en caso contrario el resultado se pone a cero. El resultado se guarda en el registro rt.

Operación:

```
32:  if GPR[rs] < (immediate15)16 || immediate15..0 then
          GPR[rt] ← 031 || 1
     else
          GPR[rt] ← 032
     endif

64:  if GPR[rs] < (immediate15)48 || immediate15..0 then
          GPR[rt] ← 063 || 1
     else
          GPR[rt] ← 064
     endif
```

Excepciones:

Ninguna.

SLTIU (Set On Less Than Immediate Unsigned)

El **formato** de la instrucción es:

SLTIU rt,rs,immediate

Descripción de la instrucción:

El dato inmediato de 16 bits hace una extensión de signo y se compara con el contenido del registro rs. Considerando ambas cantidades como enteros sin signo, si el registro rs es menor que el dato inmediato con su extensión de signo, el resultado se pone a uno; en caso contrario el resultado se pone a cero. El resultado se guarda en el registro rt.

Operación:

```
32:  if (0||GPR[rs])<(immediate15)16||immediate15..0 then
         GPR[rt] ← 031 || 1
     else
         GPR[rt] ← 032
     endif

64:  if (0||GPR[rs])<(immediate15)48||immediate15..0 then
         GPR[rt] ← 063 || 1
     else
         GPR[rt] ← 064
     endif
```

Excepciones:

Ninguna.

SLTU (Set On Less Than Unsigned)

El **formato** de la instrucción es:

SLTU rd,rs,rt

Descripción de la instrucción:

El contenido del registro rt se compara con el contenido del registro rs. Considerando ambas cantidades como enteros sin signo, si el contenido del registro rs es menor que el contenido del registro rt, el resultado se pone a uno; en caso contrario el resultado se pone a cero. El resultado se guarda en el registro rd.

Operación:

```
32:  if (0 || GPR[rs]) < (0 || GPR[rt]) then
          GPR[rd] ← 0^31 || 1
     else
          GPR[rd] ← 0^32
     endif

64:  if (0 || GPR[rs]) < (0 || GPR[rt]) then
          GPR[rd] ← 0^63 || 1
     else
          GPR[rd] ← 0^64
     endif
```

Excepciones:

Ninguna.

SRA (Shift Right Arithmetic)

El **formato** de la instrucción es:

SRA rd,rt,sa

Descripción de la instrucción:

El contenido del registro rt se desplaza hacia la derecha sa bits, haciendo una extensión de signo en los bits de orden alto. El resultado se guarda en el registro rd.

En modo 64 bits, el operando debe ser un valor de 32 bits con extensión de signo.

Operación:

```
32:  GPR[rd] ← (GPR[rt]31)^sa || GPR[rt]31..sa)

64:  s ← 0 || sa
     temp ← (GPR[rt]31)^s || GPR[rt]31..s
     GPR[rd] ← (temp31)^32 || temp
```

Excepciones:

Ninguna.

SRAV (Shift Right Arithmetic Variable)

El **formato** de la instrucción es:

SRAV rd,rt,rs

Descripción de la instrucción:

El contenido del registro rt se desplaza hacia la derecha el número de bits especificados por los cinco bits de orden bajo del registro rs, haciendo una extensión de signo de los bits del orden alto en el resultado. El resultado se guarda en el registro rd.

En modo 64 bits, el operando debe ser un valor de 32 bits con extensión de signo.

Operación:

```
32:  s ← GPR[rs]4..0
     GPR[rd] ← (GPR[rt]31)^s || GPR[rt]31..s

64:  s ← GPR[rs]4..0
     temp ← (GPR[rt]31)^s || GPR[rt]31..s
     GPR[rd] ← (temp31)^32 || temp
```

Excepciones:

Ninguna.

SRL (Shift Right Logical)

El **formato** de la instrucción es:

SRL rd,rt,sa

Descripción de la instrucción:

El contenido del registro rt se desplaza a la derecha sa bits, insertando ceros en los bits de orden alto. El resultado se guarda en el registro rd.

En modo 64 bits, el operando debe ser un valor de 32 bits con extensión de signo.

Operación:

```
32:  GPR[rd] ← 0^sa || GPR[rt]31..sa

64:  s ← 0 || sa
     temp ← 0^s || GPR[rt]31..s
     GPR[rd] ← (temp31)^32 || temp
```

Excepciones:

Ninguna.

SRLV (Shift Right Logical Variable)

El **formato** de la instrucción es:

SRLV rd,rt,rs

Descripción de la instrucción:

El contenido del registro rt se desplaza a la derecha el número de bits especificados por los cinco bits de orden bajo del registro rs, insertando ceros en los bits de orden alto. El resultado se guarda en el registro rd.

En modo 64 bits, el operando debe ser un valor de 32 bits con extensión de signo.

Operación:

```
32:  s ← GPR[rs]4..0
     GPR[rd] ← 0^s || GPR[rt]31..s

64:  s ← GPR[rs]4..0
     temp ← 0^s || GPR[rt]31..s
```

```
GPR[rd] ← (temp31)32 || temp
```

Excepciones:

Ninguna.

SUB (Subtract)

El **formato** de la instrucción es:

SUB rd,rs,rt

Descripción de la instrucción:

El contenido del registro rt se resta con el contenido del registro rs para formar el resultado que se guarda en el registro rd. En modo 64 bits, los operandos deben ser valores de 32 bits con extensión de signo.

Ocurre una "Integer overflow exception" si se produce un desbordamiento en complemento a dos. El registro rd no se modifica cuando ocurre esta excepción.

Operación:

```
32:  GPR[rd] ← GPR[rs] - GPR[rt]

64:  temp ← GPR[rs] - GPR[rt]
     GPR[rd] ← (temp31)32 || temp31..0
```

Excepciones:

Integer overflow exception.

SUBU (Subtract Unsigned)

El **formato** de la instrucción es:

SUBU rd,rs,rt

Descripción de la instrucción:

El contenido del registro rt se resta con el contenido del registro rs para formar un resultado que se guarda en el registro rd.

En modo 64 bits, los operandos deben ser valores de 32 bits con extensión de signo.

La única diferencia entre esta instrucción y la instrucción SUB es que SUBU nunca causa desbordamiento. No ocurre una "Integer overflow exception".

Operación:

```
32:  GPR[rd] ← GPR[rs] – GPR[rt]

64:  temp ← GPR[rs] – GPR[rt]
     GPR[rd] ← (temp31)32 || temp31..0
```

Excepciones:

Ninguna.

SWL (Store Word Left)

El **formato** de la instrucción es:

SWL rt,offset(base)

Descripción de la instrucción:

Esta instrucción puede usarse con la instrucción SWR para almacenar el contenido de un registro en cuatro bytes de memoria consecutivos, cuando los bytes cruzan el límite de una palabra.

Al desplazamiento de 16 bits se le hace una extensión de signo y se suma al contenido del registro base para formar una dirección la cual especifica un byte arbitrario. Se almacenarán de uno a cuatro bytes, dependiendo del byte de comienzo especificado.

Operación:

Tenemos una palabra (word) que es:

7631292A

Para Big Endian:

Registro:

rt : XX XX XX XX 76 31 29 2A

Memoria

Desplaz	Dir.	Direcciones de memoria							
0 →	0000:	76	31	29	2A	X	X	X	X
1 →	0000:	X	76	31	29	X	X	X	X
2 →	0000:	X	X	76	31	X	X	X	X
3 →	0000:	X	X	X	76	X	X	X	X

Para Little Endian:

Registro:

rt : XX XX XX XX 76 31 29 2A

Memoria:

Desplaz	Dir.	Direcciones de memoria							
0 →	0000:	76	X	X	X	X	X	X	X
1 →	0000:	31	76	X	X	X	X	X	X
2 →	0000:	29	31	76	X	X	X	X	X

3 →	0000:	2A	29	31	76	X	X	X	X

X: Lo que había en memoria antes de SWL

Excepciones:

Ninguna.

SWR (Store Word Right)

El **formato** de la instrucción es:

SWR rt,offset(base)

Descripción de la instrucción:

Esta instrucción puede usarse con la instrucción SWL para almacenar el contenido de un registro en cuatro bytes de memoria consecutivos, cuando los bytes cruzan el límite de una palabra.

Al desplazamiento de 16 bits se le hace una extensión de signo y se suma al contenido del registro base para formar una dirección la cual especifica un byte arbitrario. Se almacenarán de uno a cuatro bytes, dependiendo del byte de comienzo especificado.

Operación:

Tenemos una palabra (word) que es:

7631292A

Para Big Endian:

Registro:

rt : XX XX XX XX 76 31 29 2A

Memoria

Desplaz	Dir.	Direcciones de memoria							
0 →	0000:	2A	X	X	X	X	X	X	X

1 →	0000:	29	2A	X	X	X	X	X	X
2 →	0000:	31	29	2A	X	X	X	X	X
3 →	0000:	76	31	29	2A	X	X	X	X

Para Little Endian:

Registro:

rt : XX XX XX XX 76 31 29 2A

Memoria:

Desplaz	Dir.	Direcciones de memoria							
0 →	0000:	2A	29	31	76	X	X	X	X
1 →	0000:	X	2A	29	31	X	X	X	X
2 →	0000:	X	X	2A	29	X	X	X	X
3 →	0000:	X	X	X	2A	X	X	X	X

X: Lo que había en memoria antes de SWR

Excepciones:

Ninguna.

TEQ (Trap if Equal)

El **formato** de la instrucción es:

TEQ rs,rt

Descripción de la instrucción:

El contenido del registro rt se compara con el registro rs. Si el contenido del registro rs es igual al contenido del registro rt, ocurre una “Trap exception”.

Operación:

```
32,64:    if GPR[rs] = GPR[rt] then
```

```
            TrapException
        endif
```

Excepciones:

Trap exception.

TEQI (Trap If Equal Immediate)

El **formato** de la instrucción es:

TEQI rs,immediate

Descripción de la instrucción:

El dato inmediato de 16 bits con su extensión de signo se compara con el contenido del registro rs. Si el contenido del registro rs es igual al dato inmediato con su extensión de signo, ocurre una “Trap exception”.

Operación:

```
32:  if GPR[rs] = (immediate15)16 || immediate15..0 then
         TrapException
     endif

64:  if GPR[rs] = (immediate15)48 || immediate15..0 then
         TrapException
     endif
```

Excepciones:

Trap exception.

TGE (Trap If Greater Than Or Equal)

El **formato** de la instrucción es:

TGE rs,rt

Descripción de la instrucción:

El contenido del registro rt se compara con el contenido del registro rs. Considerando ambas cantidades como enteros con signo, si el contenido del registro rs es mayor que o igual al contenido del registro rt, ocurre una "Trap exception".

Operación:

```
32,64:    if GPR[rs] ≥ GPR[rt] then
               TrapException
          endif
```

Excepciones:

Trap exception.

TGEI (Trap If Greater Than Or Equal Immediate)

El **formato** de la instrucción es:

TGEI rs,immediate

Descripción de la instrucción:

El dato inmediato de 16 bits con su extensión de signo se compara con el contenido del registro rs. Considerando ambas cantidades como enteros con signo, si el contenido del registro rs es mayor que o igual que el dato inmediato con su extensión de signo, ocurre una "Trap exception".

Operación:

```
32:  if GPR[rs] ≥ (immediate15)16 || immediate15..0 then
          TrapException
     endif

64:  if GPR[rs] ≥ (immediate15)48 || immediate15..0 then
          TrapException
     endif
```

Excepciones:

Trap exception.

TGEIU (Trap If Greater Than Or Equal Immediate Unsigned)

El **formato** de la instrucción es:

TGEIU rs,immediate

Descripción de la instrucción:

El dato inmediato de 16 bits con su extensión de signo se compara con el contenido del registro rs. Considerando ambas cantidades como enteros sin signo, si el contenido del registro rs es mayor que o igual al dato inmediato con su extensión de signo, ocurre una “Trap exception”.

Operación:

```
32:  if (0||GPR[rs])≥(0||(immediate15)16||immediate15..0) then
          TrapException
     endif
```

```
64:  if (0||GPR[rs])≥(0||(immediate15)48||immediate15..0) then
         TrapException
     endif
```

Excepciones:

Trap exception.

TGEU (Trap If Greater Than Or Equal Unsigned)

El **formato** de la instrucción es:

TGEU rs,rt

Descripción de la instrucción:

El contenido del registro rt se compara con el contenido del registro rs. Considerando ambas cantidades como enteros sin signo, si el contenido del registro rs es mayor que o igual al contenido del registro rt, ocurre una "Trap exception".

Operación:

```
32,64:   if (0 || GPR[rs]) ≥ (0 || GPR[rt]) then
             TrapException
         endif
```

Excepciones:

Trap exception.

TLT (Trap If Less Than)

El **formato** de la instrucción es:

TLT rs,rt

Descripción de la instrucción:

El contenido del registro rt se compara con el contenido del registro rs. Considerando ambas cantidades como enteros con signo, si el contenido del registro rs es menor que el contenido del registro rt, ocurre una "Trap exception".

Operación:

```
32,64:    if GPR[rs] < GPR[rt] then
               TrapException
          endif
```

Excepciones:

Trap exception.

TLTI (Trap If Less Than Immediate)

El **formato** de la instrucción es:

TLTI rs,immediate

Descripción de la instrucción:

El dato inmediato de 16 bits con su extensión de signo se compara con el contenido del registro rs. Considerando ambas cantidades como enteros con signo, si el contenido del registro rs es menor que el dato inmediato con su extensión de signo, ocurre una "Trap exception".

Operación:

```
32:  if GPR[rs] < (immediate15)16 || immediate15..0 then
         TrapException
     endif

64:  if GPR[rs] < (immediate15)16 || immediate15..0 then
         TrapException
     endif
```

Excepciones:

Trap exception.

TLTIU (Trap If Less Than Immediate Unsigned)

El **formato** de la instrucción es:

TLTIU rs,immediate

Descripción de la instrucción:

El dato inmediato de 16 bits con su extensión de signo se compara con el contenido del registro rs. Considerando ambas cantidades como enteros con signo, si el contenido del registro rs es menor que el dato inmediato con su extensión de signo, ocurre una "Trap exception".

Operación:

```
32:  if (0||GPR[rs])<(0||(immediate15)16||immediate15..0)then
         TrapException
     endif

64:  if (0||GPR[rs])<(0||(immediate15)16||immediate15..0)then
```

```
            TrapException
        endif
```

Excepciones:

Trap exception.

TLTU (Trap If Less Than Unsigned)

El **formato** de la instrucción es:

TLTU rs,rt

Descripción de la instrucción:

El contenido del registro rt se compara con el registro rs. Considerando ambas cantidades como enteros sin signo, si el contenido del registro rs es menor que el contenido del registro general rt, ocurre una "Trap exception".

Operación:

```
32,64:    if (0||GPR[rs])<(0||GPR[rt]) then
              TrapException
          endif
```

Excepciones:

Trap exception.

TNE (Trap If Not Equal)

El **formato** de la instrucción es:

TNE rs,rt

Descripción de la instrucción:

El contenido del registro rt se compara con el contenido del registro rs. Si el contenido del registro rs no es igual al contenido del registo rt, ocurre una "Trap exception".

Operación:

```
32,64:    if GPR[rs] ≠ GPR[rt] then
               TrapException
          endif
```

Excepciones:

Trap exception.

TNEI (Trap If Not Equal Immediate)

El **formato** de la instrucción es:

TNEI rs,immediate

Descripción de la instrucción:

El dato inmediato de 16 bits con su extensión de signo se compara con el contenido del registro rs. Si el contenido del registro rs no es igual que el dato inmediato con su extensión de signo, ocurre una "Trap exception".

Operación:

```
32:  if GPR[rs] ≠ (immediate15)16 || immediate15..0 then
          TrapException
     endif
```

```
64:  if GPR[rs] ≠ (immediate15)48 || immediate15..0 then
          TrapException
     endif
```

Excepciones:

Trap exception.

XOR (Exclusive Or)

El **formato** de la instrucción es:

XOR rd,rs,rt

Descripción de la instrucción:

El contenido del registro rs se combina con el contenido del registro rt en una operación lógica XOR. El resultado se guarda en el registro rd.

Operación:

```
32,64:    GPR[rd] ← GPR[rs] xor GPR[rt]
```

Excepciones:

Ninguna.

XORI (Exclusive OR Immediate)

El **formato** de la instrucción es:

XORI rt,rs,immediate

Descripción de la instrucción:

En el dato inmediato de 16 bits se hace una extensión de ceros y se combina con el contenido del registro rs en una operación lógica XOR. El resultado se guarda en el registro rt.

Operación:

```
32:  GPR[rt] ← GPR[rs] xor (0^16 || immediate)

64:  GPR[rt] ← GPR[rs] xor (0^48 || immediate)
```

Excepciones:

Ninguna.

APÉNDICE II:

CÓDIGO FUENTE DEL SIMULADOR DE INSTRUCCIONES MIPS R4000

```
#include <stdio.h>   //Utilizamos las librerias
#include <conio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>

#define ULONG unsigned long int   //renombramos para hacerlo mas corto.
#define UCHAR unsigned char
#define NUMERO 192  //Numero de bytes de memoria que aparecen en pantalla.

void lineas (void);  //pinta las l¡neas del interfaz de usuario.
void limpia (int x1, int y1, int x2, int y2); //limpia cierta zona de
pantalla.
void menu(void);  //saca el menu de opciones por pantalla.
void creditos (void);  //saca los datos del autor del proyecto.

void ini_memoria (void); //Inicializa la memoria a cero.
void ini_regs (void);  //Inicializa los registros a cero.
void Ver_memoria (ULONG direccion); //ense¤a contenido de memoria.
void Ver_registros (int nreg); //ense¤a contenido de registros.
void Ver_CP (void);  //Ense¤a el contenido del contador de programa.
int Coger_instruccion (void);  //Coge instr. de mem. la decodifica y ejecuta.
int Registros (void);    //Modifica los registros de proposito general

int Meter_Instrucc (void);  //Mete instrucciones en memoria.
int Introducir_datos (void);  //Mete datos en memoria.
int opcion4 ();
int Reiniciar_CP(void); //pone el CP al principio del programa.
int configuracion(void);  //establece la configuraci¢n del simulador.

void Error (char c[]);   //Saca un mensaje de error si lo hay
UCHAR *lontocar (ULONG x);  //transforma 32 bits a 4 posiciones de 8bits
ULONG chartolong (UCHAR x[]);   //transforma 4 pos. de 8 bits a 32 bits.
UCHAR acarreo(ULONG n1, ULONG n2); //Mira acarreo intermedio en sumas de 64
//bits.
UCHAR Cod_op (ULONG instr, ULONG mascara); //Saca el c¢digo de op de una
//instr.
UCHAR Estado (char *c); //devuelve el valor del registro de estado corresp.
void Pon_estado(void);
UCHAR Campo (char *c,ULONG instruccion); //devuelve valor de campo que
//indiquemos.

int leeinstr (unsigned long *v); //Lee una instrucci¢n de 8 num hex.
int leedir (unsigned long *v); //Lee una direcc. de 4 n£meros hex.
int leebyte (unsigned char *v); //Lee un byte de 2 n£meros hex.
int leebytedecimal (unsigned char *v); //Lee un byte de 2 n£meros decimal.

ULONG multiplicacion (ULONG m12, ULONG m11, ULONG m22, ULONG m21, char *c);
//hace la multiplicacion de 64 bits por 64 bits.
ULONG dividir (ULONG dividendo0, ULONG dividendo, ULONG divisor0, ULONG
divisor,char *car);
//Hace la divisi¢n de 64 bits entre 64 bits.

void General_Exception (void);  //pone el registro de estado a modo kernel.
```

```
/*Aqu¡ empieza la declaraci¢n del repertorio de instrucciones del MIPS R-
4000*/

//A partir de aqui son de tabla0
void ADD (ULONG instruccion);    //Add
void ADDU (ULONG instruccion);   //Add Unsigned
void AND (ULONG instruccion);    //And
void DADD (ULONG instruccion);   //Doubleword Add
void DDIV (ULONG instruccion);   //Doubleword Divide
void DDIVU (ULONG instruccion);  //Doubleword Divide Unsigned
void DIV (ULONG instruccion);    //Divide
void DIVU (ULONG instruccion);   //Divide Unsigned
void DMULT (ULONG instruccion);  //Doubleword Multiply
void DMULTU (ULONG instruccion); //Doubleword Multiply Unsigned
void DSLL (ULONG instruccion);   //Doubleword Shift Left Logical
void DSLLV (ULONG instruccion);  //Doubleword Shift Left Logical Variable
void DSLL32 (ULONG instruccion); //Doubleword Shift Left Logical + 32
void DSRA (ULONG instruccion);   //Doubleword Shift Right Arithmetic
void DSRAV (ULONG instruccion);  //Doubleword Shift Right Arithmetic Vble.
void DSRA32 (ULONG instruccion); //Doubleword Shift Right Arithmetic + 32
void DSRL (ULONG instruccion);   //Doubleword Shift Right Logical
void DSRLV (ULONG instruccion);  //Doubleword Shift Right Logical Variable
void DSRL32 (ULONG instruccion); //Doubleword Shift Right Logical + 32
void DSUB (ULONG instruccion);   //Doubleword Subtract
void DSUBU (ULONG instruccion);  //Doubleword Subtract Unsigned
void JALR (ULONG instruccion);   //Jump And Link Register
void JR (ULONG instruccion);     //Jump Register
void MFHI (ULONG instruccion);   //Move From HI
void MFLO (ULONG instruccion);   //Move From LO
void MTHI (ULONG instruccion);   //Move To HI
void MTLO (ULONG instruccion);   //Move To LO
void MULT (ULONG instruccion);   //Multiply
void MULTU (ULONG instruccion);  //Multiply Unsigned
void NOR (ULONG instruccion);    //Nor
void OR (ULONG instruccion);     //Or
void SLL (ULONG instruccion);    //Shift Left Logical
void SLLV (ULONG instruccion);   //Shift Left Logical Variable
void SLT (ULONG instruccion);    //Set On Less Than
void SLTU (ULONG instruccion);   //Set On Less Than Unsigned
void SRA (ULONG instruccion);    //Shift Right Arithmetic
void SRAV (ULONG instruccion);   //Shift Right Arithmetic Variable
void SRL (ULONG instruccion);    //Shift Right Logical
void SRLV (ULONG instruccion);   //Shift Right Logical Variable
void SUB (ULONG instruccion);    //Subtract
void SUBU (ULONG instruccion);   //Subtract Unsigned
void TEQ (ULONG instruccion);    //Trap If Equal
void TGE (ULONG instruccion);    //Trap If Greater Than Or Equal
void TGEU (ULONG instruccion);   //Trap If Greater Than Or Equal Unsigned
void TLT (ULONG instruccion);    //Trap If Less Than
void TLTU (ULONG instruccion);   //Trap If Less Than Unsigned
void TNE (ULONG instruccion);    //Trap If Not Equal
void XOR (ULONG instruccion);    //Exclusive Or
void DADDU (ULONG instruccion);  //Doubleword Add Unsigned
```

```
//A partir de aqui son de tabla
void ADDI (ULONG instruccion);   //Add inmediate
void ADDIU (ULONG instruccion);  //Add Inmediate Unsigned
void ANDI (ULONG instruccion);      //And Inmediate
void BEQ (ULONG instruccion);    //Branch On Equal
void BEQL (ULONG instruccion);   //Branch On Greater Than Or Equal To Zero
void BGTZ (ULONG instruccion);   //Branch On Greater Than Zero
void BGTZL (ULONG instruccion);  //Branch On Greater Than Zero Likely
void BLEZ (ULONG instruccion);   //Branch On Less Than Or Equal To Zero
void BLEZL (ULONG instruccion);  //Brch On Less Thn Or Equal To Zero Likely
void BNE (ULONG instruccion);    //Branch On Not Equal
void BNEL (ULONG instruccion);   //Branch On Not Equal Likely
void DADDIU (ULONG instruccion); //Doubleword Add Inmediate Unsigned
void DADDI (ULONG instruccion);  //Doubleword Add Inmediate
void J (ULONG instruccion);      //Jump
void JAL (ULONG instruccion);    //Jump And Link
void LB (ULONG instruccion);     //Load Byte
void LBU (ULONG instruccion);    //Load Byte Unsigned
void LD (ULONG instruccion);     //Load Doubleword
void LDL (ULONG instruccion);    //Load Doubleword Left
void LDR (ULONG instruccion);    //Load Doubleword Right
void LH (ULONG instruccion);     //Load Halfword
void LHU (ULONG instruccion);    //Load Halfword Unsigned
void LUI (ULONG instruccion);    //Load Upper Inmediate
void LW (ULONG instruccion);     //Load Word
void LWL (ULONG instruccion);    //Load Word Left
void LWR (ULONG instruccion);    //Load Word Right
void LWU (ULONG instruccion);    //Load Word Unsigned
void ORI (ULONG instruccion);    //Or Inmediate
void SB (ULONG instruccion);     //Store Byte
void SD (ULONG instruccion);     //Store Doubleword
void SDL (ULONG instruccion);    //Store Doubleword Left
void SDR (ULONG instruccion);    //Store Doubleword Right
void SH (ULONG instruccion);     //Store Halfword
void SW (ULONG instruccion);     //Store Word
void SWL (ULONG instruccion);    //Store Word Left
void SWR (ULONG instruccion);    //Store Word Right
void SLTI (ULONG instruccion);   //Set On Less Than Inmediate
void SLTIU (ULONG instruccion);  //Set On Less Than Inmediate Unsigned
void XORI (ULONG instruccion);   //Exclusive OR Inmediate
void ERET (ULONG instruccion);   //Exception Return

//A partir de aqui son de tabla1
void BGEZ (ULONG instruccion);    //Branch On Greater Than Or Equal To Zero.
void BGEZAL (ULONG instruccion);  //Brch On Grtr Thn Or Eql to Zero And Link.
void BGEZALL (ULONG instruccion); //Bch On Gtr Thn Or Eql to Zro & Lnk Likely
void BGEZL (ULONG instruccion);   //Brch On Greatr Than Or Equal to 0 Likely.
void BLTZ (ULONG instruccion);    //Branch On Less Than Zero
void BLTZAL (ULONG instruccion);  //Branch On Less Than Zero And Link
void BLTZALL (ULONG instruccion); //Branch On Less Than Zero And Link Likely
void BLTZL (ULONG instruccion);   //Branch On Less Than Zero Likely
void TEQI (ULONG instruccion);    //Trap if Equal Inmediate
void TGEI (ULONG instruccion);    //Trap if Greater Than Or Equal Inmediate.
void TGEIU (ULONG instruccion);   //Trap if Greater Than Or Equal Inm Unsign.
void TLTI (ULONG instruccion);    //Trap if Less Than Inmediate
```

```
void TLTIU (ULONG instruccion);   //Trap if Less Than Inmediate Unsigned
void TNEI (ULONG instruccion);    //Trap if Not Equal Inmediate

//A partir de aqu¡ voy a declarar los formatos de cada instruccion
void Formato_cero (ULONG instruccion,UCHAR dec);
void Formato_uno (ULONG instruccion,UCHAR dec);
void Formato_dos (ULONG instruccion,UCHAR dec);
void Formato_tres (ULONG instruccion,UCHAR dec);
void Formato_cuatro (ULONG instruccion,UCHAR dec);
void Formato_cinco (ULONG instruccion,UCHAR dec);
void Formato_seis (ULONG instruccion,UCHAR dec);
void Formato_siete (ULONG instruccion,UCHAR dec);
void Formato_ocho (ULONG instruccion,UCHAR dec);
void Formato_nueve (ULONG instruccion,UCHAR dec);
void Formato_diez (ULONG instruccion,UCHAR dec);
void Formato_once (ULONG instruccion,UCHAR dec);
void Formato_doce (ULONG instruccion,UCHAR dec);
void Formato_trece (ULONG instruccion,UCHAR dec);
void Formato_catorce (ULONG instruccion,UCHAR dec);
void Formato_quince (ULONG instruccione,UCHAR dec);

typedef struct   //Utilizamos esta estructura para decodificar instrucci¢n
{
  char formato;   //Para guardar el formato de la instrucci¢n
  char *nombre;   // Para guardar el nombre de la instrucci¢n
}decodificador;  //Es el nombre del tipo de la estructura.

/*Voy a declarar las variables globales a las cuales puedo acceder en
cualquier
  zona del programa*/

decodificador formatos[256]={{3,"SLL"},{0,NULL},{3,"SRL"},{3,"SRA"},
      {1,"SLLV"},{0,NULL},{1,"SRLV"},{1,"SRAV"},{4,"JR"},{6,"JALR"},
      {0,NULL},{0,NULL},{0,NULL},{0,NULL},{0,NULL},{0,NULL},
      {5,"MFHI"},{4,"MTHI"},{5,"MFLO"},{4,"MTLO"},{1,"DSLLV"},{0,NULL},
      {1,"DSRLV"},{1,"DSRAV"},{2,"MULT"},{2,"MULTU"},{2,"DIV"},{2,"DIVU"},
      {2,"DMULT"},{2,"DMULTU"},{2,"DDIV"},{2,"DDIVU"},{0,"ADD"},{0,"ADDU"},
      {0,"SUB"},{0,"SUBU"},{0,"AND"},{0,"OR"},{0,"XOR"},{0,"NOR"},
      {0,NULL},{0,NULL},{0,"SLT"},{0,"SLTU"},{0,"DADD"},{0,"DADDU"},
      {0,"DSUB"},{0,"DSUBU"},{2,"TGE"},{2,"TGEU"},{2,"TLT"},{2,"TLTU"},
      {2,"TEQ"},{0,NULL},{2,"TNE"},{0,NULL},{3,"DSLL"},{0,NULL},
      {3,"DSRL"},{3,"DSRA"},{3,"DSLL32"},{0,NULL},{3,"DSRL32"},{3,"DSRA32"},
      {12,"BLTZ"},{12,"BGEZ"},{12,"BLTZL"},{12,"BGEZL"},{0,NULL},{0,NULL},
      {0,NULL},{0,NULL},{13,"TGEI"},{13,"TGEIU"},{13,"TLTI"},{13,"TLTIU"},
      {13,"TEQI"},{0,NULL},{13,"TNEI"},{0,NULL},{12,"BLTZAL"},{12,"BGEZAL"},
      {12,"BLTZALL"},{12,"BGEZALL"},{0,NULL},{0,NULL},{0,NULL},{0,NULL},
      {0,NULL},{0,NULL},{0,NULL},{0,NULL},{0,NULL},{0,NULL},
      {0,NULL},{0,NULL},{0,NULL},{0,NULL},{0,NULL},{0,NULL},
      {0,NULL},{0,NULL},{0,NULL},{0,NULL},{0,NULL},{0,NULL},
      {0,NULL},{0,NULL},{0,NULL},{0,NULL},{0,NULL},{0,NULL},
      {0,NULL},{0,NULL},{0,NULL},{0,NULL},{0,NULL},{0,NULL},
      {0,NULL},{0,NULL},{0,NULL},{0,NULL},{0,NULL},{0,NULL},
      {14,"J"},{14,"JAL"},{8,"BEQ"},{8,"BNE"},{9,"BLEZ"},{9,"BGTZ"},
      {7,"ADDI"},{7,"ADDIU"},{7,"SLTI"},{7,"SLTIU"},{7,"ANDI"},{7,"ORI"},
```

```
      {7,"XORI"},{10,"LUI"},{15,"ERET"},{0,NULL},{0,NULL},{0,NULL},
      {8,"BEQL"},{8,"BNEL"},{9,"BLEZL"},{9,"BGTZL"},{7,"DADDI"},{7,"DADDIU"},
      {11,"LDL"},{11,"LDR"},{0,NULL},{0,NULL},{0,NULL},{0,NULL},
      {11,"LB"},{11,"LH"},{11,"LWL"},{11,"LW"},{11,"LBU"},{11,"LHU"},
      {11,"LWR"},{11,"LWU"},{11,"SB"},{11,"SH"},{11,"SWL"},{11,"SW"},
      {11,"SDL"},{11,"SDR"},{11,"SWR"},{0,NULL},{0,NULL},{0,NULL},
      {0,NULL},{0,NULL},{0,NULL},{0,NULL},{0,NULL},{11,"LD"},
      {0,NULL},{0,NULL},{0,NULL},{0,NULL},{0,NULL},{0,NULL},
      {0,NULL},{11,"SD"},{0,NULL},{0,NULL},{0,NULL},{0,NULL},
      {0,NULL},{0,NULL},{0,NULL},{0,NULL},{0,NULL},{0,NULL},
      {0,NULL},{0,NULL},{0,NULL},{0,NULL},{0,NULL},{0,NULL},
      {0,NULL},{0,NULL},{0,NULL},{0,NULL},{0,NULL},{0,NULL},
      {0,NULL},{0,NULL},{0,NULL},{0,NULL},{0,NULL},{0,NULL},
      {0,NULL},{0,NULL},{0,NULL},{0,NULL},{0,NULL},{0,NULL},
      {0,NULL},{0,NULL},{0,NULL},{0,NULL},{0,NULL},{0,NULL},
      {0,NULL},{0,NULL},{0,NULL},{0,NULL},{0,NULL},{0,NULL},
      {0,NULL},{0,NULL},{0,NULL},{0,NULL},{0,NULL},{0,NULL},
      {0,NULL},{0,NULL},{0,NULL},{0,NULL},{0,NULL},{0,NULL},
      {0,NULL},{0,NULL},{0,NULL},{0,NULL},{0,NULL},{0,NULL}};


void (*tablaf[])(ULONG,UCHAR)= {Formato_cero,Formato_uno,Formato_dos,
                        Formato_tres,Formato_cuatro,Formato_cinco,
                        Formato_seis,Formato_siete,Formato_ocho,
                        Formato_nueve,Formato_diez,Formato_once,
                        Formato_doce,Formato_trece,Formato_catorce,
                        Formato_quince};


void (*tabla[])(ULONG)= {NULL,NULL,J,JAL,BEQ,BNE,BLEZ,BGTZ,ADDI,ADDIU,
                   SLTI,SLTIU,ANDI,ORI,XORI,LUI,ERET,NULL,NULL,
                   NULL,BEQL,BNEL,BLEZL,BGTZL,DADDI,DADDIU,LDL,
                   LDR,NULL,NULL,NULL,NULL,LB,LH,LWL,LW,LBU,LHU,
                   LWR,LWU,SB,SH,SWL,SW,SDL,SDR,SWR,NULL,NULL,
                   NULL,NULL,NULL,NULL,NULL,NULL,LD,NULL,NULL,
                   NULL,NULL,NULL,NULL,NULL,SD};


void (*tabla0[])(ULONG)={SLL,NULL,SRL,SRA,SLLV,NULL,SRLV,SRAV,JR,JALR,
                   NULL,NULL,NULL,NULL,NULL,NULL,MFHI,MTHI,MFLO,
                   MTLO,DSLLV,NULL,DSRLV,DSRAV,MULT,MULTU,DIV,DIVU,
                   DMULT,DMULTU,DDIV,DDIVU,ADD,ADDU,SUB,SUBU,AND,OR,
                   XOR,NOR,NULL,NULL,SLT,SLTU,DADD,DADDU,DSUB,DSUBU,
                   TGE,TGEU,TLT,TLTU,TEQ,NULL,TNE,NULL,DSLL,NULL,
                   DSRL,DSRA,DSLL32,NULL,DSRL32,DSRA32};


void (*tabla1[])(ULONG)={BLTZ,BGEZ,BLTZL,BGEZL,NULL,NULL,NULL,NULL,TGEI,
                   TGEIU,TLTI,TLTIU,TEQI,NULL,TNEI,NULL,BLTZAL,
                   BGEZAL,BLTZALL,BGEZALL};


FILE *pf;   //fichero que simula la memoria.
FILE *pc;    //fichero que guarda comienzo de programa y final.
ULONG rpg[35][2];  //Registros de proposito general/especial.
```

```
ULONG ultimadir;  //Guarda la ultima direccion visualizada en pantalla
UCHAR sr=0x02/*1e*//*02*/;   //Registro de estado:
00,RE,KX,SX,UX,KSU=00000010 (binario.)
/*RE: 0->Litte-Endian; 1->Big-Endian
  KX: 0->32-bit (Kernel); 1->64-bit (kernel)
  SX: 0->32-bit (Superv); 1->64-bit (Superv)
  UX: 0->32-bit (Usuario); 1->64-bit (Usuario)
  KSU: 10->Usuario;  01->Supervisor;  00->Kernel;*/

int registros=0;  //si es 0 => del 0 al 15, si es 1 =>del 16 al 31.

                   /*PROGRAMA PRINCIPAL*/
void main (void)    //funci¢n principal.
{
 char op,respuesta;   //opci¢n del men£ principal, y para coger respuesta.

 lineas();             //Dibujamos las l¡neas de la pantalla.
 Ver_memoria (0x0000); //Visualizamos la memoria por la pantalla.
 Ver_registros (0);
 Ver_CP ();
 do
 {
  limpia (2,18,56,20);  //Limpia el interfaz de comandos
  gotoxy (30,23);
  printf ("                          ");  //Limpiamos el lugar de la excepcion
  menu();    //Saca las opciones del men£.
  op=getch();    //Pide opci¢n
  op=toupper(op);   //la pasamos a may£sculas.
  switch (op)     //en caso de que op sea....
  {
   case '1': Meter_Instrucc();   //1.- Metemos instrucciones en memoria.
             break;              //se sale del switch.
   case '2': Introducir_datos(); //2.- Introducimos bytes en memoria.
             break;
   case '3': Coger_instruccion();   //3.-Ejecuta instrucciones.
             break;
   case '4': opcion4();    //opci¢n para ver la memoria desde cq. direcc.
             break;
   case '5': do
             {
               limpia(2,17,56,20);
               gotoxy(3,17);
               printf ("¨Resetear el Simulador? (S/N): ");
               respuesta=toupper(getch());
             }while ((respuesta!='S') && (respuesta!='N') && (respuesta!=27));
             if (respuesta=='S')
             {
               ini_memoria();   //inicializa toda la memoria a cero.
               ini_regs ();    //inicializa todos los registros a cero.
               Ver_memoria (0000);   //Vemos memoria a partir de la direcc. 0
               Ver_registros (0);   //Vemos los registros del cero al 15
               Ver_CP();   //Vemos el contador de programa
             }
             break;
   case '6': Registros();  //6.-Introducimos datos en los registros 1..30
```

```
            break;
   case '7': Reiniciar_CP(); //7.-Pone el CP a una direcci¢n
            break;
   case '8': configuracion();
            break;
   case '9': creditos(); //muestra por pantalla datos del autor del proyecto.
            break;
   case '*': if (registros==0)   //controlamos que al pulsar '*' cambie regs.
            {
             registros++;       //si registros es cero lo ponemos a uno...
             Ver_registros (16);  //y visualizamos los registros 16..31
            }
            else              //si registros no es cero (es uno)...
            {
             registros--;     //lo ponemos a cero
             Ver_registros (0);   //visualizamos los registros 0..15
            }
            break;
   case 'S': clrscr();   //Borra la pantalla
            exit(1);     //Se sale del programa.
  }
 }
 while (1);  //bucle infinito.
}
                  /*FIN DEL PROGRAMA PRINCIPAL*/

/*A partir de aqu¡ voy a empezar a implementar las funciones para el
interfaz de usuario.*/

void Error (char c[])
/*Esta funci¢n escribe un mensaje de error y se sale del programa*/
{
 limpia(2,4,56,15);   //limpiamos zona de la pantalla
 gotoxy (2,4);  //Se posiciona en la pantalla
 printf ("%s\n",c);   //Escribe el mensaje (pasado como par metro)
 exit(1);   //Se sale del programa
}
```

```
void lineas (void)
/*Saca las l¡neas del interfaz de usuario por pantalla*/
{
 clrscr();   //Borra la pantalla.
 printf("╔═══════════════════════════════════════════════════════╗");
 printf("║                      MIPS R-4000                      ║");
 printf("╠═════════════════════════════════════╦═════════════════╣");
 printf("║                                     ║                 ║");
 printf("║                                     ║                 ║");
 printf("║                                     ║                 ║");
 printf("║                                     ║                 ║");
 printf("║                                     ║                 ║");
 printf("║                                     ║                 ║");
 printf("║                                     ║                 ║");
 printf("║                                     ║                 ║");
 printf("║                                     ║                 ║");
 printf("║                                     ║                 ║");
 printf("╠═════════════════════════════════════╣                 ║");
 printf("║                                     ║                 ║");
 printf("║                                     ╠═════════════════╣");
 printf("╠═════════════════════════════════════╣                 ║");
 printf("║                                     ║                 ║");
 printf("╚═════════════════════════════════════╩═════════════════╝");

}

void menu(void)
/*Esta funci¢n solo saca el men£ de opciones posibles por pantalla*/
{
 limpia(2,17,56,20);
 gotoxy (4,17);   //Nos posicionamos en la pantalla.
 printf ("1.-Intro program"); //Opci¢n uno. Introducir programa.
 gotoxy (4,18);
 printf ("2.-Intro datos");   //Opci¢n dos. Introducir datos en memoria.
 gotoxy (4,19);
 printf ("3.-Ejecuci¢n");     //Opci¢n tres. Ejecuci¢n del programa.
 gotoxy (4,20);
 printf ("4.-Ver direcci¢n");  //Opci¢n cuatro. Ver a partir de una direcc.
 gotoxy (23,17);
 printf ("5.-Reset");  //Opci¢n cinco. Hace un reset del procesador.
 gotoxy (23,18);
 printf ("6.-Registros");  //Opci¢n seis. Introducimos datos en los regs.
 gotoxy (23,19);
 printf ("7.-Inicia CP");  //Opci¢n siete. Inicia CP al principio del progr.
 gotoxy (23,20);
 printf ("8.-Configurac."); //Permite cambiar la configuraci¢n de simulador.
 gotoxy (42,17);
 printf ("9.-Creditos");   //Muestra informaci¢n del realizador del proyecto.
 gotoxy (42,18);
 printf ("S.-Salir");    //Se sale del programa. Finaliza.
}

void creditos (void)
/*Esta funci¢n proporciona informaci¢n sobre el autor del proyecto*/
{
```

```
 limpia(2,17,56,20);  //limpiamos una zona de la pantalla.
 gotoxy (3,18);
 printf ("Autor  : Miguel Angel Hern ndez Pinto"); //Nombre del autor.
 gotoxy (3,17);
 printf ("Carrera: Ingeniero T,cnico en Inform tica de Sistemas");//carrera.
 gotoxy (3,19);
 printf ("A¤o    : 1999");  //A¤o en el que termin¢ el proyecto.
 gotoxy (42,20);
 printf ("Pulsa tecla...");   //Informaci¢n para el usuario.
 getch();            //espera hasta que se pulse una tecla.
}

int Introducir_datos (void)
//esta funci¢n nos permite introducir datos en la memoria byte a byte.
{
  UCHAR dato;
  char op;
  ULONG direccion;
  int x,y,escape;

  do
  {
   limpia(2,17,56,20);   //limpiamos una zona determinada de la pantalla.
   gotoxy (2,17);
   printf ("Direcci¢n de Memoria: ");  //pedimos una direcci¢n de memoria.
   escape=leedir(&direccion);  //la leemos en formato hexadecimal.
   if (escape==0)
      return 0;
  }
  while (direccion > 65535);   //controlamos que no se pase de 0xFFFF
  Ver_memoria (direccion);   //visualizamos la direcci¢n pedida.
  x=9;    //inicializamos la x y la y para poder escribir el dato intro-
  y=4;   //ducido, por pantalla.
  do              //bucle.
  {
   gotoxy (2,18);
   printf ("Escribe byte. ");  //Informamos que hay que meter un byte.
   gotoxy (2,19);
   printf ("%04lX: ",direccion);  //Imprimimos la direcci¢n a modificar.
   escape=leebyte(&dato);   //leemos el byte que vamos a meter en memoria.
   if (escape==0)  //Si se pulsa escape se sale de la opci¢n.
     return 0;

   if ((pf = fopen ("memoria","r+b"))==NULL)  //abrimos fich. para
lectura/escr.
   {
     Error ("Error al acceder a memoria");   //Escribimos mensaje
   }

   fseek (pf,direccion,SEEK_SET); //nos posicionamos en direcci¢n a escribir.

   if ((fwrite (&dato,sizeof(UCHAR),1,pf))==0)  //escribimos en fichero
   {
    Error ("Error al escribir en memoria");  //mensaje de error
   }
```

```
   fclose (pf);  //cerramos el fichero.

   gotoxy (x,y);  //se posiciona el cursor en la posici¢n x,y
   printf ("%02X",dato);  //pone en pantalla el dato que hemos metido.

   limpia(2,17,56,20);   //limpiamos una zona determinada de la pantalla.
   gotoxy (2,17);
   printf ("¨Desea meter otro dato:"); //Preguntamos si queremos meter otro
   gotoxy (9,18);
   printf ("1.- En la siguiente direcci¢n");  //en la dir. siguiente.
   gotoxy (9,19);
   printf ("2.- En otra direcci¢n distinta");  //en otra dir. distinta.
   gotoxy (9,20);
   printf ("3.- No quiero meter otro dato");  //no meter ning£n dato mas.
   do
   {
     op=getch();  //leemos la opci¢n introducida por teclado.
   }
   while (op!='1' && op!='2' && op!='3' && op!=27);

   switch(op)  //en caso de que la variable op sea....
   {
    case '1': direccion++;  //1, incrementamos la direcci¢n
            limpia(2,17,56,20);  //limpiamos zona de la pantalla.
            x+=3;   //aumentamos la x para escribir el siguiente dato.
            if (x==57)    //si la x ha llegado a 57
            {
            x=9;    //inicializamos otra vez x a 9
            y++;   //pasamos a la siguiente linea.
            }
            break;  //sale del switch.
    case '2': do      //2, vamos a meter dato en otra direcci¢n distinta.
            {
            limpia(2,17,56,20);  //limpia una zona de la pantalla
            gotoxy (2,17);
            printf ("Direcci¢n de Memoria: ");  //pide direcci¢n.
            escape=leedir(&direccion);  //la lee.
            if (escape==0)  //si se ha pulsado esc se sale de opci¢n
              return 0;
            }
            while (direccion > 65535); //controla que no sea mayor de 0xFFFF.
            Ver_memoria (direccion);  //visualizamos la memoria.
            x=9;  //inicializamos x e y.
            y=4;
            break;  //se sale del switch.
    case '3': break;  //se sale del switch.
    case 27 : return 0;  //se sale de la funci¢n.
   } //fin del switch.
  }
  while (op!='3');  //mientras que la variable op sea distinta de '3'
  return 0;
}

void ini_memoria(void)
```

```
/*Esta funci¢n inicializa el fichero que contiene la memoria todo a ceros*/
{
 ULONG i;      //i es el contador del for.
 UCHAR car=0x00;     //Inicializamos la memoria al valor de car.


 // Lo primero vamos a inicializar la memoria.

 if ((pf = fopen ("memoria","wb"))==NULL) //abrimos el fichero para escritura
 {
  Error("Error al acceder a memoria");  //Escribimos mensaje de error.
 }

 for (i=0;i<65535;i++)        //inicializamos 64 Kbytes a cero.
 {
  if ((fwrite (&car,sizeof(UCHAR),1,pf))==0)  //escribimos en el fichero.
  {
   Error ("Error al escribir en la memoria.");  //imprimimos mensaje.
  }
 }

 fclose (pf);  //Cerramos el fichero.
}

void Ver_memoria (ULONG direccion)
/*Pasamos como par metro una direcci¢n a partir de la cual vamos a ver lo que
  contiene la memoria. y el fichero que vamos a usar*/
{
 ULONG cont,i; //cont:para poner las direcc. en pantalla.-- i:contador del
for.
 UCHAR car;    //en car metemos el caracter leido del fichero.

 ultimadir = direccion;

 cont=direccion;  //en cont metemos la direcci¢n desde la que queremos ver
memoria.

 if ((pf= fopen ("memoria","rb"))==NULL)  //abrimos el fichero para lectura
 {
  Error ("Error al acceder a memoria");  //Escribimos mensaje de error.
 }

 fseek (pf,cont,SEEK_SET); /*Posicionamos el cursor del fichero en la
direcci¢n
                       indicada por la variable cont.*/

 for (i=0;i<12;i++)
 {
   gotoxy (2,4+i);
   printf ("                                                          ");
 }
 gotoxy (2,3);     //El cursor de la pantalla va a la posici¢n 2,3

 for (i=0; i<NUMERO; i++)  //El for va desde cero hasta NUMERO
 {
```

```
  if ((fread (&car,sizeof(UCHAR),1,pf))==0)  //lee un dato del fichero.
  {
   Error ("Error al leer de la memoria");  //imprimimos mensaje.
  }

  if ((i%16)==0)  //si el contador i es multiplo de 16
  {
   printf ("\n");   //pasa a la siguiente linea
   printf ("° %04X: ",cont);  //escribe la direcci¢n en la que est
  }
  printf ("%02X ",car);  //escribe el byte correspondiente en pantalla.
  if (cont==0xFFFE)
    break;
  cont++;    //incrementa la variable cont.
 }
 fclose (pf);  //cerramos el fichero.
}

void Ver_CP (void)
//Esta funci¢n visualiza el contador de programa por pantalla.
{
 gotoxy (33,22);  //Nos posicionamos
 printf ("PC:");  //Vemos el registro PC
 printf ("%08lX",rpg[34][0]);  //Imprimimos la parte alta del registro.
 printf ("%08lX",rpg[34][1]);  //Imprimimos la parte baja del registro.
}

void Ver_registros (int nreg)
/*Pasamos como par metro nreg que es el n§ de
  registro a partir del cual comenzamos a visualizarlos. Se visualizan de 16
  en 16 registros. Si nreg=0 se ven del 0 al 15 y si nreg=16 se ven del 16 al
  31 (se trata de eso, por lo tanto conviene que nreg sea 0 o 16 pero eso ya
  lo controlo yo)*/
{
 int i;   //contador para ver los registros. Toma valores del 0 al 31.
 int pos=0;  //para controlar la posici¢n vertical en la pantalla.

 gotoxy (59,4);  //Nos posicionamos.

 for (i=nreg;i<nreg+16;i++)     //Vemos a partir de nreg.
 {
  gotoxy (59,4+pos);   //seg£n se incrementa i vamos pasando de l¡nea.
  printf ("R%d:",i);
  printf ("%08lX",rpg[i][0]);  //Vemos registro por pant.
  printf ("%08lX ",rpg[i][1]);
  pos++;   //incrementamos pos.
 }

 /*Ahora voy a poner una informaci¢n en pantalla. Si se pulsa '*' se puede
ver
   de los registros 0-15 o del 16-31.*/

 gotoxy (64,21);   //Nos posicionamos
 if (nreg == 0)    //Si nreg vale 0 es que estamos viendo del 0-15.
   printf ("* => 16-31");   //Ponemos que si pulsas '*' puedes ver del 16-31.
```

```
 else if (nreg == 16)    //Si nreg vale 16 es que estamos viendo del 16-32.
   printf ("* => 0-15 ");  //Ponemos que si pulsas '*' puedes ver del 0-15.

 gotoxy (6,22);   //Nos posicionamos
 printf ("HI:");
 printf ("%08lX",rpg[32][0]);  //Vemos la parte alta del registro HI.
 printf ("%08lX",rpg[32][1]);  //Vemos la parte baja del registro HI.

 gotoxy (6,23);   //Nos posicionamos
 printf ("LO:");
 printf ("%08lX",rpg[33][0]);  //Vemos la parte alta del registro LO.
 printf ("%08lX",rpg[33][1]);  //Vemos la parte baja del registro LO.
}

void ini_regs (void)
/*Esta funci¢n lo que hace es inicializar los registros de proposito general
  (0-31) y los de proposito especial (HI,LO,CP) a un valor determinado, en
  este caso a cero.*/
{
 int i,j;   //variables para los contadores.

 for (i=0;i<35;i++)  //i es 35 porque es el n£mero de filas de la matriz.
 {
   for (j=0;j<2;j++)  //j es 2 porque es el n£mero de columnas de la matriz.
     rpg[i][j]=0x00;    //inicializamos a cero.
 }
}

void limpia (int x1, int y1, int x2, int y2)
/*A esta funci¢n le pasamos como par metro el valor de la esquina superior
  izquierda, (valor x, valor y) y el valor de la esquina inferior derecha
  y borra la pantalla*/
{
 int resta1,resta2,i,j;    //variables usadas

 gotoxy (18,16);
 printf ("Í");
 gotoxy (18,21);
 printf ("Í");
 gotoxy (57,19);
 printf ("°");
 resta1 = y2 - y1;   //calculamos la distancia vertical
 resta2 = x2 - x1;   //calculamos la distancia horizontal
 for (i=0;i<resta1+1;i++)
   for (j=0;j<resta2+1;j++)
   {
    gotoxy (x1+j,y1+i);    //nos situamos con el cursor
    printf (" ");         //borramos zona de pantalla idicada
   }
}

int Meter_Instrucc (void)
/*En esta funci¢n lo que vamos a hacer es meter instrucciones en la zona de
  memoria a partir de la direcci¢n que indiquemos.*/
{
```

```
 ULONG comienzo,cuenta,i,instruccion;
 UCHAR *car=NULL;
 int x=9,y=4,j,k;
 int escape;

 do
 {
   limpia(2,17,56,20);  //limpia una zona de la pantalla
   gotoxy (2,17);     //se posiciona
   printf ("Direcci¢n de comienzo: ");   //pide la direcci¢n de inicio de
prog.
   escape=leedir(&comienzo);  //lee la direcci¢n de comienzo
   fflush(stdin);
   if (escape==0)
     return 0;
 }
 while ((comienzo%4) != 0 || (comienzo > 65535));
 cuenta=comienzo;       //a cuenta le asigna el comienzo.
 Ver_memoria(comienzo);    //Visualizamos memoria a partir de dir. inicio.

 if ((pf = fopen ("memoria","r+b"))==NULL)  //abrimos fich. para
lectura/escr.
 {
  Error ("Error al acceder a memoria");   //Escribimos mensaje
 }

 fseek (pf,cuenta,SEEK_SET);  //El cursor del fich. se va a la direcc.
indicada.

 gotoxy (2,17);
 printf ("Escribe una instrucci¢n completa.");
 gotoxy (45,17);     //se posiciona el cursor.
 printf ("Salir: Esc");  //informaci¢n de lo que hay que poner para salir

 do
 {
  gotoxy (2,18);  //se posiciona el cursor
  printf ("                                  ");
  gotoxy (2,18);
  i=cuenta;            //a i le asignamos el valor de cuenta.
  printf ("%04X: ",i);   //Se pide valor introducido en esa direccion.
  fflush (stdin);    //limpiamos el buffer.
  escape=leeinstr(&instruccion);   //leemos el dato

  car=lontocar (instruccion);  //Se convierte el long en 4 bytes

  if (escape!=0)//si no se ha pulsado escape, se ha copiado la instr.
  {
   for (j=0;j<4;j++)
   {

    if (Estado("RE")==0)  //Si es little endian
      k=3-j;
    else if (Estado ("RE")==1)  //Si es big endian
      k=j;
```

```
    if ((fwrite (&car[k],sizeof(UCHAR),1,pf))==0)  //escribimos en fichero
    {
     Error ("Error al escribir en memoria");  //mensaje de error
    }
    gotoxy (x,y);  //posicionamos el cursor

    printf ("%02X ",car[k]);  //imprimimos car cter en pantalla

    x=x+3;    //sumamos 3 a la x
    if (x==57)    //si la x ha llegado a 57
    {
     x=9;    //inicializamos otra vez x a 9
     y++;   //pasamos a la siguiente linea.
    }
   }
   cuenta=cuenta+4;  //incrementamos cuenta al siguiente byte.
  }
 }
 while (escape!=0);   //haz mientras instruccion sea distinto de la cond de
parada.
 fclose (pf);              //cerramos el fichero

 return 0;
}

int opcion4 (void)
/*Esta funci¢n solo permite visualizar la memoria a partir de la direcci¢n
  que nosotros indiquemos*/
{
 ULONG dir;
 int escape;

 limpia(2,17,56,20);   //Limpia una zona de la pantalla
 gotoxy (2,17);   //Se posiciona
 printf ("Direcci¢n: ");   //Pide direcci¢n
 escape=leedir(&dir);   //La lee de teclado
 if (escape==0)
   return 0;
 Ver_memoria (dir);          //Muestra la memoria por pantalla
 return 0;
}

UCHAR *lontocar (ULONG x)  //Convierte dato 32 bits a 4 de 8 bits
{
 ULONG temp;   //Variable temporal
 UCHAR *conversion;   //Para la conversi¢n
 int j;        //Indice

 if ((conversion=(UCHAR *)malloc(4*sizeof(UCHAR)))==NULL) //Se crea espacio
 {
  printf ("Insuficiente espacio en memoria\n");
  exit(1);
 }
```

```
 for (j=0;j<4;j++)   //Bucle
 {
  temp=x;   //En temp metemos x
  temp=temp>>(8*j);  //Desplazamos 8*j bits hacia la derecha.
  conversion[3-j]=(UCHAR)temp;  //Lo guardamos en conversi¢n
 }
 return conversion;   //Devuelve el array.
}

int Coger_instruccion (void)
/*Esta funci¢n devuelve la instrucci¢n cogida*/
{
 ULONG auxiliar,auxiliar2,cont;
 UCHAR car[4],car2[4],cop,caracter,finbreak=0;
 char ejecucion;   //secuencial, paralelo, breakpoint
 ULONG breakpoint=0;
 int paralelo=0,escape;
 int i,j,y=17,m;

 UCHAR dec,format,tecla;  //para decodificar la instruccion y formato
 UCHAR dec2,format2,cop2;

 limpia(2,17,56,20);    //limpia una zona de la pantalla
 gotoxy (2,17);
 printf ("Modo de ejecuci¢n:");   //Pide modo de ejecuci¢n
 gotoxy (2,19);
 printf ("\t1.-Secuencial");   //Secuencial
 printf ("\t 2.-Paralelo");    //Paralelo
 printf ("\t3.-Breakpoint");   //Breakpoint (punto de parada)
 do
 {
   gotoxy (21,17);  //Nos situamos en la pantalla
   ejecucion=getch();    //Coge opci¢n de teclado
 }
 while ((ejecucion!='1') && (ejecucion!='2') && (ejecucion!='3') &&
                                          (ejecucion!=27));
 if (ejecucion==27)
   return 0;
 fflush (stdin);
 if (ejecucion=='3')   //Si la ejecuci¢n es igual a breakpoint
 {
   do
   {
     limpia(2,17,56,20);   //limpia zona de la pantalla
     gotoxy (2,17);   //Se mueve.
     printf ("Punto de parada: ");   //Pide punto de parada
     escape=leedir(&breakpoint);   //Lee punto de parada de teclado
     if (escape==0)  //Si se ha pulsado escape se sale de la opci¢n
       return 0;
   }
   while ((breakpoint%4)!=0);
 }
 limpia(2,17,56,20);   //Limpia zona de pantalla

 gotoxy (18,16);
```

```
printf ("Ë");
gotoxy (18,21);
printf ("Ê");
gotoxy (57,19);
printf ("¹");
gotoxy (2,17);
switch (ejecucion)     //En caso de que la ejecuci¢n
{
 case '1': printf ("¯ Secuencial    °");  //Sea 1: secuencial
         break;
 case '2': printf ("¯ Paralelo      °");   //Sea 2: Paralelo
         break;
 case '3': printf ("¯ Breakpoint    °");   //Sea 3: Breakpoint
         break;
}

Pon_estado();    //Escribe algunos estados del micro para la ejecuci¢n.

if ((pf=fopen ("memoria","rb+"))==NULL)  //abrimos fichero para lectura.
{
 Error ("Error al acceder a memoria.");
}

while (1) /*(rpg[34][1]!=final) && (rpg[34][1]<final)) */ ////////////
{
 if ((ejecucion=='3') && (rpg[34][1]==breakpoint))
   break;
 auxiliar = rpg[34][1];        //Cogemos la direcc. de inicio del programa.
 fseek (pf,auxiliar,SEEK_SET);    //Nos posicionamos en la memoria.

 for (i=0;i<4;i++)  //cogemos la primera instrucci¢n
 {
   if (Estado("RE")==0)  //Si es little-endian
     j=3-i;
   else if (Estado("RE")==1)   //Si es big-endian
     j=i;

   if ((fread (&car[j],sizeof (UCHAR),1,pf))==0)
   {
    Error ("Error al acceder a memoria.");
   }
 }

 for (i=0;i<4;i++)  //cogemos la instrucci¢n siguiente
 {
   if (Estado("RE")==0)  //Si es little-endian
     j=3-i;
   else if (Estado("RE")==1)   //Si es big-endian
     j=i;

   if ((fread (&car2[j],sizeof (UCHAR),1,pf))==0)
   {
    Error ("Error al acceder a memoria.");
   }
 }
```

```
  auxiliar=chartolong (car);  //auxiliar contiene la instrucci¢n a ejecutar.
  auxiliar2=chartolong (car2);  //auxiliar2 contiene la siguiente
instrucci¢n.

  if ((ejecucion=='1') || ((ejecucion=='2') && ((paralelo%2)==0)) ||
(ejecucion=='3'))
  {
    cop=Cod_op (auxiliar,0xFC000000);  //voy a formar el n£mero que dec.
instr.
    switch (cop)
    {
      case 0: dec=0;
            cop=Cod_op (auxiliar,0x0000003F);
            dec=dec | cop;
            break;
      case 1: dec=1;
            dec=dec<<6;
            cop=Cod_op (auxiliar,0x001F0000);
            dec=dec | cop;
            break;
      default:dec=2;
            dec=dec<<6;
            dec=dec | cop;
    }

    gotoxy (20,17);
    if ((formatos[dec].nombre==NULL) || (dec>191))
    {
      printf ("No encontrada");
      break;
    }
    else
    {
      format=formatos[dec].formato; //formatos es el array de estructuras
      (*tablaf[format])(auxiliar,dec); //llamada para sacar mnemonico por
pantalla
    }
  }

  if ((ejecucion=='2') && ((paralelo%2)==0)/* && ((rpg[34][1]+4)!=final)*/)
  {
    cop2=Cod_op (auxiliar2,0xFC000000);  //voy a formar el n£mero que dec.
instr.
    switch (cop2)
    {
      case 0: dec2=0;
            cop2=Cod_op (auxiliar2,0x0000003F);
            dec2=dec2 | cop2;
            break;
      case 1: dec2=1;
            dec2=dec2<<6;
            cop2=Cod_op (auxiliar2,0x001F0000);
            dec2=dec2 | cop2;
            break;
```

```
      default:dec2=2;
            dec2=dec2<<6;
            dec2=dec2 | cop2;
    }

    gotoxy (20,18);
    if ((formatos[dec].nombre==NULL) || (dec>191))
    {
      printf ("No encontrada");
      break;
    }
    else
    {
      format2=formatos[dec2].formato; //formatos es el array de estructuras
      (*tablaf[format2])(auxiliar2,dec2); //llamada para sacar mnemonico por
pantalla
    }
  }

  switch (ejecucion)   //Si ejecuci¢n es
  {
   case '1': do    //1
           {
             tecla=getch();  //lee tecla
             tecla=toupper(tecla);   //La pasa a mayusculas
             if (tecla=='*')
             {
             if (registros==0) //controlamos que al pulsar '*' cambie regs.
             {
               registros++;  //si registros es cero lo ponemos a uno...
               Ver_registros (16); //y visualizamos los registros 16..31
             }
             else      //si registros no es cero (es uno)...
             {
               registros--;     //lo ponemos a cero
               Ver_registros (0);   //visualizamos los registros 0..15
             }
             }
           }while ((tecla!='E') && (tecla!='P') && (tecla!=27));
           break;
   case '2': if ((paralelo%2)==0)
           {
             do
             {
             tecla=getch();
             tecla=toupper(tecla);
             if (tecla=='*')
             {
               if (registros==0) //controlamos que al pulsar '*' cambie regs.
               {
                 registros++;  //si registros es cero lo ponemos a uno...
                 Ver_registros (16); //y visualizamos los registros 16..31
               }
               else      //si registros no es cero (es uno)...
               {
```

```
                registros--;     //lo ponemos a cero
                Ver_registros (0);   //visualizamos los registros 0..15
              }
            }
            }while ((tecla!='E') && (tecla!='P') && (tecla!=27));
            paralelo=0;
          }
          break;
 case '3': if (rpg[34][1]==breakpoint)
          {
            finbreak=1;
          }
          break;
}
gotoxy (29,23);
printf ("                              ");  //Para limpiar la excepci¢n.

if ((tecla=='P') || (tecla==27))    //Si tecla es igual a 'P'
  break;
if (finbreak==1)    //Si finbreak es igual a uno
  break;   //Se sale de la ejecuci¢n

cop=Cod_op (auxiliar,0xFC000000);  //Ahora se va a ejecutar la instrucci¢n
switch (cop)
{
  case 0: cop=Cod_op (auxiliar,0x0000003F);
        if (cop<64)  //Para que no se pase del array y no haya error
        {
          if (tabla0[cop]!=NULL)  //Por si se ha equivocado de n£mero
          (*tabla0[cop])(auxiliar);   //ejecutamos la instrucci¢n corresp.
        }
        break;
  case 1: cop=Cod_op (auxiliar,0x001F0000);
        if (cop<20)     //Para que no se pase del array y no haya error
        {
          if (tabla1[cop]!=NULL) //Por si se ha equivocado de n£mero
          (*tabla1[cop])(auxiliar);   //ejecutamos la instrucci¢n corresp.
        }
        break;
  default:if (cop<64)  //Para que no se pase del array y no haya error
        {
          if (tabla[cop]!=NULL)  //Por si se ha equivocado de n£mero
          (*tabla[cop])(auxiliar);
        }
        //llamamos a la funci¢n que ejecuta ese codigo de operaci¢n
        break;
}
rpg[0][0]=0; //Como el registro r0 siempre tiene que tener el valor cero
rpg[0][1]=0; //y no se puede modificar, ponemos su parte alta y baja a cero

if (ejecucion=='2') paralelo++;  //incremente paralelo

 if (registros==0)          //Visualizamos los registros.
   Ver_registros (0);
 else if (registros == 1)
```

```
     Ver_registros (16);

  cont = ultimadir;
  fseek (pf,cont,SEEK_SET); /*Posicionamos el cursor del fichero en la
direcci¢n
                       indicada por la variable cont.*/
  gotoxy (2,3);     //El cursor de la pantalla va a la posici¢n 2,3

  //Ahora vamos a mostrar la memoria por pantalla
  for (m=0; m<NUMERO; m++)  //El for va desde cero hasta NUMERO
  {
   if ((fread (&caracter,sizeof(UCHAR),1,pf))==0)  //lee un dato del fichero.
   {
    Error ("Error al leer de la memoria");  //imprimimos mensaje.
   }

   if ((m%16)==0)  //si el contador i es multiplo de 16
   {
    printf ("\n");   //pasa a la siguiente linea
    printf ("° %04X: ",cont);  //escribe la direcci¢n en la que est
   }
   printf ("%02X ",caracter);  //escribe el byte correspondiente en pantalla.
   cont++;    //incrementa la variable cont.
  }
  rpg[34][1]=rpg[34][1]+4;
  Ver_CP();

 } ////////////////////////////////////////

 gotoxy (19,20);
 printf ("                                        ");  //Limpia zona de
pantalla
 gotoxy (19,20);
 if ((tecla=='P') || (tecla==27))   //Si tecla es igual a 'P'
   printf ("Pulsa una tecla...                    ");  //Mensaje.
 else if (finbreak==1)   //si finbreak es igual a uno
   printf ("Pulsa una tecla...                    ");  //Mensaje
 else   //En otro caso
   printf ("Pulsa una tecla...                    "); //Llegada al fin prog.

 getch();
 fclose (pf);   //Cierra el fichero.
 return 0;
}

ULONG chartolong (UCHAR x[])
/*Esta funci¢n pasa de un array de caracteres de 4 posiciones a un dato de
32 bits.*/
{
 ULONG var1, var2, r1,r2,r3,r4,resul;

 var2=(unsigned long int) x[0];
 var2=var2<<24;
 var1=0x00FFFFFF ^ var2;
 r1=0xFFFFFFFF & var1;
```

```
 var2=(unsigned long int) x[1];
 var2=var2<<16;
 var1=0xFF00FFFF ^ var2;
 r2=0xFFFFFFFF & var1;

 var2=(unsigned long int) x[2];
 var2=var2<<8;
 var1=0xFFFF00FF ^ var2;
 r3=0xFFFFFFFF & var1;

 var2=(unsigned long int) x[3];
 var2=var2<<0;
 var1=0xFFFFFF00 ^ var2;
 r4=0xFFFFFFFF & var1;

 resul = r1 & r2 & r3 & r4;

 return resul;
}

UCHAR Cod_op (ULONG instr, ULONG mascara)
/*A esta funci¢n se le pasa el c¢digo de la instrucci¢n y una m scara. Esta
m scara se compara y devuelve el valor del c¢digo de operaci¢n.*/
{
 ULONG auxi;  //Variable auxiliar
 UCHAR retorno;      //Lo que va a devolver.

 auxi=instr & mascara;  //Se coge el valor de la m scara

 if (mascara==0xFC000000)  //Vemos si el c¢d. de operaci¢n est  al principio
   auxi=auxi>>26;  //Desplazamos 26 bits hacia la derecha.
 else if (mascara==0x0000003F)  //Vemos si el COP est  al final
   auxi=auxi>>0;  //No desplazamos ning£n bit.
 else if (mascara==0x001F0000)  //Vemos si est  en medio aprox.
   auxi=auxi>>16;   //Desplazamos 16 bits hacia la derecha.

 retorno=(UCHAR)auxi;   //En retorno metemos auxi pasado a unsigned char.
 return retorno;   //Devuelve el valor del c¢d. de operac. de la instr.
}

UCHAR Estado (char *c)
/*A esta funci¢n se le pasa como par metro una cadena que va a comparar.
Devuelve el estado del microprocesador. Los par metros pueden ser:
RE=Reverse endian, KX=32/64, SX=32/64, UX=32/64, KSU=Usu, Kern, Sup*/
{
 UCHAR aux;   //Variable auxiliar

 aux = sr;  //En aux metemos el valor del registro de estado
 if (strcmp(c,"RE")==0)  //Compara el par metro con RE
 {
  aux=aux<<2;   //Si son iguales desplaza aux dos bits hacia la izq.
  aux=aux>>7;  //Desplaza siete bits hacia la derecha.
 }
 else if (strcmp(c,"KX")==0)   //Ocupa un bit
```

```
 {
  aux=aux<<3;
  aux=aux>>7;
 }
 else if (strcmp(c,"SX")==0)  //Ocupa un bit
 {
  aux=aux<<4;
  aux=aux>>7;
 }
 else if (strcmp(c,"UX")==0)  //Ocupa un bit
 {
  aux=aux<<5;
  aux=aux>>7;
 }
 else if (strcmp(c,"KSU")==0)  //Este campo del reg. estado ocupa 2 bits.
 {
  aux=aux<<6;
  aux=aux>>6;
 }

 return aux;   //Devuelve el estado que posee el bit de cada campo
 //Dependiendo del campo que le pasemos como par metro.
}

void Pon_estado(void)
/*Esta funci¢n pone el estado del micro, para que a la hora de ejecutar
aparezca el estado en el que estamos trabajando.*/
{
 UCHAR est,subest;

 est=Estado("RE");
 gotoxy (2,18);
 if (est==0) printf ("¯ Little-endian °");
 else if (est==1) printf ("¯ Big-endian    °");
 gotoxy (2,19);

 est=Estado("KSU");
 switch (est)
 {
  case 0: printf ("¯ Kernel        Ì");
        gotoxy (2,20);
        subest=Estado("KX");
        if (subest==0) printf ("¯ 32 bits       °");
        else if (subest==1) printf ("¯ 64 bits       °");
        break;
  case 1: printf ("¯ Supervisor    Ì");
        gotoxy (2,20);
        subest=Estado("SX");
        if (subest==0) printf ("¯ 32 bits       °");
        else if (subest==1) printf ("¯ 64 bits       °");
        break;
  case 2: printf ("¯ Usuario       Ì");
        gotoxy (2,20);
        subest=Estado("UX");
        if (subest==0) printf ("¯ 32 bits       °");
```

```
        else if (subest==1) printf ("¯ 64 bits       °");
        break;
 }
 gotoxy (19,19);
 printf ("ÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍ");
 gotoxy (19,20);
 printf ("   E: Ejecutar   P: Parar   *: Regs   ");
}

UCHAR Campo (char *c, ULONG instruccion)
/*Esta funci¢n compara el campo que le pasemos como par metro y dependiendo
de el campo que sea esta funci¢n devolver  su valor.*/
{
 ULONG auxi;   //Variable auxiliar.

 if (strcmp (c,"RS")==0)  //Si es el campo rs
 {
   auxi = instruccion & 0x03E00000;
   auxi = auxi>>21;    //En auxi mete el campo rs.
 }
 else if (strcmp (c,"RT")==0)
 {
   auxi = instruccion & 0x001F0000;
   auxi = auxi>>16;
 }
 else if (strcmp (c,"RD")==0)
 {
   auxi = instruccion & 0x0000F800;
   auxi = auxi>>11;
 }

 return (UCHAR)auxi;  //Finalmente devuelve el campo que le hemos pedido.
}

UCHAR acarreo(ULONG n1,  ULONG n2)
/*Esta funci¢n lo que hace es ver si hay acarreo cuando sumamos dos n£meros
de 32 bits. Se hace una llamada a esta funci¢n en la suma y en la resta en
modo 64 bits.*/
{
 ULONG num1i=0,num1d=0,num2i=0,num2d=0,resuld=0;
 UCHAR carry=0;

 num1d=n1;
 num2d=n2;

 num1d=num1d & 0x0000FFFF;
 num2d=num2d & 0x0000FFFF;

 resuld=num1d+num2d;

 resuld=resuld & 0x000F0000;

 resuld=resuld >> 16;
 if (resuld==1)
   carry=1;
```

```
 else if (resuld==0)
   carry=0;

 num1i=n1;
 num2i=n2;

 num1i=num1i>>16;
 num2i=num2i>>16;

 if (carry==1)
   resuld=num1i+num2i+1;
 else if (carry==0)
   resuld=num1i+num2i;

 resuld=resuld & 0x000F0000;
 resuld=resuld >> 16;

 if (resuld==1)
   carry=1;
 else if (resuld==0)
   carry=0;

 return carry;    //Devuelve el acarreo.
}

int Registros (void)
/*Esta funci¢n nos permite acceder desde el men£ principal para cambiar el
valor de los registros directamente en caso de que sea necesario. El registro
cero no se puede modificar debido a que posee un valor cero siempre. Solo
est  permitido modificar entre el registro 1 y el 31*/
{
  char tecla,resp;  //Declaro dos variables de tipo car cter.
  int escape;    //Y una variable de tipo entero.
  unsigned char registro;
  ULONG mayorpeso, menorpeso;

  do
  {
    do
    {
      limpia(2,17,56,20);  //limpia una zona de la pantalla
      gotoxy (2,17);   //Se sit£a el cursor.
      printf ("N§ de registro a modificar: ");  //Pide registro
      escape=leebytedecimal(&registro);   //Leemos n§ de reg. a modificar.
      if (escape==0)   //Si ha pulsado la tecla escape
      return 0;
    }while ((registro<1) || (registro>31));  //mientras no est, fuera
l¡mites.
    limpia(2,17,56,20);   //limpia una zona de la pantalla
    gotoxy (2,17);    //Se sit£a el cursor.
    printf ("Longitud del dato a meter: "); //Pedimos longitud de dato
    gotoxy (7,19);
    printf ("  1 -> 32 bits          2 -> 64 bits");
    gotoxy (29,17);
    do
```

```
    {
      tecla=getch();
    }
    while (tecla!='1' && tecla!='2' && tecla!=27);
    switch (tecla)
    {
      case '1': limpia(2,17,56,20);
            gotoxy (2,17);
            printf ("Escribe dato: ");  //Pide un dato
            escape=leeinstr(&menorpeso);  //lee 32 bits
            if (escape==0)    //Si ha pulsado escape
              return 0;   //Se sale de la funci¢n
            rpg[registro][1]=menorpeso;  //Se guarda en el registro
            break;
      case '2': limpia(2,17,56,20);
            gotoxy (2,17);
            printf ("32 bits mayor peso: "); //Pide bits mayor peso
            escape=leeinstr(&mayorpeso);
            if (escape==0)   //Si ha pulsado escape
              return 0;   //Se sale de la funci¢n
            gotoxy (2,18);
            printf ("32 bits menor peso: "); //Pide bits menor peso
            escape=leeinstr(&menorpeso);
            if (escape==0)  //Si ha pulsado escape
              return 0;   //Se sale de la funci¢n
            rpg[registro][0]=mayorpeso;  //Se guarda en el registro
            rpg[registro][1]=menorpeso;
            break;
      case 27:  return 0;
    }

    if (registros==0)          //Visualizamos los registros.
      Ver_registros (0);
    else if (registros == 1)
      Ver_registros (16);

    gotoxy (2,20);
    printf ("¨Desea modificar otro? (S/N) => ");
    do
      resp=toupper(getch());  //lee tecla y la pasa a may£sculas.
    while (resp!='S' && resp!='N' && resp!=27); //hasta que pulse S/N o esc
  }
  while (resp=='S');  //Hace el bucle mientras resp sea S
  return 0;
}

int Reiniciar_CP (void)
/*Esta funci¢n pide al usuario la direcci¢n de inicio que queremos poner en
el contador de programa. El usuario no se puede salir de los l¡mites estable-
cidos y la direcci¢n debe ser m£ltiplo de cuatro (debe estar alineada)*/
{
  ULONG dircp;   //La direcci¢n a la que queremos inicializar el CP
  int escape;

  do
```

```
  {
    limpia(2,17,56,20);  //limpia una zona de la pantalla
    gotoxy (2,17);  //Llevamos el cursor a una zona de la pantalla.
    printf ("Inicio de CP: ");  //Pedimos donde queremos que empiece el CP
    escape=leedir(&dircp);   //lee el dato
    if (escape==0)   //Si se ha pulsado la tecla escape
      return 0;
  }while ((dircp % 4) != 0);
  //coge el dato si es m£ltiplo de cuatro.
  rpg[34][1]=dircp;   //Guardamos la direcci¢n introducida en el CP
  Ver_CP();
  Ver_memoria(dircp);
  return 0;
}

int configuracion (void)
/*Esta funci¢n lo que hace es mostrar la configuraci¢n actual del simulador
Se pide si se quiere cambiar la configuraci¢n. Si la respuesta es s¡, nos
pide una serie de datos y el registro de estado del simulador se configura
de la forma que nosotros hemos pedido. Por defecto siempre se pone en modo
usuario.*/
{
  char modop,bigen,resp; //declaramos las variables que vamos a utilizar.

  limpia(2,17,56,20);  //limpia una zona de la pantalla
  gotoxy (2,17);  //nos situamos en la pantalla
  printf ("Configuraci¢n actual: ");  //ponemos informaci¢n

  gotoxy (25,18);   //nos situamos
  if (Estado("UX")==0) printf ("¯ 32 bits");   //miramos si es 32 o 64 bits
  else if (Estado("UX")==1) printf ("¯ 64 bits");
  gotoxy (25,19);    //nos situamos
  if (Estado("RE")==0) printf ("¯ Little-endian");  //miramos si Little- o
big-
  else if (Estado("RE")==1) printf ("¯ Big-endian");
  gotoxy (25,20);   //nos situamos
  if (Estado("KSU")==0) printf ("¯ Kernel"); //miramos si kernel, sup. o
usuar.
  else if (Estado("KSU")==1) printf ("¯ Supervisor");
  else if (Estado("KSU")==2) printf ("¯ Usuario");
  gotoxy (2,20);
  printf ("¨Cambiarla?(S/N) ");  //Pregunta
  do  //Haz
  {
    resp=getch();  //coger respuesta de teclado
    resp=toupper(resp);  //pasarla a may£sculas.
  }
  while ((resp!='S') && (resp!='N') && (resp!=27));  //mientras distinto de
si y de no

  if (resp=='S')  //si la respuesta es si
  {
    limpia(2,17,56,20);  //limpia una zona de la pantalla
    gotoxy (2,17);  //se va a una zona de la pantalla
    printf ("Modo de operaci¢n: ");  //Pide el modo de operaci¢n
```

```
    gotoxy (25,18);
    printf ("1.- 32 bits");gotoxy (25,19);   //32 bits
    printf ("2.- 64 bits");         //¢ 64 bits.
    do  //Haz
    {
      modop=getch();  //coger opci¢n de las dos mostradas.
    }
    while ((modop!='1') && (modop!='2') && modop!=27);//mientras != 1 y 2 y
27
    if (modop==27)   //Si se ha pulsado el car cter escape
    {
      return 0;   //Se sale de la funci¢n
    }

    limpia(2,17,56,20);  //limpia una zona de la pantalla
    gotoxy (25,18);   //nos situamos el cursor
    printf ("1.- Big endian");gotoxy (25,19);   //big endian
    printf ("2.- Little endian");    //Little endian
    do   //Haz
    {
      bigen=getch();   //coger opci¢n de las dos mostradas
    }
    while ((bigen!='1') && (bigen!='2') && (bigen!=27));//mientras != 1,2 y
27
    if (bigen==27)  //Si se ha pulsado el car cter escape
    {
      return 0;   //Se sale de la funci¢n
    }

    //ahora vamos a hacer los cambios (si antes no se ha pulsado la tecla
esc)
    sr=0;   //ponemos el registro de estado a cero

    if (modop=='1')   //si la variable modop es uno
      sr=sr | 0x02;   //32 bits usuario
    else if (modop=='2')     //si la variable modop es dos
      sr=sr | 0x1E;   //64 bits usuario

    if (bigen=='1')    //si la variable bigen es uno
      sr=sr | 0x22;   //big-endian usuario
    else if (bigen=='2')   //si la variable bigen es dos
      sr=sr | 0x02;   //little-endian usuario

    limpia(2,17,56,20);  //limpia una zona de la pantalla
    gotoxy (2,17);  //nos situamos en la pantalla
    printf ("Configuraci¢n actual: ");  //ponemos informaci¢n

    gotoxy (25,18);  //situamos el cursor
    if (Estado("UX")==0) printf ("¯ 32 bits"); //miramos si 32 bits ¢ 64
    else if (Estado("UX")==1) printf ("¯ 64 bits");
    gotoxy (25,19);   //situamos el cursor
    if (Estado("RE")==0) printf ("¯ Little-endian");  //Little- o big-
    else if (Estado("RE")==1) printf ("¯ Big-endian");
    gotoxy (25,20);  //situamos el cursor
```

```
    if (Estado("KSU")==0) printf ("¯ Kernel");  //Kernel, supervisor o
usuario
    else if (Estado("KSU")==1) printf ("¯ Supervisor");
    else if (Estado("KSU")==2) printf ("¯ Usuario");
    gotoxy (2,20);  //situamos el cursor
    printf ("Pulsa tecla...");  //informaci¢n
    getch();   //pulsar tecla para continuar.
  }
  return 0;
}       //fin de la funcion configuraci¢n

int leedir (unsigned long *v)
/*Esta instrucci¢n lo que hace es leer una direcci¢n de memoria desde
el teclado del ordenador. Si se pulsa escape se sale de la opci¢n indicada.
Hay que meter la direcci¢n con sus cuatro numeros*/
{
  char tecla[5], *endptr;  //array de tres caracteres y puntero.
  unsigned long temp;   //Variable temporal
  unsigned char car;
  int i=0;    //Para el contador

  do
  {
    do
    {
      car=toupper(getch());   //Lee el primer car cter introducido
      if ((car==8) && (i>0))
      {
      printf ("%c ",car);
      printf ("%c",car);
      i--;
      }
    }
    while (((car<'0') || (car>'9')) && ((car<'A') || (car>'F')) &&
(car!=27));

    if (car==27)   //Si se ha pulsado la tecla esc.
      return 0;       //Se sale de la funci¢n
    else         //Si no se ha pulsado la tecla esc.
      tecla[i]=car;
      printf ("%c",tecla[i]);

    i++;
  }while (i<4);

  tecla[i]='\0';   //Ponemos al final \0 para saber fin de cadena.

  *v=(unsigned long)strtoul (tecla,&endptr,16); //Lo pasamos a unsigned long

  return 1;   //Devuelve el n£mero.
}

int leeinstr (unsigned long *v)
/*Esta instrucci¢n lo que hace es leer una instrucci¢n desde el teclado
del ordenador. Si se pulsa escape se sale de la opci¢n indicada.
```

```
Hay que meter la instrucci¢n con sus ocho numeros*/
{
  char tecla[9], *endptr;  //array de tres caracteres y puntero.
  unsigned long temp;   //Variable temporal
  unsigned char car;
  int i=0;    //Para el contador

  do
  {
    do
    {
      car=toupper(getch());   //Lee el primer car cter introducido
      if ((car==8) && (i>0))
      {
      printf ("%c ",car);
      printf ("%c",car);
      i--;
      }
    }
    while (((car<'0') || (car>'9')) && ((car<'A') || (car>'F')) &&
(car!=27));

    if (car==27)   //Si se ha pulsado la tecla esc.
      return 0;       //Se sale de la funci¢n
    else         //Si no se ha pulsado la tecla esc.
      tecla[i]=car;
      printf ("%c",tecla[i]);

    i++;
  }while (i<8);

  tecla[i]='\0';   //Ponemos al final \0 para saber fin de cadena.

  *v=(unsigned long)strtoul (tecla,&endptr,16); //Lo pasamos a unsigned long

  return 1;   //Devuelve el n£mero.
}

int leebyte (unsigned char *v)
/*Esta instrucci¢n lo que hace es leer un byte desde el teclado del
ordenador. Si se pulsa escape se sale de la opci¢n indicada. Hay que
meter el byte con sus dos numeros*/
{
  char tecla[3], *endptr;  //array de tres caracteres y puntero.
  unsigned long temp;   //Variable temporal
  unsigned char car;
  int i=0;    //Para el contador

  do
  {
    do
    {
      car=toupper(getch());   //Lee el primer car cter introducido
      if ((car==8) && (i>0))
      {
```

```
      printf ("%c ",car);
      printf ("%c",car);
      i--;
      }
    }
    while (((car<'0') || (car>'9')) && ((car<'A') || (car>'F')) &&
(car!=27));

    if (car==27)   //Si se ha pulsado la tecla esc.
      return 0;       //Se sale de la funci¢n
    else         //Si no se ha pulsado la tecla esc.
      tecla[i]=car;
      printf ("%c",tecla[i]);

    i++;
  }while (i<2);

  tecla[i]='\0';   //Ponemos al final \0 para saber fin de cadena.

  *v=(unsigned char)strtoul (tecla,&endptr,16); //Lo pasamos a unsigned char

  return 1;   //Devuelve el n£mero.
}

int leebytedecimal (unsigned char *v)
/*Esta instrucci¢n lo que hace es leer un byte en decimal desde el
teclado del ordenador. Si se pulsa escape se sale de la opci¢n
indicada. Hay que meter el byte con sus dos numeros*/
{
  char tecla[3], *endptr;  //array de tres caracteres y puntero.
  unsigned long temp;   //Variable temporal
  unsigned char car;
  int i=0;    //Para el contador

  do
  {
    do
    {
      car=toupper(getch());   //Lee el primer car cter introducido
      if ((car==8) && (i>0))
      {
      printf ("%c ",car);
      printf ("%c",car);
      i--;
      }
    }
    while (((car<'0') || (car>'9')) && (car!=27));

    if (car==27)   //Si se ha pulsado la tecla esc.
      return 0;       //Se sale de la funci¢n
    else         //Si no se ha pulsado la tecla esc.
      tecla[i]=car;
      printf ("%c",tecla[i]);

    i++;
```

```
  }while (i<2);

  tecla[i]='\0';   //Ponemos al final \0 para saber fin de cadena.

  *v=(unsigned char)strtoul (tecla,&endptr,10); //Lo pasamos a unsigned char

  return 1;   //Devuelve el n£mero.
}

/*A partir de aqui voy a implementar todas las instrucciones*/
/*******************************************************/
/*******************INSTRUCCIONES***********************/
/*******************************************************/

void ADD (ULONG instruccion)  //Add
/*Esta funci¢n hace la suma con signo de los registros rs y rt y el resul-
tado lo mete en el registro rd. Para 64 bits, los 32 bits de mayor peso del
registro rd son una extensi¢n del bit 31 del resultado*/
{
 UCHAR rs,rt,rd;  //indican el n£mero de registro.
 signed long int regrs, regrt, regrd; //porque es suma con signo.
 ULONG auxrs,auxrt,auxrd;  //es para ver el bit 31 de cada registro.

 rs=Campo ("RS",instruccion);   //cogemos el campo rs
 rt=Campo ("RT",instruccion);   //cogemos el campo rt
 rd=Campo ("RD",instruccion);   //cogemos el campo rd

 regrs=rpg[rs][1];    //en regrs metemos los 32 bits bajos del reg rs.
 regrt=rpg[rt][1];    //en regrt metemos los 32 bits bajos del reg rt.

 //hacemos una suma con signo y el resultado se mete en regrd.
 regrd = regrs + regrt;
 auxrd=regrd;       //Metemos los valores en var. auxiliares para...
 auxrs=regrs;       //...ver el bit 31 y poder comprobar si hay...
 auxrt=regrt;       //...overflow.
 auxrd=auxrd >> 31;   //cada una de estas variables auxiliares se...
 auxrs=auxrs >> 31;   //...desplazan 31 bit a la derecha.
 auxrt=auxrt >> 31;

 if (Estado("UX")==0)  //para 32 bits
 {
   if ((auxrs == auxrt) && (auxrd != auxrs))
   /*la £nica condici¢n para que haya overflow es que los bit de signo de
     los dos operandos sean iguales y el bit de signo del resultado sea
     distinto que el de los operandos*/
   {
     General_Exception();        //Integer overflow exception
     gotoxy (30,23);
     printf ("Integer overfl. exception");
   }
   else  //en caso de que no haya overflow
   {
     rpg[rd][1] = regrd;  //metemos el resultado en el registro rd.
   }
 }
```

```
 else if (Estado("UX")==1)  //para 64 bits
 {
   if ((auxrs == auxrt) && (auxrd != auxrs))
   {
     General_Exception();        //Integer overflow exception
     gotoxy (30,23);
     printf ("Integer overfl. exception");
   }
   else
   {
     rpg[rd][1] = regrd;   //metemos el resultado en el registro destino.

     //ahora vamos a hacer la extensi¢n de signo.
     if (auxrd != 0)     //si el bit de signo del resultado es != 0
       rpg[rd][0]=0xFFFFFFFF;   //los 32 bits mas altos se ponen a uno.
     else if (auxrd == 0)    //si el bit de signo del resultado es 0
       rpg[rd][0]=0x00000000;   //los 32 bits mas altos e ponen a cero.
   }
 }
}  //fin de la funci¢n ADD.

void ADDI (ULONG instruccion)   //Add Immediate.
/*Esta funci¢n hace la suma con signo de los registros rs y un dato inme-
diato y el resultado lo mete en el registro rt. Para 64 bits, los 32 bits
de mayor peso del registro rt son una extensi¢n del bit 31 del resultado.
Si hay desbordamiento se produce una excepci¢n.*/
{
  UCHAR rs,rt;    //indican el n£mero de registro.
  signed long int regrs, regrt,inmediato; //porque es suma con signo.
  ULONG auxi,auxrs,auxrt,inme; //son variables intermedias.

  rs=Campo ("RS",instruccion);  //cogemos el campo rs de la instrucci¢n.
  rt=Campo ("RT",instruccion);  //cogemos el campo rt de la instrucci¢n.

  regrs=rpg[rs][1]; //en regrs metemos el dato del registro rs.

  auxi = instruccion & 0x0000FFFF;  //vamos a calcular el dato inmediato
  inmediato=auxi;          //lo guardamos en la variable inmediato.
  auxi = auxi>>15;     //miramos el bit 15 del dato inmediato.
  if (auxi==0)   //si es cero
    inmediato = inmediato | 0x00000000; //se hace la extensi¢n.
  else if (auxi!=0)
    inmediato = inmediato | 0xFFFF0000;

 regrt = regrs + inmediato;   //se hace la suma y el resultado en regrt.
 inme =inmediato;    //en inme se guarda el dato inmediato con su extensi¢n.
 auxrs=regrs;   //en la var auxiliar auxrs se guarda el dato del reg. rs.
 auxrt=regrt;   //en auxrt se guarda el resultado.
 inme =inme >> 31;  //se mira el bit 31
 auxrs=auxrs >> 31;
 auxrt=auxrt >> 31;

 if (Estado("UX")==0)  //para 32 bits
 {
   if ((auxrs == inme) && (auxrt != auxrs))
```

```
   /*la £nica condici¢n para que haya overflow es que los bit de signo de
     los dos operandos sean iguales y el bit de signo del resultado sea
     distinto que el de los operandos*/
   {
     General_Exception();        //Integer overflow exception
     gotoxy (30,23);
     printf ("Integer overfl. exception");
   }
   else   //si no hay overflow
   {
     rpg[rt][1] = regrt;      //metemos el resultado en el registro rt.
   }
 }
 else if (Estado("UX")==1)  //para 64 bits
 {
   if ((auxrs == inme) && (auxrt != auxrs))
   {
     General_Exception();        //Integer overflow exception
     gotoxy (30,23);
     printf ("Integer overfl. exception");
   }
   else
   {
     rpg[rt][1] = regrt;   //metemos el resultado en el registro destino.

     //ahora vamos a hacer la extensi¢n de signo.
     if (auxrt != 0)         //si el bit 31 no es cero
       rpg[rt][0]=0xFFFFFFFF;   //la parte alta del reg rt se pone a unos.
     else if (auxrt == 0)   //si es cero
       rpg[rt][0]=0x00000000;  //la parte alta del reg. rt se pone a ceros.
   }
 }
}  //fin de la funci¢n ADDI

void AND (ULONG instruccion)        //And
/*Esta funci¢n hace la AND logica de los registros rs y rt y el resultado
se guarda en el registro rd.*/
{
 UCHAR rs,rt,rd;   //indican el n£mero de registro.
 ULONG auxi;       //variable auxiliar.

 rs=Campo ("RS",instruccion);  //coge el campo rs de la instrucci¢n.
 rt=Campo ("RT",instruccion);    //coge el campo rt de la instrucci¢n.
 rd=Campo ("RD",instruccion);   //coge el campo rd de la instrucci¢n.

 if (Estado("UX")==0)  //para 32 bits
   rpg[rd][1] = rpg[rs][1] & rpg[rt][1];  //hace la operaci¢n And y...
                              //...el resultado se guarda en rd.
 else if (Estado("UX")==1)  //para 64 bits
 {
   rpg[rd][1] = rpg[rs][1] & rpg[rt][1];  //hacemos la and de la parte baja.
   rpg[rd][0] = rpg[rs][0] & rpg[rt][0];  //hacemos la and de la parte alta.
 }
}  //fin de la instrucci¢n AND.
```

```
void ADDIU (ULONG instruccion)  //Add Immediate Unsigned
/*Esta funci¢n hace la suma sin signo de los registros rs y un dato inme-
diato y el resultado lo mete en el registro rt. Para 64 bits, los 32 bits
de mayor peso del registro rt son una extensi¢n del bit 31 del resultado.
No ocurre ninguna excepci¢n de desbordamiento*/
{
  UCHAR rs,rt;   //indican el n£mero de registro.
  ULONG auxi,inmediato; //para variable auxiliar y dato inmediato.
  ULONG temp[1][2];   //una variable temporal.

  rs=Campo ("RS",instruccion);  //cogemos el campo rs
  rt=Campo ("RT",instruccion);  //cogemos el campo rt

  auxi = instruccion & 0x0000FFFF;  //vamos a calcular el dato inmediato
  inmediato=auxi;  //metemos en inmediato el dato cogido de la instrucci¢n.

  auxi = auxi>>15;  //le miramos el bit de signo para hacer la extensi¢n ...
  if (auxi==0)   //...de signo. si el bit 15 es cero
    inmediato = inmediato | 0x00000000;  //se extienden ceros
  else if (auxi!=0)                     //si no es cero
    inmediato = inmediato | 0xFFFF0000;   //se extienden 15 unos.

  if (Estado("UX")==0)  //para 32 bits
    rpg[rt][1]=rpg[rs][1]+inmediato; //sumamos sin signo y guardamos en rt

  else if (Estado("UX")==1)  //para 64 bits
  {
    temp[0][1]=rpg[rs][1]+inmediato;  //sumamos sin signo y guardamos en temp
    auxi=temp[0][1]; //metemos el resultado temporal en una variable
//auxiliar.
    auxi=auxi & 0x80000000;  //vamos a mirar el valor del bit 31
    auxi=auxi >> 31;
    if (auxi!=0) rpg[rt][0]=0xFFFFFFFF;  //si no es cero extendemos unos
    else if (auxi==0) rpg[rt][0]=0x00000000; //si es cero extendemos ceros.
    //la extensi¢n se ha hecho en la parte alta del registro.
    rpg[rt][1]=temp[0][1];  //finalmente guardamos el resultado en la parte
                     //baja del registro rt.
  }
}  //fin de la instrucci¢n ADDIU

void ADDU (ULONG instruccion)         //Add Unsigned
/*Esta funci¢n hace la suma sin signo de los registros rs y rt y el resul-
tado lo mete en el registro rd. Para 64 bits, los 32 bits de mayor peso del
registro rd son una extensi¢n del bit 31 del resultado*/
{
 UCHAR rs,rt,rd;      //indican el n£mero de registro.
 ULONG temp[1][2], aux; //variable temporal y variable auxiliar.

 rs=Campo ("RS",instruccion);  //cogemos el campo rs.
 rt=Campo ("RT",instruccion);   //cogemos el campo rt.
 rd=Campo ("RD",instruccion);  //cogemos el campo rd.

 if (Estado("UX")==0)  //para 32 bits
   rpg[rd][1] = rpg[rs][1] + rpg[rt][1]; //sumamos sin signo y guardamos...
                              //...el resultado en registro rd.
```

```
 else if (Estado("UX")==1)  //para 64 bits
 {
   temp[0][1] = rpg[rs][1] + rpg[rt][1]; //sumamos sin signo.
   aux = temp[0][1];      //pasamos el resultado temporal a la var aux.
   aux = aux & 0x80000000;   //vamos a ver el valor del bit 31
   aux = aux >> 31;
   if (aux != 0) rpg[rd][0]=0xFFFFFFFF;  //si no es cero extendemos con unos.
   else if (aux == 0) rpg[rd][0]=0x00000000; //si es cero extendemos con
ceros.
   rpg[rd][1] = temp[0][1];  //finalmente guardamos el resultado que est
en..
                        //la variable temporal, en la parte baja de rd.
 }
}  //fin de la instrucci¢n ADDU

void SUB (ULONG instruccion)   //Subtract
/*Resta el registro rs - rt y el resultado se guarda en el registro rd.
Para 64 bits se hace una extensi¢n de signo. Si se produce desbordamiento
se produce una excepci¢n*/
{
 UCHAR rs,rt,rd;    //Para guardar el n£mero del registro
 ULONG regrs, regrt, regrd;  //Variables intermedias
 ULONG auxrs,auxrt,auxrd;  //Variables auxiliares para ver signo

 rs=Campo ("RS",instruccion);  //Cogemos el campo rs de la instrucci¢n
 rt=Campo ("RT",instruccion);  //Cogemos el campo rt de la instrucci¢n
 rd=Campo ("RD",instruccion);  //Cogemos el campo rd de la instrucci¢n

 regrs=rpg[rs][1];  //Guardamos parte baja de rs en regrs
 regrt=~rpg[rt][1]+1;  //Lo cambiamos de signo

 regrd = regrs + regrt;   //Sumamos y guardamos en regrd
 auxrd=regrd;   //En aux.. metemos lo que hay en reg.. porque queremos
 auxrs=regrs;   //Ver el signo (bit 32)
 auxrt=regrt;
 auxrd=auxrd >> 31;   //Los desplazamos 31 bits hacia la derecha.
 auxrs=auxrs >> 31;
 auxrt=auxrt >> 31;

 if (Estado("UX")==0)  //para 32 bits
 {
   if ((auxrs == auxrt) && (auxrd != auxrs)) //Vemos si hay desbordamiento
   {
     General_Exception();        //Integer overflow exception
     gotoxy (30,23);
     printf ("Integer overfl. exception");
   }
   else
   {
     rpg[rd][1] = regrd;  //Guardamos en el registro el resultado
   }
 }
 else if (Estado("UX")==1)  //para 64 bits
 {
   if ((auxrs == auxrt) && (auxrd != auxrs))  //Vemos si hay desbordamiento
```

```
   {
     General_Exception();        //Integer overflow exception
     gotoxy (30,23);
     printf ("Integer overfl. exception");
   }
   else
   {
     rpg[rd][1] = regrd;   //metemos el resultado en el registro destino.

     //ahora vamos a hacer la extensi¢n de signo.
     if (auxrd != 0)         //Si auxrd es distinto de cero
       rpg[rd][0]=0xFFFFFFFF;  //Ponemos la parte alta toda a unos
     else if (auxrd == 0)  //Si auxrd es igual a cero
       rpg[rd][0]=0x00000000;   //Ponemos la parte alta toda a ceros
   }
 }
}    //Fin de la instrucci¢n SUB

void SUBU (ULONG instruccion)  //Subtract Unsigned
/*Resta el registro rs - rt y el resultado se guarda en el registro rd.
Para 64 bits se hace una extensi¢n de signo. No se produce desbordamiento*/
{
 UCHAR rs,rt,rd;    //Para guardar el n£mero del registro
 ULONG regrs, regrt, regrd;  //Variables intermedias
 ULONG auxrd;  //Variable auxiliar

 rs=Campo ("RS",instruccion);  //Cogemos el campo rs de la instrucci¢n
 rt=Campo ("RT",instruccion);  //Cogemos el campo rt de la instrucci¢n
 rd=Campo ("RD",instruccion);  //Cogemos el campo rd de la instrucci¢n

 regrs=rpg[rs][1];  //Guardamos parte baja de rs en regrs
 regrt=~rpg[rt][1]+1;  //Lo cambiamos de signo

 regrd = regrs + regrt;   //Sumamos y guardamos en regrd
 auxrd=regrd;   //En aux.. metemos lo que hay en reg.. porque queremos
 auxrd=auxrd>>31; //Desplazamos hacia la derecha 31 bits para ver signo.

 if (Estado("UX")==0)  //para 32 bits
   rpg[rd][1] = regrd;  //Guardamos en el registro el resultado
 else if (Estado("UX")==1)  //para 64 bits
 {
   rpg[rd][1] = regrd;   //metemos el resultado en el registro destino.

   //ahora vamos a hacer la extensi¢n de signo.
   if (auxrd != 0)         //Si auxrd es distinto de cero
     rpg[rd][0]=0xFFFFFFFF;  //Ponemos la parte alta toda a unos
   else if (auxrd == 0)  //Si auxrd es igual a cero
     rpg[rd][0]=0x00000000;   //Ponemos la parte alta toda a ceros
 }
}  //Fin de la instrucci¢n SUBU

void ANDI (ULONG instruccion)   //And Immediate
/*Esta operaci¢n hace la and l¢gica de un dato inmediato de 16 bits con
  los 16 bits de menor peso del registro rs. El resultado se guarda en
  el registro rt.*/
```

```
{
  UCHAR rs,rt;        //indican el n£mero de registro.
  ULONG auxi,inmediato; //variable auxiliar y dato inmediato.

  rs=Campo ("RS",instruccion);   //cogemos el campo rs de la instrucci¢n.
  rt=Campo ("RT",instruccion);   //cogemos el campo rt de la instrucci¢n.

  auxi = instruccion & 0x0000FFFF;  //vamos a calcular el dato inmediato
  inmediato=auxi;   //metemos el dato en inmediato.

  if (Estado("UX")==0)  //para 32 bits
  {
    auxi= inmediato & rpg[rs][1];  //se hace la and de inmediato y reg. rs.
    rpg[rt][1]= auxi & 0x0000FFFF; //se cogen los 16 bits de menor peso...
                          //...del resultado.
  }
  else if (Estado ("UX")==1)  //para 64 bits
  {
    rpg[rt][0]=0x00000000;   //la parte alta del registro destino a cero.
    auxi= inmediato & rpg[rs][1];  //se hace la and de inmediato y reg rs.
    rpg[rt][1]= auxi & 0x0000FFFF; //se cogen los 16 bits bajos del resultado
  }
}    //fin de la instrucci¢n ANDI

void DIV (ULONG instruccion)        //Divide
/*Esta instrucci¢n hace la divisi¢n con signo de los registros rs y rt. El
  cociente es guardado en el registro LO y el resto es guardado en el regis-
  tro HI. Para 64 bits se hace una extensi¢n de signo.*/
{
  ULONG auxi,cociente,resto;  //variable auxiliar, cociente y resto de div.
  signed long int regrs,regrt; //para guardar lo que hay en los regs con
signo.
  UCHAR rs,rt;    //indican el n£mero de registro.

  rs=Campo ("RS",instruccion);   //cogemos el campo rs de la instrucci¢n.
  rt=Campo ("RT",instruccion);   //cogemos el campo rt de la instrucci¢n.

  regrs = rpg[rs][1];   //en regrs metemos lo que hay en el registro rs.
  regrt = rpg[rt][1];   //en regrt metemos lo que hay en el registro rt.

  if (Estado("UX")==0)  //para 32 bits
  {
    if (rpg[rt][1] == 0)  //si el divisor es cero (no se puede hacer
divisi¢n)
    {
      gotoxy (30,23);
      printf ("    Divisi¢n por cero    ");  //informamos de que no se puede.
    }
    else   //si el divisor es distinto de cero (se puede hacer divisi¢n)
    {
      rpg[33][1]= regrs / regrt; //el cociente de la div. lo guardamos en LO
      rpg[32][1]= regrs % regrt; //el resto de la div. lo guardamos en HI
    }
  }
  else if (Estado ("UX")==1)  //para 64 bits
```

```
  {
    if (rpg[rt][1] == 0)  //si el divisor es cero (no se puede hacer
divisi¢n)
    {
      gotoxy (30,23);
      printf ("    Divisi¢n por cero    "); //informamos de que no se puede.
    }
    else  //si el divisor es distinto de cero (se puede hacer la divisi¢n).
    {
      cociente= regrs / regrt; //en cociente se guarda el cociente.
      resto= regrs % regrt;    //en resto se guarda el resto de la divisi¢n.
      auxi=cociente;    //en la variable auxi guardamos el cociente.
      auxi=auxi & 0x80000000;   //vamos a ver el valor que tiene el bit 31.
      auxi=auxi>>31;
      if (auxi!=0) rpg[33][0]=0xFFFFFFFF; //si !=0 la parte alta del reg. a 1
      else if (auxi==0) rpg[33][0]=0x00000000; //si ==0 la parte alta a 0
      rpg[33][1]=cociente;   //guardamos el cociente en el registro LO

      auxi=resto;   //en la variable auxi guardamos el resto de la divisi¢n.
      auxi=auxi & 0x80000000;  //vamos a ver el valor que tiene el bit 31.
      auxi=auxi>>31;
      if (auxi!=0) rpg[32][0]=0xFFFFFFFF; //si !=0 la parte alta del reg. a 1
      else if (auxi==0) rpg[32][0]=0x00000000; //si ==0 la parte alta a 0
      rpg[32][1]=resto;   //guardamos el resto en el registro HI
    }
  }
}            //fin de la instrucci¢n DIV

void DIVU (ULONG instruccion)     //Divide Unsigned
/*Esta instrucci¢n hace la divisi¢n sin signo de los registros rs y rt. El
  cociente es guardado en el registro LO y el resto es guardado en el regis-
  tro HI. Para 64 bits se hace una extensi¢n de signo.*/
{
  ULONG auxi,cociente,resto,regrs,regrt;   //Variables sin signo
  UCHAR rs,rt;       //Indican el n£mero de registro

  rs=Campo ("RS",instruccion);  //Cogemos el campo rs de la instrucci¢n.
  rt=Campo ("RT",instruccion);  //Cogemos el campo rt de la instrucci¢n.

  regrs = rpg[rs][1];  //En regrs metemos el contenido del registro rs.
  regrt = rpg[rt][1];  //En regrt metemos el contenido del registro rt.

  if (Estado("UX")==0)  //para 32 bits
  {
    if (rpg[rt][1] == 0)   //Si el divisor es igual a cero
    {
      gotoxy (30,23);
      printf ("    Divisi¢n por cero    ");  //Informamos que no se puede
    }
    else   //si el divisor no es igual a cero
    {
      rpg[33][1]= regrs / regrt; //En LO guardamos el cociente de la divisi¢n
      rpg[32][1]= regrs % regrt; //En HI guardamos el resto de la divisi¢n.
    }
  }
```

```
  else if (Estado ("UX")==1)  //para 64 bits
  {
    if (rpg[rt][1] == 0)   //Si el divisor es igual a cero
    {
      gotoxy (30,23);
      printf ("    Divisi¢n por cero    ");  //Informamos de el error.
    }
    else   //si el divisor no es igual a cero.
    {
      cociente= regrs / regrt;  //En cociente se guarda el cociente de la
div.
      resto= regrs % regrt;    //En resto se guarda el resto de la divisi¢n.
      auxi=cociente;    //En la variable auxiliar auxi se mete el cociente.
      auxi=auxi & 0x80000000;  //Vamos a ver el valor del bit 32
      auxi=auxi>>31;
      if (auxi!=0) rpg[33][0]=0xFFFFFFFF; //Hacemos la extensi¢n de signo.
      else if (auxi==0) rpg[33][0]=0x00000000;
      rpg[33][1]=cociente;    //En el registro LO guardamos el cociente.

      auxi=resto;   //En la variable auxiliar auxi se mete el resto.
      auxi=auxi & 0x80000000;  //Vamos a ver el valor del bit 32.
      auxi=auxi>>31;
      if (auxi!=0) rpg[32][0]=0xFFFFFFFF;   //Hacemos la extensi¢n de signo.
      else if (auxi==0) rpg[32][0]=0x00000000;
      rpg[32][1]=resto;   //En el registro HI guardamos el resto.
    }
  }
}     //Fin de la instrucci¢n DIVU

ULONG dividir (ULONG dividendo0, ULONG dividendo, ULONG divisor0, ULONG
divisor,char *car)
/*Esta divisi¢n solo se puede hacer si el dividendo es mayor que el
divisor.*/
{
  ULONG temp,cociente0,cociente,resto0,resto;
  UCHAR dividres[17][16],diviscoc[2][16];
  int i,j,c,arco,arcoaux;
  UCHAR tdivid,tcoc,tres,aux,limit,repetir=0;
  UCHAR fila=1,columna=0,filazar=0,dato=0,lmt;
  UCHAR coci[16],rest[16];

  for (i=0;i<16;i++)  //Inicializamos coci y rest a ceros
  {
    coci[i]=0;
    rest[i]=0;
  }

  for (i=0;i<17;i++)    //Iniciamos la matriz dividres a cero
    for (j=0;j<16;j++)  //Se llama as¡ porque contiene el dividendo y el
      dividres[i][j]=0; //resto de la divisi¢n

  for (i=0;i<2;i++)    //Iniciamos la matriz diviscoc a cero
    for (j=0;j<16;j++) //Se llama as¡ porque contiene el divisor y el
      diviscoc[i][j]=0; //cociente de la divisi¢n
```

```
//Cogemos el dividendo
for (i=0;i<8;i++)
{
  temp=dividendo0<<(4*i);  //0,4,8,12,16
  dividres[0][i]=temp>>28;  //28 siempre
}
for (i=8;i<16;i++)
{
  temp=dividendo<<(4*(i-8));  //0,4,8,12,16
  dividres[0][i]=temp>>28;  //28 siempre
}

/*Cogemos el divisor*/
for (i=0;i<8;i++)
{
  temp=divisor0<<(4*i);  //0,4,8,12,16
  diviscoc[0][i]=temp>>28;  //28 siempre
}
for (i=8;i<16;i++)
{
  temp=divisor<<(4*(i-8));  //0,4,8,12,16
  diviscoc[0][i]=temp>>28;  //28 siempre
}

//ahora vamos a calcular a partir de donde acaban los ceros
i=0;
while (dividres[0][i]==0)
  i++;       //Buscamos el l¡mite del dividendo

j=0;
while (diviscoc[0][j]==0)
  j++;                    //Buscamos el l¡mite del divisor

if (i==j)
  arco=15;
else if (i<j)
{
  if (dividres[0][i]>diviscoc[0][j])
    arco=i+(15-j);
  else if (dividres[0][i]<diviscoc[0][j])
    arco=i+((15-j)+1);
  else if (dividres[0][i]==diviscoc[0][j])
  {
    c=0;
    while ((dividres[0][i+c]==diviscoc[0][j+c]) && (c<(16-j)))
    {
    c++;
    }

    if (c==2)
    arco=i+(15-j);
    else if (dividres[0][i+c]<diviscoc[0][j+c])
    arco=i+((15-j)+1);
    else if (dividres[0][i+c]>diviscoc[0][j+c])
    arco=i+(15-j);
```

```
  }
}
  //ahora hacemos la divisi¢n
while (arco!=16)
{
  repetir=0;
  do
  {
    limit=15;
    arcoaux=arco;
    tdivid=0;

    //El primer n£mero para sacar el cociente
    if ((arcoaux-i)>(15-j))
    {
    dato=dividres[filazar][i];
    dato=dato<<4;
    dato=dividres[filazar][i+1] | dato;
    }
    else if ((arcoaux-i)==(15-j))
    dato=dividres[filazar][i];

    tcoc=dato/diviscoc[0][j];
    tcoc=tcoc-repetir;
    diviscoc[1][columna]=tcoc;

    if ((arco-i)>(15-j))
    lmt=i+2;
    else if ((arco-i)==(15-j))
    lmt=i+1;

    for (c=arco;c>=lmt;c--)
    {
    aux=tcoc*diviscoc[0][limit];
    tdivid=tdivid>>4;
    aux=aux+tdivid;
    tdivid=dividres[filazar][arcoaux];
    if (tdivid<aux)
    {
      tdivid=tdivid+(aux & 0xF0);
      if (tdivid<aux)
        tdivid=tdivid+0x10;
    }
    tres=tdivid-aux;
    dividres[fila][arcoaux]=tres;
    arcoaux--;
    limit--;
    }

    aux=tcoc*diviscoc[0][j];
    tdivid=tdivid>>4;
    aux=aux+tdivid;
    tdivid=dato;
    if ((tdivid > aux) || (tdivid==aux))
    {
```

```
    tres=tdivid-aux;
    if ((arco-i)>(15-j))
      dividres[fila][i+1]=tres;
    else if ((arco-i)==(15-j))
      dividres[fila][i]=tres;
    }
    else
    {
    repetir++;
    }
  }while (tdivid < aux);
//Ahora bajamos el n£mero
  arco++;
  dividres[fila][arco]=dividres[0][arco];
  if (dividres[fila][i]==0)
    i++;

  fila++;
  filazar++;
  columna++;
}

for (i=columna-1,j=15;i>=0;i--,j--)
  coci[j]=diviscoc[1][i];

for (i=0;i<16;i++)
  rest[i]=dividres[fila-1][i];

//Ahora lo convertimos a long
cociente0=0;
resto0=0;
cociente=0;
resto=0;

for (i=0;i<8;i++)
{
  temp=(ULONG)coci[i];
  temp=temp<<(28-(4*i));
  cociente0=cociente0 | temp;
  temp=(ULONG)rest[i];
  temp=temp<<(28-(4*i));
  resto0=resto0 | temp;
}
for (i=8;i<16;i++)
{
  temp=(ULONG)coci[i];
  temp=temp<<(60-(4*i));
  cociente=cociente | temp;
  temp=(ULONG)rest[i];
  temp=temp<<(60-(4*i));
  resto=resto | temp;
}

if (strcmp(car,"COCIENTE0")==0)
  return cociente0;
```

```
  else if (strcmp(car,"COCIENTE")==0)
    return cociente;
  else if (strcmp(car,"RESTO0")==0)
    return resto0;
  else if (strcmp(car,"RESTO")==0)
    return resto;
}

void DDIV (ULONG instruccion)        //Doubleword Divide
/*Esta instrucci¢n hace la divisi¢n de 64 bits con signo de los registros
rs y rt. El cociente es guardado en el registro LO y el resto es guardado
en el registro HI. Para 32 se produce una Reserved Instruction Exception.*/
{
  ULONG auxi,cociente,resto;  //variable auxiliar, cociente y resto de div.
  signed long int rssigno,rtsigno; //guarda lo que hay en los regs con signo.
  ULONG rsnosigno,rtnosigno;
  ULONG regrs0,regrs1,regrt0,regrt1;
  UCHAR rs,rt,cambio=0;    //indican el n£mero de registro.

  rs=Campo ("RS",instruccion);   //cogemos el campo rs de la instrucci¢n.
  rt=Campo ("RT",instruccion);   //cogemos el campo rt de la instrucci¢n.

  regrs0=rpg[rs][0]>>31;  //Guardamos en regrs0 la parte alta del reg. rs.
  regrt0=rpg[rt][0]>>31;  //Guardamos en regrt0 la parte alta del reg. rt.

  if ((regrs0!=0) && (regrt0!=0)) //Si los dos operandos son negativos
  {
    regrs1= (~rpg[rs][1]+1);  //hacemos el positivo en complemento a dos
    regrs0= (~rpg[rs][0]);   //invertimos todos los bits
    if (regrs1==0) regrs0++; //si hay acarreo al sumar a regrs1 1, inc.
regrs0
    regrt1= (~rpg[rt][1]+1);  //el otro operando tambi,n le cambiamos el
signo
    regrt0= (~rpg[rt][0]);
    if (regrt1==0) regrt0++;
    cambio=0;          //No hay que hacer cambio de signo al final.
  }
  else if ((regrs0!=0) && (regrt0==0)) //Si solo el reg. rs es negativo
  {
    regrs1= (~rpg[rs][1]+1);  //hacemos el positivo en complemento a dos
    regrs0= (~rpg[rs][0]); //invertimos todos los bits
    if (regrs1==0) regrs0++; //si hay acarreo al sumar a regrs1 1, inc.
regrs0
    cambio=1;     //Hay que hacer un cambio de signo al resultado
    regrt1= rpg[rt][1];  //El registro rt al ser positivo se queda como est .
    regrt0= rpg[rt][0];
  }
  else if ((regrt0!=0) && (regrs0==0)) //Si solo el reg. rt es negativo
  {
    regrt1= (~rpg[rt][1]+1);  //hacemos el positivo en complemento a dos
    regrt0= (~rpg[rt][0]);  //invertimos todos los bits
    if (regrt1==0) regrt0++; //si hay acarreo al sumar a regrt1 1, inc.
regrt0
    cambio=1;    //Hay que hacer un cambio de signo al resultado final
    regrs1= rpg[rs][1];  //el registro rs al ser positivo se queda como est .
```

```
    regrs0= rpg[rs][0];
  }
  else if ((regrs0==0) && (regrt0==0)) //Si los dos reg. son positivos
  {
    regrs1=rpg[rs][1];  //El registro rs se queda como est  porque es
positivo
    regrs0=rpg[rs][0];
    regrt1=rpg[rt][1];  //El registro rt se queda como est  porque es
positivo
    regrt0=rpg[rt][0];
    cambio=0;   //No hay que hacer cambio de signo al resultado final
  }

  rssigno=regrs0;
  rsnosigno=regrs1;
  rtsigno=regrt0;
  rtnosigno=regrt1;

  if (Estado("UX")==0)  //para 32 bits
  {
    General_Exception();        //Reserved instruction exception
    gotoxy (30,23);
    printf ("Reserved Instr. exception");
  }
  else if (Estado ("UX")==1)  //para 64 bits
  {
    if ((rpg[rt][1] == 0) && (rpg[rt][0]==0))  //si el divisor es cero
    {
      gotoxy (30,23);
      printf ("    Divisi¢n por cero    "); //informamos de que no se puede.
    }
    else  //si el divisor es distinto de cero (se puede hacer la divisi¢n).
    {
       if (rpg[rs][0]!=rpg[rt][0])  //Si las dos partes altas son distintas
       {
       if ((rssigno>rtsigno) || (rssigno==rtsigno))  //y rssigno > rtsigno
       {
         rpg[33][0]=dividir(regrs0,regrs1,regrt0,regrt1,"COCIENTE0");
         rpg[33][1]=dividir(regrs0,regrs1,regrt0,regrt1,"COCIENTE");
         rpg[32][0]=dividir(regrs0,regrs1,regrt0,regrt1,"RESTO0");
         rpg[32][1]=dividir(regrs0,regrs1,regrt0,regrt1,"RESTO");
         if (cambio==1)
         {
           rpg[33][1]= (~rpg[33][1]+1);  //hacemos el positivo en complemento
a dos
           rpg[33][0]= (~rpg[33][0]);  //invertimos todos los bits
           if (rpg[33][1]==0) rpg[33][0]++; //si hay acarreo al sumar a
regrt1 1, inc. regrt0
           rpg[32][1]= (~rpg[32][1]+1);  //hacemos el positivo en complemento
a dos
           rpg[32][0]= (~rpg[32][0]);  //invertimos todos los bits
           if (rpg[32][1]==0) rpg[32][0]++; //si hay acarreo al sumar a
regrt1 1, inc. regrt0
         }
       }
```

```
       else if (rssigno<rtsigno)
       {
         rpg[33][0]=0;  //En el cociente hay un cero
         rpg[33][1]=0;
         rpg[32][0]=rpg[rs][0];  //El resto es igual que el dividendo
         rpg[32][1]=rpg[rs][1];
       }
       }
       else if (rpg[rs][0]==rpg[rt][0]) //Si las dos partes altas son iguales
       {
       if ((rsnosigno>rtnosigno) || (rsnosigno==rtnosigno))  //si mayor
       {
         rpg[33][0]=dividir(regrs0,regrs1,regrt0,regrt1,"COCIENTE0");
         rpg[33][1]=dividir(regrs0,regrs1,regrt0,regrt1,"COCIENTE");
         rpg[32][0]=dividir(regrs0,regrs1,regrt0,regrt1,"RESTO0");
         rpg[32][1]=dividir(regrs0,regrs1,regrt0,regrt1,"RESTO");
         if (cambio==1)
         {
           rpg[33][1]= (~rpg[33][1]+1);  //hacemos el positivo en complemento
a dos
           rpg[33][0]= (~rpg[33][0]);  //invertimos todos los bits
           if (rpg[33][1]==0) rpg[33][0]++; //si hay acarreo al sumar a
regrt1 1, inc. regrt0
           rpg[32][1]= (~rpg[32][1]+1);  //hacemos el positivo en complemento
a dos
           rpg[32][0]= (~rpg[32][0]);  //invertimos todos los bits
           if (rpg[32][1]==0) rpg[32][0]++; //si hay acarreo al sumar a
regrt1 1, inc. regrt0
         }
       }
       else if (rsnosigno<rtnosigno)
       {
         rpg[33][0]=0;  //En el cociente hay un cero
         rpg[33][1]=0;
         rpg[32][0]=rpg[rs][0];  //El resto es igual que el dividendo
         rpg[32][1]=rpg[rs][1];
       }
       }
    }
  }
}            //fin de la instrucci¢n DDIV

void DDIVU (ULONG instruccion)        //Doubleword Divide Unsigned
/*Esta instrucci¢n hace la divisi¢n de 64 bits sin signo de los registros
rs y rt. El cociente es guardado en el registro LO y el resto es guardado
en el registro HI. Para 32 se produce una Reserved Instruction Exception.*/
{
  ULONG auxi,cociente,resto;  //variable auxiliar, cociente y resto de div.
  ULONG regrs,regrt;
  UCHAR rs,rt;    //indican el n£mero de registro.

  rs=Campo ("RS",instruccion);   //cogemos el campo rs de la instrucci¢n.
  rt=Campo ("RT",instruccion);   //cogemos el campo rt de la instrucci¢n.

  if (Estado("UX")==0)  //para 32 bits
```

```
  {
    General_Exception();        //Reserved instruction exception
    gotoxy (30,23);
    printf ("Reserved Instr. exception");
  }
  else if (Estado ("UX")==1)  //para 64 bits
  {
    if ((rpg[rt][1] == 0) && (rpg[rt][0]==0))  //si el divisor es cero
    {
      gotoxy (30,23);
      printf ("    Divisi¢n por cero    "); //informamos de que no se puede.
    }
    else  //si el divisor es distinto de cero (se puede hacer la divisi¢n).
    {
       if (rpg[rs][0]!=rpg[rt][0])  //Si las dos partes altas son distintas
       {
       regrs=rpg[rs][0];
       regrt=rpg[rt][0];
       if ((regrs>regrt) || (regrs==regrt))  //y regrs >= regrt
       {

rpg[33][0]=dividir(rpg[rs][0],rpg[rs][1],rpg[rt][0],rpg[rt][1],"COCIENTE0");

rpg[33][1]=dividir(rpg[rs][0],rpg[rs][1],rpg[rt][0],rpg[rt][1],"COCIENTE");

rpg[32][0]=dividir(rpg[rs][0],rpg[rs][1],rpg[rt][0],rpg[rt][1],"RESTO0");

rpg[32][1]=dividir(rpg[rs][0],rpg[rs][1],rpg[rt][0],rpg[rt][1],"RESTO");
       }
       else if (regrs<regrt)
       {
         rpg[33][0]=0;  //En el cociente hay un cero
         rpg[33][1]=0;
         rpg[32][0]=rpg[rs][0];  //El resto es igual que el dividendo
         rpg[32][1]=rpg[rs][1];
       }
       }
       else if (rpg[rs][0]==rpg[rt][0]) //Si las dos partes altas son iguales
       {
       regrs=rpg[rs][1];
       regrt=rpg[rt][1];
       if ((regrs>regrt) || (regrs==regrt))  //si mayor o igual
       {

rpg[33][0]=dividir(rpg[rs][0],rpg[rs][1],rpg[rt][0],rpg[rt][1],"COCIENTE0");

rpg[33][1]=dividir(rpg[rs][0],rpg[rs][1],rpg[rt][0],rpg[rt][1],"COCIENTE");

rpg[32][0]=dividir(rpg[rs][0],rpg[rs][1],rpg[rt][0],rpg[rt][1],"RESTO0");

rpg[32][1]=dividir(rpg[rs][0],rpg[rs][1],rpg[rt][0],rpg[rt][1],"RESTO");
       }
       else if (regrs<regrt)
       {
         rpg[33][0]=0;  //En el cociente hay un cero
```

```
          rpg[33][1]=0;
          rpg[32][0]=rpg[rs][0];  //El resto es igual que el dividendo
          rpg[32][1]=rpg[rs][1];
        }
        }
    }
  }
}            //fin de la instrucci¢n DDIVU

void MFHI (ULONG instruccion)  //Move From Hi
/*El contenido del registro especial HI se carga en el registro rd*/
{
  UCHAR rd;  //Para tener el n£mero del registro

  rd=Campo ("RD",instruccion);  //Cogemos el campo rd de la instrucci¢n

  if (Estado("UX")==0)  //para 32 bits
  {
    rpg[rd][1] = rpg[32][1]; //Metemos parte baja de HI en parte baja de rd
  }
  else if (Estado("UX")==1)    //para 64 bits.
  {
    rpg[rd][0] = rpg[32][0]; //Metemos parte alta de HI en parte alta de rd
    rpg[rd][1] = rpg[32][1]; //Metemos parte baja de HI en parte baja de rd
  }
}    //Fin de la instrucci¢n MFHI

void MFLO (ULONG instruccion)  //Move From LO
/*El contenido del registro especial LO se guarda en el registro rd*/
{
  UCHAR rd;  //Para tener el n£mero de registro

  rd=Campo ("RD",instruccion);  //Cogemos el campo rd de la instrucci¢n

  if (Estado("UX")==0)  //para 32 bits
  {
    rpg[rd][1] = rpg[33][1]; //Metemos parte baja de LO en parte baja de rd
  }
  else if (Estado("UX")==1)  //para 64 bits.
  {
    rpg[rd][0] = rpg[33][0];  //Metemos parte alta de LO en parte alta de rd
    rpg[rd][1] = rpg[33][1];  //Metemos parte baja de LO en parte baja de rd
  }
}   //Fin de la instrucci¢n MFLO

void MTHI (ULONG instruccion)  //Move to HI
/*El contenido del registro rs es cargado en el registro especial HI*/
{
  UCHAR rs;  //Para el n£mero de registro.

  rs=Campo ("RS",instruccion);  //Cogemos el campo rs de la instrucci¢n

  if (Estado("UX")==0)   //para 32 bits
  {
    rpg[32][1] = rpg[rs][1];  //El contenido de rs se mete en HI (parte baja)
```

```
  }
  else if (Estado("UX")==1)  //para 64 bits.
  {
    rpg[32][0] = rpg[rs][0];  //Parte alta de rs se mete en parte alta de HI
    rpg[32][1] = rpg[rs][1];  //Parte baja de rs se mete en parte baja de HI
  }
}   //Fin de la instrucci¢n MTHI

void MTLO (ULONG instruccion) //Move to LO
/*El contenido del registro rs se carga en el registro especial LO*/
{
  UCHAR rs;  //Para guardar el n£mero de registro

  rs=Campo ("RS",instruccion);  //Cogemos el campo rs de la instrucci¢n

  if (Estado("UX")==0)  //para 32 bits
  {
    rpg[33][1] = rpg[rs][1]; //Parte baja de rs se mete en parte baja de LO
  }
  else if (Estado("UX")==1)  //para 64 bits.
  {
    rpg[33][0] = rpg[rs][0];  //Parte alta de rs se mete en parte alta de LO
    rpg[33][1] = rpg[rs][1];  //Parte baja de rs se mete en parte baja de LO
  }
}    //Fin de la instrucci¢n MTLO

void NOR (ULONG instruccion)
/*Esta instrucci¢n lo que hace es hacer la operaci¢n NOR l¢gica de los re-
gistros rs y rt y el resultado se guarda en el registro rd.*/
{
  UCHAR rs,rt,rd;  //Para coger el n£mero del registro
  ULONG auxi[1][2];   //variable auxiliar.

  rs=Campo ("RS",instruccion);  //Cogemos campo rs de la instrucci¢n
  rt=Campo ("RT",instruccion);  //Cogemos campo rt de la instrucci¢n
  rd=Campo ("RD",instruccion);  //Cogemos campo rd de la instrucci¢n

  if (Estado("UX")==0)  //Para 32 bits.
  {
    auxi[0][1] = rpg[rs][1] | rpg[rt][1];  //Hacemos la or del reg. rs y rt
    rpg[rd][1] = auxi[0][1] ^ 0xFFFFFFFF; //hacemos la xor de auxi y
0xFFFFF..
  }
  else if (Estado("UX")==1)   //Para 64 bits.
  {
    auxi[0][1] = rpg[rs][1] | rpg[rt][1]; //Or del reg. rs y rt (parte baja)
    rpg[rd][1] = auxi[0][1] ^ 0xFFFFFFFF; //Xor de auxi y 0xFFFFFFFF
    auxi[0][0] = rpg[rs][0] | rpg[rt][0]; //Or del reg. rs y rt (parte alta)
    rpg[rd][0] = auxi[0][0] ^ 0xFFFFFFFF; //Xor de auxi y 0xFFFFFFFF
  }
}    //Fin de la instrucci¢n NOR

void OR (ULONG instruccion)
/*Esta instrucci¢n lo que hace es hacer la operaci¢n OR l¢gica de los re-
gistros rs y rt y el resultado se guarda en el registro rd.*/
```

```
{
  UCHAR rs,rt,rd;   //Para coger el n£mero del registro
  ULONG auxi[1][2];  //variable auxiliar.

  rs=Campo ("RS",instruccion);  //cogemos el campo rs de la instrucci¢n
  rt=Campo ("RT",instruccion);  //cogemos el campo rt de la instrucci¢n
  rd=Campo ("RD",instruccion);  //cogemos el campo rd de la instrucci¢n

  if (Estado("UX")==0)  //Para 32 bits.
    rpg[rd][1] = rpg[rs][1] | rpg[rt][1];  //se hace la or directamente
  else if (Estado("UX")==1)   //Para 64 bits.
  {
    rpg[rd][1] = rpg[rs][1] | rpg[rt][1]; //Se hace la or de la parte baja
    rpg[rd][0] = rpg[rs][0] | rpg[rt][0];  //Se hace la or de la parte alta
  }
}   //fin de la instrucci¢n OR

void ORI (ULONG instruccion)   //Or Immediate
/*El dato inmediato se extiende con ceros y se combina con el contenido
del registro rs en una operaci¢n or l¢gica. El resultado se guarda en
el registro rt.*/
{
  UCHAR rs,rt;   //Para el n£mero de los registros
  ULONG inmediato, regori, resul, izqreg;  //variables sin signo de 32 bits

  rs=Campo ("RS",instruccion);  //Cogemos el campo rs de la instrucci¢n
  rt=Campo ("RT",instruccion);  //Cogemos el campo rt de la instrucci¢n
  inmediato = instruccion & 0x0000FFFF;  //Cogemos el campo inmediato.

  regori = rpg[rs][1] & 0x0000FFFF;  //Cogemos los 16 bits mas bajos
  resul = inmediato | regori;    //hace la or con los 16 bits de inmediato.
  izqreg = rpg[rs][1] & 0xFFFF0000;  //bits 31..16 del registro.
  izqreg = izqreg | 0x0000FFFF;  //Los 16 bits mas bajos se ponen a 1
  resul = resul | 0xFFFF0000;  //Ponemos los 16 bits mas altos a 1

  resul = resul & izqreg;   //Esto es lo que vamos a meter en el registro.

  if (Estado("UX")==0)  //Para 32 bits.
  {
    rpg[rt][1]= resul;   //Guardamos el resultado en la parte baja del reg.
  }
  else if (Estado("UX")==1)  //Para 64 bits.
  {
    rpg[rt][0] = rpg[rs][0];   //Dejamos la parte alta como est
    rpg[rt][1] = resul;   //Guardamos el resultado en la parte baja del reg.
  }
}     //fin de la instrucci¢n ori.

void XOR (ULONG instruccion)  //Exclusive Or
/*Se hace la xor l¢gica del registro rs y rt y el resultado se guarda en el
registro rd*/
{
  UCHAR rs,rt,rd;  //Para guardar el n£mero del registro

  rs=Campo ("RS",instruccion); //Cogemos el campo rs de la instrucci¢n
```

```
  rt=Campo ("RT",instruccion);  //Cogemos el campo rt de la instrucci¢n
  rd=Campo ("RD",instruccion);  //Cogemos el campo rd de la instrucci¢n

  if (Estado("UX")==0)  //Para 32 bits.
    rpg[rd][1] = rpg[rs][1] ^ rpg[rt][1];  //Se hace la xor
  else if (Estado("UX")==1)  //Para 64 bits.
  {
    rpg[rd][0] = rpg[rs][0] ^ rpg[rt][0];  //Se hace la xor
    rpg[rd][1] = rpg[rs][1] ^ rpg[rt][1];   //Se hace la xor
  }
}   //Fin de la instrucci¢n XOR

void XORI (ULONG instruccion)  //Xor Immediate
/*El dato de 16 bits se extiende con ceros y se combina con el contenido
del registro rs en una operaci¢n or exclusiva.*/
{
  UCHAR rs,rt;  //Para coger el n£mero de los registros
  ULONG inmediato;  //Para el dato inmediato

  rs=Campo ("RS",instruccion);  //Cogemos el campo rs de la instrucci¢n
  rt=Campo ("RT",instruccion);  //Cogemos el campo rt de la instrucci¢n

  inmediato = instruccion & 0x0000FFFF;  //Cogemos el dato inmediato de
instr.

  if (Estado("UX")==0)  //Para 32 bits.
  {
    rpg[rt][1] = rpg[rs][1] ^ inmediato;  //Se hace la xor de reg. y dato
inm.
  }
  else if (Estado("UX")==1)  //Para 64 bits.
  {
    rpg[rt][0] = rpg[rs][0] ^ 0x00000000;  //Se hace la xor de la parte alta.
    rpg[rt][1] = rpg[rs][1] ^ inmediato;  //Se hace la xor del reg. y dato
inm.
  }
}  //fin de la instrucci¢n XORI

void DADDIU (ULONG instruccion)  //Doubleword Add Immediate Unsigned
/*Se hace una extensi¢n del dato inmediato y se suma al contenido del reg.
rs para formar el resultado. El resultado se guara en el registro general
rt. No ocurre ninguna excepci¢n de desbordamiento y si se ejecuta en modo
32 bits se produce una Reserved instruction exception*/
{
  UCHAR rs,rt,aca;  //Para coger el n£mero de registros y acarreo
  ULONG inmediato, auxi;  //Para el dato inmediato y variable auxiliar.

  rs=Campo ("RS",instruccion);  //Cogemos el campo rs de la instrucci¢n
  rt=Campo ("RT",instruccion);  //Cogemos el campo rt de la instrucci¢n
  inmediato = instruccion & 0x0000FFFF;  //Cogemos el dato inmediato de
instr.
  auxi = inmediato; //En auxi metemos el dato inmediato
  auxi = auxi>>15;   //Lo desplazamos 15 bits hacia la derecha.

  if (Estado("UX")==0)  //Para 32 bits.
```

```
  {
    General_Exception();        //Reserved instruction exception
    gotoxy (30,23);
    printf ("Reserved Instr. exception");
  }
  else if (Estado("UX")==1)  //Para 64 bits.
  {
    if (auxi!=0)  //Si auxi es distinto de cero
    {
      rpg[rt][0]= rpg[rs][0] + 0xFFFFFFFF;  //Sumamos las partes altas.
      inmediato = inmediato | 0xFFFF0000;  //Extendemos el dato inmediato
    }
    else if (auxi==0)   //Si auxi es igual a cero
    {
      rpg[rt][0]= rpg[rs][0] + 0x00000000;  //Sumamos las partes altas.
    }
    rpg[rt][1] = rpg[rs][1] + inmediato;  //Sumamos las partes bajas
    aca=acarreo (rpg[rs][1],inmediato);  //Miramos si hay acarreo
    if (aca==1) rpg[rt][0]++;  //Si acarreo en parte baja incrementamos alta
  }
}     //Fin de la instrucci¢n DADDIU

void DADDI (ULONG instruccion)    //Doubleword Add Immediate
/*Se hace una extensi¢n de signo al dato inmediato y se suma al contenido
del registro rs para formar el resultado, que se guarda en el reg. rt. Si
se ejecuta en modo 32 bits se produce una reserved instruction exception
y si se produce desbordamiento, entonces habr  una integer overflow
exception.*/
{
  UCHAR rs,rt,aca;  //Para el n£mero de registros y acarreo
  ULONG auxi,auxrs0,inme0,auxrt0;  //Variables auxiliares para ver bit signo.
  signed long int inmediato, regrs1,regrs0,regrt0,regrt1;  //Var. intemedias

  rs=Campo ("RS",instruccion);  //Cogemos el campo rs de la instrucci¢n
  rt=Campo ("RT",instruccion);  //Cogemos el campo rt de la instrucci¢n
  inmediato = instruccion & 0x0000FFFF;  //Cogemos el dato inmediato de
instr.
  auxi = inmediato;  //El dato inmediato lo metemos en auxi.
  auxi = auxi>>15;  //Lo desplazamos 15 bits hacia la derecha.

  if (auxi!=0)   //Si auxi es distinto de cero
  {
    inme0 = 0xFFFFFFFF;   //La parte alta de inmediato es todo unos
    inmediato = inmediato | 0xFFFF0000;  //Hacemos extensi¢n del dato inmedi.
  }
  else if (auxi==0)  //Si auxi es igual a cero
  {
    inme0 = 0x00000000;  //La parte alta de inmediato es todo ceros.
  }

  regrs0=rpg[rs][0];    //En regrs0 metemos parte alta del registro rs
  regrs1=rpg[rs][1];    //En regrs1 metemos parte baja del registro rs

  regrt0 = regrs0 + inme0;  //Sumamos las partes altas
  regrt1 = regrs1 + inmediato;   //se hace la suma y el resultado en regrt.
```

```
  aca=acarreo (regrs1,inmediato); //Si hay acarreo en las partes bajas
  if (aca==1) regrt0++;  //se suma uno a la parte alta del resultado.

  auxrs0=regrs0;   //en la var auxiliar auxrs se guarda el dato del reg. rs.
  auxrt0=regrt0;   //en auxrt se guarda el resultado.
  inme0 =inme0 >> 31;  //se mira el bit 31
  auxrs0=auxrs0 >> 31;
  auxrt0=auxrt0 >> 31;

  if (Estado("UX")==0)   //Para 32 bits.
  {
    General_Exception();        //Reserved instruction exception
    gotoxy (30,23);
    printf ("Reserved Instr. exception");
  }
  else if (Estado("UX")==1)   //Para 64 bits.
  {
    if ((auxrs0 == inme0) && (auxrt0 != auxrs0))  //Si hay overflow
    {
      General_Exception();        //Integer overflow exception
      gotoxy (30,23);
      printf ("Integer overfl. exception");
    }
    else       //Si no hay overflow
    {
      rpg[rt][0] = regrt0;  //Metemos la parte alta del resultado en el reg.
      rpg[rt][1] = regrt1;  //Metemos la parte baja del resultado en el reg.
    }
  }
}     //fin de la instrucci¢n DADDI

void DADDU (ULONG instruccion)
/*Si esta instrucci¢n se ejecuta en modo 32 bits se produce una reserved
instruction exception. Solo se puede ejecutar en modo 64 bits para poder
obtener un resultado. Suma sin signo el registro rt y el rs y el resultado
se guarda en el registro rd. No se produce excepci¢n de desbordamiento*/
{
  UCHAR rs,rt,rd,aca;  //variables para coger los registros y acarreo

  rs=Campo ("RS",instruccion);  //cogemos el campo rs de la instrucci¢n
  rt=Campo ("RT",instruccion);  //cogemos el campo rt de la instrucci¢n
  rd=Campo ("RD",instruccion);  //cogemos el campo rt de la instrucci¢n

  if (Estado("UX")==0)  //Para 32 bits.
  {
    General_Exception();        //Reserved instruction exception
    gotoxy (30,23);
    printf ("Reserved Instr. exception");
  }
  else if (Estado("UX")==1)  //Para 64 bits.
  {
    rpg[rd][0]= rpg[rs][0] + rpg[rt][0];  //sumamos la parte alta
    rpg[rd][1]= rpg[rs][1] + rpg[rt][1];  //sumamos la parte baja
    aca=acarreo (rpg[rs][1],rpg[rt][1]);  //vemos si hay acarreo en parte
baja
```

```
    if (aca==1) rpg[rd][0]++;   //si hay acarreo incrementamos parte alta
  }
}         //fin de la instrucci¢n DADDU

void DADD (ULONG instruccion)
/*Si esta instrucci¢n se ejecuta en modo 32 bits se produce una reserved
instruction exception. Solo se puede ejecutar en modo 64 bits para poder
obtener un resultado. Suma con signo el registro rt y el rs y el resultado
se guarda en el registro rd. Se puede producir una excepci¢n de
desbordamiento*/
{
  UCHAR rs,rt,rd,aca;  //variables de los registros y acarreo
  signed long int regrs0,regrs1,regrt0,regrt1,regrd0,regrd1;//var.
intermedias
  ULONG auxrs0,auxrt0,auxrd0;  //variables auxiliares para ver bit 64.

  rs=Campo ("RS",instruccion);  //cogemos el n£mero de registro rs
  rt=Campo ("RT",instruccion);  //cogemos el n£mero de registro rt
  rd=Campo ("RD",instruccion);  //cogemos el n£mero de registro rd

  regrs0=rpg[rs][0];   //metemos la parte alta del reg. rs en regrs0
  regrs1=rpg[rs][1];   //metemos la parte baja del reg. rs en regrs1
  regrt0=rpg[rt][0];   //metemos la parte alta del reg. rt en regrt0
  regrt1=rpg[rt][1];   //metemos la parte baja del reg. rt en regrt1

  regrd0=regrs0+regrt0;  //sumamos la parte alta de los reg. rs+rt
  regrd1=regrs1+regrt1;  //sumamos la parte baja de los reg. rs+rt
  aca=acarreo (regrs1,regrt1);  //miramos si hay acarreo en la parte baja
  if (aca==1) regrd0++;   //si hay acarreo incrementamos en uno la parte
alta.

  auxrs0=regrs0;  //en auxrs0 metemos lo que hay en regrs0
  auxrt0=regrt0;  //en auxrt0 metemos lo que hay en regrt0
  auxrd0=regrd0;  //en auxrt0 metemos lo que hay en regrd0
  auxrs0=auxrs0>>31;  //desplazamos a la derecha 31 bits...
  auxrt0=auxrt0>>31;  //...para ver el valor del bit 64 de los tres...
  auxrd0=auxrd0>>31;  //...registros rs, rt, rd y as¡ sabemos si hay
overflow.

  if (Estado("UX")==0)  //Para 32 bits.
  {
    General_Exception();        //Reserved instruction exception
    gotoxy (30,23);
    printf ("Reserved Instr. exception");
  }
  else if (Estado("UX")==1)  //Para 64 bits.
  {
   if ((auxrs0 == auxrt0) && (auxrd0 != auxrs0))
   /*la £nica condici¢n para que haya overflow es que los bit de signo de
     los dos operandos sean iguales y el bit de signo del resultado sea
     distinto que el de los operandos*/
   {
     General_Exception();        //Integer overflow exception
     gotoxy (30,23);
     printf ("Integer overfl. exception");
```

```
   }
   else  //en caso de que no haya overflow
   {
     rpg[rd][0] = regrd0;
     rpg[rd][1] = regrd1;  //metemos el resultado en el registro rd.
   }
  }
}      //Fin de la instrucci¢n DADD


void DSUBU (ULONG instruccion)
/*Si el modo de operaci¢n es de 32 bits se produce una Reserved instr. excep-
tion. En 64 bits hace la resta de los registro rs y rt y el resultado lo
mete en el registro rd. No se produce ninguna excepci¢n de desbordamiento.*/
{
  UCHAR rs,rt,rd,aca;  //Para coger n£mero de regs. y acarreo.
  ULONG regrs0,regrs1,regrt0,regrt1,regrd0,regrd1;  //var. auxiliares

  rs=Campo ("RS",instruccion);  //cogemos el campo rs de la instrucci¢n
  rt=Campo ("RT",instruccion);  //cogemos el campo rt de la instrucci¢n
  rd=Campo ("RD",instruccion);  //cogemos el campo rd de la instrucci¢n

  if (Estado("UX")==0)  //Para 32 bits.
  {
    General_Exception();        //Reserved instruction exception
    gotoxy (30,23);
    printf ("Reserved Instr. exception");
  }
  else if (Estado("UX")==1)  //Para 64 bits.
  {
    regrs0=rpg[rs][0];  //El primer operando se deja como est
    regrs1=rpg[rs][1];

    regrt0=~rpg[rt][0];   //Al segundo operando se le cambia de signo.
    regrt1=~rpg[rt][1]+1;
    if (regrt1==0) regrt0++;

    regrd0=regrs0+regrt0;    //Se hace la suma de la parte alta
    regrd1=regrs1+regrt1;   //Se hace la suma de la parte baja
    aca=acarreo (regrs1,regrt1); //si hay acarreo intermedio
    if (aca==1) regrd0++;  //se le suma uno a la parte alta

    rpg[rd][0]=regrd0;   //metemos el resultado en la parte alta
    rpg[rd][1]=regrd1;   //metemos el resultado en la parte baja
  }
}         //fin de la instrucci¢n DSUBU

void DSUB (ULONG instruccion)
/*Si el modo de operaci¢n es de 32 bits se produce una Reserved instr. excep-
tion. En 64 bits hace la resta de los registro rs y rt y el resultado lo
mete en el registro rd. Si hay overflow se produce una excepci¢n.*/
{
  UCHAR rs,rt,rd,aca;  //Para coger n£mero de regs. y acarreo.
  ULONG regrs0,regrs1,regrt0,regrt1,regrd0,regrd1;  //var. auxiliares
  ULONG auxrs,auxrt,auxrd;  //Estas se utilizan para ver bit 64.
```

```
  rs=Campo ("RS",instruccion);  //cogemos el campo rs de la instrucci¢n
  rt=Campo ("RT",instruccion);  //cogemos el campo rt de la instrucci¢n
  rd=Campo ("RD",instruccion);  //cogemos el campo rd de la instrucci¢n

  if (Estado("UX")==0)  //Para 32 bits.
  {
    General_Exception();        //Reserved instruction exception
    gotoxy (30,23);
    printf ("Reserved Instr. exception");
  }
  else if (Estado("UX")==1)  //Para 64 bits.
  {
    regrs0=rpg[rs][0];  //El primer operando se deja como est
    regrs1=rpg[rs][1];

    regrt0=~rpg[rt][0];   //Al segundo operando se le cambia de signo.
    regrt1=~rpg[rt][1]+1;
    if (regrt1==0) regrt0++;

    regrd0=regrs0+regrt0;    //Se hace la suma de la parte alta
    regrd1=regrs1+regrt1;   //Se hace la suma de la parte baja
    aca=acarreo (regrs1,regrt1); //si hay acarreo intermedio
    if (aca==1) regrd0++;  //se le suma uno a la parte alta

    auxrs=regrs0;  //metemos en las variables aux.. el contenido de...
    auxrt=regrt0;  //las variables reg..0
    auxrd=regrd0;
    auxrs=auxrs>>31;  //Desplazamos 31 bits hacia la derecha para..
    auxrt=auxrt>>31;  //poder quedarnos con el bit 64
    auxrd=auxrd>>31;

    if ((auxrs==auxrt) && (auxrs!=auxrd)) //Vemos si se produce overflow.
    {
      General_Exception();        //Integer overflow exception
      gotoxy (30,23);
      printf ("Integer Overfl. exception");
    }
    else   //Si no se produce overflow.
    {
      rpg[rd][0]=regrd0;   //metemos el resultado en la parte alta
      rpg[rd][1]=regrd1;   //metemos el resultado en la parte baja
    }
  }
}         //fin de la instrucci¢n DSUB


void DSLL (ULONG instruccion)   //Doubleword Shift Left Logical
{
  UCHAR rt,rd,sa;    //Indican el n£mero de registro y desplazamiento
  ULONG auxi;     //Variable auxiliar.

  rt=Campo ("RT",instruccion);  //Cogemos el campo rt de la instrucci¢n.
  rd=Campo ("RD",instruccion);  //Cogemos el campo rd de la instrucci¢n.
```

```
  auxi = instruccion & 0x00007C0;  //Cogemos el campo desplazamiento.
  auxi = auxi >> 6;    //Lo desplazamos seis bits a la derecha.
  sa = (UCHAR) auxi;  //Y lo convertimos a unsigned char.

  if (Estado("UX")==0)  //Para 32 bits.
  {
    General_Exception();        //Reserved instruction exception
    gotoxy (30,23);
    printf ("Reserved Instr. exception");  //Ponemos nombre de excepci¢n.
  }
  else if (Estado("UX")==1)  //Para 64 bits.
  {
    rpg[rd][0]=rpg[rt][0]<<sa;  //Desplazamos la parte alta sa bits a la izq.
    auxi = rpg[rt][1];        //En auxi metemos la parte baja.
    auxi = auxi >> (32-sa);   //Lo desplazamos (32-sa) bits a la derecha.
    rpg[rd][0]=rpg[rd][0] | auxi;  //Hacemos Or logica de parte alta con
auxi.
    rpg[rd][1] = rpg[rt][1] << sa; //Desplazamos parte baja sa bits hacia
izq.
  }
}         //Fin de la instrucci¢n DSLL

void DSLLV (ULONG instruccion)
/*Si se ejecuta en 32 bits se produce una reserved instruction excepction.
En 64 bits puede desplazar de cero a 64 bits hacia la izquierda*/
{
  UCHAR rt,rd,rs,s; //declaramos las variables de los reg. y desplazamiento
  ULONG auxi; //auxi es variable auxiliar.

  rd=Campo ("RD",instruccion);  //cogemos el campo rd de la instrucci¢n
  rs=Campo ("RS",instruccion);   //cogemos el campo rs de la instrucci¢n
  rt=Campo ("RT",instruccion);  //cogemos el campo rt de la instrucci¢n
  auxi = rpg[rs][1] & 0x0000003F;  //en auxi metemos el desplazamiento de
bits.
  s=(UCHAR) auxi;  //metemos el desplazamiento en la variable s

  if (Estado("UX")==0)  //Para 32 bits.
  {
    General_Exception();        //Reserved instruction exception
    gotoxy (30,23);
    printf ("Reserved Instr. exception");
  }
  else if (Estado("UX")==1)  //Para 64 bits.
  {
    if (s>32)   //si el desplazamiento es mayor de 32 bits
    {
      rpg[rd][0]=rpg[rt][1];  //metemos la parte baja de rt en alta de rd
      rpg[rd][1]=0x00000000;  //ponemos ceros en la baja de rd
      rpg[rd][0]=rpg[rd][0]<<(s-32);  //desplazamos hacia la izq. la alta
    }
    else  //si el desplazamiento no es mayor de 32 bits.
    {
      rpg[rd][0]=rpg[rt][0]<<s; //metemos en rd la parte baja de rt
desplazada.
      auxi = rpg[rt][1];  //en auxi mentemos la parte baja de rt
```

```
      auxi = auxi >> (32-s);   //lo desplazamos hacia la derecha 32-s bits
      rpg[rd][0]=rpg[rd][0] | auxi;  //hacemos la or de parte alta y auxi
      rpg[rd][1] = rpg[rt][1] << s;  //desplazamos hacia la izq. la parte
baja.
    }
  }
}   //Fin de la instrucci¢n DSLLV

void DSLL32 (ULONG instruccion)  //Doubleword Shift Left Logical+32
/*El contenido del registro rt se desplaza 32+sa bits, insertando ceros en
los bits de orden bajo. La ejecuci¢n de esta instrucci¢n en modo 32 bits
causa una reserved instruction exception*/
{
  UCHAR rt,rd,sa;  //Para el n£mero de los registros y desplazamiento
  ULONG auxi;  //Variable auxiliar

  rt=Campo ("RT",instruccion);  //Cogemos el campo rt de la instrucci¢n
  rd=Campo ("RD",instruccion);  //Cogemos el campo rd de la instrucci¢n
  auxi = instruccion & 0x00007C0;  //Cogemos el campo sa de la instrucci¢n
  auxi = auxi >> 6;  //Lo desplazamos 6 bits hacia la derecha.
  sa = (UCHAR) auxi;  //Lo convertimos a char.

  sa = sa | 0x20;  //sumamos a sa 32.

  if (Estado("UX")==0)  //Para 32 bits.
  {
    General_Exception();        //Reserved instruction exception
    gotoxy (30,23);
    printf ("Reserved Instr. exception");
  }
  else if (Estado("UX")==1)  //Para 64 bits.
  {
    if (sa>32)   //Si sa es mayor de 32
    {
      rpg[rd][0]=rpg[rt][1]; //En la parte alta del registro se mete la baja
      rpg[rd][1]=0x00000000;  //En la parte baja del registro se pone a cero
      rpg[rd][0]=rpg[rd][0]<<(sa-32);  //La parte alta se despl. sa-32 bits
    }
    else     //Si sa no es mayor de 32
    {
      rpg[rd][0]=rpg[rt][0]<<sa;  //La parte alta se desplaza sa bits izq.
      auxi = rpg[rt][1];  //en auxi metemos la parte baja del reg. rt
      auxi = auxi >> (32-sa);  //auxi lo desplazamos 32-sa bits derecha.
      rpg[rd][0]=rpg[rd][0] | auxi;  //Hacemos la or de auxi y parte alta
      rpg[rd][1] = rpg[rt][1] << sa;  //Desplazamos la parte baja sa bits
    }
  }
}   //fin de la instrucci¢n DSLL32

void DSRA (ULONG instruccion)  //Doubleword Shift Right Arithmetic
/*El contenido del registro rt se desplaza a la derecha sa bits, extendiendo
el bit de mayor peso. el resultado se guarda en el registro rd*/
{
  UCHAR rt,rd,sa;  //Para n£mero de registro y desplazamiento
  ULONG auxi,auxi2;  //Variables auxiliares.
```

```
  rt = Campo ("RT",instruccion);  //Cogemos el campo rt de la instrucci¢n
  rd = Campo ("RD",instruccion);  //Cogemos el campo rd de la instrucci¢n
  auxi = instruccion & 0x00007C0;  //Cogemos el campo sa de la instrucci¢n
  auxi = auxi >> 6;  //Lo desplazamos 6 bits hacia la derecha.
  sa = (UCHAR) auxi;  //Lo convertimos al tipo unsigned char.

  if (Estado("UX")==0)  //Para 32 bits.
  {
    General_Exception();        //Reserved instruction exception
    gotoxy (30,23);
    printf ("Reserved Instr. exception");
  }
  else if (Estado("UX")==1) //Para 64 bits.
  {
    auxi=rpg[rt][0];   //En auxi metemos la parte alta del reg. rt.
    auxi = auxi>>31;  //Lo desplazamos hacia la derecha 31 bits
    auxi2=rpg[rt][0];  //En auxi2 metemos la parte alta del reg. rt
    auxi2=auxi2<<(32-sa);  //La desplazamos hacia la izq. 32-sa bits.
    if (auxi!=0)  //Si auxi es distinto de cero
    {
      auxi=0xFFFFFFFF;  //En auxi metemos todo unos
      auxi=auxi>>(32-sa);  //corremos hacia la derecha 32-sa bits.
      auxi=auxi<<(32-sa);  //Tenemos la extension de signo
    }
    else if (auxi==0)  //Si auxi es igual a cero
    {
      auxi=0x00000000;  //ponemos todo a ceros
    }
    rpg[rd][0]=rpg[rt][0]>>sa;  //Desplazamos la parte alta sa bits drcha.
    rpg[rd][1]=rpg[rt][1]>>sa;  //Desplazamos la parte baja sa bits drcha.
    rpg[rd][0]=rpg[rd][0] | auxi;  //Hacemos parte alta or auxi
    rpg[rd][1]=rpg[rd][1] | auxi2;  //Hacemos parte baja or auxi2
  }
}    //fin de la instrucci¢n DSRA

void DSRAV (ULONG instruccion)
/*Si se ejecuta en 32 bits se produce una reserved instruction excepction.
En 64 bits puede desplazar de cero a 64 bits hacia la derecha haciendo una
extensi¢n del bit 64*/
{
  UCHAR rt,rd,rs,sa;  //Para coger los registros y desplazamiento de bits.
  ULONG auxi,auxi2;  //variables auxiliares.

  rs = Campo ("RS",instruccion);  //cogemos el campo rs de la instruccion.
  rt = Campo ("RT",instruccion);  //cogemos el campo rt de la instrucci¢n.
  rd = Campo ("RD",instruccion);  //cogemos el campo rd de la instrucci¢n.
  auxi = rpg[rs][1] & 0x0000003F;  //en auxi metemos el desplazamiento de
bits.
  sa = (UCHAR) auxi;  //lo guardamos en la variable sa.

  if (Estado("UX")==0) //Para 32 bits.
  {
    General_Exception();        //Reserved instruction exception
    gotoxy (30,23);
```

```
    printf ("Reserved Instr. exception");
  }
  else if (Estado("UX")==1)  //Para 64 bits.
  {
    auxi=rpg[rt][0];   //Metemos en auxi el dato de mayor peso del reg. rt
    auxi = auxi>>31;  //lo desplazamos hacia la derecha para ver bit de
signo.
    if (sa>32)  //si sa es mayor de 32
    {
      if (auxi!=0)  //si auxi es distinto de cero
      {
      rpg[rd][1]=rpg[rt][0];  //en parte baja de rd metemos parte alta de rt
      rpg[rd][1]=rpg[rd][1]>>(sa-32); //desplazamos a la drcha sa-32 bits.
      rpg[rd][0]=0xFFFFFFFF;  //en parte alta de rd extendemos signo.
      auxi=0xFFFFFFFF<<(64-sa);  //desplazamos izq. 64-sa bits.
      rpg[rd][1]=rpg[rd][1] | auxi;  //hacemos la or de rd y auxi
      }
      else if (auxi==0)  //si auxi es igual a cero
      {
      rpg[rd][1]=rpg[rt][0]; //en parte baja de rd metemos parte alta de rt.
      rpg[rd][1]=rpg[rd][1]>>(sa-32);  //desplazamos a la drcha sa-32 bits.
      rpg[rd][0]=0x00000000;  //la parte alta la ponemos a cero
      }
    }
    else  //si sa no es mayor de 32
    {
      auxi2=rpg[rt][0];  //en auxi2 metemos el registro rt (parte alta).
      auxi2=auxi2<<(32-sa); //lo desplazamos hacia la izq. 32-sa bits.
      if (auxi!=0)  //Si auxi es distinto de cero
      {
      auxi=0xFFFFFFFF;  //en auxi metemos todo a unos
      auxi=auxi>>(32-sa);  //lo desplazamos hacia la derecha 32-sa bits.
      auxi=auxi<<(32-sa);  //Tenemos la extension de signo
      }
      else if (auxi==0)  //si auxi es igual a cero
      {
      auxi=0x00000000;   //en auxi metemos todo a ceros.
      }
      rpg[rd][0]=rpg[rt][0]>>sa; //metemos parte alta de rt despl drch sa
bits
      rpg[rd][1]=rpg[rt][1]>>sa; //metemos parte baja de rt despl drch sa
bits
      rpg[rd][0]=rpg[rd][0] | auxi; //hacemos or con auxi
      rpg[rd][1]=rpg[rd][1] | auxi2;  //hacemos la or con auxi2
    }
  }
}      //fin de la instrucci¢n DSRAV

void DSRA32 (ULONG instruccion) //Doubleword Shift Right Arithmetic+32
/*El contenido del registro rt se desplaza a la derecha 32+sa bits
extendiendo el bit de mayor peso. El resultado se guarda en el registro rd*/
{
  UCHAR rt,rd,sa;  //Para el n£mero de registro y desplazamiento de bits.
  ULONG auxi,auxi2;  //Variables auxiliares.
```

```
  rt = Campo ("RT",instruccion);  //Cogemos el campo rt de la instrucci¢n
  rd = Campo ("RD",instruccion);  //Cogemos el campo rd de la instrucci¢n
  auxi = instruccion & 0x00007C0;  //Cogemos el campo sa de la instrucci¢n
  auxi = auxi >> 6;  //Desplazamos auxi 6 bits hacia la derecha.
  sa = (UCHAR) auxi;  //Lo convertimos a tipo unsigned char.

  sa = sa | 0x20;  //sumamos a sa 32.

  if (Estado("UX")==0)  //Para 32 bits.
  {
    General_Exception();        //Reserved instruction exception
    gotoxy (30,23);
    printf ("Reserved Instr. exception");
  }
  else if (Estado("UX")==1)  //Para 64 bits.
  {
    auxi=rpg[rt][0];  //En auxi metemos la parte alta del registro rt.
    auxi = auxi>>31;  //Lo desplazamos 31 bits hacia la derecha.
    if (sa>32)   //Si sa es mayor de 32 bits.
    {
      if (auxi!=0)   //Si auxi es distinto de cero.
      {
      rpg[rd][1]=rpg[rt][0];  //En la parte baja metemos la parte alta
      rpg[rd][1]=rpg[rd][1]>>(sa-32);  //Desplazamos a la drcha sa-32 bits.
      rpg[rd][0]=0xFFFFFFFF;  //En la parte alta metemos todo a unos
      auxi=0xFFFFFFFF<<(64-sa);  //en auxi metemos todo a unos << 64-sa bits
      rpg[rd][1]=rpg[rd][1] | auxi;  //hacemos la or de parte baja y auxi
      }
      else if (auxi==0)  //Si auxi es igual a cero
      {
      rpg[rd][1]=rpg[rt][0];  //En parte baja metemos parte alta
      rpg[rd][1]=rpg[rd][1]>>(sa-32);  //Desplazamos sa-32 bits.
      rpg[rd][0]=0x00000000;  //En parte alta metemos todo a ceros.
      }
    }
    else  //Si sa no es mayor de 32 bits.
    {
      auxi2=rpg[rt][0];  //En auxi2 metemos la parte alta del reg. rt
      auxi2=auxi2<<(32-sa);  //Desplazamos auxi2 32-sa bits.
      if (auxi!=0)    //Si auxi es distinto de cero
      {
      auxi=0xFFFFFFFF;  //Lo ponemos todo a unos
      auxi=auxi>>(32-sa);  //Desplazamos hacia derecha 32-sa bits.
      auxi=auxi<<(32-sa);  //Tenemos la extension de signo
      }
      else if (auxi==0)  //Si auxi es igual a cero
      {
      auxi=0x00000000;   //Lo ponemos todo a ceros.
      }
      rpg[rd][0]=rpg[rt][0]>>sa;  //En parte alta desplazamos derecha sa
bits.
      rpg[rd][1]=rpg[rt][1]>>sa;  //En parte baja desplazamos derecha sa
bits.
      rpg[rd][0]=rpg[rd][0] | auxi;  //Hacemos la or de parte alta y auxi
      rpg[rd][1]=rpg[rd][1] | auxi2;  //Hacemos la or de parte baja y auxi2
```

```
    }
  }
}   //fin de la instrucci¢n DSRA32

void DSRL (ULONG instruccion)   //Doubleword Shift Right Logical
/*El contenido del registro rt se desplaza sa bits, insertando ceros en
los bits de mayhor peso. El resultado se guarda en el registro rd.*/
{
  UCHAR rt,rd,sa;  //Para el n£mero de registro y desplazamiento
  ULONG auxi;  //Variable auxiliar.

  rt = Campo ("RT",instruccion);  //Cogemos el campo rt de la instrucci¢n
  rd = Campo ("RD",instruccion);  //Cogemos el campo rd de la instrucci¢n
  auxi = instruccion & 0x00007C0;  //Cogemos el campo sa de la instrucci¢n
  auxi = auxi >> 6;  //Lo desplazamos 6 bits hacia la derecha.
  sa = (UCHAR) auxi;  //Lo pasamos a tipo unsigned char.

  if (Estado("UX")==0)  //Para 32 bits.
  {
    General_Exception();        //Reserved instruction exception
    gotoxy (30,23);
    printf ("Reserved Instr. exception");
  }
  else if (Estado("UX")==1)  //Para 64 bits.
  {
    auxi=rpg[rt][0]<<(32-sa);   //metemos parte alta despla. izq. 32-sa bits.
    rpg[rd][0]=rpg[rt][0]>>sa;  //Desplazamos sa bits derecha parte alta
    rpg[rd][1]=rpg[rt][1]>>sa;  //Desplazamos sa bits derecha parte baja
    rpg[rd][1]=rpg[rd][1] | auxi;  //Hacemos la or de parte baja y auxi.
  }
}           //fin de la instrucci¢n DSRL

void DSRLV (ULONG instruccion)
/*Si est  en modo 32 bits se produce una reserved instruction exception.
En modo 64 bits se hace un desplazamiento l¢gico hacia la derecha.*/
{
  UCHAR rt,rd,rs,sa;  //Para los n£meros de registros y desplazam. de bits.
  ULONG auxi;   //Variable auxiliar

  rt = Campo ("RT",instruccion);  //Cogemos el campo rt de la instrucci¢n
  rd = Campo ("RD",instruccion);  //Cogemos el campo rd de la instrucci¢n
  rs = Campo ("RS",instruccion);  //Cogemos el campo rs de la instrucci¢n

  auxi = rpg[rs][1] & 0x0000003F; //en auxi metemos el desplaz. de bits.
  sa = (UCHAR) auxi;  //en sa metemos el desplazamiento

  if (Estado("UX")==0) //Para 32 bits.
  {
    General_Exception();        //Reserved instruction exception
    gotoxy (30,23);
    printf ("Reserved Instr. exception");
  }
  else if (Estado("UX")==1)  //Para 64 bits.
  {
    if (sa>32)      //Si sa es mayor de 32
```

```
    {
      rpg[rd][0]=0;   //La parte alta del registro rd se pone a cero.
      rpg[rd][1]=rpg[rt][0]>>(sa-32);
    }
    else  //si sa no es mayor de 32 bits.
    {
      auxi=rpg[rt][0]<<(32-sa); //parte alta de rt desplazada izq. 32-sa
bits.
      rpg[rd][0]=rpg[rt][0]>>sa; //parte alta de rt se despl. sa bits derch.
      rpg[rd][1]=rpg[rt][1]>>sa; //parte baja de rt se despl. sa bits derech.
      rpg[rd][1]=rpg[rd][1] | auxi; //Se hace una or de parte baja y auxi.
    }
  }
}     //fin de la instrucci¢n DSRLV

void DSRL32 (ULONG instruccion)  //Doubleword Shift Right Logical + 32
/*El contenido del registro rt se desplaza a la derecha 32+sa bits, inser
tando ceros en los bits de orden alto. El resultado se guarda en el registro
rd.*/
{
  UCHAR rt,rd,sa;  //Para el n£mero de registro y desplazamiento.
  ULONG auxi;   //Variable auxiliar.

  rt = Campo ("RT",instruccion);  //Cogemos el campo rt de la instrucci¢n
  rd = Campo ("RD",instruccion);  //Cogemos el campo rd de la instrucci¢n
  auxi = instruccion & 0x00007C0;  //Cogemos el campo sa de la instrucci¢n
  auxi = auxi >> 6;  //Lo desplazamos 6 bits hacia la derecha.
  sa = (UCHAR) auxi;  //lo convertimos a unsigned char

  sa = sa | 0x20;  //sumamos a sa 32.

  if (Estado("UX")==0)  //Para 32 bits.
  {
    General_Exception();        //Reserved instruction exception
    gotoxy (30,23);
    printf ("Reserved Instr. exception");
  }
  else if (Estado("UX")==1)  //Para 64 bits.
  {
    if (sa>32)  //Si sa es mayor de 32
    {
      rpg[rd][0]=0;  //En la parte alta ponemos un cero
      rpg[rd][1]=rpg[rt][0]>>(sa-32);  //En parte baja ponemos parte alta
dsp.
    }
    else  //Si sa no es mayor de 32
    {
      auxi=rpg[rt][0]<<(32-sa);  //En auxi metemos parte alta desp. 32-sa
      rpg[rd][0]=rpg[rt][0]>>sa; //En parte alta metemos parte alta desp sa.
      rpg[rd][1]=rpg[rt][1]>>sa; //En parte baja metemos parte baja desp sa.
      rpg[rd][1]=rpg[rd][1] | auxi;  //Hacemos or de parte baja y auxi
    }
  }
}      //Fin de la instrucci¢n DSRL32
```

```
void SLL (ULONG instruccion)   //Shift Left Logical
/*El contenido del registro rt se desplaza hacia la izquierda sa bits,
insertando ceros en los bits de orden bajo*/
{
  UCHAR rt,rd,sa;   //Para coger el n£mero de registro y desplazamiento
  ULONG auxi;  //Variable auxiliar

  rt = Campo ("RT",instruccion); //Cogemos el campo rt de la instrucci¢n
  rd = Campo ("RD",instruccion);  //Cogemos el campo rd de la instrucci¢n
  auxi = instruccion & 0x00007C0;  //Cogemos el campo sa de la instrucci¢n
  auxi = auxi >> 6;  //Desplazamos 6 bits hacia la derecha.
  sa = (UCHAR) auxi;  //Lo convertimos a unsigned char.

  if (Estado("UX")==0)  //Para 32 bits.
    rpg[rd][1]=rpg[rt][1]<<sa;
  else if (Estado("UX")==1)  //Para 64 bits.
  {
    auxi=rpg[rt][1]<<sa;  //Metemos en auxi parte baja despl. izq. sa bits.
    auxi=auxi>>31;   //Lo desplazamos 31 bits hacia la derecha.
    if (auxi!=0)   //Si auxi es distinto de cero
      rpg[rd][0]=0xFFFFFFFF;  //Ponemos la parte alta del reg. todo a unos.
    else if (auxi==0)   //Si auxi es igual a cero
      rpg[rd][0]=0x00000000;   //Ponemos la parte alta del reg. todo a ceros.
    rpg[rd][1]=rpg[rt][1]<<sa;  //Metemos en parte baja el dato final despl.
  }
}   //Fin de la instrucci¢n SLL

void SLLV (ULONG instruccion)  //Shift Left Logical Variable
/*Esta instrucci¢n hace un desplazamiento l¢gico hacia la izquierda al
dato de 32 bits del registro rt y el resultado se guarda en el registro rd */
{
  UCHAR rt,rd,rs,sa;  //Para el n£mero de los registros y desplazamiento
  ULONG auxi;  //variable auxiliar.

  rt = Campo ("RT",instruccion);  //Cogemos el campo rt de la instrucci¢n
  rd = Campo ("RD",instruccion);  //Cogemos el campo rd de la instrucci¢n
  rs = Campo ("RS",instruccion);  //Cogemos el campo rs de la instrucci¢n
  auxi = rpg[rs][1] & 0x0000001F;
  sa = (UCHAR) auxi;             //este "sa" se refiere a "s"

  if (Estado("UX")==0)  //Para 32 bits.
    rpg[rd][1]=rpg[rt][1]<<sa;  //Se desplaza sa bits hacia la izquierda.
  else if (Estado("UX")==1)  //Para 64 bits.
  {
    auxi=rpg[rt][1]<<sa; //Desplaza sa bits hacia la izq. y se mete en auxi.
    auxi=auxi>>31;  //Mira el bit 31 del dato ya desplazado
    if (auxi!=0)  //Si auxi es distinto de cero (deber¡a ser uno)
      rpg[rd][0]=0xFFFFFFFF; //la parte alta del registro se pone toda a
unos.
    else if (auxi==0)  //Si auxi es igual a cero
      rpg[rd][0]=0x00000000;   //la parte alta del reg. se pone toda a ceros.
    rpg[rd][1]=rpg[rt][1]<<sa;  //en la parte baja se mete el dato
desplazado.
  }
}  //fin de la instrucci¢n SLLV
```

```
void SRL (ULONG instruccion)  //Shift Right Logical
/*El contenido del registro rt se desplaza a la derecha sa bits, insertando
ceros en los bits de orden alto. El resultado se guarda en el registro rd.
En 64 bits se hace una extensi¢n de signo.*/
{
  UCHAR rt,rd,sa;   //Para el n£mero de registro y desplazamiento.
  ULONG auxi;   //Variable auxiliar.

  rt = Campo ("RT",instruccion);  //Cogemos el campo rt de la instrucci¢n
  rd = Campo ("RD",instruccion);  //Cogemos el campo rd de la instrucci¢n
  auxi = instruccion & 0x00007C0;  //Cogemos el campo sa de la instrucci¢n
  auxi = auxi >> 6;  //Desplazamos 6 bits a la derecha.
  sa = (UCHAR) auxi;  //Lo convertimos a unsigned char.

  if (Estado("UX")==0)  //Para 32 bits.
    rpg[rd][1]=rpg[rt][1]>>sa;   //Se desplaza sa bits drcha parte baja
  else if (Estado("UX")==1)  //Para 64 bits.
  {
    auxi=rpg[rt][1]>>sa;   //En auxi metemos la parte baja >> sa bits
    auxi=auxi>>31;  //Se desplaza 31 bits hacia la derecha.
    if (auxi!=0)   //Si auxi es distinto de cero
      rpg[rd][0]=0xFFFFFFFF;  //Ponemos la parte alta toda a unos.
    else if (auxi==0)   //Si auxi es igual a cero
      rpg[rd][0]=0x00000000;   //Ponemos la parte alta toda a ceros.
    rpg[rd][1]=rpg[rt][1]>>sa;  //Ponemos resultado en parte baja.
  }
}    //fin de la instrucci¢n SRL

void SRLV (ULONG instruccion)   //Shift Right Logical Variable
/*El registro rt se desplaza hacia la derecha el n£mero de bits especificados
por la variable sa*/
{
  UCHAR rt,rd,rs,sa;  //Para guardar el n£mero de los reg. y el desplaz.
  ULONG auxi;  //Variable auxiliar.

  rt = Campo ("RT",instruccion);  //Cogemos el campo rt de la instrucci¢n
  rd = Campo ("RD",instruccion);  //Cogemos el campo rd de la instrucci¢n
  rs = Campo ("RS",instruccion);  //Cogemos el campo rs de la instrucci¢n
  auxi = rpg[rs][1] & 0x0000001F;  //Cogemos el desplazamiento de bits.
  sa = (UCHAR) auxi;             //este "sa" se refiere a "s"

  if (Estado("UX")==0)  //Para 32 bits.
    rpg[rd][1]=rpg[rt][1]>>sa;   //Desplazamos hacia la derecha
  else if (Estado("UX")==1)  //Para 64 bits.
  {
    auxi=rpg[rt][1]>>sa;   //Guardamos en auxi el dato desplazado
    auxi=auxi>>31;  //lo desplazamos 31 bits hacia la derecha
    if (auxi!=0)  //Si auxi es distinto de cero
      rpg[rd][0]=0xFFFFFFFF;  //ponemos la parte alta todo a unos
    else if (auxi==0)  //Si auxi es igual a cero
      rpg[rd][0]=0x00000000;  //ponemos la parte alta todo a ceros.
    rpg[rd][1]=rpg[rt][1]>>sa;  //guardamos el dato en el registro.
  }
}    //Fin de la instrucci¢n SRLV
```

```
void SRA (ULONG instruccion)  //Shift Right Arithmetic
/*El contenido del registro rt se desplaza hacia la derecha sa bits, haciendo
una extensi¢n de signo del bit de mayor peso.*/
{
  UCHAR rt,rd,sa;  //Para el n£mero de registro y desplazamiento
  ULONG auxi;   //Variable auxiliar.

  rt = Campo ("RT",instruccion);  //Cogemos el campo rt de la instrucci¢n
  rd = Campo ("RD",instruccion);  //Cogemos el campo rd de la instrucci¢n
  auxi = instruccion & 0x00007C0;  //Cogemos el campo sa de la instrucci¢n
  auxi = auxi >> 6;  //Desplazamos hacia la derecha 6 bits.
  sa = (UCHAR) auxi;   //Pasamos a UCHAR.

  if (Estado("UX")==0)  //Para 32 bits.
  {
    auxi=rpg[rt][1]>>31;  //Metemos en auxi la parte baja desplaza. 31 bits.
    if (auxi!=0)  //Si es distinto de cero
    {
      auxi=0xFFFFFFFF>>(32-sa);  //En auxi metemos todo 1 despl dercha 32-sa.
      auxi=auxi<<(32-sa);  //Desplazamos 32-sa bits hacia la izquierda.
    }
    else if (auxi==0)  //Si auxi es igual a cero
    {
      auxi=0x00000000;   //En auxi metemos un cero
    }
    rpg[rd][1]=rpg[rt][1]>>sa;    //En parte baja metemos parte baja >> sa
    rpg[rd][1]=rpg[rd][1] | auxi;  //Hacemos la or con auxi.
  }
  else if (Estado("UX")==1)  //Para 64 bits.
  {
    auxi=rpg[rt][1]>>31;  //Metemos en auxi la parte baja desplaza. 31 bits.
    if (auxi!=0)  //Si auxi es distinto de cero
    {
      rpg[rd][0]=0xFFFFFFFF;  //En parte alta metemos todo unos
      auxi=0xFFFFFFFF>>(32-sa); //En auxi metemos todo 1 despl. drcha 32-sa.
      auxi=auxi<<(32-sa);  //Desplazamos 32-sa bits hacia la izquierda.
    }
    else if (auxi==0)  //Si auxi es igual a cero
    {
      rpg[rd][0]=0x00000000;  //Ponemos la parte alta a cero
      auxi=0x00000000;   //En auxi metemos un cero
    }
    rpg[rd][1]=rpg[rt][1]>>sa;   //En parte baja metemos parte baja >> sa
    rpg[rd][1]=rpg[rd][1] | auxi;  //Hacemos la or con auxi.
  }
}    //Fin de la instrucci¢n SRA

void SRAV (ULONG instruccion)    //Shift Right Arithmetic Variable
/*Esta instrucci¢n hace un desplazamiento de bits aritm,tico hacia la derecha
y el resultado se guarda en el registro rd*/
{
  UCHAR rt,rd,rs,sa; //Para coger los registro y el desplazamiento
  ULONG auxi;  //Variable auxiliar
```

```
  rs = Campo ("RS",instruccion);  //Cogemos el campo rs de la instrucci¢n
  rt = Campo ("RT",instruccion);  //Cogemos el campo rt de la instrucci¢n
  rd = Campo ("RD",instruccion);  //Cogemos el campo rd de la instrucci¢n
  auxi = rpg[rs][1] & 0x0000001F;  //Cogemos el desplazamiento
  sa = (UCHAR) auxi;             //este "sa" se refiere a "s"

  if (Estado("UX")==0)  //Para 32 bits.
  {
    auxi=rpg[rt][1]>>31; //En auxi metemos el reg rt desplaz. a la drcha 31
bts
    if (auxi!=0)   //Si auxi es distinto de cero
    {
      auxi=0xFFFFFFFF>>(32-sa);  //En auxi metemos todo 1 despl. drch 32-sa
bit
      auxi=auxi<<(32-sa); //Desplazamos a la izq. 32-sa bits a auxi
    }
    else if (auxi==0)  //Si auxi es igual a cero
    {
      auxi=0x00000000;   //En auxi metemos un cero
    }
    rpg[rd][1]=rpg[rt][1]>>sa;  //Desplazamos sa bits a la derecha
    rpg[rd][1]=rpg[rd][1] | auxi;  //Hacemos la or del registro con auxi
  }
  else if (Estado("UX")==1)  //Para 64 bits.
  {
    auxi=rpg[rt][1]>>31; //En auxi metemos el reg rt desplaz. a la drcha 31
bts
    if (auxi!=0)   //Si auxi es distinto de cero
    {
      rpg[rd][0]=0xFFFFFFFF;  //En la parte alta se pone todo a unos
      auxi=0xFFFFFFFF>>(32-sa);  //Se desplaza a la derecha 32-sa bits
      auxi=auxi<<(32-sa);  //Se vuelve a desplazar a la izq. 32-sa bits.
    }
    else if (auxi==0)  //Si auxi es igual a cero
    {
      rpg[rd][0]=0x00000000;  //En la parte alta se pone todo a ceros
      auxi=0x00000000;  //En auxi ponemos un cero
    }
    rpg[rd][1]=rpg[rt][1]>>sa; //Desplazamos la parte baja sa bits.
    rpg[rd][1]=rpg[rd][1] | auxi;  //Hacemos la or del reg y auxi.
  }
}     //Fin de la instrucci¢n SRAV

void SLT (ULONG instruccion)  //Set On Less Than
/*Si el contenido del registro rs es menor que el contenido del registro rt
el resultado se pone a uno, en caso de que no sea menor el resultado se pone
a cero, los datos tienen que ser con signo*/
{
  UCHAR rs,rt,rd;  //Para coger el n£mero de los registros
  signed long int rssigno,rtsigno;    //signed long int
  ULONG rsnosigno,rtnosigno;

  rs = Campo ("RS",instruccion);  //Cogemos el campo rs de la instrucci¢n
  rt = Campo ("RT",instruccion);  //Cogemos el campo rt de la instrucci¢n
  rd = Campo ("RD",instruccion);  //Cogemos el campo rd de la instrucci¢n
```

```
  if (Estado("UX")==0)  //Para 32 bits.
  {
    rssigno=rpg[rs][1];  //En rssig ponemos lo que hay en el reg. rs (parte
baja)
    rtsigno=rpg[rt][1];  //En rtsig ponemos lo que hay en el reg. rt (parte
baja)
    //Hacemos esto porque los operandos tienen que ser con signo.

    if (rssigno < rtsigno) //Si rssigno es menor que rtsigno
      rpg[rd][1]=0x00000001;  //El registro rd se pone a uno
    else          //si rssigno no es menor que rtsigno
      rpg[rd][1]=0x00000000;  //El registro rd se pone a cero.
  }
  else if (Estado("UX")==1)  //Para 64 bits.
  {
    rssigno=rpg[rs][0]; //En rssig ponemos lo que hay en el reg. rs (parte
alta)
    rtsigno=rpg[rt][0]; //En rtsig ponemos lo que hay en el reg. rt (parte
alta)
    rsnosigno=rpg[rs][1]; //En rsnosigno est  la parte baja del reg rs.
    rtnosigno=rpg[rt][1]; //En rtnosigno est  la parte baja del reg rt.

    if ((rpg[rs][0]!=0) || (rpg[rt][0]!=0)) /*Si la parte alta de cualquiera
    de los dos es distinta de cero...*/
    {
      if (rpg[rs][0]!=rpg[rt][0])  //...Si las dos son distintas
      {
       if (rssigno < rtsigno)  //Si rssigno es menor que rtsigno
       {
      rpg[rd][1]=0x00000001;  //El registro rd se pone a uno.
      rpg[rd][0]=0;
       }
       else   //Si rssigno no es menor que rtsigno
       {
      rpg[rd][1]=0x00000000;  //El registro rd se pone a cero.
      rpg[rd][0]=0;
       }
      }
      else if (rpg[rs][0]==rpg[rt][0])  //Si las dos son iguales
      {
       if (rsnosigno < rtnosigno)   //Si rsnosig es menor que rtnosig (bajas)
       {
      rpg[rd][1]=0x00000001;   //El registro rd se pone a uno
      rpg[rd][0]=0;
       }
       else       //Si rsnosigno no es menor que rtnosigno
       {
      rpg[rd][1]=0x00000000;  //El registro rd se pone a cero
      rpg[rd][0]=0;
       }
      }
    }
    else   //Si las partes altas son igual a cero
    {
```

```
      if (rsnosigno < rtnosigno)  //Si rsnosigno es menor que rtnosigno
      {
      rpg[rd][1]=0x00000001;  //El registro rd se pone a uno
      rpg[rd][0]=0;
      }
      else   //Si rsnosigno no es menor que rtnosigno
      {
      rpg[rd][1]=0x00000000;   //El registro rd se pone a cero
      rpg[rd][0]=0;
      }
    }
  }
}        //fin de la instrucci¢n SLT

void SLTU (ULONG instruccion)   //Set On Less Than
/*Si el contenido del registro rs es menor que el contenido del registro rt
el resultado se pone a uno, en caso de que no sea menor el resultado se pone
a cero, los datos tienen que ser sin signo*/
{
  UCHAR rs,rt,rd;   //Para coger el n£mero del registro
  ULONG regrs,regrt;    //unsigned long int

  rs = Campo ("RS",instruccion);  //Cogemos el campo rs de la instrucci¢n
  rt = Campo ("RT",instruccion);  //Cogemos el campo rt de la instrucci¢n
  rd = Campo ("RD",instruccion);  //Cogemos el campo rd de la instrucci¢n

  regrs=rpg[rs][1];  //En regrs metemos el dato del registro rs (parte baja)
  regrt=rpg[rt][1];  //En regrt metemos el dato del registro rt (parte baja)

  if (Estado("UX")==0)  //Para 32 bits.
  {
    if (regrs < regrt)  //Si regrs es menor que regrt
      rpg[rd][1]=0x00000001;  //Ponemos el registro rd a uno.
    else   //Si regrs no es menor que regrt
      rpg[rd][1]=0x00000000;   //Ponemos el registro rd a cero.
  }
  else if (Estado("UX")==1)  //Para 64 bits.
  {
    if ((rpg[rs][0]!=0) || (rpg[rt][0]!=0))  //Si las partes altas son !=0
    {
      if (rpg[rs][0]!=rpg[rt][0])  //Si son distintas
      {
       regrs=rpg[rs][0];   //guardamos la parte alta en regrs
       regrt=rpg[rt][0];  //guardamos la parte alta en regrt
       if (regrs < regrt)  //Si regrs es menor que regrt
       {
      rpg[rd][1]=0x00000001;  //El registro rd se pone a uno
      rpg[rd][0]=0;
       }
       else   //Si regrs no es menor que regrt
       {
      rpg[rd][1]=0x00000000;  //El registro rd se pone a cero
      rpg[rd][0]=0;
       }
      }
```

```
      else if (rpg[rs][0]==rpg[rt][0])  //Si son iguales
      {
       if (regrs < regrt)  //Comparamos las partes bajas de cada registro
       {
      rpg[rd][1]=0x00000001;   //El registro se pone a uno
      rpg[rd][0]=0;
       }
       else    //Si regrs no es menor que regrt
       {
      rpg[rd][1]=0x00000000;   //El registro rd se pone a cero.
      rpg[rd][0]=0;
       }
      }
    }
    else   //Si las partes altas son igual a cero
    {
      if (regrs < regrt)   //Comparamos las partes bajas
      {
      rpg[rd][1]=0x00000001;   //Se pone el registro rd a uno
      rpg[rd][0]=0;
      }
      else   //si regrs no es menor que regrt
      {
      rpg[rd][1]=0x00000000;   //Se pone el registro rd a cero.
      rpg[rd][0]=0;
      }
    }
  }
}      //Fin de la instrucci¢n SLTU

void SLTI (ULONG instruccion)   //Set on Less Than Immediate
/*El dato de 16 bits se extende y se compara con el contenido del registro
rs. Las dos cantidades tienen que ser enteros con signo. Si rs es menor
que el dato inmediato extendido, el resultado se pone a uno y si no es as¡ se
pone a cero. El resultado se guarda en el registro rt.*/
{
  UCHAR rs,rt;   //Para guardar el n£mero de registro.
  signed long int rssigno,inmesigno,inme0,inmediato;    //signed long int
  ULONG rsnosigno,inmenosigno;   //Datos sin signo.
  ULONG auxi;  //variable auxiliar
  //inme0 son los 32 bits de mayor peso del dato inmediato en 64 bits.

  rs = Campo ("RS",instruccion);  //Cogemos el campo rs de la instrucci¢n
  rt = Campo ("RT",instruccion);  //Cogemos el campo rt de la instrucci¢n
  inmediato = instruccion & 0x0000FFFF;  //Cogemos el dato inmediato de
instr.
  auxi=inmediato;  //En auxi metemos el dato inmediato
  auxi=auxi>>15;   //Desplazamos hacia la derecha 15 bits.

  if (auxi!=0)  //Si auxi es distinto de cero
  {
    inmediato=inmediato | 0xFFFF0000;  //Extendemos el dato inmediato con 1
    inme0=0xFFFFFFFF;  //La parte alta del dato inmediato se pone a unos.
  }
  else if (auxi==0)  //Si auxi es igual a cero
```

```
    inme0=0x00000000;   //La parte alta del dato inmediato se pone a ceros.

  if (Estado("UX")==0)  //Para 32 bits.
  {
    rssigno=rpg[rs][1];  //En rssigno metemos la parte baja del registro rs.
    inmesigno=inmediato;  //En inmesigno metemos parte baja del dato
inmediato
    if (rssigno < inmesigno)   //Si rssigno es menor que inmesigno
      rpg[rt][1]=0x00000001;  //Ponemos el registro rt a uno.  (parte baja)
    else    //Si rssigno no es menor que inmesigno
      rpg[rt][1]=0x00000000;  //Ponemos el registro rt a cero. (parte baja)
  }
  else if (Estado("UX")==1)  //Para 64 bits.
  {
    rssigno=rpg[rs][0];  //En rssigno metemos la parte alta del reg. rs.
    inmesigno=inme0;  //En inmesigno metemos la parte alta del dato inmediato
    rsnosigno=rpg[rs][1]; //En rsnosigno metemos la parte baja del reg. rs.
    inmenosigno=inmediato; //En inmenosigno metemos la parte baja de
inmediato
    if ((rpg[rs][0]!=0) || (inme0!=0)) //Si la parte alta es distinta de cero
    {
      if (rpg[rs][0]!=inme0)   //y los dos dos datos son distintos
      {
       if (rssigno < inmesigno)  //Comparamos las partes altas (con signo)
       {
       rpg[rt][1]=0x00000001;  //ponemos parte baja del reg rt a uno
       rpg[rt][0]=0;   //ponemos la parte alta del reg. rt a cero.
       }
       else   //Si rssingo no es menor que inmesigno
       {
       rpg[rt][1]=0x00000000;  //ponemos la parte baja del reg. rt a cero
       rpg[rt][0]=0;    //ponemos la parte alta del reg. rt a cero.
       }
      }
      else if (rpg[rs][0]==inme0)  //Si las dos partes altas son iguales
      {
       if (rsnosigno < inmenosigno)  //Comparamos las partes bajas (sin
signo)
       {
       rpg[rt][1]=0x00000001;  //Ponemos parte baja del registro rt a uno
       rpg[rt][0]=0;   //Ponemos parte alta del regis. rt a cero.
       }
       else   //Si rsnosigon no es menor que inmenosigno
       {
       rpg[rt][1]=0x00000000;  //Ponemos parte baja del registro rt a cero
       rpg[rt][0]=0;  //Ponemos parte alta del registro rt a cero.
       }
      }
    }
    else   //Si la parte alta de los dos datos es igual a cero
    {
      if (rsnosigno < inmenosigno)   //Si rsnosigno es menor que inmenosigno
      {
      rpg[rt][1]=0x00000001;  //Ponemos parte baja del registro rt a uno
      rpg[rt][0]=0;  //Ponemos parte alta del registro rt a cero.
```

```
      }
      else  //Si rsnosigno no es menor que inmenosigno
      {
      rpg[rt][1]=0x00000000;  //Ponemos parte baja del registro rt a cero.
      rpg[rt][0]=0;  //Ponemos parte alta del registro rt a cero.
      }
    }
  }
}    //Fin de la instrucci¢n SLTI

void SLTIU (ULONG instruccion)  //Set on Less Than Immediate Unsigned
/*Se extiende el dato inmediato de 16 bits y se compara con el contenido
del registro rs. Las dos cantidades son sin signo. Si el registro rs es
menor que el dato inmediato, el resultado se pone a uno. y en caso contrario
el resultado que se guarda en el registro rt se pone a cero.*/
{
  UCHAR rs,rt;  //Para guradar el n£mero de los registros.
  ULONG regrs,inmediato,inme0,auxi;    //unsigned long int
  //inme0 son los 32 bits de mayor peso del dato inmediato en 64 bits.

  rs = Campo ("RS",instruccion);   //Cogemos el campo rs de la instrucci¢n
  rt = Campo ("RT",instruccion);   //Cogemos el campo rt de la instrucci¢n
  inmediato = instruccion & 0x0000FFFF;  //Cogemos el dato inmediato de
instr.
  auxi=inmediato;   //En auxi metemos el dato inmediato
  auxi=auxi>>15;   //Lo desplazamos 15 bits hacia la derecha.

  regrs=rpg[rs][1];   //En regrs metemos el registro rs. (parte baja)

  if (auxi!=0)   //Si auxi es distinto de cero
  {
    inmediato=inmediato | 0xFFFF0000;  //Extendemos el dato inmediato
    inme0=0xFFFFFFFF;   //La parte alta del dato inmediato se pone a unos.
  }
  else if (auxi==0)     //Si auxi es igual a cero
    inme0=0x00000000;  //La parte alta del dato inmediato se pone a ceros.

  if (Estado("UX")==0)  //Para 32 bits.
  {
    if (regrs < inmediato)   //Si regrs es menor que inmediato
      rpg[rt][1]=0x00000001;  //La parte baja del reg. rt se pone a uno.
    else
      rpg[rt][1]=0x00000000;   //La parte baja del reg. rt se pone a cero.
  }
  else if (Estado("UX")==1)  //Para 64 bits.
  {
    if ((rpg[rs][0]!=0) || (inme0!=0))  //Si la parte alta es distinta de
cero
    {
      if (rpg[rs][0]!=inme0)  //y son distintas entre ellas.
      {
       regrs=rpg[rs][0];   //en regrs metemos la parte alta del reg. rs.
       if (regrs < inme0)  //Si regrs es menor que inme0
       {
       rpg[rt][1]=0x00000001;  //La parte baja del reg. rt se pone a uno.
```

```
       rpg[rt][0]=0;  //La parte alta del reg. rt se pone a cero.
       }
       else   //Si regrs no es menor que inme0
       {
       rpg[rt][1]=0x00000000;  //La parte baja del reg. rt se pone a cero
       rpg[rt][0]=0;   //La parte alta del reg. rt se pone a cero.
       }
      }
      else if (rpg[rs][0]==inme0)  //Si son iguales entre ellas.
      {
       if (regrs < inmediato)  //Comparamos las partes bajas
       {
       rpg[rt][1]=0x00000001;  //Se pone a uno la parte baja de rt.
       rpg[rt][0]=0;  //Se pone a cero la parte alta de rt.
       }
       else   //Si regrs no es menor que inmediato
       {
       rpg[rt][1]=0x00000000;  //Se pone a cero la parte baja de rt.
       rpg[rt][0]=0;   //Se pone a cero la parte alta de rt.
       }
      }
    }
    else   //Si la parte alta de los dos datos es cero
    {
      if (regrs < inmediato)  //Si regrs es menor que inmediato
      {
      rpg[rt][1]=0x00000001;  //La parte baja del reg. rt se pone a uno
      rpg[rt][0]=0;  //La parte alta del reg. rt se pone a cero.
      }
      else   //Si regrs no es menor que inmediato
      {
      rpg[rt][1]=0x00000000;  //La parte baja del reg. rt se pone a cero.
      rpg[rt][0]=0;  //La parte alta del reg. rt se pone a cero.
      }
    }
  }
}    //Fin de la instrucci¢n SLTIU

void BEQ (ULONG instruccion)   //Branch On Equal
/*Se comparan los contenidos de los registros rs y rt. Si los dos registros
son iguales, el programa hace un salto relativo marcado por el campo offset*/
{
  UCHAR rs,rt;  //Para coger el n£mero de los registros
  signed long int offset;  //El offset para que se desplace
  ULONG auxi;   //Variable auxiliar

  rs = Campo ("RS",instruccion);  //Cogemos el campo rs de la instrucci¢n
  rt = Campo ("RT",instruccion);  //Cogemos el campo rt de la instrucci¢n
  offset = instruccion & 0x0000FFFF;  //Cogemos el campo offset de instr.

  offset = offset << 2;  //Lo desplazamos dos bits hacia la izquierda.
  auxi=offset;  //En auxi metemos offset
  auxi=auxi>>17;      //Desplazamos auxi 17 bits hacia la derecha.

  if (auxi!=0)   //Si auxi es distinto de cero
```

```
    offset=offset | 0xFFFC0000;  //Hacemos la extensi¢n de signo.

  if (Estado("UX")==0)  //Para 32 bits.
  {
    if (rpg[rs][1]==rpg[rt][1])  //Si la parte baja de los dos reg. es igual
       rpg[34][1]=rpg[34][1]+offset;  //El contador de programa salta
  }
  else if (Estado("UX")==1)   //Para 64 bits.
  {
    if ((rpg[rs][0]==rpg[rt][0]) && (rpg[rs][1]==rpg[rt][1])) //Si son
iguales
       rpg[34][1]=rpg[34][1]+offset; //El contador de programa salta.
  }
}    //Fin de la instrucci¢n BEQ.

void BEQL (ULONG instruccion)  //Branch On Equal Likely
/*Se comparan los contenidos de los registros rs y rt. Si los dos registros
son iguales, el programa hace un salto relativo marcado por el campo offset.
Si la condici¢n no se cumple la instrucci¢n a continuaci¢n de esta se anula
(no se ejecuta)*/
{
  UCHAR rs,rt;  //Para coger el n£mero de los registros
  signed long int offset;  //Para el offset
  ULONG auxi;   //Variable auxiliar

  rs = Campo ("RS",instruccion);  //Cogemos el campo rs de la instrucci¢n
  rt = Campo ("RT",instruccion);  //Cogemos el campo rt de la instrucci¢n
  offset = instruccion & 0x0000FFFF;  //Cogemos el offset de la instrucci¢n

  offset = offset << 2;  //Lo desplazamos dos bits hacia la izquierda
  auxi=offset;  //en auxi metemos offset
  auxi=auxi>>17;   //Lo desplazamos 17 bits hacia la derecha.

  if (auxi!=0)   //Si auxi es distinto de cero
    offset=offset | 0xFFFC0000;   //Hacemos extensi¢n de signo con unos

  if (Estado("UX")==0)  //Para 32 bits.
  {
    if (rpg[rs][1]==rpg[rt][1])  //Si la parte baja de los registros es igual
       rpg[34][1]=rpg[34][1]+offset;  //Se suma el offset al CP
    else   //Si la parte baja de los registros es distinta
       rpg[34][1]=rpg[34][1]+4;   //NullifyCurrentInstruction (pasa de la
sig)
  }
  else if (Estado("UX")==1)   //Para 64 bits.
  {
    if ((rpg[rs][0]==rpg[rt][0]) && (rpg[rs][1]==rpg[rt][1])) //Si son
iguales
       rpg[34][1]=rpg[34][1]+offset; //Se suma el offset al CP
    else
       rpg[34][1]=rpg[34][1]+4;   //NullifyCurrentInstruction (pasa de la
sig)
  }
}       //Fin de la instrucci¢n BEQL
```

```
void BNE (ULONG instruccion)  //Branch If Not Equal
/*Se comparan los contenidos de los registros rs y rt. Si los dos registros
son distintos, el programa hace un salto relativo marcado por el campo
offset*/
{
  UCHAR rs,rt; //Para el n£mero de los registros
  signed long int offset;   //Para el offset
  ULONG auxi;    //Variable auxiliar

  rs = Campo ("RS",instruccion);  //Cogemos el campo rs de la instrucci¢n
  rt = Campo ("RT",instruccion);  //Cogemos el campo rt de la instrucci¢n
  offset = instruccion & 0x0000FFFF;  //Cogemos el campo offset de la instr.

  offset = offset << 2;  //Desplazamos hacia la izquierda dos bits
  auxi=offset;   //En auxi metemos offset
  auxi=auxi>>17;   //Lo desplazamos 17 bits hacia la derecha.

  if (auxi!=0)   //Si auxi es distinto de cero
    offset=offset | 0xFFFC0000;   //Se hace una extensi¢n de signo con unos

  if (Estado("UX")==0)  //Para 32 bits.
  {
    if (rpg[rs][1]!=rpg[rt][1])  //Si la parte baja de los regs. es distinta
       rpg[34][1]=rpg[34][1]+offset;  //Al contador CP se le suma el offset
  }
  else if (Estado("UX")==1)   //Para 64 bits.
  {
    if ((rpg[rs][0]!=rpg[rt][0]) || (rpg[rs][1]!=rpg[rt][1])) //Si son
distintos
       rpg[34][1]=rpg[34][1]+offset;  //Al contador CP se le suma el offset
  }
}    //Fin de la instrucci¢n BNE

void BNEL (ULONG instruccion)  //Branch if Not Equal Likely
/*Se comparan los contenidos de los registros rs y rt. Si los dos registros
son distintos, el programa hace un salto relativo marcado por el campo
offset.
Si la condici¢n no se cumple la instrucci¢n a continuaci¢n de esta se anula
(no se ejecuta)*/
{
  UCHAR rs,rt; //Para el n£mero de los registros
  signed long int offset;  //Para el desplazamiento
  ULONG auxi;   //Variable auxiliar

  rs = Campo ("RS",instruccion);  //Cogemos el campo rs de la instrucci¢n
  rt = Campo ("RT",instruccion);  //Cogemos el campo rt de la instrucci¢n
  offset = instruccion & 0x0000FFFF;  //Cogemos el campo offset de la instru.

  offset = offset << 2;  //Desplazamos dos bits hacia la izquierda.
  auxi=offset;   //En auxi metemos offset
  auxi=auxi>>17;   //Lo desplazamos 17 bits hacia la derecha.

  if (auxi!=0)    //Si auxi es distinto de cero
    offset=offset | 0xFFFC0000;  //Hacemos la extensi¢n de signo con unos.
```

```
  if (Estado("UX")==0)  //Para 32 bits.
  {
    if (rpg[rs][1]!=rpg[rt][1])  //Si la parte baja de los registros es
distinta
       rpg[34][1]=rpg[34][1]+offset; //Al CP se le suma el offset
    else   //Si la parte baja de los registros no es distinta.
       rpg[34][1]=rpg[34][1]+4;          //NullifyCurrentInstruction
  }
  else if (Estado("UX")==1)   //Para 64 bits.
  {
    if ((rpg[rs][0]!=rpg[rt][0]) || (rpg[rs][1]!=rpg[rt][1]))//Si son
distintos
       rpg[34][1]=rpg[34][1]+offset;  //Al CP se le suma el offset
    else
       rpg[34][1]=rpg[34][1]+4; //NullifyCurrentInstruction (se anula la
sig.)
  }
}      //Fin de la instrucci¢n BNEL

void BGEZ (ULONG instruccion)  //Branch On Greater Than Or Equal to Zero
{
  UCHAR rs;  //Para coger el n£mero de registro
  signed long int offset;  //Para el desplazamiento
  ULONG auxi,condicion;  //Variable auxiliar y condici¢n

  rs = Campo ("RS",instruccion);  //Cogemos el campo rs de la instrucci¢n
  offset = instruccion & 0x0000FFFF;  //Cogemos el campo offset de la instr.

  offset = offset << 2;  //Desplazamos dos bits hacia la izquierda
  auxi=offset;  //En auxi metemos offset
  auxi=auxi>>17;  //Desplazamos 17 bits hacia la derecha.

  if (auxi!=0)  //Si auxi es distinto de cero
    offset=offset | 0xFFFC0000;  //Se extiende el signo

  if (Estado("UX")==0)  //Para 32 bits.
  {
    condicion=rpg[rs][1] >> 31;  //miramos el bit 32
    if (condicion==0)  //Si el bit 31 del registro es cero
       rpg[34][1]=rpg[34][1]+offset;  //Sumamos el desplazam. al CP
  }
  else if (Estado("UX")==1)   //Para 64 bits.
  {
    condicion=rpg[rs][0] >> 31;     //miramos el bit 64
    if (condicion==0)  //Si es cero es que es mayor o igual a cero
       rpg[34][1]=rpg[34][1]+offset;  //Sumamos el desplazam. al CP
  }
}   //Fin de la instrucci¢n BGEZ

void BGTZ (ULONG instruccion)  //Branch On Greater Than Zero
/*Si el registro rs es mayor que cero, el programa hace un salto relativo
marcado por el campo offset*/
{
  UCHAR rs;  //Para coger el n£mero del registro rs
  signed long int offset;  //Para coger el desplazamiento
```

```
  ULONG condicion,auxi;  //Para la condici¢n y variable auxiliar.

  rs = Campo ("RS",instruccion);  //Cogemos el campo rs de la instrucci¢n
  offset = instruccion & 0x0000FFFF;  //Cogemos el campo offset de la instr.

  offset = offset << 2;   //Se desplaza dos bits hacia la izquierda.
  auxi=offset;  //Metemos offset en auxi
  auxi=auxi>>17;   //Desplazamos auxi hacia la derecha 17 bits.

  if (auxi!=0)      //Si auxi es distinto de cero
    offset=offset | 0xFFFC0000;  //Hacemos la extensi¢n de signo.

  if (Estado("UX")==0)  //Para 32 bits.
  {
    condicion=rpg[rs][1] >> 31;  //miramos el bit 32
    if ((condicion==0) && (rpg[rs][1]!=0))  //Si es mayor que cero
       rpg[34][1]=rpg[34][1]+offset;  //Al CP se le suma el offset
  }
  else if (Estado("UX")==1)   //Para 64 bits.
  {
    condicion=rpg[rs][0] >> 31;     //miramos el bit 64
    if ((condicion==0) && (rpg[rs][0]!=0 || rpg[rs][1]!=0)) //Si mayor que
cero
       rpg[34][1]=rpg[34][1]+offset;   //Al CP se le suma el offset.
  }
}      //Fin de la instrucci¢n BGTZ

void BGTZL (ULONG instruccion) //Branch if Greater Than Zero Likely
/*Si el registro rs es mayor que cero, el programa hace un salto relativo
marcado por el campo offset. Si la condici¢n no se cumple, la siguiente
instrucci¢n que hay en memoria despu,s de esta no se ejecuta.*/
{
  UCHAR rs;   //Para coger el n£mero del registro rs.
  signed long int offset;  //Para el desplazamiento relativo
  ULONG auxi,condicion;  //Para variable auxiliar y condici¢n

  rs = Campo ("RS",instruccion);   //Cogemos el campo rs de la instrucci¢n
  offset = instruccion & 0x0000FFFF;  //Cogemos el campo offset de la instr.

  offset = offset << 2;  //Desplazamos dos bits hacia la izquierda.
  auxi=offset;   //En auxi metemos lo que hay en offset
  auxi=auxi>>17;    //Desplazamos 17 bits hacia la derecha.

  if (auxi!=0)  //Si auxi es distinto de cero
    offset=offset | 0xFFFC0000;   //Hacemos extensi¢n de signo todo a unos.

  if (Estado("UX")==0)  //Para 32 bits.
  {
    condicion=rpg[rs][1] >> 31;  //miramos el bit 32
    if ((condicion==0) && (rpg[rs][1]!=0))   //Si es mayor que cero
       rpg[34][1]=rpg[34][1]+offset;  //Al CP se le suma el desplazamiento
    else    //Si no es mayor que cero
       rpg[34][1]=rpg[34][1]+4;  //NullifyCurrentInstruction (se anula)
  }
  else if (Estado("UX")==1)   //Para 64 bits.
```

```
  {
    condicion=rpg[rs][0] >> 31;     //miramos el bit 64
    if ((condicion==0) && (rpg[rs][0]!=0 || rpg[rs][1]!=0)) //Si mayor que
cero
       rpg[34][1]=rpg[34][1]+offset;  //Al CP se le suma el desplazamiento
    else   //Si no es mayor que cero
       rpg[34][1]=rpg[34][1]+4;     //NullifyCurrentInstruction (se anula)
  }
}    //fin de la instrucci¢n BGTZL

void BLEZ (ULONG instruccion)  //Branch on Less Than or Equal To Zero
/*Si el registro rs es menor o igual que cero, el programa hace un salto
relativo marcado por el campo offset*/
{
  UCHAR rs;  //Para coger el n£mero del registro rs.
  signed long int offset;  //Para el desplazamiento
  ULONG auxi,condicion;  //Para variable auxiliar y condici¢n

  rs = Campo ("RS",instruccion);  //Cogemos el campo rs de la instrucci¢n
  offset = instruccion & 0x0000FFFF;  //Cogemos el campo offset de la instr.

  offset = offset << 2;   //Desplazamos hacia la izquierda dos bits.
  auxi=offset;   //En auxi metemos lo que hay en offset.
  auxi=auxi>>17;   //Desplazamos auxi 17 bits hacia la derecha.

  if (auxi!=0)  //Si auxi es distinto de cero
    offset=offset | 0xFFFC0000;   //Se hace una extensi¢n de signo con unos.

  if (Estado("UX")==0)  //Para 32 bits.
  {
    condicion=rpg[rs][1] >> 31;  //miramos el bit 32
    if ((condicion!=0) || (rpg[rs][1]==0))  //Si es menor o igual que cero
       rpg[34][1]=rpg[34][1]+offset;  //Al CP se le suma el offset
  }
  else if (Estado("UX")==1)   //Para 64 bits.
  {
    condicion=rpg[rs][0] >> 31;     //miramos el bit 64
    if ((condicion!=0) || (rpg[rs][0]==0 && rpg[rs][1]==0)) //Si menor o cero
       rpg[34][1]=rpg[34][1]+offset;  //Al CP se le suma el offset
  }
}   //Fin de la instrucci¢n BLEZ

void BLEZL (ULONG instruccion)  //Branch on Less Than Or Equal to Zero Likely
/*Si el registro rs es menor o igual que cero, el programa hace un salto
relativo marcado por el campo offset. Si la condici¢n no se cumple, la
siguiente instrucci¢n que hay en memoria despu,s de esta no se ejecuta.*/
{
  UCHAR rs; //Para coger el n£mero de registro rs
  signed long int offset;  //Para el desplazamiento
  ULONG auxi,condicion;  //Para variable auxiliar y condici¢n

  rs = Campo ("RS",instruccion);  //Cogemos el campo rs de la instrucci¢n
  offset = instruccion & 0x0000FFFF;  //Cogemos el camo offset de la instr.

  offset = offset << 2;   //Desplazamos offset hacia la izquierda dos bits
```

```
  auxi=offset;   //En auxi metemos lo que hay en offset
  auxi=auxi>>17;   //Desplazamos 17 bits hacia la derecha.

  if (auxi!=0)   //Si auxi es distinto de cero
    offset=offset | 0xFFFC0000;  //Hacemos la extensi¢n de signo a unos.

  if (Estado("UX")==0)  //Para 32 bits.
  {
    condicion=rpg[rs][1] >> 31;  //miramos el bit 32
    if ((condicion!=0) || (rpg[rs][1]==0))  //Si es menor o igual que cero
       rpg[34][1]=rpg[34][1]+offset;  //Al CP le sumamos el offset
    else
       rpg[34][1]=rpg[34][1]+4;    //NullifyCurrentInstruction
  }
  else if (Estado("UX")==1)   //Para 64 bits.
  {
    condicion=rpg[rs][0] >> 31;     //miramos el bit 64
    if ((condicion!=0) || (rpg[rs][0]==0 && rpg[rs][1]==0)) //Si menor o
igual
       rpg[34][1]=rpg[34][1]+offset;  //Al CP le sumamos el offset
    else
       rpg[34][1]=rpg[34][1]+4;     //NullifyCurrentInstruction
  }
}     //fin de la instrucci¢n BLEZL

void BLTZ (ULONG instruccion)  //Branch On Less Than Zero
{
  UCHAR rs;  //Para coger el n£mero de registro
  signed long int offset; //Para el desplazamiento
  ULONG auxi,condicion;  //Variable auxiliar y condici¢n

  rs = Campo ("RS",instruccion);  //Cogemos el campo rs de la instrucci¢n
  offset = instruccion & 0x0000FFFF;  //Cogemos el campo offset de la instr.

  offset = offset << 2;  //Desplazamos dos bits hacia la izquierda
  auxi=offset;  //En auxi metemos el offset.
  auxi=auxi>>17;  //Desplazamos 17 bits hacia la derecha.

  if (auxi!=0)  //Si auxi es distinto de cero
    offset=offset | 0xFFFC0000;  //Hacemos extensi¢n de signo con unos.

  if (Estado("UX")==0)  //Para 32 bits.
  {
    condicion=rpg[rs][1] >> 31;  //miramos el bit 32
    if (condicion!=0)  //Si condici¢n es distinto de cero
       rpg[34][1]=rpg[34][1]+offset; //Sumamos offset al CP
  }
  else if (Estado("UX")==1)   //Para 64 bits.
  {
    condicion=rpg[rs][0] >> 31;     //miramos el bit 64
    if (condicion!=0) //Si condici¢n es distinto de cero
       rpg[34][1]=rpg[34][1]+offset;  //Sumamos offset al CP
  }
}    //Fin de la instrucci¢n BLTZ
```

```
void BLTZL (ULONG instruccion)  //Branch On Less Than Zero Likely
{
  UCHAR rs;  //Para el n£mero de registro
  signed long int offset; //Para el desplazamiento
  ULONG auxi,condicion;  //Para variable auxiliar y condici¢n

  rs = Campo ("RS",instruccion);  //Cogemos el campo rs de la instrucci¢n
  offset = instruccion & 0x0000FFFF; //Cogemos el campo offset de instrucci¢n

  offset = offset << 2;  //Desplazamos dos bits hacia la izquierda
  auxi=offset;  //En auxi metemos el contenido de offset
  auxi=auxi>>17;  //Desplazamos 17 bits hacia la derecha.

  if (auxi!=0)  //Si auxi es distinto de cero
    offset=offset | 0xFFFC0000;  //Hacemos la extensi¢n de signo con unos

  if (Estado("UX")==0)  //Para 32 bits.
  {
    condicion=rpg[rs][1] >> 31;  //miramos el bit 32
    if (condicion!=0)  //Si el bit 32 es distinto de cero
       rpg[34][1]=rpg[34][1]+offset;  //Sumamos al CP el offset
    else
       rpg[34][1]=rpg[34][1]+4;         //NullifyCurrentInstruction
  }
  else if (Estado("UX")==1)   //Para 64 bits.
  {
    condicion=rpg[rs][0] >> 31;     //miramos el bit 64
    if (condicion!=0)   //Si el bit 64 es distinto de cero
       rpg[34][1]=rpg[34][1]+offset;  //Sumamos al CP el offset
    else
       rpg[34][1]=rpg[34][1]+4;        //NullifyCurrentInstruction
  }
}   //Fin de la instrucci¢n BLTZL

void BLTZAL (ULONG instruccion) //Branch On Less Than Zero and Link
{
  UCHAR rs;  //Para coger el n£mero de registro
  signed long int offset;  //Para el desplazamiento
  ULONG auxi,condicion;  //Variable auxiliar y condici¢n

  rs = Campo ("RS",instruccion);  //Cogemos el campo rs de la instrucci¢n
  offset = instruccion & 0x0000FFFF;  //Cogemos el desplazamiento

  offset = offset << 2;  //Desplazamos dos bits hacia la izquierda
  auxi=offset;  //En auxi metemos el contenido de offset
  auxi=auxi>>17;  //Desplazamos 17 bits hacia la derecha.

  if (auxi!=0)  //Si auxi es distinto de cero
    offset=offset | 0xFFFC0000;  //Se hace una extensi¢n de signo con unos

  if (Estado("UX")==0)  //Para 32 bits.
  {
    condicion=rpg[rs][1] >> 31;  //miramos el bit 32
    rpg[31][1]=rpg[34][1]+8;//Metemos la direc de la instr. despu,s de delay
slot
```

```
    if (condicion!=0)   //Si el bit 32 es distinto de cero
       rpg[34][1]=rpg[34][1]+offset;   //Sumamos al CP el desplazamiento
  }
  else if (Estado("UX")==1)   //Para 64 bits.
  {
    condicion=rpg[rs][0] >> 31;     //miramos el bit 64
    rpg[31][1]=rpg[34][1]+8;//Metemos la direc de la instr. despu,s de delay
slot
    if (condicion!=0)  //Si el bit 64 es distinto de cero
       rpg[34][1]=rpg[34][1]+offset;  //Sumamos al CP el desplazamiento
  }
}   //Fin de la instrucci¢n BLTZAL

void BLTZALL (ULONG instruccion)//Brch On Lss Thn Zro And Link Likely
{
  UCHAR rs;  //Para coger el n£mero de registro
  signed long int offset;  //Para el desplazamiento
  ULONG auxi,condicion;  //Para variable auxiliar y condici¢n

  rs = Campo ("RS",instruccion);  //Cogemos el campo rs de la instrucci¢n
  offset = instruccion & 0x0000FFFF;  //Cogemos el campo desplazamiento

  offset = offset << 2;  //Desplazamos 2 bits hacia la izquierda
  auxi=offset;   //En auxi metemos lo que hay en offset
  auxi=auxi>>17;  //Desplazamos 17 bits hacia la derecha

  if (auxi!=0)   //Si auxi es distinto de cero
    offset=offset | 0xFFFC0000;  //Hacemos la extensi¢n de signo con unos

  if (Estado("UX")==0)  //Para 32 bits.
  {
    condicion=rpg[rs][1] >> 31;  //miramos el bit 32
    rpg[31][1]=rpg[34][1]+8;//Metemos direc de la instr. siguiente al delay
slt
    if (condicion!=0)   //Si condici¢n es distinto de cero
       rpg[34][1]=rpg[34][1]+offset;  //Sumamos al desplazamiento el CP
    else   //Si no es distinto de cero
       rpg[34][1]=rpg[34][1]+4;      //NullifyCurrentInstruction
  }
  else if (Estado("UX")==1)   //Para 64 bits.
  {
    condicion=rpg[rs][0] >> 31;     //miramos el bit 64
    rpg[31][1]=rpg[34][1]+8;//Metemos direc de la isntr. siguiente al delay
slt
    if (condicion!=0)   //Si el bit 64 es distinto de cero
       rpg[34][1]=rpg[34][1]+offset;  //Sumamos al CP el offset
    else  //Si no es distinto de cero
       rpg[34][1]=rpg[34][1]+4;         //NullifyCurrentInstruction
  }
}    //fin de la instrucci¢n BLTZALL

void BGEZL (ULONG instruccion)//Branch if Greater Than or equal to zero
likely
{
  UCHAR rs;  //Para coger el n£mero de registro
```

```
  signed long int offset;  //Para el desplazamiento
  ULONG auxi,condicion;  //Variable auxiliar y condici¢n

  rs = Campo ("RS",instruccion);  //Cogemos el campo rs de la instrucci¢n
  offset = instruccion & 0x0000FFFF;  //Cogemos el offset de la instrucci¢n

  offset = offset << 2;  //Desplazamos 2 bits hacia la izquierda
  auxi=offset;  //En auxi metemos el offset
  auxi=auxi>>17;  //Desplazamos 17 bits hacia la derecha

  if (auxi!=0)  //Si auxi es distinto de cero
    offset=offset | 0xFFFC0000; //Hacemos extensi¢n de signo con unos

  if (Estado("UX")==0)  //Para 32 bits.
  {
    condicion=rpg[rs][1] >> 31;  //miramos el bit 32
    if (condicion==0)   //Si el bit 32 es igual a cero
       rpg[34][1]=rpg[34][1]+offset;  //Sumamos desplazamiento al CP
    else  //Si no es igual a cero
       rpg[34][1]=rpg[34][1]+4;  //NullifyCurrentInstruction
  }
  else if (Estado("UX")==1)   //Para 64 bits.
  {
    condicion=rpg[rs][0] >> 31;     //miramos el bit 64
    if (condicion==0)  //Si el bit 64 es igual a cero
       rpg[34][1]=rpg[34][1]+offset;  //Sumamos desplazamiento al CP
    else  //Si no es igual a cero
       rpg[34][1]=rpg[34][1]+4;  //NullifyCurrentInstruction
  }
}   //Fin de la instrucci¢n BGEZL

void BGEZAL (ULONG instruccion)//Brch On Greater than or Equal to zro and
Link
{
  UCHAR rs;  //Para coger el n£mero de registro
  signed long int offset;  //Para el desplazamiento
  ULONG auxi,condicion;  //Variable auxiliar y condici¢n

  rs = Campo ("RS",instruccion);  //Cogemos el campo rs de la instrucci¢n
  offset = instruccion & 0x0000FFFF;  //Cogemos el offset de la instrucci¢n

  offset = offset << 2;  //Desplazamos 2 bits hacia la izquierda
  auxi=offset;   //En auxi metemos offset
  auxi=auxi>>17;   //Desplazamos 17 bits hacia la derecha.

  if (auxi!=0)  //Si auxi es distinto de cero
    offset=offset | 0xFFFC0000;  //Hacemos la extensi¢n de signo con unos

  if (Estado("UX")==0)  //Para 32 bits.
  {
    condicion=rpg[rs][1] >> 31;  //miramos el bit 32
    rpg[31][1]=rpg[34][1]+8;  //Metemos direcci¢n de instr. despu,s de delay
slot
    if (condicion==0)  //Si el bit 32 es igual a cero
       rpg[34][1]=rpg[34][1]+offset;   //Sumamos al CP el desplazamiento
```

```
  }
  else if (Estado("UX")==1)   //Para 64 bits.
  {
    condicion=rpg[rs][0] >> 31;     //miramos el bit 64
    rpg[31][1]=rpg[34][1]+8;  //Metemos direcci¢n de instr. despu,s de delay
slot
    if (condicion==0)   //Si el bit 64 es igual a cero
       rpg[34][1]=rpg[34][1]+offset;  //Sumamos al CP el desplazamiento
  }
}    //Fin de la instrucci¢n BGEZAL

void BGEZALL (ULONG instruccion)//Brch On Grtr Thn Or Eql to Zro and Lnk
Likely
{
  UCHAR rs;  //Para el n£mero de registro
  signed long int offset;  //Para el desplazamiento
  ULONG auxi,condicion;  //Variable auxiliar y condici¢n

  rs = Campo ("RS",instruccion);  //Cogemos el campo rs de la instrucci¢n
  offset = instruccion & 0x0000FFFF;  //Cogemos el campo offset de la instr.

  offset = offset << 2;   //Desplazamos 2 bits hacia la izquierda.
  auxi=offset;   //En auxi metemos lo que hay en offset
  auxi=auxi>>17;   //Desplazamos 17 bits hacia la derecha.

  if (auxi!=0)   //Si auxi es distinto de cero
    offset=offset | 0xFFFC0000;  //Hacemos la extensi¢n de signo

  if (Estado("UX")==0)  //Para 32 bits.
  {
    condicion=rpg[rs][1] >> 31;  //miramos el bit 32
    rpg[31][1]=rpg[34][1]+8;  //Metemos direcci¢n de instr despues de delay
slot
    if (condicion==0)  //Si el bit 32 es igual a cero
       rpg[34][1]=rpg[34][1]+offset;  //Sumamos el desplazamiento al CP
    else
       rpg[34][1]=rpg[34][1]+4;   //NullifyCurrentInstruction (pasa de la
sig)
  }
  else if (Estado("UX")==1)   //Para 64 bits.
  {
    condicion=rpg[rs][0] >> 31;     //miramos el bit 64
    rpg[31][1]=rpg[34][1]+8;  //Metemos direcci¢n de intr despues de delay
slot
    if (condicion==0)   //Si el bit 64 es igual a cero
       rpg[34][1]=rpg[34][1]+offset;  //Sumamos el desplazamiento al CP
    else
       rpg[34][1]=rpg[34][1]+4;   //NullifyCurrentInstruction (pasa de la
sig)
  }
}      //Fin de la instrucci¢n BGEZALL

void TEQI (ULONG instruccion)  //Trap If Equal Immediate
/*Al offset se le hace una extensi¢n de signo y se compara con el contenido
del registro rs. Si el contenido del registro rs es igual al dato inmediato
```

```
extendido (el signo), ocurre una Trap exception*/
{
  UCHAR rs;  //Para el n£mero del registro rs
  ULONG inmediato,inme0,auxi; //Parte alta y baja de dato inmediato y
auxiliar

  rs = Campo ("RS",instruccion);  //Cogemos el campo rs de la instrucci¢n
  inmediato = instruccion & 0x0000FFFF; //Cogemos el dato inmediato
  auxi=inmediato>>15;  //En auxi metemos inmediato desplazado 15 bits drcha

  if (auxi!=0)  //Si auxi es distinto de cero
  {
    inmediato=inmediato | 0xFFFF0000;  //Hacemos la extensi¢n de signo a unos
    inme0=0xFFFFFFFF;  //Ponemos la parte alta todo a unos
  }
  else if (auxi==0)   //Si auxi es igual a cero
    inme0=0x00000000;  //Ponemos la parte alta de inmediato todo a unos

  if (Estado("UX")==0)  //Para 32 bits.
  {
    if (rpg[rs][1]==inmediato)  //Si parte baja del reg. es igual a inmediato
    {
      General_Exception();     //Trap exception
      gotoxy (30,23);
      printf ("     Trap exception.     ");
    }
  }
  else if (Estado("UX")==1)   //Para 64 bits.
  {
    if ((rpg[rs][0]==inme0) && (rpg[rs][1]==inmediato))  //Si son iguales
    {
      General_Exception();     //Trap exception
      gotoxy (30,23);
      printf ("     Trap exception.     ");
    }
  }
}    //Fin de la instrucci¢n TEQI

void TNEI (ULONG instruccion)  //Trap If Not Equal Immediate
/*Al dato inmediato se le hace una extensi¢n de signo y se compara con el
contenido del registro rs. Si el contenido del registro rs no es igual que
el dato inmediato extendido, ocurre una trap exception*/
{
  UCHAR rs;  //Para coger el n£mero de registro rs
  ULONG inmediato,inme0,auxi;  //Variables sin signo

  rs = Campo ("RS",instruccion);  //Cogemos el campo rs de la instrucci¢n
  inmediato = instruccion & 0x0000FFFF;  //Cogemos el campo inmediato
  auxi=inmediato>>15;  //En auxi metemos inmediato desplazado 15 bits drcha.

  if (auxi!=0)  //Si auxi es distinto de cero
  {
    inmediato=inmediato | 0xFFFF0000;  //Hacemos extensi¢n de signo con unos
    inme0=0xFFFFFFFF;  //La parte alta de inmediato la ponemos a unos.
  }
```

```
  else if (auxi==0)  //Si auxi es igual a cero
    inme0=0x00000000;  //Ponemos la parte alta de inmediato a ceros

  if (Estado("UX")==0)  //Para 32 bits.
  {
    if (rpg[rs][1]!=inmediato) //Si parte baja de reg. distinto que inmediato
    {
      General_Exception();     //Trap exception
      gotoxy (30,23);
      printf ("     Trap exception.     ");
    }
  }
  else if (Estado("UX")==1)   //Para 64 bits.
  {
    if ((rpg[rs][0]!=inme0) || (rpg[rs][1]!=inmediato)) //Si son distintos
    {
      General_Exception();     //Trap exception
      gotoxy (30,23);
      printf ("     Trap exception.     ");
    }
  }
}   //Fin de la instrucci¢n TNEI

void TGEI (ULONG instruccion)    //Trap If Greater Than Or Equal Immediate
/*Al dato inmediato se le extiende el signo y se compara con el contenido
del registro rs. Considerando ambas cantidades como enteros con signo, si
el contenido del registro rs es mayor o igual al dato inmediato extendido
ocurre una Trap Exception*/
{
  UCHAR rs;  //Para coger el n£mero del registro rs.
  signed long int rssigno, inmesigno; //Variables con signo
  ULONG inmediato,inme0,auxi,rsnosigno,inmenosigno;//Variables sin signo

  rs = Campo ("RS",instruccion);  //Cogemos el campo rs de la instrucci¢n
  inmediato = instruccion & 0x0000FFFF; //Cogemos el dato inmediato
  auxi=inmediato>>15;  //En auxi metemos el dato inmediato desplzado 15 bits.

  if (auxi!=0) //Si auxi es distinto de cero
  {
    inmediato=inmediato | 0xFFFF0000; //Se extiende el signo a unos
    inme0=0xFFFFFFFF;  //La parte alta de inmediato se pone a unos
  }
  else if (auxi==0)  //Si auxi es igual a cero
    inme0=0x00000000;   //La parte alta de inmediato se pone a ceros

  if (Estado("UX")==0)  //Para 32 bits.
  {
    rssigno=rpg[rs][1];  //En rssigno metemos parte baja del reg. rs
    inmesigno=inmediato;  //En inmesigno metemos parte baja de dato inmediato
    if ((rssigno>inmesigno) || (rssigno==inmesigno)) //Si es mayor o igual
    {
      General_Exception();     //Trap exception
      gotoxy (30,23);
      printf ("     Trap exception.     ");
    }
```

```
  }
  else if (Estado("UX")==1)   //Para 64 bits.
  {
    rssigno=rpg[rs][0];  //En rssigno metemos parte alta de reg. rs.
    rsnosigno=rpg[rs][1];  //En rsnosigno metemos parte baja de reg. rs.
    inmesigno=inme0;  //En inmesigno metemos parte alta de dato inmediato
    inmenosigno=inmediato;  //En inmenosigno metemos parte baja de dato
inmed.
    if (rpg[rs][0]!=0 || inme0!=0)//Si la parte alta de los datos es !=0
    {
       if (rpg[rs][0]!=inme0) //Si las dos partes altas son distintas
       {
       if ((rssigno>inmesigno) || (rssigno==inmesigno)) //Si mayor o igual
       {
         General_Exception();     //Trap exception
         gotoxy (30,23);
         printf ("     Trap exception.     ");
       }
       }
       else if (rpg[rs][0]==inme0)
       {
       if ((rsnosigno>inmenosigno) || (rsnosigno==inmenosigno))//Si > o igual
       {
         General_Exception();     //Trap exception
         gotoxy (30,23);
         printf ("     Trap exception.     ");
       }
       }
    }
    else  //Si la parte alta de los dos datos es igual a cero
    {
      if ((rsnosigno>inmenosigno) || (rsnosigno==inmenosigno))//Si mayor o
igual
      {
      General_Exception();     //Trap exception
      gotoxy (30,23);
      printf ("     Trap exception.     ");
      }
    }
  }
}   //Fin de la instrucci¢n TGEI

void TGEIU (ULONG instruccion) //Trap If Greater Than Or Equal Immed.
Unsigned
/*Al dato inmediato se le extiende el signo y se compara con el contenido
del registro rs. Considerando ambas cantidades como enteros sin signo, si
el contenido del registro rs es mayor o igual que el dato inmediato
extendido,
ocurre una Trap Exception*/
{
  UCHAR rs;  //Para coger el n£mero de registro rs.
  ULONG inmediato,inme0,auxi,regrs;  //Variables sin signo.

  rs = Campo ("RS",instruccion);  //Cogemos el campo rs de la instrucci¢n
  regrs=rpg[rs][1];   //En regrs metemos la parte baja del registro rs.
```

```
  inmediato = instruccion & 0x0000FFFF;  //Cogemos el campo inmediato
  auxi=inmediato>>15;  //En auxi metemos el dato inmediato despl. 15 bits
drcha

  if (auxi!=0)  //Si auxi es distinto de cero
  {
    inmediato=inmediato | 0xFFFF0000;  //Hacemos extensi¢n de signo con unos
    inme0=0xFFFFFFFF;  //Ponemos la parte alta de dato inmediato todo a unos
  }
  else if (auxi==0)  //Si auxi es igual a cero
    inme0=0x00000000;  //Ponemos la parte alta de dato inmediato todo a ceros

  if (Estado("UX")==0)  //Para 32 bits.
  {
    if ((regrs>inmediato) || (regrs==inmediato)) //Si es mayor o igual
    {
      General_Exception();     //Trap exception
      gotoxy (30,23);
      printf ("     Trap exception.     ");
    }
  }
  else if (Estado("UX")==1)   //Para 64 bits.
  {
    if (rpg[rs][0]!=0 || inme0!=0)  //Si la parte alta es distinta de cero
    {
       if (rpg[rs][0]!=inme0)  //Si las dos partes altas son distintas.
       {
       regrs=rpg[rs][0];  //En regrs metemos la parte alta del reg. rs.
       if ((regrs>inme0) || (regrs==inme0)) //Si es mayor o igual
       {
         General_Exception();     //Trap exception
         gotoxy (30,23);
         printf ("     Trap exception.     ");
       }
       }
       else if (rpg[rs][0]==inme0)  //Si las dos partes altas son iguales
       {
       if ((regrs>inmediato) || (regrs==inmediato))//Comp. partes bajas.
       {
         General_Exception();     //Trap exception
         gotoxy (30,23);
         printf ("     Trap exception.     ");
       }
       }
    }
    else  //Si las dos partes altas son igual a cero
    {
      if ((regrs>inmediato) || (regrs==inmediato))  //Comparamos partes bajas
      {
      General_Exception();     //Trap exception
      gotoxy (30,23);
      printf ("     Trap exception.     ");
      }
    }
  }
```

```
}    //Fin de la instrucci¢n TGEIU

void TLTI (ULONG instruccion)  //Trap If Less Than Immediate
/*Al dato inmediato se le extiende el signo y se compara con el contenido
del registro rs. Considerando ambas cantidades como enteros con signo, si
el contenido del registro rs es menor que el dato inmediato extendido, ocurre
una Trap Exception*/
{
  UCHAR rs;  //Para coger el n£mero de registro
  signed long int rssigno,inmesigno;  //Variables con signo
  ULONG inmediato,inme0,auxi,rsnosigno,inmenosigno;//Variables sin signo

  rs = Campo ("RS",instruccion); //Cogemos el campo rs de  la instrucci¢n
  inmediato = instruccion & 0x0000FFFF;  //Cogemos el campo inmediato
  auxi=inmediato>>15;  //En auxi metemos dato inmediato desplzado 15 bits.

  if (auxi!=0) //Si auxi es distinto de cero
  {
    inmediato=inmediato | 0xFFFF0000;  //Hacemos extensi¢n de signo de
inmediato
    inme0=0xFFFFFFFF;  //En parte alta de inmediato metemos todo a unos
  }
  else if (auxi==0)  //Si auxi es igual a cero
    inme0=0x00000000;  //En parte alta de inmediato metemos todo a ceros

  if (Estado("UX")==0)  //Para 32 bits.
  {
    rssigno=rpg[rs][1];  //En rssigno metemos la parte baja de reg. rs.
    inmesigno=inmediato;  //En inmesigno metemos parte baja de dato inmediato
    if (rssigno<inmesigno)  //Si rssigno es menor que inmesigno
    {
      General_Exception();     //Trap exception
      gotoxy (30,23);
      printf ("     Trap exception.     ");
    }
  }
  else if (Estado("UX")==1)   //Para 64 bits.
  {
    rssigno=rpg[rs][0];  //En rssigno metemos parte alta de reg. rs
    rsnosigno=rpg[rs][1];  //En rsnosigno metemos parte baja de reg. rs.
    inmesigno=inme0;  //En inmesigno metemos parte alta de dato inmediato
    inmenosigno=inmediato;   //En inmenosigno metemos parte baja de dato
inmedi
    if (rpg[rs][0]!=0 || inme0!=0)  //Si la parte alta es distinta de cero
    {
       if (rpg[rs][0]!=inme0)  //Si una parte alta es distinta que la otra.
       {
       if (rssigno<inmesigno)  //Comparamos las partes altas de los datos
       {
         General_Exception();     //Trap exception
         gotoxy (30,23);
         printf ("     Trap exception.     ");
       }
       }
       else if (rpg[rs][0]==inme0)  //Si las partes altas son iguales
```

```
       {
       if (rsnosigno<inmenosigno)  //Comparamos las partes bajas
       {
         General_Exception();     //Trap exception
         gotoxy (30,23);
         printf ("     Trap exception.     ");
       }
       }
    }
    else  //Si las partes altas son cero
    {
      if (rsnosigno<inmenosigno)  //Comparamos las partes bajas
      {
      General_Exception();     //Trap exception
      gotoxy (30,23);
      printf ("     Trap exception.     ");
      }
    }
  }
}    //fin de la instrucci¢n TLTI

void TLTIU (ULONG instruccion)  //Trap If Less Than Immediate Unsigned
/*Al dato inmediato de 16 bits se le hace una extensi¢n de signo y se compara
con el contenido del registro rs. Considerando ambas cantidades como enteros
sin signo, si el contenido del registro rs es menor que el dato inmediato
con extensi¢n, ocurre una Trap exception*/
{
  UCHAR rs;  //Para coger el n£mero de registro rs.
  ULONG inmediato,inme0,auxi,regrs;  //Variables sin signo

  rs = Campo ("RS",instruccion); //Cogemos el campo rs de la instrucci¢n
  regrs=rpg[rs][1];  //En regrs metemos parte baja del reg. rs.
  inmediato = instruccion & 0x0000FFFF;  //Cogemos el dato inmediato
  auxi=inmediato>>15;  //En auxi metemos inmediato desplazado 15 bits drcha.

  if (auxi!=0) //Si auxi es distinto de cero
  {
    inmediato=inmediato | 0xFFFF0000;  //Hacemos la extensi¢n de signo
    inme0=0xFFFFFFFF;  //La parte alta de inmediato se pone todo a unos
  }
  else if (auxi==0)  //Si auxi es igual a cero
    inme0=0x00000000;  //La parte alta de inmediato se pone todo a ceros

  if (Estado("UX")==0)  //Para 32 bits.
  {
    if (regrs<inmediato)  //Si regrs es menor que inmediato
    {
      General_Exception();     //Trap exception
      gotoxy (30,23);
      printf ("     Trap exception.     ");
    }
  }
  else if (Estado("UX")==1)   //Para 64 bits.
  {
    if (rpg[rs][0]!=0 || inme0!=0)  //Si la parte alta es distinta de cero
```

```
    {
       if (rpg[rs][0]!=inme0)   //Y las dos partes son distintas
       {
       regrs=rpg[rs][0];  //En regrs metemos parte alta del registro rs.
       if (regrs<inme0)  //Si regrs es menor que parte alta de inmediato
       {
         General_Exception();     //Trap exception
         gotoxy (30,23);
         printf ("     Trap exception.     ");
       }
       }
       else if (rpg[rs][0]==inme0)  //Si las dos partes altas son iguales
       {
       if (regrs<inmediato)  //Si regrs es menor que parte baja de inmediato
       {
         General_Exception();     //Trap exception
         gotoxy (30,23);
         printf ("     Trap exception.     ");
       }
       }
    }
    else  //Si las dos partes altas son igual a cero
    {
      if (regrs<inmediato)  //Comparamos parte baja del reg. y baja de
inmediato
      {
      General_Exception();     //Trap exception
      gotoxy (30,23);
      printf ("     Trap exception.     ");
      }
    }
  }
}   //Fin de la instrucci¢n TLTIU

void TEQ (ULONG instruccion)   //Trap if equal
/*Esta instrucci¢n causa una Trap exception si el contenido del registro
rs es igual que el contenido del registro rt*/
{
  UCHAR rs,rt;    //Para coger el n£mero del registro

  rs = Campo ("RS",instruccion);  //Cogemos el campo rs de la instrucci¢n
  rt = Campo ("RT",instruccion);  //Cogemos el campo rt de la instrucci¢n

  if (Estado("UX")==0)  //Para 32 bits.
  {
    if (rpg[rs][1]==rpg[rt][1])  //si el reg rs es igual al rt
    {
      General_Exception();     //Trap exception
      gotoxy (30,23);
      printf ("     Trap exception.     ");
    }
  }
  else if (Estado("UX")==1)   //Para 64 bits.
  {
    if ((rpg[rs][0]==rpg[rt][0]) && (rpg[rs][1]==rpg[rt][1]))
```

```
    //si el registro rs es igual al rt
    {
      General_Exception();     //Trap exception
      gotoxy (30,23);
      printf ("     Trap exception.     ");
    }
  }
}            //Fin de la instrucci¢n TEQ

void TNE (ULONG instruccion)  //Trap if not equal
/*Esta instrucci¢n produce una excepci¢n si el contenido del registro rs
no es igual al contenido del registro rt.*/
{
  UCHAR rs,rt;  //Para el n£mero de los registros

  rs = Campo ("RS",instruccion);  //Cogemos el campo rs de la instrucci¢n
  rt = Campo ("RT",instruccion);  //Cogemos el campo rt de la instrucci¢n

  if (Estado("UX")==0)  //Para 32 bits.
  {
    if (rpg[rs][1]!=rpg[rt][1])   //Si contenido de rs es distinto del de rt
    {
      General_Exception();     //Trap exception
      gotoxy (30,23);
      printf ("     Trap exception.     ");
    }
  }
  else if (Estado("UX")==1)   //Para 64 bits.
  {
    if ((rpg[rs][0]!=rpg[rt][0]) || (rpg[rs][1]!=rpg[rt][1]))
    //Si alguna de las dos partes (alta,baja) de cada registro son distintas
    {
      General_Exception();     //Trap exception
      gotoxy (30,23);
      printf ("     Trap exception.     ");
    }
  }
}     //fin de la instrucci¢n TNE

void TGE (ULONG instruccion)   //Trap If Greater Than Or Equal
/*Si el contenido del registro rs es mayor o igual que el contenido del
registro rt ocurre una Trap exception. Los dos regs. deben ser con signo*/
{
  UCHAR rs,rt;   //Para coger el n£mero de registro
  signed long int rssigno, rtsigno; //para tener los valores de regs con
signo.
  ULONG rsnosigno, rtnosigno;//para tener valores sin signo.

  rs = Campo ("RS",instruccion);  //cogemos el campo rs de la instrucci¢n
  rt = Campo ("RT",instruccion);  //cogemos el campo rt de la instrucci¢n

  if (Estado("UX")==0)  //Para 32 bits.
  {
    rssigno=rpg[rs][1];  //Metemos el dato de 32 bits en rssigno.
    rtsigno=rpg[rt][1];  //Metemos el dato de 32 bits en rtsigno.
```

```
    if ((rssigno>rtsigno) || (rssigno==rtsigno))  //Si es mayor o igual
    {
      General_Exception();     //Trap exception
      gotoxy (30,23);
      printf ("     Trap exception.     ");
    }
  }
  else if (Estado("UX")==1)   //Para 64 bits.
  {
    rssigno=rpg[rs][0]; //Metemos la parte alta del reg. rs en rssigno
    rsnosigno=rpg[rs][1]; //Metemos la parte baja del reg. rs en rsnosigno
    rtsigno=rpg[rt][0];  //Metemos la parte alta del reg. rt en rtsigno
    rtnosigno=rpg[rt][1];  //Metemos la parte baja del reg. rt en rtnosigno

    if (rpg[rs][0]!=0 || rpg[rt][0]!=0) //si parte alta de alg£n reg != 0
    {
       if (rpg[rs][0]!=rpg[rt][0])  //Si las dos partes altas son distintas
       {
       if ((rssigno>rtsigno) || (rssigno==rtsigno)) //y rssigno >= rtsigno
       {
         General_Exception();     //Trap exception
         gotoxy (30,23);
         printf ("     Trap exception.     ");
       }
       }
       else if (rpg[rs][0]==rpg[rt][0]) //Si las dos partes altas son iguales
       {
       if ((rsnosigno>rtnosigno) || (rsnosigno==rtnosigno)) //si mayor o
igual
       {
         General_Exception();     //Trap exception
         gotoxy (30,23);
         printf ("     Trap exception.     ");
       }
       }
    }
    else   //Si la parte alta de los dos registros es cero
    {
      if ((rsnosigno>rtnosigno) || (rsnosigno==rtnosigno)) //si mayor o igual
      {
      General_Exception();     //Trap exception
      gotoxy (30,23);
      printf ("     Trap exception.     ");
      }
    }
  }
}   //Fin de la instrucci¢n TGE

void TGEU (ULONG instruccion)  //Trap if Greater Than or equal Unsigned
/*Si el contenido del registro rs es mayor o igual que el contenido del
registro rt ocurre una Trap exception. Los dos regs. deben ser sin signo*/
{
  UCHAR rs,rt;  //Para coger el n£mero del registro
  ULONG regrs,regrt;  //Para coger datos intermedios de los registros
```

```
  rs = Campo ("RS",instruccion);  //Cogemos el campo rs de la instrucci¢n
  rt = Campo ("RT",instruccion);  //Cogemos el campo rt de la instrucci¢n

  regrs=rpg[rs][1];  //En regrs metemos la parte baja del registro rs
  regrt=rpg[rt][1];  //En regrt metemos la parte baja del registro rt

  if (Estado("UX")==0)  //Para 32 bits.
  {
    if ((regrs>regrt) || (regrs==regrt)) //Si regrs es mayor o igual que
regrt
    {
      General_Exception();     //Trap exception
      gotoxy (30,23);
      printf ("     Trap exception.     ");
    }
  }
  else if (Estado("UX")==1)   //Para 64 bits.
  {
    if (rpg[rs][0]!=0 || rpg[rt][0]!=0) //Si la parte alta de alguno de los
    {                     //dos registros es distinta de cero
       if (rpg[rs][0]!=rpg[rt][0])  //Si las partes altas son distintas
       {
       regrs=rpg[rs][0];  //en regrs se mete la parte alta del reg. rs
       regrt=rpg[rt][0];  //en regrt se mete la parte alta del reg. rt
       if ((regrs>regrt) || (regrs==regrt)) //si regrs es mayor o igual
       {
         General_Exception();     //Trap exception
         gotoxy (30,23);
         printf ("     Trap exception.     ");
       }
       }
       else if (rpg[rs][0]==rpg[rt][0]) //Si las partes altas son iguales
       {
       if ((regrs>regrt) || (regrs==regrt)) //si regrs mayor o igual q regrt
       {
         General_Exception();     //Trap exception
         gotoxy (30,23);
         printf ("     Trap exception.     ");
       }
       }
    }
    else   //Si la parte alta de los dos registros es cero
    {
      if ((regrs>regrt) || (regrs==regrt)) //Si regrs es mayor o igual q
regrt
      {
      General_Exception();     //Trap exception
      gotoxy (30,23);
      printf ("     Trap exception.     ");
      }
    }
  }
}    //Fin de la instrucci¢n TGEU

void TLT (ULONG instruccion) //Trap if Less Than
```

```
/*Esta instrucci¢n provoca una trap exception si el contenido del registro
rs es menor que el contenido del registro rt. Los datos son con signo*/
{
  UCHAR rs,rt;  //Para coger el n£mero de registro
  signed long int rssigno, rtsigno; //Datos con signo
  ULONG rsnosigno, rtnosigno;  //Datos sin signo

  rs = Campo ("RS",instruccion);  //Cogemos el campo rs de la instrucci¢n
  rt = Campo ("RT",instruccion);  //Cogemos el campo rt de la instrucci¢n

  if (Estado("UX")==0)  //Para 32 bits.
  {
    rssigno=rpg[rs][1];  //En rssigno metemos la parte baja del reg. rs
    rtsigno=rpg[rt][1];  //En rtsigno metemos la parte baja del reg. rt
    if (rssigno<rtsigno)  //si rssigno es menor que rtsigno
    {
      General_Exception();     //Trap exception
      gotoxy (30,23);
      printf ("     Trap exception.     ");
    }
  }
  else if (Estado("UX")==1)   //Para 64 bits.
  {
    rssigno=rpg[rs][0]; //En rssigno metemos la parte alta del reg. rs
    rsnosigno=rpg[rs][1];  //En rsnosigno metemos la parte baja del reg. rs
    rtsigno=rpg[rt][0];  //En rtsigno metemos la parte alta del reg. rt
    rtnosigno=rpg[rt][1];   //En rtnosigno metemos la parte baja del reg. rt
    if (rpg[rs][0]!=0 || rpg[rt][0]!=0) //Si parte alta de alguno es !=0
    {
       if (rpg[rs][0]!=rpg[rt][0])  //Si son distintas
       {
       if (rssigno<rtsigno) //Si rssigno es menor
       {
         General_Exception();     //Trap exception
         gotoxy (30,23);
         printf ("     Trap exception.     ");
       }
       }
       else if (rpg[rs][0]==rpg[rt][0])  //si son iguales
       {
       if (rsnosigno<rtnosigno)  //Comparamos las partes bajas
       {
         General_Exception();     //Trap exception
         gotoxy (30,23);
         printf ("     Trap exception.     ");
       }
       }
    }
    else   //En caso de que la parte alta de los dos registros sea cero
    {
      if (rsnosigno<rtnosigno)   //Comparamos las partes bajas.
      {
      General_Exception();     //Trap exception
      gotoxy (30,23);
      printf ("     Trap exception.     ");
```

```
      }
    }
  }
}    //Fin de la instrucci¢n TLT

void TLTU (ULONG instruccion)   //Trap if Less Than Unsigned
/*Esta instrucci¢n provoca una trap exception si el contenido del registro
rs es menor que el contenido del registro rt. Los datos son sin signo*/
{
  UCHAR rs,rt;  //Para coger el n£mero de los registros
  ULONG regrs,regrt;  //Variables intermedias sin signo.

  rs = Campo ("RS",instruccion);   //Cogemos el campo rs de la instrucci¢n
  rt = Campo ("RT",instruccion);  //Cogemos el campo rt de la instrucci¢n
  regrs=rpg[rs][1];  //En regrs metemos la parte baja del reg. rs.
  regrt=rpg[rt][1];  //En regrt metemos la parte baja del reg. rt

  if (Estado("UX")==0)  //Para 32 bits.
  {
    if (regrs<regrt)  //Si regrs es menor que regrt
    {
      General_Exception();     //Trap exception
      gotoxy (30,23);
      printf ("     Trap exception.     ");
    }
  }
  else if (Estado("UX")==1)   //Para 64 bits.
  {
    if (rpg[rs][0]!=0 || rpg[rt][0]!=0) //Si la parte alta de alguno !=0
    {
       if (rpg[rs][0]!=rpg[rt][0])   //Si las partes altas son distintas
       {
       regrs=rpg[rs][0];  //Metemos la parte alta del reg rs en regrs
       regrt=rpg[rt][0];  //Metemos la parte alta del reg. rt en regrt
       if (regrs<regrt)  //Las comparamos
       {
         General_Exception();     //Trap exception
         gotoxy (30,23);
         printf ("     Trap exception.     ");
       }
       }
       else if (rpg[rs][0]==rpg[rt][0]) //Si las partes altas son iguales
       {
       if (regrs<regrt)  //Comparamos las partes bajas
       {
         General_Exception();     //Trap exception
         gotoxy (30,23);
         printf ("     Trap exception.     ");
       }
       }
    }
    else   //En caso de que la parte alta de los dos registro sea cero
    {
      if (regrs<regrt)  //Comparamos las partes bajas de cada registro
      {
```

```
      General_Exception();     //Trap exception
      gotoxy (30,23);
      printf ("    Trap exception.      ");
      }
    }
  }
}     //fin de la instrucci¢n TLTU

void LB (ULONG instruccion)   //Load Byte
/*Esta instrucci¢n extiende el signo del offset y se suma al contenido
del registro base para formar una direcci¢n. El contenido del byte en esa
localizaci¢n de memoria hace una extensi¢n de signo y se guarda en el
registro
rt.*/
{
  UCHAR rt,base,dato,aca; //base es el campo de rs, dato y aca para dato y
acarreo
  signed long int off0, offset, dest0, destino; //parte alta y baja de off y
dest
  ULONG auxi;  //Variable auxiliar.

  rt = Campo ("RT",instruccion);  //Cogemos el campo rt de la instrucci¢n
  base = Campo ("RS",instruccion);  //Cogemos el campo base de la instrucci¢n
  offset = instruccion & 0x0000FFFF; //Cogemos el campo offset de la
instrucc.
  auxi = offset;   //En auxi metemos offset.
  auxi = auxi >> 15;  //Desplazamos 15 bits hacia la derecha.

  if (auxi!=0)  //Si auxi es distinto de cero
  {
    offset = offset | 0xFFFF0000;  //Al offset se le hace extensi¢n signo a 1
    off0 = 0xFFFFFFFF;  //La parte alta del offset se pone todo a unos.
  }
  else if (auxi==0)  //Si auxi es igual a cero
    off0 = 0;

  destino=rpg[base][1]+offset;  //En destino metemos partebaja de base +
offset
  dest0 = rpg[base][0]+off0;  //En dest0 metemos parte alta de base+off0
  aca=acarreo(rpg[base][1],offset);  //Miramos si hay acarreo
  if (aca==1)  //Si hay acarreo
    dest0++;    //La parte alta del destino se incrementa en uno.

  if (Estado("UX")==0)  //Para 32 bits.
  {
    if ((destino > 65535) || (destino < 0))  //Si se sale de los l¡mites
    {
      gotoxy (30,23);     //Nos colocamos en la pantalla
      printf ("Direcci¢n fuera de rango.");  //Informamos
    }
    else   //Si no se sale de los l¡mites
    {
     fseek (pf,destino,SEEK_SET);  //Nos colocamos en memoria en posici¢n
indicada
     if ((fread (&dato,sizeof(UCHAR),1,pf))==0)  //Leemos el dato
```

```
     {
      Error ("Error al acceder a memoria.");
     }
     rpg[rt][1]=dato;      //En parte baja del reg. rt metemos el dato
     dato=dato >> 7;   //Desplazamos el dato 7 bits hacia la derecha.
     if (dato!=0)    //Si dato es distinto de cero
       rpg[rt][1]=rpg[rt][1] | 0xFFFFFF00;  //Ponemos el resto del reg. a
uno.
     else if (dato==0)  //Si dato es igual a cero
       rpg[rt][1]=rpg[rt][1] & 0x000000FF;  //Ponemos el resto del reg. a
cero
    }
  }
  else if (Estado("UX")==1)   //Para 64 bits.
  {
    if ((dest0!=0) || (dest0==0 && (destino>65535 || destino<0))) //Si se
sale
    {
      gotoxy (30,23);   //Nos posicionamos
      printf ("Direcci¢n fuera de rango.");  //Informamos
    }
    else  //Si no se sale de los l¡mites.
    {
     fseek (pf,destino,SEEK_SET);  //Nos posicionamos en el fichero de
memoria.
     if ((fread (&dato,sizeof(UCHAR),1,pf))==0)  //Leemos el dato
     {
      Error ("Error al acceder a memoria.");
     }
     rpg[rt][1]=dato;  //Metemos el dato en la parte baja del reg. rt.
     dato=dato >> 7;  //Desplazamos el dato 7 bits hacia la derecha.
     if (dato!=0)   //Si dato es distinto de cero
     {
       rpg[rt][1]=rpg[rt][1] | 0xFFFFFF00;  //Ponemos el resto del reg a unos
       rpg[rt][0]=0xFFFFFFFF;  //La parte alta la ponemos toda a unos
     }
     else if (dato==0)  //Si el dato es igual a cero
     {
       rpg[rt][1]=rpg[rt][1] & 0x000000FF;  //Ponemos el resto de reg a ceros
       rpg[rt][0]=0;     //La parte alta la ponemos toda a ceros.
     }
    }
  }
}    //Fin de la instrucci¢n LB

void LBU (ULONG instruccion)        //Load Byte Unsigned
/*Esta instrucci¢n extiende el signo del offset y se suma al contenido
del registro base para formar una direcci¢n. El contenido del byte en esa
localizaci¢n de memoria hace una extensi¢n con ceros y se guarda en el
registro rt.*/
{
  UCHAR rt,base,dato,aca; //base es el campo de rs, aca y dato: dato y
acarreo
  signed long int off0,offset,dest0,destino;  //Parte alta y baja de offs y
dest
```

```
  ULONG auxi;  //Variable auxiliar

  rt = Campo ("RT",instruccion);  //Cogemos el campo rt de la instrucci¢n
  base = Campo ("RS",instruccion);  //Cogemos el campo base de la instrucci¢n
  offset = instruccion & 0x0000FFFF; //Cogemos el campo offset de la
instrucci¢n
  auxi = offset;  //En auxi metemos el offset
  auxi = auxi >> 15;  //Desplazamos auxi 15 bits hacia la derecha.

  if (auxi!=0)  //Si auxi es distinto de cero
  {
    offset = offset | 0xFFFF0000;  //Hacemos extensi¢n de signo con unos
    off0 = 0xFFFFFFFF;  //Extendemos a unos la parte alta del offset
  }
  else if (auxi==0)   //Si auxi es igual a cero
    off0 = 0;    //Extendemos a ceros la parte alta del offset.

  destino=rpg[base][1]+offset;  //Metemos en destino  reg. base + offset
  dest0 = rpg[base][0]+off0;  //Metemos en dest0 parte alta reg.base+off0
  aca=acarreo(rpg[base][1],offset);  //Vemos si hay acarreo
  if (aca==1)   //Si aca es igual a uno
    dest0++;   //Incrementamos la parte alta de destino en uno.

  if (Estado("UX")==0)  //Para 32 bits.
  {
    if ((destino > 65535) || (destino < 0))  //Si se sale del rango
    {
      gotoxy (30,23);   //Nos posicionamos
      printf ("Direcci¢n fuera de rango."); //Informamos
    }
    else   //Si no se sale del rango de direcciones
    {
     fseek (pf,destino,SEEK_SET);   //Nos posicionamos en el fichero memoria
     if ((fread (&dato,sizeof(UCHAR),1,pf))==0)  //Leemos el dato
     {
      Error ("Error al acceder a memoria.");
     }
     rpg[rt][1]=dato;   //Metemos el dato en el registro
     rpg[rt][1]=rpg[rt][1] & 0x000000FF;  //Extend. el resto del reg con
ceros
    }
  }
  else if (Estado("UX")==1)   //Para 64 bits.
  {
    if ((dest0!=0) || (dest0==0 && (destino>65535 || destino<0))) //Si se
sale
    {
      gotoxy (30,23);   //Nos posicionamos
      printf ("Direcci¢n fuera de rango.");  //Informamos
    }
    else   //Si no se sale del rango de direcciones.
    {
     fseek (pf,destino,SEEK_SET);   //Nos posicionamos en el fichero memoria.
     if ((fread (&dato,sizeof(UCHAR),1,pf))==0)  //Leemos el dato
     {
```

```
      Error ("Error al acceder a memoria.");
     }
     rpg[rt][1]=dato;   //En la parte baja del reg. rt metemos el dato
     rpg[rt][1]=rpg[rt][1] & 0x000000FF;  //Hacemos extensi¢n con ceros
     rpg[rt][0]=0;  //En la parte alta del registro rt metemos todo ceros.
    }
  }
}    //Fin de la instrucci¢n LBU

void LH (ULONG instruccion)  //Load Halfword
/*El offset se hace una extensi¢n de signo y se suma al contenido del
registro base para formar una direcci¢n. El contenido de la doble palabra
en la localizaci¢n de memoria especificada por la direcci¢n efectiva se
extiende el signo y se carga en el registro rt. Si la direcci¢n no es
m£ltiplo de 2 ocurre un error de direccionamiento*/
{
  UCHAR rt,base,dato1,dato2,aca; //base es el campo de rs
  signed long int off0,offset,dest0,destino; //Parte alta y baja de off. y
dest.
  ULONG auxi;  //Variable auxiliar.

  rt = Campo ("RT",instruccion);  //Cogemos el campo rt de la instrucci¢n
  base = Campo ("RS",instruccion);  //Cogemos el campo rs de la instrucci¢n
  offset = instruccion & 0x0000FFFF;  //Cogemos el campo offset de la
instrucc.
  auxi = offset;   //En auxi metemos lo que hay en offset.
  auxi = auxi>> 15;  //Desplazamos 15 bits hacia la derecha.

  if (auxi!=0)  //Si auxi es distinto de cero
  {
    offset = offset | 0xFFFF0000;  //Hacemos la extensi¢n de signo de offset.
    off0 = 0xFFFFFFFF;  //Ponemos la parte alta de offset todo a unos
  }
  else if (auxi==0)  //Si auxi es igual a cero
    off0 = 0;   //Ponemos la parte alta de offset todo a ceros.

  destino=rpg[base][1]+offset;  //Hacemos registro base mas offset
  dest0 = rpg[base][0]+off0;  //Hacemos registro base + off0  (parte alta)
  aca=acarreo (rpg[base][1],offset);  //Miramos si hay acarreo
  if (aca==1)  //Si aca es igual a uno
    dest0++;   //Incrementamos la parte alta del destino en uno.

  if (Estado("UX")==0)  //Para 32 bits.
  {
    if ((destino > 65535) || (destino < 0))  //Si se sale del rango de
memoria
    {
      gotoxy (30,23);   //Nos posicionamos en la pantalla
      printf ("Direcci¢n fuera de rango.");  //Informamos
    }
    else if (destino%2 != 0)  //Si no es multiplo de dos
    {
      General_Exception();    //Error de direccionamiento
      gotoxy (30,23);
      printf ("Address Error exception. ");
```

```
    }
    else  //Si no se sale del rango de memoria y no hay error de direcc.
    {
     fseek (pf,destino,SEEK_SET);  //Nos posicionamos en el fichero de mem.

     if ((fread (&dato1,sizeof(UCHAR),1,pf))==0)  //leemos un byte
     {
      Error ("Error al acceder a memoria.");
     }

     if ((fread (&dato2,sizeof(UCHAR),1,pf))==0)  //Leemos el otro byte
     {
      Error ("Error al acceder a memoria.");
     }

     auxi=0;
     if (Estado("RE")==0)    //Si es little-endian
     {
       auxi=dato2;      //En auxi metemos el dato2
       auxi=auxi<<8;  //Desplazamos auxi 8 bits hacia la izquierda.
       auxi=auxi | dato1;  //Hacemos la or con dato1
       rpg[rt][1]=auxi;   //En la parte baja del reg. rt metemos auxi
     }
     else if (Estado("RE")==1)     //Si es big-endian
     {
       auxi=dato1;  //En auxi metemos el dato1
       auxi=auxi<<8;   //Desplazamos 8 bits hacia la izquierda.
       auxi=auxi | dato2;  //Hacemos la or con dato2
       rpg[rt][1]=auxi;  //En la parte baja del reg. rt metemos auxi
     }
     //Ahora vamos a hacer la extensi¢n de signo
     auxi=auxi >> 15;  //Desplazamos auxi 15 bits hacia la derecha.
     if (auxi!=0)   //Si auxi es distinto de cero
       rpg[rt][1]=rpg[rt][1] | 0xFFFF0000;  //Extendemos signo con unos
     else if (auxi==0)   //Si auxi es igual a cero
       rpg[rt][1]=rpg[rt][1] & 0x0000FFFF;  //Extendemos signo con ceros.
    }
  }
  else if (Estado("UX")==1)   //Para 64 bits.
  {
    if ((dest0!=0) || (dest0==0 && (destino>65535 || destino<0)))//Si se sale
    {
      gotoxy (30,23);
      printf ("Direcci¢n fuera de rango.");
    }
    else if (destino%2 != 0)  //Si la direcci¢n es multiplo de dos
    {
      General_Exception();    //Error de direccionamiento
      gotoxy (30,23);
      printf ("Address Error exception. ");
    }
    else  //Si no se sale del rango de direcciones y no error de direcc.
    {
     fseek (pf,destino,SEEK_SET);  //Se posiciona en el fichero de memoria.
```

```
     if ((fread (&dato1,sizeof(UCHAR),1,pf))==0)  //Lee un byte
     {
      Error ("Error al acceder a memoria.");
     }
     if ((fread (&dato2,sizeof(UCHAR),1,pf))==0)  //Lee el siguiente byte
     {
      Error ("Error al acceder a memoria.");
     }

     auxi=0;
     if (Estado("RE")==0)    //Si es little-endian
     {
       auxi=dato2;       //En auxi mete el dato2
       auxi=auxi<<8;     //Desplaza 8 bits hacia la izquierda
       auxi=auxi | dato1;  //Hace la or de auxi con dato1
       rpg[rt][1]=auxi;    //En la parte baja del registro metemos auxi.
     }
     else if (Estado("RE")==1)     //Si es big-endian
     {
       auxi=dato1;   //En auxi metemos dato1
       auxi=auxi<<8;  //Desplazamos 8 bits hacia la izquierda
       auxi=auxi | dato2;  //Hacemos la or de auxi y dato2
       rpg[rt][1]=auxi;  //En la parte baja del registro metemos auxi.
     }

     auxi=auxi >> 15;     //Desplazamos auxi 15 bits hacia la derecha.
     if (auxi!=0)   //Si auxi es distinto de cero
     {
       rpg[rt][1]=rpg[rt][1] | 0xFFFF0000;  //Se hace la extensi¢n con unos
       rpg[rt][0]=0xFFFFFFFF;  //En la parte alta se pone todo a unos
     }
     else if (auxi==0)  //Si auxi es igual a cero
     {
       rpg[rt][1]=rpg[rt][1] & 0x0000FFFF;  //Se hace la extensi¢n con ceros
       rpg[rt][0]=0;   //En la parte alta se pone todo a ceros
     }
    }
  }
}   //Fin de la instrucci¢n LH

void LHU (ULONG instruccion)    //Load Halfword Unsigned
{
  UCHAR rt,base,dato1,dato2,aca; //base es el campo de rs, datos y acarreo
  signed long int off0,offset,dest0,destino;//Partes alta y baja de off. y
dest.
  ULONG auxi;  //Variable auxiliar.

  rt = Campo ("RT",instruccion);  //Cogemos el campo rt de la instrucci¢n
  base = Campo ("RS",instruccion);  //Cogemos el campo rs de la isntrucci¢n
  offset = instruccion & 0x0000FFFF;  //Cogemos el campo offset de la
instrucc.
  auxi = offset;   //En auxi metemos el contenido de offset
  auxi = auxi >> 15;  //Desplazamos auxi 15 bits hacia la derecha.

  if (auxi!=0)   //Si auxi es distinto de cero
```

```
{
  offset = offset | 0xFFFF0000;  //Hacemos la extensi¢n de offset con unos
  off0 = 0xFFFFFFFF;   //La parte alta de offset lo ponemos todo a unos
}
else if (auxi==0)  //Si auxi es igual a cero
  off0 = 0;  //La parte alta de offset lo ponemos todo a ceros.

destino=rpg[base][1]+offset;  //Hacemos registro base mas desplazamiento
dest0 = rpg[base][0]+off0;   //Hacemos registro base + off0 (parte alta)
aca=acarreo(rpg[base][1],offset);  //Miramos si hay acarreo
if (aca==1)  //Si aca es igual a uno
  dest0++;   //Incrementamos la parte alta del destino.

if (Estado("UX")==0)  //Para 32 bits.
{
  if ((destino > 65535) || (destino < 0))   //Si se sale del rango de dirs.
  {
    gotoxy (30,23);
    printf ("Direcci¢n fuera de rango.");
  }
  else if (destino%2 != 0)  //Si la direcci¢n es m£ltiplo de dos
  {
    General_Exception();    //Error de direccionamiento
    gotoxy (30,23);
    printf ("Address Error exception. ");
  }
  else   //Si no se sale del rango de dirs. y no hay error de direcc.
  {
   fseek (pf,destino,SEEK_SET);  //Nos posicionamos en el fichero memoria.

   if ((fread (&dato1,sizeof(UCHAR),1,pf))==0)  //cogemos un byte
   {
    Error ("Error al acceder a memoria.");
   }

   if ((fread (&dato2,sizeof(UCHAR),1,pf))==0)  //leemos el siguiente byte.
   {
    Error ("Error al acceder a memoria.");
   }

   if (Estado("RE")==0)    //Si es little-endian
   {
     auxi=dato2;   //En auxi metemos dato2
     auxi=auxi<<8;  //Desplazamos 8 bits hacia la izquerda
     auxi=auxi | dato1;  //Hacemos la or de auxi y dato1
     rpg[rt][1]=auxi;  //En la parte baja del registro rt metemos auxi.
   }
   else if (Estado("RE")==1)     //Si es big-endian
   {
     auxi=dato1;    //En auxi metemos dato1
     auxi=auxi<<8;   //Desplazamos 8 bits hacia la izquierda
     auxi=auxi | dato2;  //Hacemos la or de auxi y dato2
     rpg[rt][1]=auxi;  //En la parte baja del registro metemos auxi
   }
   rpg[rt][1]=rpg[rt][1] & 0x0000FFFF;  //Hacemos una extensi¢n con ceros
```

```
    }
  }
  else if (Estado("UX")==1)   //Para 64 bits.
  {
    if ((dest0!=0) || (dest0==0 && (destino>65535 || destino<0)))//Si se sale
    {
      gotoxy (30,23);
      printf ("Direcci¢n fuera de rango.");
    }
    else if (destino%2 != 0)  //Si la direcci¢n es multiplo de dos
    {
      General_Exception();     //Se produce un error de direccionamiento
      gotoxy (30,23);
      printf ("Address Error exception. ");
    }
    else  //Si no se sale del rango y no se produce error de direccionamiento
    {
     fseek (pf,destino,SEEK_SET);  //Nos posicionamos en el fichero de
memoria

     if ((fread (&dato1,sizeof(UCHAR),1,pf))==0)  //leemos el dato1
     {
      Error ("Error al acceder a memoria.");
     }
     if ((fread (&dato2,sizeof(UCHAR),1,pf))==0)  //leemos el dato2
     {
      Error ("Error al acceder a memoria.");
     }

     if (Estado("RE")==0)    //Si es little-endian
     {
       auxi=dato2;   //En auxi metemos dato2
       auxi=auxi<<8;   //Desplazamos 8 bits hacia la izquierda
       auxi=auxi | dato1;   //Hacemos la or de auxi con dato1
       rpg[rt][1]=auxi;  //En la parte baja del registro rt metemos auxi
     }
     else if (Estado("RE")==1)     //Si es big-endian
     {
       auxi=dato1;   //En auxi metemos dato1
       auxi=auxi<<8;  //Desplazamos 8 bits hacia la izquierda
       auxi=auxi | dato2;  //Hacemos la or de auxi y dato2
       rpg[rt][1]=auxi;  //En la parte baja del reg. rt metemos auxi.
     }
     rpg[rt][1]=rpg[rt][1] & 0x0000FFFF; // hacemos una extensi¢n con ceros
     rpg[rt][0]=0;  //ponemos la parte alta del registro destino todo a
ceros.
    }
  }
}    //fin de la instrucci¢n LHU

void LW (ULONG instruccion)  //Load Word
/*Al offset se le hace una extensi¢n de signo y se suma al contenido del
registro base para formar una direcci¢n. El contenido de la palabra en
direcci¢n de memoria especificada por la direcci¢n efectiva se carga en el
registro rt. En modo 64 bits, la palabra cargada se extiende. Si alguno
```

```
de los dos bits menos significativos de la direcci¢n no es cero ocurre un
error de direccionamiento*/
{
  UCHAR rt,base,dato[4],aca; //base es el campo de rs
  signed long int off0,offset,dest0,destino;//parte alta y baja de offs. y
desti.
  ULONG auxi;  //Variable auxiliar
  int i,j;   //Los utilizaremos como ¡ndices.

  rt = Campo ("RT",instruccion);  //Cogemos el campo rt de la instrucci¢n
  base = Campo ("RS",instruccion);  //Cogemos el campo base de la instrucci¢n
  offset = instruccion & 0x0000FFFF;  //Cogemos el campo offset de la
instrucc.
  auxi = offset;   //En auxi metemos lo que hay en offset
  auxi = auxi >> 15;  //Desplazamos lo que hay en auxi 15 bits hacia derecha.

  if (auxi!=0)  //Si auxi es distinto de cero
  {
    offset = offset | 0xFFFF0000;  //Hacemos la extensi¢n de signo de offset
    off0 = 0xFFFFFFFF;  //En la parte alta de offset metemos todo unos
  }
  else if (auxi==0)  //Si auxi es igual a cero
    off0 = 0;   //En la parte alta del offset metemos todo ceros.

  destino=rpg[base][1]+offset;  //Hacemos registro base+ desplazamiento
  dest0 = rpg[base][0]+off0;  //Hacemos reg. base+off0 (parte alta)
  aca=acarreo (rpg[base][1],offset);  //Miramos si hay acarreo.
  if (aca==1)  //Si acarreo es igual a uno
    dest0++;   //Incrementamos la parte alta de la direcci¢n destino.

  if (Estado("UX")==0)  //Para 32 bits.
  {
    if ((destino > 65535) || (destino < 0))  //Si se sale del rango de dirs.
    {
      gotoxy (30,23);
      printf ("Direcci¢n fuera de rango.");
    }
    else if (destino%4 != 0)   //Si la direcci¢n no es m£ltiplo de 4
    {
      General_Exception();     //Error de direccionamiento.
      gotoxy (30,23);
      printf ("Address Error exception. ");
    }
    else  //Si no se sale del rango y si no hay error de direccionamiento
    {
     fseek (pf,destino,SEEK_SET);  //Se situa en el fichero de la memoria
     for (i=0;i<4;i++)
     {
      if (Estado("RE")==0)  //Si es little-endian
      j=3-i;
      else if (Estado("RE")==1)   //Si es big-endian
      j=i;

      if ((fread (&dato[j],sizeof(UCHAR),1,pf))==0)  //Cogemos 4 bytes
      {
```

```
       Error ("Error al acceder a memoria.");
      }
     }

     rpg[rt][1]=chartolong(dato); //En la parte baja del reg. rt mete dato
    }
  }
  else if (Estado("UX")==1)   //Para 64 bits.
  {
    if ((dest0!=0) || (dest0==0 && (destino>65535 || destino<0)))//Si se sale
    {
      gotoxy (30,23);
      printf ("Direcci¢n fuera de rango.");
    }
    else if (destino%4 != 0)  //Si la direcci¢n no es m£ltiplo de 4
    {
      General_Exception();  //Error de direccionamiento
      gotoxy (30,23);
      printf ("Address Error exception. ");
    }
    else   //Si no se sale del rango de direcc. y no hay error de direcc.
    {
     fseek (pf,destino,SEEK_SET);  //Nos posicionamos en el fichero memoria
     for (i=0;i<4;i++)
     {
      if (Estado("RE")==0)  //Si es little-endian
      j=3-i;
      else if (Estado("RE")==1)   //Si es big-endian
      j=i;

      if ((fread (&dato[j],sizeof(UCHAR),1,pf))==0)  //Cogemos 4 bytes
      {
       Error ("Error al acceder a memoria.");
      }
     }
     auxi=chartolong(dato);  //Metemos el dato en auxi (lo convertimos a
long)
     rpg[rt][1]=auxi;  //En la parte baja del reg. rt metemos auxi

     auxi=auxi >> 31;  //Desplazamos 31 bits hacia derecha para ver signo.
     if (auxi!=0)  //Si auxi es distinto de cero
       rpg[rt][0]=0xFFFFFFFF;   //La parte alta del reg. dest. se pone a unos
     else if (auxi==0)  //Si auxi es igual a cero
       rpg[rt][0]=0;  //La parte alta del reg. destino se pone toda a ceros.
    }
  }
}    //fin de la instrucci¢n LW

void LWL (ULONG instruccion)    //Load Word Left
/*Esta instrucci¢n puede usarse en combinaci¢n con la instrucci¢n LWR para
cargar un registro con cuatro bytes consecutivos desde memoria cuando los
bytes atraviesan el l¡mite de palabra. LWL carga la parte izquierda del
registro con la parte correspondiente de mayor peso de la palabra. En modo
64 bits, la palabra cargada hace una extensi¢n de signo.*/
{
```

```
  UCHAR rt,base,dato[4],desplaz,aca; //Para reg, dato, desplazam y acarreo
  signed long int off0,offset,dest0,destino;//Parte alta y baja de offs. y
dest.
  ULONG auxi; //Variable auxiliar.
  int i,j;  //Se utilizan como ¡ndice.

  rt = Campo ("RT",instruccion);  //Cogemos el campo rt de la instrucci¢n
  base = Campo ("RS",instruccion);  //Cogemos el campo base de la instrucci¢n
  offset = instruccion & 0x0000FFFF; //Cogemos el campo offset de la instr.
  auxi = offset;  //En auxi metemos el contenido de offset
  auxi = auxi >> 15;  //Desplazamos 15 bits hacia la derecha.

  if (auxi!=0)  //Si auxi es distinto de cero
  {
    offset = offset | 0xFFFF0000;  //Hacemos la extensi¢n de signo con unos
    off0 = 0xFFFFFFFF;  //Ponemos la parte alta de offset todo a unos
  }
  else if (auxi==0)  //Si auxi es igual a cero
    off0 = 0;   //Ponemos la parte alta de offset a cero.

  destino=rpg[base][1]+offset;  //Hacemos registro base mas desplazamiento
  dest0 = rpg[base][0]+off0;  //Hacemos reg. base + off0 (parte alta).
  aca=acarreo(rpg[base][1],offset);  //Miramos si hay acarreo
  if (aca==1)   //Si aca es igual a uno
    dest0++;  //Incrementamos la parte alta de destino en uno.

  desplaz=destino & 0x00000003;//Cogemos el desplazamiento a partir del
l¡mite
  destino=destino & 0xFFFFFFFC;//Cogemos el l¡mite de la palabra.

  if (Estado("UX")==0)  //Para 32 bits.
  {
    if ((destino > 65535) || (destino < 0))  //Si se sale del l¡mite.
    {
      gotoxy (30,23);
      printf ("Direcci¢n fuera de rango.");
    }
    else   //Si no se sale del l¡mite
    {
     fseek (pf,destino,SEEK_SET);  //Nos situamos en el fichero destino
     for (i=0;i<4;i++)
     {
      if (Estado("RE")==0)  //Si es little-endian
      j=3-i;
      else if (Estado("RE")==1)   //Si es big-endian
      j=i;

      if ((fread (&dato[j],sizeof(UCHAR),1,pf))==0)  //Cogemos 4 bytes
      {
       Error ("Error al acceder a memoria.");
      }
     }
     auxi=chartolong(dato);    //Lo pasamos a long y lo metemos en auxi
     if (Estado("RE")==0)  //Si es little-endian
       auxi=auxi<<((3-desplaz)*8);  //Desplazamos
```

```
     else if (Estado("RE")==1)  //Si es Big-endian
       auxi=auxi<<(desplaz*8);  //Desplazamos
     if (desplaz==0)  //Si desplazamiento es igual a cero
     {
       if (Estado("RE")==1) rpg[rt][1]=auxi;  //En reg. rt metemos auxi
       else if (Estado("RE")==0)  //Si es little-endian
       {
       rpg[rt][1]=rpg[rt][1] & 0x00FFFFFF;  //hacemos and con parte baja
       rpg[rt][1]=rpg[rt][1] | auxi;  //Hacemos or con auxi
       }
     }
     else if (desplaz==1)  //Si desplazamiento es igual a uno
     {
       if (Estado("RE")==1) rpg[rt][1]=rpg[rt][1] & 0x000000FF;//Si Big-
endian
       else if (Estado("RE")==0) rpg[rt][1]=rpg[rt][1] & 0x0000FFFF;//Little-
endian
       rpg[rt][1]=rpg[rt][1] | auxi; //Hacemos or de parte baja con auxi
     }
     else if (desplaz==2)
     {
       if (Estado("RE")==1) rpg[rt][1]=rpg[rt][1] & 0x0000FFFF;//Si Big-
endian
       else if (Estado("RE")==0) rpg[rt][1]=rpg[rt][1] & 0x000000FF;//Little
       rpg[rt][1]=rpg[rt][1] | auxi;  //Hacemos or de parte baja con auxi
     }
     else if (desplaz==3)  //Si el desplazamiento es igual a tres
     {
       if (Estado("RE")==1)  //Si Big-endian
       {
       rpg[rt][1]=rpg[rt][1] & 0x00FFFFFF;  //Hacemos la and con parte baja
       rpg[rt][1]=rpg[rt][1] | auxi;  //Hacemos la or de auxi con parte baja
       }
       else if (Estado("RE")==0) rpg[rt][1]=auxi; //En parte baja metemos
auxi
     }
    }
  }
  else if (Estado("UX")==1)   //Para 64 bits.
  {
    if ((dest0!=0) || (dest0==0 && (destino>65535 || destino<0)))//Si se sale
    {
      gotoxy (30,23);
      printf ("Direcci¢n fuera de rango.");
    }
    else   //Si no se sale de los l¡mites de la memoria
    {
     fseek (pf,destino,SEEK_SET);  //Nos situamos en el fichero destino
     for (i=0;i<4;i++)
     {
      if (Estado("RE")==0)  //Si es little-endian
      j=3-i;
      else if (Estado("RE")==1)   //Si es big-endian
      j=i;
```

```
      if ((fread (&dato[j],sizeof(UCHAR),1,pf))==0)  //Leemos 4 bytes
      {
       Error ("Error al acceder a memoria.");
      }
     }
     auxi=chartolong(dato);  //Pasamos la palabra leida a long
     if (Estado("RE")==0)  //Si es little-endian
       auxi=auxi<<((3-desplaz)*8);  //Desplazamos
     else if (Estado("RE")==1)  //Si es big-endian
       auxi=auxi<<(desplaz*8);  //Desplazamos
     if (desplaz==0)    //Si el desplazamiento es igual a cero
     {
       if (Estado("RE")==1) rpg[rt][1]=auxi;  //Metemos en parte baja auxi
       else if (Estado("RE")==0)  //Si es little-endian
       {
       rpg[rt][1]=rpg[rt][1] & 0x00FFFFFF;  //Hacemos and con parte baja
       rpg[rt][1]=rpg[rt][1] | auxi; //Hacemos or de auxi con parte baja
       }
     }
     else if (desplaz==1)  //Si desplazamiento es igula a uno
     {
       if (Estado("RE")==1) rpg[rt][1]=rpg[rt][1] & 0x000000FF;
       else if (Estado("RE")==0) rpg[rt][1]=rpg[rt][1] & 0x0000FFFF;
       rpg[rt][1]=rpg[rt][1] | auxi;  //Hacemos or de auxi con parte baja
     }
     else if (desplaz==2)  //Si desplazamiento es igual a dos
     {
       if (Estado("RE")==1) rpg[rt][1]=rpg[rt][1] & 0x0000FFFF;
       else if (Estado("RE")==0) rpg[rt][1]=rpg[rt][1] & 0x000000FF;
       rpg[rt][1]=rpg[rt][1] | auxi;  //Hacemos or de auxi con parte baja
     }
     else if (desplaz==3)   //Si desplazamiento es igual a tres
     {
       if (Estado("RE")==1)  //Si es Big-endian
       {
       rpg[rt][1]=rpg[rt][1] & 0x00FFFFFF;  //Hacemos and con parte baja
       rpg[rt][1]=rpg[rt][1] | auxi;  //Hacemos or de parte baja y auxi
       }
       else if (Estado("RE")==0) rpg[rt][1]=auxi; //Metemos auxi en parte
baja
     }

     auxi=rpg[rt][1] >> 31; //En auxi metemos parte baja desplaz. 31 bits
drcha
     if (auxi!=0)  //Si auxi es distinto de cero
       rpg[rt][0]=0xFFFFFFFF;  //En la parte alta del reg. metemos todo unos
     else if (auxi==0)  //Si auxi es igual a cero
       rpg[rt][0]=0;  //Metemos en la parte alta todo ceros.
    }
  }
}      //fin de la instrucci¢n LWL

void LWR (ULONG instruccion)  //Load Word Right
/*Esta instrucci¢n puede usarse en combinaci¢n con la instrucci¢n LWL para
cargar un registro con cuatro bytes consecutivos desde memoria cuando los
```

```
bytes cruzan un l¡mite de palabra. LWR carga la parte derecha del registro
con la parte correspondiente de orden bajo de la palabra.*/
{
  UCHAR rt,base,dato[4],desplaz,aca; //Para registro, dato, desplazam, y
acarreo
  signed long int off0,offset,dest0,destino;//Parte alta y baja de offs. y
dest.
  ULONG auxi;  //Variable auxiliar.
  int i,j;   //Para los ¡ndices.

  rt = Campo ("RT",instruccion);  //Cogemos el campo rt de la instrucci¢n
  base = Campo ("RS",instruccion);  //Cogemos el campo base de la instrucci¢n
  offset = instruccion & 0x0000FFFF;  //Cogemos el offset de la instrucci¢n
  auxi = offset;   //En auxi metemos lo que hay en offset
  auxi = auxi >> 15;  //Desplazamos 15 bits hacia la derecha.

  if (auxi!=0)  //Si auxi es distinto de cero
  {
    offset = offset | 0xFFFF0000;  //Hacemos la extensi¢n de signo con unos
    off0 = 0xFFFFFFFF;  //En la parte alta de offset ponemos todo unos
  }
  else if (auxi==0)  //Si auxi es igual a cero
    off0 = 0;  //En la parte alta de offset ponemos todo ceros.

  destino=rpg[base][1]+offset;  //Hacemos registro base mas desplazamiento
  dest0 = rpg[base][0]+off0;  //Hacemos reg. base + off0 (parte alta).
  aca=acarreo(rpg[base][1],offset); //Miramos si hay acarreo
  if (aca==1)  //Si aca es igual a uno
    dest0++;  //Incrementamos la parte alta de destino en uno

  desplaz=destino & 0x00000003;  //Cogemos es desplazamiento desde el l¡mite.
  destino=destino & 0xFFFFFFFC;  //Cogemos el l¡mite de palabra.

  if (Estado("UX")==0)  //Para 32 bits.
  {
    if ((destino > 65535) || (destino < 0))  //Si se sale del rango de direc.
    {
      gotoxy (30,23);
      printf ("Direcci¢n fuera de rango.");
    }
    else   //Si no se sale del rango de direcciones
    {
     fseek (pf,destino,SEEK_SET);  //Nos posicionamos en el fichero memoria
     for (i=0;i<4;i++)
     {
      if (Estado("RE")==0)  //Si es little-endian
      j=3-i;
      else if (Estado("RE")==1)   //Si es big-endian
      j=i;

      if ((fread (&dato[j],sizeof(UCHAR),1,pf))==0)  //Cogemos 4 bytes.
      {
       Error ("Error al acceder a memoria.");
      }
     }
```

```
     auxi=chartolong(dato);  //En auxi metemos el dato convertido a long
     if (Estado("RE")==0)  //Si es little-endian
       auxi=auxi>>(desplaz*8);  //Desplazamos
     else if (Estado("RE")==1)  //Si es Big-endian
       auxi=auxi>>((3-desplaz)*8);  //Desplazamos
     if (desplaz==0)  //Si desplazamiento es igual a cero
     {
       if (Estado("RE")==1)  //Si es Big-endian
       {
       rpg[rt][1]=rpg[rt][1] & 0xFFFFFF00;  //Hacemos and con parte baja
       rpg[rt][1]=rpg[rt][1] | auxi;  //Hacemos or de auxi con parte baja
       }
       else if (Estado("RE")==0)  rpg[rt][1]=auxi;  //En parte baja metemos
auxi
     }
     else if (desplaz==1)  //Si desplazamiento es igual a uno
     {
       if (Estado("RE")==1) rpg[rt][1]=rpg[rt][1] & 0xFFFF0000;
       else if (Estado("RE")==0) rpg[rt][1]=rpg[rt][1] & 0xFF000000;
       rpg[rt][1]=rpg[rt][1] | auxi;  //hacemos la or de auxi con parte baja
     }
     else if (desplaz==2)   //Si desplazamiento es igual a dos
     {
       if (Estado("RE")==1) rpg[rt][1]=rpg[rt][1] & 0xFF000000;
       else if (Estado("RE")==0) rpg[rt][1]=rpg[rt][1] & 0xFFFF0000;
       rpg[rt][1]=rpg[rt][1] | auxi;  //Hacemos la or de auxi con parte baja
     }
     else if (desplaz==3)  //Si desplazamiento es igual a tres
     {
       if (Estado("RE")==1) rpg[rt][1]=auxi;  //En parte baja metemos auxi
       else if (Estado("RE")==0)  //Si es little endian
       {
       rpg[rt][1]=rpg[rt][1] & 0xFFFFFF00; //Hacemos and con parte baja
       rpg[rt][1]=rpg[rt][1] | auxi;  //Hacemos or de auxi con parte baja
       }
     }
    }
  }
  else if (Estado("UX")==1)   //Para 64 bits.
  {
    if ((dest0!=0) || (dest0==0 && (destino>65535 || destino<0)))//Si se sale
    {
      gotoxy (30,23);
      printf ("Direcci¢n fuera de rango.");
    }
    else  //Si no se sale del rango de direcciones
    {
     fseek (pf,destino,SEEK_SET);  //Nos situamos en el fichero de memoria
     for (i=0;i<4;i++)
     {
      if (Estado("RE")==0)  //Si es little-endian
      j=3-i;
      else if (Estado("RE")==1)   //Si es big-endian
      j=i;
```

```
      if ((fread (&dato[j],sizeof(UCHAR),1,pf))==0)  //Leemos 4 bytes
      {
       Error ("Error al acceder a memoria.");
      }
     }
     auxi=chartolong(dato);  //En auxi metemos el dato convertido a long
     if (Estado("RE")==0)  //Si es little-endian
       auxi=auxi>>(desplaz*8);  //Desplazamos
     else if (Estado("RE")==1)  //Si es Big-endian
       auxi=auxi>>((3-desplaz)*8);  //Desplazamos

     if (desplaz==0)  //Si desplazamiento es igual a cero
     {
       if (Estado("RE")==1)  //Si es Big-endian
       {
       rpg[rt][1]=rpg[rt][1] & 0xFFFFFF00;  //Hacemos and con parte baja
       rpg[rt][1]=rpg[rt][1] | auxi;  //Hacemos or de parte baja y auxi
       }
       else if (Estado("RE")==0)  rpg[rt][1]=auxi;  //En parte baja metemos
auxi
     }
     else if (desplaz==1)  //Si desplazamiento es igual a uno
     {
       if (Estado("RE")==1) rpg[rt][1]=rpg[rt][1] & 0xFFFF0000;
       else if (Estado("RE")==0) rpg[rt][1]=rpg[rt][1] & 0xFF000000;
       rpg[rt][1]=rpg[rt][1] | auxi;  //Hacemos or de auxi y parte baja
     }
     else if (desplaz==2)  //Si desplazamiento es igual a dos
     {
       if (Estado("RE")==1) rpg[rt][1]=rpg[rt][1] & 0xFF000000;
       else if (Estado("RE")==0) rpg[rt][1]=rpg[rt][1] & 0xFFFF0000;
       rpg[rt][1]=rpg[rt][1] | auxi;  //Hacemos or de parte baja y auxi
     }
     else if (desplaz==3)  //Si desplazamiento es igual a tres
     {
       if (Estado("RE")==1) rpg[rt][1]=auxi;  //Si es big-endian
       else if (Estado("RE")==0)  //Si es little endian
       {
       rpg[rt][1]=rpg[rt][1] & 0xFFFFFF00;  //Hacemos and con parte baja
       rpg[rt][1]=rpg[rt][1] | auxi;  //Hacemos or de parte baja y auxi
       }
     }
      //Ahora vamos a hacer la extensi¢n de signo
     auxi=rpg[rt][1] >> 31;  //En auxi metemos parte baja desplazada 31 bits.
     if (auxi!=0)  //Si auxi es distinto de cero
       rpg[rt][0]=0xFFFFFFFF;  //En parte alta metemos todo a unos
     else if (auxi==0)  //Si auxi es igual a cero
       rpg[rt][0]=0;  //En parte alta metemos todo a ceros
    }
  }
}   //Fin de la instrucci¢n LWR

void LWU (ULONG instruccion)      //Load Word Unsigned
/*Al offset se le extiende el signo y se suma al contenido del registro
base para formar una direcci¢n. El contenido de la palabra de la direcci¢n
```

```
especificada se carga en el registro rt. La palabra cargada se extiende con
ceros. La direcci¢n debe ser m£ltiplo de 4. Si se ejecuta en modo 32 bits
se produce una address error exception (error de direccionamiento)*/
{
  UCHAR rt,base,dato[4],aca; //base es el campo de rs, dato y acarreo
  signed long int off0,offset,dest0,destino;//Parte alta y baja de offs. y
dest.
  ULONG auxi;  //Variable auxiliar.
  int i,j;  //Las utilizamos como ¡ndice.

  rt = Campo ("RT",instruccion);  //Cogemos el campo rt de la instrucci¢n
  base = Campo ("RS",instruccion);  //Cogemos el campo base de la instrucci¢n
  offset = instruccion & 0x0000FFFF;  //Cogemos el offset de la instrucci¢n
  auxi = offset;  //En auxi metemos lo que hay en la variable offset
  auxi = auxi >> 15;  //Desplazamo 15 bits hacia la derecha.

  if (auxi!=0)  //Si auxi es distinto de cero
  {
    offset = offset | 0xFFFF0000;  //Se extiende el signo con unos
    off0 = 0xFFFFFFFF;  //La parte alta del offset se pone todo a unos
  }
  else if (auxi==0)  //Si auxi es igual a cero
    off0 = 0;  //La parte alta del offset se pone todo a ceros

  destino=rpg[base][1]+offset;  //Hacemos registro base mas desplazamiento
  dest0 = rpg[base][0]+off0;  //Hacemos reg. base + off0 (parte alta)
  aca=acarreo(rpg[base][1],offset); //Miramos si hay acarreo
  if (aca==1)  //Si aca es igual a uno
    dest0++;   //Incrementamos la parte alta del destino en uno

  if (Estado("UX")==0)  //Para 32 bits.
  {
    General_Exception();        //Reserved instruction exception
    gotoxy (30,23);
    printf ("Reserved Instr. exception");
  }
  else if (Estado("UX")==1)  //Para 64 bits.
  {
    if ((dest0!=0) || (dest0==0 && (destino>65535 || destino<0)))//Si se sale
    {
      gotoxy (30,23);
      printf ("Direcci¢n fuera de rango.");
    }
    else if (destino%4 != 0)  //Si no es multiplo de 4
    {
      General_Exception();   //Error de direccionamiento
      gotoxy (30,23);
      printf ("Address Error exception. ");
    }
    else  //Si no se sale del rango de direcc y no hay error de
direccionamiento
    {
     fseek (pf,destino,SEEK_SET); //Nos situamos en el fichero memoria
     for (i=0;i<4;i++)
     {
```

```
      if (Estado("RE")==0)  //Si es little-endian
      j=3-i;
      else if (Estado("RE")==1)   //Si es big-endian
      j=i;

      if ((fread (&dato[j],sizeof(UCHAR),1,pf))==0)  //Leemos 4 bytes
      {
       Error ("Error al acceder a memoria.");
      }
     }
     auxi=chartolong(dato);  //En auxi metemos el dato pasado a long
     rpg[rt][1]=auxi;  //En la parte baja del registro destino metemos auxi
     rpg[rt][0]=0x00000000;  //La parte alta del reg. destino todo a ceros.
    }
  }
}    //Fin de la instrucci¢n LWU

void LD (ULONG instruccion)  //Load Doubleword
/*Hace una extensi¢n del signo de offset y se suma al contenido del registro
base para formar una direcci¢n. El contenido de la doble palabra en la
direcci¢n de memoria especificada por la direcci¢n se carga en el registro
rt. Ocurre un error de direccionamiento si la direcci¢n no es m£ltiplo de 8.
Si se ejecuta en modo 32 bits usuario o supervisor, se produce una reserved
instruction exception*/
{
  UCHAR rt,base,aca; //base es el campo de rs
  signed long int off0,offset,dest0,destino;//Parte alta y baja de off. y
dest.
  ULONG auxi;   //Variable auxiliar.
  ULONG datomenorpeso,datomayorpeso; //Para el dato de mayor y menor peso.
  UCHAR doble0[4],doble1[4];   //Para coger los datos de memoria.
  int i,j;   //Para utilizarlas como ¡ndices.

  rt = Campo ("RT",instruccion);  //Cogemos el campo rt de la instrucci¢n
  base = Campo ("RS",instruccion);  //Cogemos el campo base de la instrucci¢n
  offset = instruccion & 0x0000FFFF;  //Cogemos el campo offset de la instr.
  auxi = offset; //En auxi metemos lo que hay en la variable offset
  auxi = auxi >> 15;  //Desplazamos 15 bits hacia la derecha

  if (auxi!=0) //Si auxi es distinto de cero
  {
    offset = offset | 0xFFFF0000;  //Hacemos una extensi¢n de signo a unos
    off0 = 0xFFFFFFFF;  //Ponemos la parte alta de offset a unos.
  }
  else if (auxi==0)   //Si auxi es igual a cero
    off0 = 0;   //Ponemos la parte alta de offset a ceros

  destino=rpg[base][1]+offset; //Sumamos registro base + offset
  dest0 = rpg[base][0]+off0;   //Sumamos registro base + off0 (parte alta).
  aca=acarreo(rpg[base][1],offset);  //Miramos si hay acarreo
  if (aca==1)  //Si aca es igual a uno
    dest0++;   //Incrementamos la parte alta de la direcci¢n destino

  if ((Estado("UX")==0) && ((Estado("KSU")==1) || (Estado("KSU")==2))) //Para
32 bits.
```

```
{
  General_Exception();        //Reserved instruction exception
                      //Si modo kernel no se ejecuta excepcion
  gotoxy (30,23);
  printf ("Reserved Instr. exception");
}
else if ((Estado("UX")==1) || (Estado ("KSU")==0)) //Para 64 bits O kERNEL.
{
  if ((dest0!=0) || (dest0==0 && (destino>65535 || destino<0)))//Si se sale
  {
    gotoxy (30,23);   //Nos posicionamos
    printf ("Direcci¢n fuera de rango.");  //Informamos
  }
  else if (destino%8 != 0)   //Si la direcci¢n destino no es multiplo de 8
  {
    General_Exception();   //Se produce un error de direccionamiento
    gotoxy (30,23);
    printf ("Address Error exception. ");
  }
  else    //Si no se sale del rango de direcc. y no hay error de direcc.
  {
   fseek (pf,destino,SEEK_SET);  //Nos posicionamos en el fichero memoria
   for (i=0;i<4;i++)
   {
    if (Estado("RE")==0)  //Si es little-endian
    j=3-i;
    else if (Estado("RE")==1)   //Si es big-endian
    j=i;
    if ((fread (&doble0[j],sizeof(UCHAR),1,pf))==0)  //Cogemos 4 bytes
    {
     Error ("Error al acceder a memoria.");
    }
   }
   for (i=0;i<4;i++)
   {
    if (Estado("RE")==0)  //Si es little-endian
    j=3-i;
    else if (Estado("RE")==1)   //Si es big-endian
    j=i;
    if ((fread (&doble1[j],sizeof(UCHAR),1,pf))==0)  //leemos otros 4 bytes
    {
     Error ("Error al acceder a memoria.");
    }
   }

   if (Estado("RE")==0)    //Si es little-endian
   {
     datomenorpeso=chartolong(doble0);  //Dato de menor peso es doble0
     datomayorpeso=chartolong(doble1);  //Dato de mayor peso es doble1
   }
   else if (Estado("RE")==1)    //Si el big-endian
   {
     datomayorpeso=chartolong(doble0);  //Dato de mayor peso es doble0
     datomenorpeso=chartolong(doble1);  //Dato de menor peso es doble1
   }
```

```
     rpg[rt][0]=datomayorpeso;   //Metemos en parte alta de reg.
datomayorpeso
     rpg[rt][1]=datomenorpeso;   //Metemos en parte baja de reg.
datomenorpeso
    }
  }
}   //Fin de la instrucci¢n LD

void LDL (ULONG instruccion)   //Load Doubleword Left
/*Esta instrucci¢n puede ser usada en combinaci¢n con la instrucci¢ LDR para
cargar un registro con ocho bytes consecutivos de memoria, cuando los bytes
cruzan un l¡mite de doble palabra. En modo 32 bits causa una reserved
instruction exception*/
{
  UCHAR rt,base,desplaz,aca; //base es el campo de rs, desplazamiento y
acarreo
  signed long int off0,offset,dest0,destino;  //parte alta y baja de offs. y
dest.
  ULONG datomenorpeso,datomayorpeso,auxi;  //dato de mayor y menor peso y
auxiliar
  UCHAR doble0[4],doble1[4];  //Para coger los bytes de la memoria
  int i,j;   //Variables que se utilizan como ¡ndice.

  rt = Campo ("RT",instruccion);  //Cogemos el campo rt de la instrucci¢n
  base = Campo ("RS",instruccion);  //Cogemos el campo base de la instrucci¢n
  offset = instruccion & 0x0000FFFF;  //Cogemos el campo offset de la instr.
  auxi = offset;   //En auxi metemos el contenido de la variable offset.
  auxi = auxi >> 15;  //Desplazamos auxi 15 bits hacia la derecha.

  if (auxi!=0)  //Si auxi es distinto de cero
  {
    offset = offset | 0xFFFF0000; //Extendemos el signo con unos
    off0 = 0xFFFFFFFF;  //La parte alta del offset la ponemos toda a unos
  }
  else if (auxi==0)  //Si auxi es igual a cero
    off0 = 0;   //La parte alta del offset la ponemos toda a ceros.

  destino=rpg[base][1]+offset; //hacemos registro base + desplazamiento
  dest0 = rpg[base][0]+off0; //hacemos registro base+off0 (parte alta.)
  aca=acarreo(rpg[base][1],offset);  //Miramos si hay acarreo
  if (aca==1)  //Si aca es igual a uno
    dest0++;  //Incrementamos la parte alta del destino en uno.

  desplaz=destino & 0x00000007;  //Cogemos el desplazamiento desde el l¡mite.
  destino=destino & 0xFFFFFFF8;  //Cogemos el l¡mite de dobleword m s
pr¢ximo.

  if (Estado("UX")==0)  //Para 32 bits.
  {
    General_Exception();        //Reserved instruction exception
    gotoxy (30,23);
    printf ("Reserved Instr. exception");
  }
  else if (Estado("UX")==1)  //Para 64 bits.
```

```
{
  if ((dest0!=0) || (dest0==0 && (destino>65535 || destino<0)))//Si se sale
  {
    gotoxy (30,23);   //Nos situamos
    printf ("Direcci¢n fuera de rango.");  //Informamos
  }
  else  //Si no se sale de los l¡mites de la memoria.
  {
   fseek (pf,destino,SEEK_SET);  //Nos situamos en el fichero de la memoria
   for (i=0;i<4;i++)
   {
    if (Estado("RE")==0)  //Si es little-endian
    j=3-i;
    else if (Estado("RE")==1)   //Si es big-endian
    j=i;
    if ((fread (&doble0[j],sizeof(UCHAR),1,pf))==0)  //Cogemos 4 bytes
    {
     Error ("Error al acceder a memoria.");
    }
   }
   for (i=0;i<4;i++)
   {
    if (Estado("RE")==0)  //Si es little-endian
    j=3-i;
    else if (Estado("RE")==1)   //Si es big-endian
    j=i;
    if ((fread (&doble1[j],sizeof(UCHAR),1,pf))==0)  //Cogemos 4 bytes
    {
     Error ("Error al acceder a memoria.");
    }
   }

   if (Estado("RE")==0)    //Si es little-endian
   {
     datomenorpeso=chartolong(doble0); //En datomenorpeso metemos doble0
     datomayorpeso=chartolong(doble1); //En datomayorpeso metemos doble1
     if (desplaz==0)   //Si el desplazamiento es cero
     {
     datomenorpeso=datomenorpeso<<24;  //Desplazamos 24 bits hacia izq.
     rpg[rt][0]=rpg[rt][0] & 0x00FFFFFF;  //Hacemos la and en parte alta
     rpg[rt][0]=rpg[rt][0] | datomenorpeso;  //Hacemos or con datomenorp.
     }
     else if (desplaz==1)  //Si el desplazamiento es uno
     {
     datomenorpeso=datomenorpeso<<16;  //Se desplaza 16 bits hacia la izq.
     rpg[rt][0]=rpg[rt][0] & 0x0000FFFF;  //Hacemos la and en parte alta
     rpg[rt][0]=rpg[rt][0] | datomenorpeso;  //Hacemos or con datomenorp
     }
     else if (desplaz==2)  //Si el desplazamiento es dos
     {
     datomenorpeso=datomenorpeso<<8;  //Desplazamos 8 bits hacia la izq.
     rpg[rt][0]=rpg[rt][0] & 0x000000FF;  //Hacemos la and en parte alta
     rpg[rt][0]=rpg[rt][0] | datomenorpeso; //hacemos la or con datomenorp
     }
```

```
       else if (desplaz==3) rpg[rt][0]=datomenorpeso; //Si desplaz==3 mete
todo
       else if (desplaz==4)  //Si desplazamiento es igual a cuatro
       {
       auxi=datomenorpeso<<24;  //En auxi metemos datomenorpeso desp 24 bits
       rpg[rt][1]=rpg[rt][1] & 0x00FFFFFF; //Hacemos la and en parte baja
       datomenorpeso=datomenorpeso>>8; //Desplazamos 8 bits hacia la derecha
       datomayorpeso=datomayorpeso<<24; //Desplazamos 24 hacia la izq.
       rpg[rt][0]=datomenorpeso | datomayorpeso;  //hacemos or de uno y otro
       rpg[rt][1]=rpg[rt][1] | auxi;  //Hacemos la or de parte baja y auxi
       }
       else if (desplaz==5)   //Si desplazamiento es igual a cinco
       {
       auxi=datomenorpeso<<16;   //En auxi metemos dato menorpeso desplazado
       rpg[rt][1]=rpg[rt][1] & 0x0000FFFF; //Hacemos and con parte baja
       datomenorpeso=datomenorpeso>>16;  //Desplazamos 16 bits hacia derecha
       datomayorpeso=datomayorpeso<<16;  //Desplazamos 16 bits hacia izq.
       rpg[rt][0]=datomenorpeso | datomayorpeso; //Hacemos or de uno y otro
       rpg[rt][1]=rpg[rt][1] | auxi;  //Hacemos or de parte baja y auxi.
       }
       else if (desplaz==6)  //Si desplazamiento es igual a seis
       {
       auxi=datomenorpeso<<8;  //En auxi metemos datomenorpeso desplazado.
       rpg[rt][1]=rpg[rt][1] & 0x000000FF;  //Hacemos la and con parte baja
       datomenorpeso=datomenorpeso>>24; //Desplazamos 24 bits hacia derecha.
       datomayorpeso=datomayorpeso<<8;  //Desplazamos 8 hacia izquierda.
       rpg[rt][0]=datomenorpeso | datomayorpeso;  //Hacemos or de uno y otro
       rpg[rt][1]=rpg[rt][1] | auxi;  //Hacemos or de parte baja y auxi.
       }
       else if (desplaz==7)  //Si desplazamiento es igual a siete
       {
       rpg[rt][0]=datomayorpeso;  //En la parte alta metemos el datomayorp.
       rpg[rt][1]=datomenorpeso;  //En la parte baja mentemos datomenorpeso.
       }
     }
     else if (Estado("RE")==1)    //Si el big-endian
     {
       datomayorpeso=chartolong(doble0); //En datomayorpeso metemos doble0
       datomenorpeso=chartolong(doble1); //En datomenorpeso metemos doble1
       if (desplaz==0)  //Si el desplazamiento es igual a cero
       {
       rpg[rt][0]=datomayorpeso;  //En parte alta metemos datomayorpeso
       rpg[rt][1]=datomenorpeso;  //En parte baja metemos datomenorpeso
       }
       else if (desplaz==1)  //Si el desplazamiento es igual a uno
       {
       auxi=datomenorpeso<<8;  //En auxi metemos datomenorpeso desplazado
       rpg[rt][1]=rpg[rt][1] & 0x000000FF;  //Hacemos and con parte baja
       datomenorpeso=datomenorpeso>>24;  //Desplazamos hacia derecha 24 bits
       datomayorpeso=datomayorpeso<<8;  //Desplazamos hacia izq. 8 bits.
       rpg[rt][0]=datomenorpeso | datomayorpeso; //Hacemos la or de uno y
otro
       rpg[rt][1]=rpg[rt][1] | auxi;  //Hacemos la or de menorpeso y auxi
       }
       else if (desplaz==2)  //Si desplazamiento es igual a dos
```

```
      {
      auxi=datomenorpeso<<16;  //En auxi metemos datomenorpeso desplazado
      rpg[rt][1]=rpg[rt][1] & 0x0000FFFF;  //Hacemos and con parte baja
      datomenorpeso=datomenorpeso>>16; //Desplazamos 16 bits hacia derecha
      datomayorpeso=datomayorpeso<<16;  //Desplazamos 16 bits hacia izq.
      rpg[rt][0]=datomenorpeso | datomayorpeso;  //Hacemos or de ambas.
      rpg[rt][1]=rpg[rt][1] | auxi;  //Hacemos or de menor peso y auxi
      }
      else if (desplaz==3)   //Si desplazamiento es igual a tres
      {
      auxi=datomenorpeso<<24;  //En auxi metemos datomenorpeso desplazado.
      rpg[rt][1]=rpg[rt][1] & 0x00FFFFFF;  //Hacemos la and con parte baja
      datomenorpeso=datomenorpeso>>8;  //Desplazamos 8 bits hacia derecha
      datomayorpeso=datomayorpeso<<24;  //Desplazamos 24 bits hacia izq.
      rpg[rt][0]=datomenorpeso | datomayorpeso;  //Hacemos or de uno y otro.
      rpg[rt][1]=rpg[rt][1] | auxi; //Hacemos or de parte baja y auxi
      }
      else if (desplaz==4) rpg[rt][0]=datomenorpeso; //Si es 4 mete todo
      else if (desplaz==5)  //Si desplazamiento es igual a cinco
      {
      datomenorpeso=datomenorpeso<<8;  //Desplazamos datomenorp 8 bits izq.
      rpg[rt][0]=rpg[rt][0] & 0x000000FF; //Hacemos la and con parte alta
      rpg[rt][0]=rpg[rt][0] | datomenorpeso; //Hacemos or con datomenorpeso
      }
      else if (desplaz==6)  //Si desplazamiento es igual a seis
      {
      datomenorpeso=datomenorpeso<<16;  //Desplazamos datomenorpso 16 izq.
      rpg[rt][0]=rpg[rt][0] & 0x0000FFFF; //Hacemos and con parte alta.
      rpg[rt][0]=rpg[rt][0] | datomenorpeso; //Hacemos or con datomenorpso
      }
      else if (desplaz==7) //Y si el desplazamiento es igual a siete.
      {
      datomenorpeso=datomenorpeso<<24;  //Desplazamos 24 bits hacia izq.
      rpg[rt][0]=rpg[rt][0] & 0x00FFFFFF; //Hacemos and con parte alta
      rpg[rt][0]=rpg[rt][0] | datomenorpeso;  //Hacemos or con datomenorpso
      }
     }
    }
  }
}   //fin de la instrucci¢n LDL

void LDR (ULONG instruccion)   //Load Doubleword Right
/*Esta instrucci¢n puede usarse en combinaci¢n con la instrucci¢n LDL para
cargar un registro con ocho bytes consecutivos de memoria, cuando los bytes
atraviesan un l¡mite de doblepalabra. Carga la parte derecha del registro
con la parte apropiada de orden bajo de la doble palabra.*/
{
  UCHAR rt,base,desplaz,aca; //base es el campo de rs,desplazamiento y
acarreo
  signed long int off0,offset,dest0,destino; //parte alta y baja de off. y
despl
  ULONG datomenorpeso,datomayorpeso,auxi;  //Dato de mayor y menor peso y
auxiliar
  UCHAR doble0[4],doble1[4];  //Para coger los bytes de la memoria
  int i,j;   //Estas variables se utilizar n como ¡ndices
```

```
  rt = Campo ("RT",instruccion);  //Cogemos el campo rt de la instrucci¢n
  base = Campo ("RS",instruccion);  //Cogemos el campo rs de la instrucci¢n
  offset = instruccion & 0x0000FFFF;  //Cogemos el campo offset de la instr.
  auxi = offset;    //En auxi metemos el contenido de la variable offset.
  auxi = auxi >> 15;  //Se desplaza 15 bits hacia la derecha.

  if (auxi!=0)  //Si auxi es distinto de cero
  {
    offset = offset | 0xFFFF0000; //Se hace una extensi¢n con unos
    off0 = 0xFFFFFFFF;  //La parte alta se pone todo a unos
  }
  else if (auxi==0)  //Si auxi es igual a cero
    off0 = 0;   //La parte alta se pone todo a ceros

  destino=rpg[base][1]+offset; //Hacemos registro base + offset
  dest0 = rpg[base][0]+off0;  //Hacemos registro base+off0 (parte alta)
  aca=acarreo (rpg[base][1],offset);  //Miramos si hay acarreo
  if (aca==1)  //Si aca es igual a uno
    dest0++;  //Incrementamos la parte superior de destino en uno.

  desplaz=destino & 0x00000007;  //Cogemos el desplazamiento
  destino=destino & 0xFFFFFFF8;  //Ponemos el l¡mite de doublword m s cercano

  if (Estado("UX")==0)  //Para 32 bits.
  {
    General_Exception();        //Reserved instruction exception
    gotoxy (30,23);
    printf ("Reserved Instr. exception");
  }
  else if (Estado("UX")==1)  //Para 64 bits.
  {
    if ((dest0!=0) || (dest0==0 && (destino>65535 || destino<0))) //Si se
sale
    {
      gotoxy (30,23);            //Nos situamos
      printf ("Direcci¢n fuera de rango.");  //Informamos
    }
    else  //Si no se sale del rango de direcciones.
    {
     fseek (pf,destino,SEEK_SET);  //Nos situamos en el fichero de la memoria
     for (i=0;i<4;i++)
     {
      if (Estado("RE")==0)  //Si es little-endian
      j=3-i;
      else if (Estado("RE")==1)   //Si es big-endian
      j=i;
      if ((fread (&doble0[j],sizeof(UCHAR),1,pf))==0)  //Leemos 4 bytes
      {
       Error ("Error al acceder a memoria.");
      }
     }
     for (i=0;i<4;i++)
     {
      if (Estado("RE")==0)  //Si es little-endian
```

```
      j=3-i;
      else if (Estado("RE")==1)   //Si es big-endian
      j=i;
      if ((fread (&doble1[j],sizeof(UCHAR),1,pf))==0)  //Leemos 4 bytes
      {
       Error ("Error al acceder a memoria.");
      }
     }

     if (Estado("RE")==0)    //Si es little-endian
     {
       datomenorpeso=chartolong(doble0);  //Datomenorpeso metemos doble0
       datomayorpeso=chartolong(doble1);  //Datomayorpeso metemos doble1
       if (desplaz==0)  //Si el desplazamiento es igual a cero
       {
       rpg[rt][0]=datomayorpeso;  //En parte alta metemos datomayorpeso
       rpg[rt][1]=datomenorpeso;  //En parte baja metemos datomenorpeso
       }
       else if (desplaz==1)  //Si desplazamiento es igual a uno
       {
       auxi=datomayorpeso>>8;  //En auxi metemos datomayorpeso desplazado
       rpg[rt][0]=rpg[rt][0] & 0xFF000000;  //Hacemos and con parte alta.
       datomenorpeso=datomenorpeso>>8;  //Desplazamos datomenorpeso 8 bits.
       datomayorpeso=datomayorpeso<<24;  //Desplazamos hacia izq. 24 bits.
       rpg[rt][1]=datomenorpeso | datomayorpeso;  //Hacemos or de ambas.
       rpg[rt][0]=rpg[rt][0] | auxi;  //Hacemos or de parte alta y auxi
       }
       else if (desplaz==2)  //Si desplazamiento es igual a dos
       {
       auxi=datomayorpeso>>16;  //En auxi metemos datomayorpeso desplazado
       rpg[rt][0]=rpg[rt][0] & 0xFFFF0000;   //Hacemos and con parte alta
       datomenorpeso=datomenorpeso>>16;  //Desplazamos datomenor peso 16
bits.
       datomayorpeso=datomayorpeso<<16; //Desplazamos 16 bits hacia la izq.
       rpg[rt][1]=datomenorpeso | datomayorpeso;  //Hacemos or de ambas.
       rpg[rt][0]=rpg[rt][0] | auxi;  //Hacemos or de parte alta y auxi
       }
       else if (desplaz==3)  //Si desplazamiento es igual a tres
       {
       auxi=datomayorpeso>>24;  //En auxi metemos datomayorpeso desplazado.
       rpg[rt][0]=rpg[rt][0] & 0xFFFFFF00;  //Hacemos and de parte alta
       datomenorpeso=datomenorpeso>>24;  //Desplazamos 24 bits hacia derecha.
       datomayorpeso=datomayorpeso<<8;  //Desplazamos 8 bits hacia la
izquierda.
       rpg[rt][1]=datomenorpeso | datomayorpeso; //hacemos la or de ambas.
       rpg[rt][0]=rpg[rt][0] | auxi;  //Hacemos la or de parte alta y auxi.
       }
       else if (desplaz==4) rpg[rt][1]=datomayorpeso;  //Si es 4 metemos
todo.
       else if (desplaz==5)  //Si desplazamiento es igual a cinco
       {
       datomayorpeso=datomayorpeso>>8;  //Desplazamos datomayorp 8 bits drcha
       rpg[rt][1]=rpg[rt][1] & 0xFF000000; //Hacemos la and con parte baja
       rpg[rt][1]=rpg[rt][1] | datomayorpeso;  //Hacemos la or con parte baja
       }
```

```
    else if (desplaz==6)  //Si desplazamiento es igual  a seis
    {
    datomayorpeso=datomayorpeso>>16;  //Desplazamos 16 bits hacia derecha
    rpg[rt][1]=rpg[rt][1] & 0xFFFF0000;  //Hacemos la and con parte baja
    rpg[rt][1]=rpg[rt][1] | datomayorpeso;  //Hacemos la or con parte baja
    }
    else if (desplaz==7)  //Si desplazamiento es igual a siete
    {
    datomayorpeso=datomayorpeso>>24;  //Desplaza. 24 bits hacia derecha.
    rpg[rt][1]=rpg[rt][1] & 0xFFFFFF00;  //Hacemos la and con parte baja
    rpg[rt][1]=rpg[rt][1] | datomayorpeso;  //Hacemos la or con parte baja
    }
  }
  else if (Estado("RE")==1)    //Si el big-endian
  {
    datomayorpeso=chartolong(doble0);  //En datomayorpeso metemos doble0
    datomenorpeso=chartolong(doble1);  //En datomenorpeso metemos doble1
    if (desplaz==0)  //Si desplazamiento es igual a cero
    {
    datomayorpeso=datomayorpeso>>24;  //Desplazamos 24 bits hacia derecha.
    rpg[rt][1]=rpg[rt][1] & 0xFFFFFF00; //Hacemos la and con parte baja
    rpg[rt][1]=rpg[rt][1] | datomayorpeso;  //hacemos la or con parte baja
    }
    else if (desplaz==1)  //Si desplazamiento es igual a uno
    {
    datomayorpeso=datomayorpeso>>16;  //Desplazamos hacia derecha 16 bts
    rpg[rt][1]=rpg[rt][1] & 0xFFFF0000; //Hacemos and con parte baja
    rpg[rt][1]=rpg[rt][1] | datomayorpeso;  //Hacemos or con parte baja
    }
    else if (desplaz==2)  //Si desplazamiento es igual a dos
    {
    datomayorpeso=datomayorpeso>>8;  //Desplazamos hacia derecha 8 bits.
    rpg[rt][1]=rpg[rt][1] & 0xFF000000;  //Hacemos la and con parte baja
    rpg[rt][1]=rpg[rt][1] | datomayorpeso;  //Hacemos or con parte baja
    }
    else if (desplaz==3) rpg[rt][1]=datomayorpeso;  //Si 3 metemos todo
    else if (desplaz==4)  //Si desplazamiento es igual a cuatro
    {
    auxi=datomayorpeso>>24;  //Desplazamos hacia la derecha 24 bits.
    rpg[rt][0]=rpg[rt][0] & 0xFFFFFF00;  //Hacemos la and con parte alta.
    datomenorpeso=datomenorpeso>>24;  //Desplazamos 24 bits hacia derecha.
    datomayorpeso=datomayorpeso<<8;  //Desplazmos 8 bits hacia izq.
    rpg[rt][1]=datomenorpeso | datomayorpeso;  //Hacemos la or entre ambas
    rpg[rt][0]=rpg[rt][0] | auxi;  //Hacemos or de parte alta y auxi
    }
    else if (desplaz==5)  //Si desplazamiento es igual a cinco
    {
    auxi=datomayorpeso>>16;  //En auxi metemos datomayorpeso desplazado
    rpg[rt][0]=rpg[rt][0] & 0xFFFF0000;  //Hacemos and con parte alta
    datomenorpeso=datomenorpeso>>16;  //Desplazamos 16 bits hacia derecha
    datomayorpeso=datomayorpeso<<16;  //Desplazamos 16 bits hacia la izq.
    rpg[rt][1]=datomenorpeso | datomayorpeso;  //Hacemos la or de ambas.
    rpg[rt][0]=rpg[rt][0] | auxi;  //Hacemos or de parte alta y auxi.
    }
    else if (desplaz==6)   //Si desplazamiento es igual a seis
```

```
       {
       auxi=datomayorpeso>>8;  //En auxi metemos datomayorpeso desplazado.
       rpg[rt][0]=rpg[rt][0] & 0xFF000000;  //Hacemos la and con parte alta
       datomenorpeso=datomenorpeso>>8;  //Desplazamos 8 bits hacia derecha.
       datomayorpeso=datomayorpeso<<24;  //Desplazamos 24 bits hacia izq.
       rpg[rt][1]=datomenorpeso | datomayorpeso;  //Hacemos la or de ambas.
       rpg[rt][0]=rpg[rt][0] | auxi;  //Hacemos la or de parte alta y auxi
       }
       else if (desplaz==7)  //Si desplazamiento es igual a siete.
       {
       rpg[rt][0]=datomayorpeso;  //En la parte alta metemos datomayorpeso
       rpg[rt][1]=datomenorpeso;  //En la parte baja metemos datomenorpeso
       }
     }
    }
  }
}       //fin de la instrucci¢n LDR

void LUI (ULONG instruccion)  //Load Upper Immediate
/*El dato inmediato se desplaza hacia la izquierda 16 bits y se concatena
con 16 bits de ceros. El resultado se guarda en el registro rt. En modo 64
bits, la palabra cargada hace una extensi¢n de signo.*/
{
  UCHAR rt;   //Para coger el n£mero de registro rt.
  signed long int inmediato;  //Para coger el dato inmediato
  ULONG auxi;   //Variable auxiliar.

  rt = Campo ("RT",instruccion);  //Cogemos el campo rt de la instrucci¢n.
  inmediato = instruccion & 0x0000FFFF;  //Cogemos el campo inmediato de la
ins.
  auxi = inmediato >> 15;  //En auxi guardamos inmediato despl. 15 bits
drcha.

  if (Estado("UX")==0)  //Para 32 bits.
  {
    inmediato=inmediato<<16;  //Desplazamos inmediato 16 bits hacia la izq.
    rpg[rt][1]=inmediato;  //En la parte baja del reg. rt metemos inmediato.
  }
  else if (Estado("UX")==1)    //Para 64 bits.
  {
    inmediato=inmediato<<16;  //Desplazamos inmediato 16 bits hacia la izq.
    rpg[rt][1]=inmediato;  //En la parte baja del reg. rt metemos inmediato.
    if (auxi!=0)  //Si auxi es distinto de cero
      rpg[rt][0]=0xFFFFFFFF;   //Hacemos la extensi¢n de signo a unos. (parte
alta)
    else if (auxi==0)  //Si auxi es igual a cero
      rpg[rt][0]=0x00000000;   //Hacemos extensi¢n de signo a ceros. (parte
alta)
  }
}      //Fin de la instrucci¢n LUI

void SB (ULONG instruccion)  //Store Byte
/*Al offset se le extiende el signo y se suma con el contenido del registro
base para formar una direcci¢n virtual. El byte menos significativo del
registro rt se almacena en la direcci¢n.*/
```

```
{
  UCHAR rt,base,dato,aca; //base es el campo de rs, dato y acarreo
  signed long int off0,offset,dest0,destino;//Parte alta y baja de off. y
dest.
  ULONG auxi;  //Variable auxiliar.

  rt = Campo ("RT",instruccion);   //Cogemos el campo rt de la instrucci¢n
  base = Campo ("RS",instruccion);  //Cogemos el campo rs de la instrucci¢n
  offset = instruccion & 0x0000FFFF;  //Cogemos el offset de la instrucci¢n
  auxi = offset;    //En auxi metemos el contenido de offset
  auxi = auxi >> 15;  //Desplazamos a la derecha 15 bits.

  if (auxi!=0)  //Si auxi es distinto de cero
  {
    offset = offset | 0xFFFF0000;  //Hacemos la extensi¢n de signo con unos
    off0 = 0xFFFFFFFF;  //La parte alta del offset la ponemos a unos
  }
  else if (auxi==0)  //Si auxi es igual a cero
    off0 = 0;  //Ponemos la parte alta del offset a cero.

  destino=rpg[base][1]+offset;  //Hacemos registro base + desplazamiento
  dest0 = rpg[base][0]+off0;  //Hacemos registro base+off0 (parte alta).
  aca=acarreo(rpg[base][1],offset);  //Miramos si hay acarreo
  if (aca==1)  //Si aca es igual a uno
    dest0++;  //Incrementamos la parte alta de destino en uno.

  dato=rpg[rt][1] & 0x000000FF;  //Cogemos el byte m s bajo del reg. rt.

  if (Estado("UX")==0)  //Para 32 bits.
  {
    if ((destino > 65535) || (destino < 0))  //Si se sale del rango de
direcc.
    {
      gotoxy (30,23);
      printf ("Direcci¢n fuera de rango.");
    }
    else  //Si no se sale del rango de direcciones
    {
     fseek (pf,destino,SEEK_SET);  //Nos situamos en el fichero de memoria
     if ((fwrite (&dato,sizeof(UCHAR),1,pf))==0)  //Escribimos dato en
memoria
     {
      Error ("Error al acceder a memoria.");
     }
    }
  }
  else if (Estado("UX")==1)   //Para 64 bits.
  {
    if ((dest0!=0) || (dest0==0 && (destino>65535 || destino<0)))//Si se sale
    {
      gotoxy (30,23);
      printf ("Direcci¢n fuera de rango.");
    }
    else   //Si no se sale del rango de direcciones
    {
```

```
     fseek (pf,destino,SEEK_SET);  //Nos vamos a la direcci¢n destino
     if ((fwrite (&dato,sizeof(UCHAR),1,pf))==0)//Escribimos dato en memoria
     {
      Error ("Error al acceder a memoria.");
     }
    }
  }
}   //Fin de la instrucci¢n SB

void SH (ULONG instruccion)  //Store Halfword
/*Al offset se le hace una extensi¢n de signo y se suma al contenido del
registro base para formar una direcci¢n. La media palabra menos significativa
del registro rt se almacena en la direcci¢n. La direcci¢n debe ser m£ltiplo
de dos, si no es as¡ se produce un error de direccionamiento*/
{
  UCHAR rt,base,dato1,dato2,aca; //base es el campo de rs
  signed long int off0,offset,dest0,destino;//Partes alta y baja de off y
dest.
  ULONG auxi;  //Variable auxiliar.

  rt = Campo ("RT",instruccion);  //Cogemos el campo rt de la instrucci¢n
  base = Campo ("RS",instruccion);  //Cogemos el campo base de la instrucci¢n
  offset = instruccion & 0x0000FFFF;  //Cogemos el campo offset de
instrucci¢n
  auxi = offset;   //En auxi metemos el contenido de offset
  auxi = auxi >> 15;  //Desplazamos auxi 15 bits hacia la derecha.

  if (auxi!=0)  //Si auxi es distinto de cero
  {
    offset = offset | 0xFFFF0000;  //Hacemos la extensi¢n de signo con unos
    off0 = 0xFFFFFFFF;  //La parte alta de offset la ponemos toda a unos
  }
  else if (auxi==0)  //Si auxi es igual a cero
    off0 = 0;   //Ponemos la parte alta de offset a cero.

  destino=rpg[base][1]+offset;  //Hacemos registro base + desplazamiento
  dest0 = rpg[base][0]+off0;  //Hacemos reg. base + desplaz (parte alta)
  aca=acarreo (rpg[base][1],offset);  //Miramos si hay acarreo
  if (aca==1)  //Si aca es igual a uno
    dest0++;   //Incrementamos la parte alta de destino en uno

  if (Estado("RE")==0)   //coger el dato en little endian
  {
    dato1=rpg[rt][1] & 0x000000FF;  //En dato1 metemos el byte mas bajo
    dato2=(rpg[rt][1] & 0x0000FF00) >> 8; //En dato2 metemos el 2§ byte +
bajo
  }
  else if (Estado("RE")==1)      //cogemos el dato en big-endian
  {
    dato1=(rpg[rt][1] & 0x0000FF00) >> 8; //En dato1 metemos el 2§ byte+bajo
    dato2=rpg[rt][1] & 0x000000FF;  //En dato2 metemos el byte mas bajo
  }

  if (Estado("UX")==0)  //Para 32 bits.
  {
```

```
    if ((destino > 65535) || (destino < 0))  //Si se sale del rango
    {
      gotoxy (30,23);
      printf ("Direcci¢n fuera de rango.");
    }
    else if (destino%2 != 0)   //Si no es m£ltipo de dos
    {
      General_Exception();   //Se produce un error de direccionamiento
      gotoxy (30,23);
      printf ("Address Error exception. ");
    }
    else   //Si no se sale del rango de direcciones y no se produce excepci¢n
    {
     fseek (pf,destino,SEEK_SET);  //Nos posicionamos en el fichero memoria

     if ((fwrite (&dato1,sizeof(UCHAR),1,pf))==0)  //Escribimos el dato1
     {
      Error ("Error al acceder a memoria.");
     }

     if ((fwrite (&dato2,sizeof(UCHAR),1,pf))==0)  //Escribimos el dato2
     {
      Error ("Error al acceder a memoria.");
     }
    }
  }
  else if (Estado("UX")==1)   //Para 64 bits.
  {
    if ((dest0!=0) || (dest0==0 && (destino>65535 || destino<0)))//Si se sale
    {
      gotoxy (30,23);
      printf ("Direcci¢n fuera de rango.");
    }
    else if (destino%2 != 0)  //Si no es multiplo de dos
    {
      General_Exception();   //Error de direccionamiento
      gotoxy (30,23);
      printf ("Address Error exception. ");
    }
    else   //Si no se sale del rango y no hay error de direccionamiento
    {
     fseek (pf,destino,SEEK_SET);   //Nos posicionamos en el fichero memoria

     if ((fwrite (&dato1,sizeof(UCHAR),1,pf))==0)  //Escribimos el dato1
     {
      Error ("Error al acceder a memoria.");
     }
     if ((fwrite (&dato2,sizeof(UCHAR),1,pf))==0)  //Escribimos el dato2
     {
      Error ("Error al acceder a memoria.");
     }
    }
  }
}    //Fin de la instrucci¢n SH
```

```
void SW (ULONG instruccion)   //Store Word
/*Al offset se le extiende el signo y se suma al contenido del registro
base para formar la direcci¢n. El contenido del registro rt se almacena
en la direcci¢n de memoria especificada. Si no es multiplo de cuatro
se produce un error de direccionamiento*/
{
  UCHAR rt,base,*dato=NULL,aca; //base es el campo de rs
  signed long int off0,offset,dest0,destino;//Parte alta y baja de off. y
dest.
  ULONG auxi,datoalmac;  //Para variable auxiliar y dato almacenado.
  int i,j;  //Se utilizan como ¡ndices.

  rt = Campo ("RT",instruccion);  //Cogemos el campo rt de la instrucci¢n
  base = Campo ("RS",instruccion);   //Cogemos el campo base de la instrucc.
  offset = instruccion & 0x0000FFFF; //Cogemos el offset de la instrucci¢n
  auxi = offset;  //En auxi metemos el offset
  auxi = auxi >> 15;  //Desplazamos 15 bits hacia la derecha.

  if (auxi!=0)  //Si auxi  es distinto de cero
  {
    offset = offset | 0xFFFF0000; //Hacemos la extensi¢n de signo con unos
    off0 = 0xFFFFFFFF;  //La parte alta de offset se pone todo a unos
  }
  else if (auxi==0)  //Si auxi es igual a cero
    off0 = 0;  //Metemos la parte alta de offset todo a ceros.

  destino=rpg[base][1]+offset;  //Hacemos registro base+desplazamiento
  dest0 = rpg[base][0]+off0;  //Hacemos reg. base + off0 (parte alta)
  aca=acarreo(rpg[base][1],offset);  //Miramos si hay acarreo
  if (aca==1)  //Si aca es igual a uno
    dest0++;   //Incrementamos la parte alta del destino en uno.

  datoalmac=rpg[rt][1];  //En datoalmac metemos la parte baja del registro.
  dato=lontocar(datoalmac); //Pasamos el dato long a array de caracteres.

  if (Estado("UX")==0)  //Para 32 bits.
  {
    if ((destino > 65535) || (destino < 0))  //Si se sale del rango
    {
      gotoxy (30,23);
      printf ("Direcci¢n fuera de rango.");
    }
    else if (destino%4 != 0)   //Si no es multiplo de 4
    {
      General_Exception();   //Error de direccionamiento
      gotoxy (30,23);
      printf ("Address Error exception. ");
    }
    else   //Si no se sale del rango y no hay error de direccionamiento
    {
     fseek (pf,destino,SEEK_SET);  //nos posicionamos en el fichero memoria
     for (i=0;i<4;i++)
     {
      if (Estado("RE")==0)  //Si es little-endian
      j=3-i;
```

```
      else if (Estado("RE")==1)   //Si es big-endian
      j=i;

      if ((fwrite (&dato[j],sizeof(UCHAR),1,pf))==0)  //Escribe en fichero
      {
       Error ("Error al acceder a memoria.");
      }
     }
    }
  }
  else if (Estado("UX")==1)   //Para 64 bits.
  {
    if ((dest0!=0) || (dest0==0 && (destino>65535 || destino<0)))//Si se sale
    {
      gotoxy (30,23);
      printf ("Direcci¢n fuera de rango.");
    }
    else if (destino%4 != 0)  //Si no es m£ltiplo de 4
    {
      General_Exception();   //Error de direccionamiento
      gotoxy (30,23);
      printf ("Address Error exception. ");
    }
    else  //Si no se sale del rango y no es m£ltiplo de 4
    {
     fseek (pf,destino,SEEK_SET);  //Nos situamos en el fichero memoria
     for (i=0;i<4;i++)
     {
      if (Estado("RE")==0)  //Si es little-endian
      j=3-i;
      else if (Estado("RE")==1)   //Si es big-endian
      j=i;

      if ((fwrite (&dato[j],sizeof(UCHAR),1,pf))==0)  //Escribe en memoria
      {
       Error ("Error al acceder a memoria.");
      }
     }
    }
  }
}    //Fin de la instrucci¢n SW

void SWL (ULONG instruccion)  //Store Word Left
{
  UCHAR rt,base,desplaz,*dato=NULL,aca; //base es el campo de rs
  signed long int off0,offset,dest0,destino;//Parte alta y baja de off. y
dest.
  ULONG auxi,datoalmac;  //Variable auxiliar y dato almacenado.
  int i,j;  //Se van a utilizar como ¡ndices.

  rt = Campo ("RT",instruccion);  //Cogemos el campo rt de la instrucci¢n
  base = Campo ("RS",instruccion);  //Cogemos el campo base de la instrucci¢n
  offset = instruccion & 0x0000FFFF; //Cogemos el campo offset de la instr.
  auxi = offset;  //En auxi metemos lo que hay en offset
  auxi = auxi >> 15;  //Desplazamos 15 bits hacia la derecha.
```

```
  if (auxi!=0)  //Si auxi es distinto de cero
  {
    offset = offset | 0xFFFF0000;  //Hacemos la extensi¢n con unos
    off0 = 0xFFFFFFFF;  //Ponemos la parte alta de offset a unos
  }
  else if (auxi==0)  //Si auxi es igual a cero
    off0 = 0;  //Ponemos la parte alta de offset a ceros

  destino=rpg[base][1]+offset;  //Hacemos registro base+offset
  dest0 = rpg[base][0]+off0;  //Hacemos registro base+off0 (parte alta)
  aca=acarreo (rpg[base][1],offset);  //Miramos si hay acarreo
  if (aca==1)  //Si aca es igual a uno
    dest0++;  //Incrementamos la parte alta de destino en uno

  desplaz=destino & 0x00000003;  //Cogemos el desplazamiento a partir del
limite
  destino=destino & 0xFFFFFFFC;//Cogemos el l¡mite

  datoalmac=rpg[rt][1];  //En datoalmac metemos la parte baja del reg.
  dato=lontocar(datoalmac);  //Convertimos de long a array de caracteres.

  if (Estado("UX")==0)  //Para 32 bits.
  {
    if ((destino > 65535) || (destino < 0))  //Si no est  dentro del rango
    {
      gotoxy (30,23);
      printf ("Direcci¢n fuera de rango.");
    }
    else   //si est  dentro del rango
    {
     if (Estado("RE")==0)  //si es little-endian
     {
       fseek (pf,destino,SEEK_SET);  //Nos situamos en el fichero memoria
       for (i=0;i<(desplaz+1);i++)
       {
       j=desplaz-i;
       if ((fwrite (&dato[j],sizeof(UCHAR),1,pf))==0) //Escribe dato
       {
         Error ("Error al acceder a memoria.");
       }
       }
     }
     else if (Estado("RE")==1)   //si es big-endian
     {
       fseek (pf,(destino+desplaz),SEEK_SET);  //Nos situamos en el fichero
       for (i=0;i<(4-desplaz);i++)
       {
       j=i;
       if ((fwrite (&dato[j],sizeof(UCHAR),1,pf))==0) //Escribe dato
       {
         Error ("Error al acceder a memoria.");
       }
       }
     }
```

```
    }
  }
  else if (Estado("UX")==1)   //Para 64 bits.
  {
    if ((dest0!=0) || (dest0==0 && (destino>65535 || destino<0)))//Si se sale
    {
      gotoxy (30,23);
      printf ("Direcci¢n fuera de rango.");
    }
    else  //Si no se sale del rango de direcciones
    {
     if (Estado("RE")==0)  //si es little-endian
     {
       fseek (pf,destino,SEEK_SET);  //Nos situamos en el fichero memoria
       for (i=0;i<(desplaz+1);i++)
       {
       j=desplaz-i;
       if ((fwrite (&dato[j],sizeof(UCHAR),1,pf))==0) //Escribe dato
       {
         Error ("Error al acceder a memoria.");
       }
       }
     }
     else if (Estado("RE")==1)   //si es big-endian
     {
       fseek (pf,(destino+desplaz),SEEK_SET);  //Nos situamos en el fich.
       for (i=0;i<(4-desplaz);i++)
       {
       j=i;
       if ((fwrite (&dato[j],sizeof(UCHAR),1,pf))==0) //Escribe dato
       {
         Error ("Error al acceder a memoria.");
       }
       }
     }
    }
  }
}    //fin de la instrucci¢n SWL

void SWR (ULONG instruccion)    //Store Word Right
{
  UCHAR rt,base,desplaz,*dato=NULL,aca; //base es el campo de rs
  signed long int off0,offset,dest0,destino;//Parte alta y baja de off. y
dest.
  ULONG auxi,datoalmac;  //Variable auxiliar y dato almacenado
  int i,j;  //Se utilizan como ¡ndices.

  rt = Campo ("RT",instruccion);  //Cogemos el campo rt de la instrucci¢n
  base = Campo ("RS",instruccion);  //Cogemos el campo base de la instrucci¢n
  offset = instruccion & 0x0000FFFF; //Cogemos el offset de la instrucci¢n
  auxi = offset;   //En auxi metemos lo que hay en offset
  auxi = auxi >> 15;   //Desplazamos 15 bits hacia la derecha.

  if (auxi!=0)  //Si auxi es distinto de cero
  {
```

```
  offset = offset | 0xFFFF0000;  //Hacemos la extensi¢n de signo con unos
  off0 = 0xFFFFFFFF;  //Ponemos la parte alta de offset a unos
}
else if (auxi==0)  //Si auxi es igual a cero
  off0 = 0;   //Ponemos la parte alta de offset a cero

destino=rpg[base][1]+offset;   //Hacemos registro base + offset
dest0 = rpg[base][0]+off0;  //Hacemos reg. base + off0
aca=acarreo(rpg[base][1],offset);  //Miramos si hay acarreo
if (aca==1)  //Si aca es igual a uno
  dest0++;  //Incrementamos la parte alta de destino en uno

desplaz=destino & 0x00000003;  //Cogemos el desplazamiento
destino=destino & 0xFFFFFFFC;  //Cogemos el l¡mite de palabra.

datoalmac=rpg[rt][1];   //En datoalmac metemos parte baja de reg. rt.
dato=lontocar(datoalmac);  //Pasamos a array de caracteres un long

if (Estado("UX")==0)  //Para 32 bits.
{
  if ((destino > 65535) || (destino < 0))  //Si se sale del rango
  {
    gotoxy (30,23);
    printf ("Direcci¢n fuera de rango.");
  }
  else    //Si no se sale del rango
  {
    if (Estado("RE")==0)  //si es little-endian
    {
    fseek (pf,(destino+desplaz),SEEK_SET);  //Nos situamos en el fichero
    for (i=0;i<(4-desplaz);i++)
    {
      j=3-i;
      if ((fwrite (&dato[j],sizeof(UCHAR),1,pf))==0)  //Escribe dato
      {
        Error ("Error al acceder a memoria.");
      }
    }
    }
    else if (Estado("RE")==1)   //si es big-endian
    {
    fseek (pf,destino,SEEK_SET);  //Nos situamos en el fichero
    for (i=(desplaz);i>=0;i--)
    {
      j=3-i;
      if ((fwrite (&dato[j],sizeof(UCHAR),1,pf))==0)  //Escribe dato
      {
        Error ("Error al acceder a memoria.");
      }
    }
    }
  }
}
else if (Estado("UX")==1)   //Para 64 bits.
{
```

```
    if ((dest0!=0) || (dest0==0 && (destino>65535 || destino<0)))//Si se sale
    {
      gotoxy (30,23);
      printf ("Direcci¢n fuera de rango.");
    }
    else   //si no se sale
    {
      if (Estado("RE")==0)  //si es little-endian
      {
      fseek (pf,(destino+desplaz),SEEK_SET); //Nos situamos en el fichero
      for (i=0;i<(4-desplaz);i++)
      {
        j=3-i;
        if ((fwrite (&dato[j],sizeof(UCHAR),1,pf))==0)  //Escribe dato
        {
          Error ("Error al acceder a memoria.");
        }
      }
      }
      else if (Estado("RE")==1)   //si es big-endian
      {
      fseek (pf,destino,SEEK_SET);  //Nos situamos en el fchero
      for (i=(desplaz);i>=0;i--)
      {
        j=3-i;
        if ((fwrite (&dato[j],sizeof(UCHAR),1,pf))==0)  //Escribe dato
        {
          Error ("Error al acceder a memoria.");
        }
      }
      }
    }
  }
}   //fin de la instrucci¢n SWR

void SD (ULONG instruccion)    //Store Double
/*Al offset se le hace una extensi¢n de signo y se suma al contenido del
registro base para formar una direcci¢n. El contenido del registro rt
es almacenado en la direcci¢n de memoria especificada. La direcci¢n debe ser
m£ltiplo de 8. Si se ejecuta en modo 32 bits se produce una reserved
instruction exception*/
{
  UCHAR rt,base,aca; //base es el campo de rs, y acarreo
  signed long int off0,offset,dest0,destino;//Parte alta y baja de off. y
dest.
  ULONG datomenorpeso,datomayorpeso,auxi;//Para dato de mayor y menor peso  y
auxi
  UCHAR *doble0=NULL,*doble1=NULL; //Para convertir el long a array de char.
  int i,j;  //Se utilizan como ¡ndices.

  rt = Campo ("RT",instruccion);  //Cogemos el campo rt de la instrucci¢n
  base = Campo ("RS",instruccion);  //Cogemos el campo base de la instrucci¢n
  offset = instruccion & 0x0000FFFF; //Cogemos el offset de la instrucci¢n
  auxi = offset;  //En auxi metemos el contenido de offset
  auxi = auxi >> 15;  //Desplazamos auxi 15 bits hacia la derecha.
```

```
  if (auxi!=0)  //Si auxi es distinto de cero
  {
    offset = offset | 0xFFFF0000;  //Se extiende el signo con unos
    off0 = 0xFFFFFFFF;  //La parte alta de offset se pone todo a unos
  }
  else if (auxi==0)  //Si auxi es igual a cero
    off0 = 0;  //La parte alta de offset se pone todo a ceros.

  destino=rpg[base][1]+offset;  //Hacemos registro base + offset
  dest0 = rpg[base][0]+off0;  //Hacemos registro base+off0
  aca=acarreo(rpg[base][1],offset);  //Miramos si hay acarreo
  if (aca==1)  //Si aca es igual a uno
    dest0++;  //Incrementamos la parte alta de destino en uno.

  if (Estado("RE")==0)  //si es little-endian
  {
    doble0=lontocar(rpg[rt][1]);  //En doble0 metemos parte baja de reg. rt.
    doble1=lontocar(rpg[rt][0]);  //Pasamos de long a array de caracteres.
  }
  if (Estado("RE")==1)   //si es big-endian
  {
    doble0=lontocar(rpg[rt][0]);   //En doble0 metemos la parte alta de rt.
    doble1=lontocar(rpg[rt][1]);  //Pasamos de long a array de caracteres.
  }

  if ((Estado("UX")==0) && ((Estado("KSU")==1) || (Estado("KSU")==2))) //Para
32 bits.
  {
    General_Exception();        //Reserved instruction exception
                        //Si modo kernel no se ejecuta excepcion
    gotoxy (30,23);
    printf ("Reserved Instr. exception");
  }
  else if ((Estado("UX")==1) || (Estado ("KSU")==0)) //Para 64 bits O kERNEL.
  {
    if ((dest0!=0) || (dest0==0 && (destino>65535 || destino<0)))//Si se sale
    {
      gotoxy (30,23);
      printf ("Direcci¢n fuera de rango.");
    }
    else if (destino%8 != 0)   //Si no es m£ltiplo de ocho
    {
      General_Exception();   //Error de direccionamiento
      gotoxy (30,23);
      printf ("Address Error exception. ");
    }
    else  //Si no se sale del rango de direcc. y no se produce error de
direcc.
    {
     fseek (pf,destino,SEEK_SET); //Nos situamos en el fichero de memoria
     for (i=0;i<4;i++)
     {
      if (Estado("RE")==0)  //Si es little-endian
      j=3-i;
```

```
      else if (Estado("RE")==1)   //Si es big-endian
      j=i;
      if ((fwrite (&doble0[j],sizeof(UCHAR),1,pf))==0) //Escribimos doble0
      {
       Error ("Error al acceder a memoria.");
      }
     }
     for (i=0;i<4;i++)
     {
      if (Estado("RE")==0)  //Si es little-endian
      j=3-i;
      else if (Estado("RE")==1)   //Si es big-endian
      j=i;
      if ((fwrite (&doble1[j],sizeof(UCHAR),1,pf))==0)  //Escribimos doble1
      {
       Error ("Error al acceder a memoria.");
      }
     }
    }
  }
}    //Fin de la instrucci¢n SD

void SDL (ULONG instruccion)  //Store Doubleword Left
/*Esta instrucci¢n puede usarse con la instrucci¢n SDR para almacenar
el contenido de un registro en 8 bytes consecutivos de memoria cuando
los bytes cruzan un l¡mite de palabra. SDL almacena la parte izquierda
del registro en la parte alta correspondiente de memoria.*/
{
  UCHAR rt,base,desplaz,aca; //base es el campo de rs
  signed long int off0,offset,dest0,destino;//Parte alta y baja de off. y
dest.
  ULONG datomenorpeso,datomayorpeso,auxi;//Dato de mayor y menor peso y auxi
  UCHAR *doble0=NULL,*doble1=NULL; //Para convertir long a cadena de
caracteres
  int i,j;  //Se utilizan como ¡ndices.

  rt = Campo ("RT",instruccion);  //Cogemos el campo rt de la instrucci¢n
  base = Campo ("RS",instruccion);  //Cogemos el campo base de la instrucci¢n
  offset = instruccion & 0x0000FFFF;  //Cogemos el offset de la instrucci¢n
  auxi = offset;   //En auxi metemos lo que hay en offset
  auxi = auxi >> 15;  //Desplazamos hacia la derecha 15 bits.

  if (auxi!=0)  //Si auxi es distinto de cero
  {
    offset = offset | 0xFFFF0000;  //Hacemos la extensi¢n de signo con unos
    off0 = 0xFFFFFFFF; //La parte alta de offset la ponemos toda a unos
  }
  else if (auxi==0)  //Si auxi es igual a cero
    off0 = 0;  //Ponemos la parte alta de offset toda a ceros.

  destino=rpg[base][1]+offset;  //Hacemos registro base mas offset
  dest0 = rpg[base][0]+off0;  //Hacemos reg. base + off0 (parte alta)
  aca=acarreo(rpg[base][1],offset); //Miramos si hay acarreo
  if (aca==1)  //Si aca es igual a uno
    dest0++;   //Incrementamos la parte alta de destino en uno.
```

```
  desplaz=destino & 0x00000007;  //Cogemos el desplazamiento hasta el l¡mite
  destino=destino & 0xFFFFFFF8;  //Ponemos destino en el l¡mite de doubleword

  if (Estado("UX")==0)  //Para 32 bits.
  {
    General_Exception();        //Reserved instruction exception
    gotoxy (30,23);
    printf ("Reserved Instr. exception");
  }
  else if (Estado("UX")==1) //Para 64 bits
  {
    if ((dest0!=0) || (dest0==0 && (destino>65535 || destino<0)))//Si se sale
    {
      gotoxy (30,23);
      printf ("Direcci¢n fuera de rango.");
    }
    else  //si no se sale del rango de direcciones
    {
     if (Estado("RE")==0)  //si es little-endian
     {
       doble0=lontocar(rpg[rt][1]); //En doble0 metemos la parte baja del
reg.
       doble1=lontocar(rpg[rt][0]); //Convertimos long a array de caracteres.
       fseek (pf,destino,SEEK_SET); //Nos posicionamos en el fichero de
memoria
       if ((desplaz==0) || (desplaz==1) || (desplaz==2) || (desplaz==3))
       {  //Si desplazamiento es cero ¢ uno ¢ dos ¢ tres
       for (i=0;i<(desplaz+1);i++)
       {
         j=desplaz-i;
         if ((fwrite (&doble1[j],sizeof(UCHAR),1,pf))==0) //Escribimos en mem
         {
           Error ("Error al acceder a memoria.");
         }
       }
       }
       else if ((desplaz==4) || (desplaz==5) || (desplaz==6) || (desplaz==7))
       {  //Si desplazamiento es cuatro ¢ cinco ¢ seis ¢ siete
       for (i=0;i<((desplaz-4)+1);i++)
       {
         j=(desplaz-4)-i;
         if ((fwrite (&doble0[j],sizeof(UCHAR),1,pf))==0)//Escribe en memoria
         {
           Error ("Error al acceder a memoria.");
         }
       }
       for (i=0;i<4;i++)
       {
         j=3-i;
         if ((fwrite (&doble1[j],sizeof(UCHAR),1,pf))==0)//Escribe en memoria
         {
           Error ("Error al acceder a memoria.");
         }
       }
```

```
        }
      }
      if (Estado("RE")==1)   //si es big-endian
      {
        doble0=lontocar(rpg[rt][0]); //Metemos en doble0 la parte alta del
reg.
        doble1=lontocar(rpg[rt][1]); //Convertimos un long a array de
caracteres
        fseek (pf,(destino+desplaz),SEEK_SET); //Nos situamos en el fichero
        if ((desplaz==0) || (desplaz==1) || (desplaz==2) || (desplaz==3))
        {  //Si desplazamiento es cero ¢ uno ¢ dos ¢ tres
        for (i=0;i<4;i++)
        {
          j=i;
          if ((fwrite (&doble0[j],sizeof(UCHAR),1,pf))==0)//Escribe en memoria
          {
            Error ("Error al acceder a memoria.");
          }
        }
        for (i=0;i<(4-desplaz);i++)
        {
          j=i;
          if ((fwrite (&doble1[j],sizeof(UCHAR),1,pf))==0)//Escribe en memoria
          {
            Error ("Error al acceder a memoria.");
          }
        }
        }
        else if ((desplaz==4) || (desplaz==5) || (desplaz==6) || (desplaz==7))
        {  //Si desplazamiento es cuatro ¢ cinco ¢ seis ¢ siete
        for (i=0;i<((7-desplaz)+1);i++)
        {
          j=i;
          if ((fwrite (&doble0[j],sizeof(UCHAR),1,pf))==0)//Escribe en memoria
          {
            Error ("Error al acceder a memoria.");
          }
        }
        }
      }
     }
   }
}   //Fin de la instrucci¢n SDL

void SDR (ULONG instruccion)  //Store Doubleword Right
/*Esta instrucci¢n puede usarse con la instrucci¢n SDL para almacenar
el contenido de un registro en 8 bytes consecutivos de memoria cuando
los bytes cruzan un l¡mite de palabra. SDR almacena la parte derecha
del registro en la parte baja correspondiente de memoria.*/
{
  UCHAR rt,base,desplaz,aca; //base es el campo de rs, desplazam y aca.
  signed long int off0,offset,dest0,destino;//Parte alta y baja de off y
dest.
  ULONG datomenorpeso,datomayorpeso,auxi; //Dato de menor y mayor peso y auxi
  UCHAR *doble0=NULL,*doble1=NULL; //Para pasar de long a array de char.
```

```
int i,j;  //Se utilizan como ¡ndice

rt = Campo ("RT",instruccion);  //Cogemos el campo rt de la instrucci¢n
base = Campo ("RS",instruccion);  //Cogemos el campo base de la instrucci¢n
offset = instruccion & 0x0000FFFF; //Cogemos el campo offset de la instru.
auxi = offset;  //En auxi metemos lo que hay en offset
auxi = auxi >> 15; //Desplazamos 15 bits hacia la derecha.

if (auxi!=0)  //Si auxi es distinto de cero
{
  offset = offset | 0xFFFF0000;  //Hacemos la extensi¢n de signo a unos
  off0 = 0xFFFFFFFF;  //Ponemos la parte alta de offset a unos
}
else if (auxi==0)  //Si auxi es igual a cero
  off0 = 0;  //Ponemos la parte alta de offset a ceros.

destino=rpg[base][1]+offset;  //Hacemos registro base + desplazamiento
dest0 = rpg[base][0]+off0;  //Hacemos registro base+off0 (parte alta)
aca=acarreo(rpg[base][1],offset); //Miramos si hay acarreo
if (aca==1)   //Si aca es igual a uno
  dest0++;   //Incrementamos la parte alta de destino en uno.

desplaz=destino & 0x00000007;  //Cogemos el desplazamiento desde el l¡mite
destino=destino & 0xFFFFFFF8;  //Cogemos el l¡mite de doblepalabra

if (Estado("UX")==0)  //Para 32 bits.
{
  General_Exception();        //Reserved instruction exception
  gotoxy (30,23);
  printf ("Reserved Instr. exception");
}
else if (Estado("UX")==1) //Para 64 bits
{
  if ((dest0!=0) || (dest0==0 && (destino>65535 || destino<0)))//Si se sale
  {
    gotoxy (30,23);
    printf ("Direcci¢n fuera de rango.");
  }
  else   //si no se sale del rango de memoria
  {
   if (Estado("RE")==0)  //si es little-endian
   {
     doble0=lontocar(rpg[rt][1]);  //Metemos en doble0 la parte baja de rt
     doble1=lontocar(rpg[rt][0]);  //Metemos en doble1 la parte alta de rt.
     fseek (pf,(destino+desplaz),SEEK_SET);  //Nos situamos en el fichero
     if ((desplaz==0) || (desplaz==1) || (desplaz==2) || (desplaz==3))
     {  //Si desplazamiento es cero ¢ uno ¢ dos ¢ tres
     for (i=0;i<4;i++)
     {
       j=3-i;
       if ((fwrite (&doble0[j],sizeof(UCHAR),1,pf))==0)//Escribe en fich.
       {
         Error ("Error al acceder a memoria.");
       }
     }
```

```
       for (i=0;i<(4-desplaz);i++)
       {
         j=3-i;
         if ((fwrite (&doble1[j],sizeof(UCHAR),1,pf))==0)//Escribe en memoria
         {
           Error ("Error al acceder a memoria.");
         }
       }
       }
       else if ((desplaz==4) || (desplaz==5) || (desplaz==6) || (desplaz==7))
       {  //Si desplazamiento es cuatro ¢ cinco ¢ seis ¢ siete
       for (i=0;i<((7-desplaz)+1);i++)
       {
         j=3-i;
         if ((fwrite (&doble0[j],sizeof(UCHAR),1,pf))==0)//Escribe en memoria
         {
           Error ("Error al acceder a memoria.");
         }
       }
       }
     }
     if (Estado("RE")==1)   //si es big-endian
     {
       doble0=lontocar(rpg[rt][0]); //En doble0 metemos la parte alta de rt.
       doble1=lontocar(rpg[rt][1]);  //Pasamos de long a cadena de
caracteres.
       fseek (pf,destino,SEEK_SET); //Nos situamos en el fichero de memoria
       if ((desplaz==0) || (desplaz==1) || (desplaz==2) || (desplaz==3))
       {  //Si desplazamiento es cero ¢ uno ¢ dos ¢ tres
       for (i=(desplaz);i>=0;i--)
       {
         j=3-i;
         if ((fwrite (&doble1[j],sizeof(UCHAR),1,pf))==0)//Escribe en memoria
         {
           Error ("Error al acceder a memoria.");
         }
       }
       }
       else if ((desplaz==4) || (desplaz==5) || (desplaz==6) || (desplaz==7))
       {  //Si desplazamiento es cuatro ¢ cinco ¢ seis ¢ siete
       for (i=(desplaz);i>=4;i--)
       {
         j=7-i;
         if ((fwrite (&doble0[j],sizeof(UCHAR),1,pf))==0)//Escribe en memoria
         {
           Error ("Error al acceder a memoria.");
         }
       }
       for (i=0;i<4;i++)
       {
         j=i;
         if ((fwrite (&doble1[j],sizeof(UCHAR),1,pf))==0)//Escribe en memoria
         {
           Error ("Error al acceder a memoria.");
         }
```

```
        }
        }
      }
    }
  }
}    //fin de la instrucci¢n SDR

void J (ULONG instruccion)  //Jump
{
  ULONG target; //Para el campo target
  signed long int temp;  //Variable temporal

  target = instruccion & 0x03FFFFFF;  //Cogemos el campo target de la instr.
  temp = target << 2;  //En temp metemos target desplazado 2 bits hacia izq.

  if (Estado("UX")==0)  //para 32 bits
  {
    if ((temp > 65535) || (temp < 0))  //temp porque es la direcci¢n completa
    {
      gotoxy (30,23);
      printf ("Direcci¢n fuera de rango.");
    }
    else  //Si no se sale del rango de direcciones
    {
      temp=temp-4;    //porque al salir de esta funci¢n se incrementa en 4
      rpg[34][1]=temp;  //Metemos temp en el contador de programa
    }
  }
  else if (Estado("UX")==1)    //para 64 bits.
  {
    if ((temp > 65535) || (temp < 0))  //temp porque es la direcci¢n completa
    {
      gotoxy (30,23);
      printf ("Direcci¢n fuera de rango.");
    }
    else   //Si no se sale del rango de direcciones
    {
      temp=temp-4;    //porque al salir de esta funci¢n se incrementa en 4
      rpg[34][1]=temp;  //Metemos tempo en el contador de programa.
    }
  }
}     //Fin de la instrucci¢n J

void JAL (ULONG instruccion)  //Jump And Link
{
  ULONG target;   //Para el campo target
  signed long int temp;  //Variable temporal

  target = instruccion & 0x03FFFFFF;  //Cogemos el campo target de la instr.
  temp = target << 2;  //Desplazamos 2 bits hacia la izquierda

  if (Estado("UX")==0)  //para 32 bits
  {
    if ((temp > 65535) || (temp < 0))  //temp porque es la direcci¢n completa
    {
```

```
      gotoxy (30,23);
      printf ("Direcci¢n fuera de rango.");
    }
    else  //si no se sale del rango de direcciones
    {
      rpg[31][1]=rpg[34][1]+8;//Metemos direcc. de sig. instr al delay slot
      temp=temp-4;    //porque al salir de esta funci¢n se incrementa en 4
      rpg[34][1]=temp;  //En el CP metemos temp
    }
  }
  else if (Estado("UX")==1)    //para 64 bits.
  {
    if ((temp > 65535) || (temp < 0))  //temp porque es la direcci¢n completa
    {
      gotoxy (30,23);
      printf ("Direcci¢n fuera de rango.");
    }
    else   //Si no se sale del rango de direcciones
    {
      rpg[31][1]=rpg[34][1]+8;//Metemos direcc. de sig. instr. al delay slot
      temp=temp-4;    //porque al salir de esta funci¢n se incrementa en 4
      rpg[34][1]=temp;  //En CP metemos temp
    }
  }
}   //Fin de la instrucci¢n JAL

void JALR (ULONG instruccion)  //Jump And Link Register
/*El programa salta a la direcci¢n contenida en el registro rs. La direcci¢n
de la instrucci¢n siguiente al delay slot (siguiente de la siguiente) se
guarda en el registro rd. Si rd vale cero se guarda en el reg. 31.*/
{
  UCHAR rs, rd;  //Para guardar el n£mero de registro
  signed long int destino;  //Para el destino

  rs = Campo ("RS",instruccion);  //Cogemos el campo rs de la instrucci¢n
  rd = Campo ("RD",instruccion);  //Cogemos el campo rd de la instrucci¢n

  destino = rpg[rs][1];  //Metemos la parte baja del reg. rs en destino.

  if (Estado("UX")==0)  //para 32 bits
  {
    if ((destino > 65535) || (destino < 0))  //Si se sale del rango de 64 k
    {
      gotoxy (30,23);
      printf ("Direcci¢n fuera de rango.");  //Direcci¢n fuera de rango
    }
    else if (destino%4 != 0)   //Si la direcci¢n no es alineada (word)
    {
      General_Exception();   //Se produce un error de direccionamiento
      gotoxy (30,23);
      printf ("Address Error exception. ");
    }
    else   //en otro caso
    {
      if (rd==0)   //Si el n£mero rd es igual a cero
```

```
      rpg[31][1]=rpg[34][1]+8;  //guardamos la direcc. en el reg. 31
      else if (rd!=0)  //Si es distinto de cero
      rpg[rd][1]=rpg[34][1]+8;  //Guardamos la direcc. en el reg. rd
      destino=destino-4;   //restamos 4 a destino
      rpg[34][1]=destino; //En el contador de programa metemos direcc.
destino
    }
  }
  else if (Estado("UX")==1)    //para 64 bits.
  {
    if ((rpg[rs][0]!=0) || (rpg[rs][0]==0 && (destino>65535 || destino<0)))
    {
      gotoxy (30,23);
      printf ("Direcci¢n fuera de rango.");  //Direcci¢n fuera de rango
    }
    else if (destino%4 != 0)   //Si la direcci¢n no es alineada (word)
    {
      General_Exception();   //Se produce un error de direccionamiento
      gotoxy (30,23);
      printf ("Address Error exception. ");
    }
    else   //en otro caso
    {
      if (rd==0)   //Si el n£mero rd es igual a cero
      rpg[31][1]=rpg[34][1]+8;  //Guardamos direcc. en el registro 31
      else if (rd!=0)   //Si es distinto de cero
      rpg[rd][1]=rpg[34][1]+8;   //Guardamos direcc. en el registro rd.
      destino=destino-4;   //restamos 4 a destino
      rpg[34][1]=destino; //En el contador de programa metemos direcc.
destino
    }
  }
}         //Fin de la instrucci¢n JALR

void JR (ULONG instruccion)  //Jump Register
/*El programa salta incondicionalmente a la direcci¢n contenido en el regis-
tro general rs, con un retardo de una instrucci¢n. Las instrucciones deben
estar alineadas. Los dos bits de orden bajo del target del registro rs deben
ser cero. Si no es as¡ cocurrir  una address error exception*/
{
  UCHAR rs;   //Para el n£mero del registro
  signed long int destino;  //Para irse al destino.

  rs = Campo ("RS",instruccion);  //Cogemos el campo rs de la instrucci¢n

  destino = rpg[rs][1];  //En destino metemos la parte baja del reg. rs.

  if (Estado("UX")==0)  //para 32 bits
  {
    if ((destino > 65535) || (destino < 0))  //Si est  fuera de un rango
    {
      gotoxy (30,23);
      printf ("Direcci¢n fuera de rango.");
    }
    else if (destino%4 != 0)  //Si no est  alineada
```

```
    {
      General_Exception();    //Se produce un error de direccionamiento
      gotoxy (30,23);
      printf ("Address Error exception. ");
    }
    else   //En otro caso
    {
      destino=destino-4;   //restamos 4 a destino
      rpg[34][1]=destino; //En el contador de programa metemos direcc.
destino
    }
  }
  else if (Estado("UX")==1)    //para 64 bits.
  {
    if ((rpg[rs][0]!=0) || (rpg[rs][0]==0 && (destino>65535 || destino<0)))
    {
      gotoxy (30,23);
      printf ("Direcci¢n fuera de rango.");
    }
    else if (destino%4 != 0)  //Si la palabra no est  alineada
    {
      General_Exception();   //Se produce un error de direccionamiento
      gotoxy (30,23);
      printf ("Address Error exception. ");
    }
    else   //En otro caso
    {
      destino=destino-4;  //Restamos 4 a destino.
      rpg[34][1]=destino;  //En el contador de programa metemos direc.
destino
    }
  }
}       //Fin de la instrucci¢n JR

ULONG multiplicacion (ULONG m12, ULONG m11, ULONG m22, ULONG m21, char *c)
{
  UCHAR a[8],b[8];   //a es el multiplicando y b el multiplicador.
  UCHAR resul[9][16];   //guardamos el resultado en fila 9.
  unsigned int acarreo,suma;
  int i,j,k;
  UCHAR alta[4],altamedia[4],bajamedia[4],baja[4];
  ULONG uno,dos,tres,cuatro;

  a[0] = m11 & 0x000000FF;   //i
  a[1] = (m11 & 0x0000FF00) >> 8; //j
  a[2] = (m11 & 0x00FF0000) >> 16;
  a[3] = (m11 & 0xFF000000) >> 24;
  a[4] = m12 & 0x000000FF;
  a[5] = (m12 & 0x0000FF00) >> 8;
  a[6] = (m12 & 0x00FF0000) >> 16;
  a[7] = (m12 & 0xFF000000) >> 24;

  b[0] = m21 & 0x000000FF;
  b[1] = (m21 & 0x0000FF00) >> 8;
  b[2] = (m21 & 0x00FF0000) >> 16;
```

```
b[3] = (m21 & 0xFF000000) >> 24;
b[4] = m22 & 0x000000FF;
b[5] = (m22 & 0x0000FF00) >> 8;
b[6] = (m22 & 0x00FF0000) >> 16;
b[7] = (m22 & 0xFF000000) >> 24;

for (i=0;i<9;i++)
  for (j=0;j<16;j++)
    resul[i][j]=0;

for (i=0;i<8;i++)
{
  k=i;
  acarreo=0;
  for (j=0;j<8;j++)
  {
    resul[i][k] = (b[i] * a[j]) + acarreo;
    acarreo = (unsigned int)((b[i] * a[j]) + acarreo) >> 8;
    if ((j == 7) && (acarreo != 0))
    resul[i][k+1] = acarreo;
    k++;
  }
}

acarreo=0;
for (i=0;i<16;i++)
{
  suma=0;
  for (j=0;j<8;j++)
  {
    suma = suma + resul[j][i];
  }
  suma = suma + acarreo;
  resul[8][i] = suma;
  acarreo = suma >> 8;
}

//ahora guardamos la parte alta y la baja

for (i=15;i>=12;i--)
  alta[15-i]=resul[8][i];

for (i=11;i>=8;i--)
  altamedia[11-i]=resul[8][i];

for (i=7;i>=4;i--)
  bajamedia[7-i]=resul[8][i];

for (i=3;i>=0;i--)
  baja[3-i]=resul[8][i];

uno=chartolong(alta);
dos=chartolong(altamedia);
tres=chartolong(bajamedia);
cuatro=chartolong(baja);
```

```
  if (strcmp(c,"ALTA")==0)
    return uno;
  else if (strcmp(c,"ALTAMEDIA")==0)
    return dos;
  else if (strcmp(c,"BAJAMEDIA")==0)
    return tres;
  else if (strcmp(c,"BAJA")==0)
    return cuatro;
}


void DMULT (ULONG instruccion) //Doubleword Multiply
/*En modo 32 bits causa una reserved instruction exception. En 64 bits
multiplica con signo el contenido de los registros rs y rt. Los 64 bits mas
bajos del resultado se guardan en el registro LO y los 64 bits m s altos
del resultado se guardan en el registro HI.*/
{
  UCHAR rt,rs,sa,cambio=0;  //Para coger el n£mero de registro
  ULONG auxi,regrs0,regrs1,regrt0,regrt1; //Variable auxiliar y para regs.

  rs=Campo ("RS",instruccion);  //Cogemos el campo rs de la instrucci¢n
  rt=Campo ("RT",instruccion);  //Cogemos el campo rt de la instrucci¢n

  regrs0=rpg[rs][0]>>31;  //Guardamos en regrs0 la parte alta del reg. rs.
  regrt0=rpg[rt][0]>>31;  //Guardamos en regrt0 la parte alta del reg. rt.

  if ((regrs0!=0) && (regrt0!=0)) //Si los dos operandos son negativos
  {
    regrs1= (~rpg[rs][1]+1);  //hacemos el positivo en complemento a dos
    regrs0= (~rpg[rs][0]);   //invertimos todos los bits
    if (regrs1==0) regrs0++; //si hay acarreo al sumar a regrs1 1, inc.
regrs0
    regrt1= (~rpg[rt][1]+1);  //el otro operando tambi,n le cambiamos el
signo
    regrt0= (~rpg[rt][0]);
    if (regrt1==0) regrt0++;
    cambio=0;          //No hay que hacer cambio de signo al final.
  }
  else if ((regrs0!=0) && (regrt0==0)) //Si solo el reg. rs es negativo
  {
    regrs1= (~rpg[rs][1]+1);  //hacemos el positivo en complemento a dos
    regrs0= (~rpg[rs][0]); //invertimos todos los bits
    if (regrs1==0) regrs0++; //si hay acarreo al sumar a regrs1 1, inc.
regrs0
    cambio=1;     //Hay que hacer un cambio de signo al resultado
    regrt1= rpg[rt][1];  //El registro rt al ser positivo se queda como est .
    regrt0= rpg[rt][0];
  }
  else if ((regrt0!=0) && (regrs0==0)) //Si solo el reg. rt es negativo
  {
    regrt1= (~rpg[rt][1]+1);  //hacemos el positivo en complemento a dos
    regrt0= (~rpg[rt][0]);  //invertimos todos los bits
    if (regrt1==0) regrt0++; //si hay acarreo al sumar a regrt1 1, inc.
regrt0
```

```
    cambio=1;    //Hay que hacer un cambio de signo al resultado final
    regrs1= rpg[rs][1];  //el registro rs al ser positivo se queda como est .
    regrs0= rpg[rs][0];
  }
  else if ((regrs0==0) && (regrt0==0)) //Si los dos reg. son positivos
  {
    regrs1=rpg[rs][1];  //El registro rs se queda como est  porque es
positivo
    regrs0=rpg[rs][0];
    regrt1=rpg[rt][1];  //El registro rt se queda como est  porque es
positivo
    regrt0=rpg[rt][0];
    cambio=0;   //No hay que hacer cambio de signo al resultado final
  }

  if (Estado("UX")==0)  //Para 32 bits.
  {
    General_Exception();        //Reserved instruction exception
    gotoxy (30,23);
    printf ("Reserved Instr. exception");
  }
  else if (Estado("UX")==1)  //Para 64 bits.
  {
    /*Hacemos la multiplicaci¢n y lo guardamos en su correspondiente reg.*/
    rpg[32][0]=multiplicacion(regrs0,regrs1,regrt0,regrt1,"ALTA");
    rpg[32][1]=multiplicacion(regrs0,regrs1,regrt0,regrt1,"ALTAMEDIA");
    rpg[33][0]=multiplicacion(regrs0,regrs1,regrt0,regrt1,"BAJAMEDIA");
    rpg[33][1]=multiplicacion(regrs0,regrs1,regrt0,regrt1,"BAJA");
    if (cambio==1) //Ahora vamos a ver si hay que hacer el cambio de signo
    {   //en el resultado de la multiplicaci¢n
      rpg[33][1]=(~rpg[33][1]+1);  //cambiamos de signo en Ca2 (0..31)
      rpg[33][0]=(~rpg[33][0]); //invertimos todos los bits (32..63)
      rpg[32][1]=(~rpg[32][1]); //invertimos todos los bits (64..95)
      rpg[32][0]=(~rpg[32][0]);  //invertimos todos los bits (96..127)
      if (rpg[33][1]==0) rpg[33][0]++; //Transmitimos el acarreo si lo hay
      if (rpg[33][0]==0) rpg[32][1]++;
      if (rpg[32][1]==0) rpg[32][0]++;
    }
  }
}    //Fin de la instrucci¢n DMULT

void DMULTU (ULONG instruccion)  //Doubleword Multiply Unsigned
/*En modo 32 bits causa una reserved instruction exception. En 64 bits
multiplica sin signo el contenido de los registros rs y rt. Los 64 bits mas
bajos del resultado se guardan en el registro LO y los 64 bits m s altos
del resultado se guardan en el registro HI.*/
{
  UCHAR rt,rs,sa;   //Para coger el n£mero de los registros
  ULONG auxi;   // Variable auxiliar

  rs=Campo ("RS",instruccion);  //Cogemos el campo rs de la instrucci¢n
  rt=Campo ("RT",instruccion);  //Cogemos el campo rt de la instrucci¢n

  if (Estado("UX")==0)  //Para 32 bits.
  {
```

```
    General_Exception();        //Reserved instruction exception
                        //Si modo kernel no se ejecuta excepcion
    gotoxy (30,23);
    printf ("Reserved Instr. exception");
  }
  else if (Estado("UX")==1)  //Para 64 bits.
  {
    /*Hacemos la multiplicaci¢n y el result. lo guardamos en su corresp
reg.*/

rpg[32][0]=multiplicacion(rpg[rs][0],rpg[rs][1],rpg[rt][0],rpg[rt][1],"ALTA")
;

rpg[32][1]=multiplicacion(rpg[rs][0],rpg[rs][1],rpg[rt][0],rpg[rt][1],"ALTAME
DIA");

rpg[33][0]=multiplicacion(rpg[rs][0],rpg[rs][1],rpg[rt][0],rpg[rt][1],"BAJAME
DIA");

rpg[33][1]=multiplicacion(rpg[rs][0],rpg[rs][1],rpg[rt][0],rpg[rt][1],"BAJA")
;
  }
}   //Fin de la instrucci¢n DMULTU

void MULT (ULONG instruccion)  //Multiply
/*Multiplica con signo el contenido de los registros rs y rt. Los 32 bits mas
bajos del resultado se guardan en el registro LO y los 32 bits m s altos
del resultado se guardan en el registro HI. En 64 bits se hace una extensi¢n
de signo en el correspondiente registro del resultado*/
{
  UCHAR rt,rs,sa,cambio=0;  //Para el n£mero de los registros y cambio signo.
  ULONG auxi,regrs,regrt;  //variables auxiliares y para los registros

  rs=Campo ("RS",instruccion);  //Cogemos el campo RS de la instrucci¢n
  rt=Campo ("RT",instruccion);  //Cogemos el campo rt de la instrucci¢n

  regrs=rpg[rs][1] >> 31;  //vamos a ver el signo que tiene cada registro
  regrt=rpg[rt][1] >> 31;  //Para ello desplazamos a la derecha 31 bits.

  if ((regrs!=0) && (regrt!=0))  //Si los dos registros son negativos
  {
    regrs= (~rpg[rs][1]+1);  //hacemos el positivo en complemento a dos
    regrt= (~rpg[rt][1]+1);
    cambio=0;        //No hace falta hacer el cambio de signo en el resultado
  }
  else if ((regrs!=0) && (regrt==0)) //Si solo el registro rs es negativo
  {
    regrs= (~rpg[rs][1]+1);  //hacemos el positivo en complemento a dos
    cambio=1;       //Para hacer el cambio de signo en el resultado final
    regrt= rpg[rt][1];   //El registro rt se queda como est
  }
  else if ((regrt!=0) && (regrs==0))  //Si solo el registro rt es negativo
  {
    regrt= (~rpg[rt][1]+1);   //Hacemos el positivo en complemento a dos
    cambio=1;    //Para hacer el cambio de signo en el resultado final
```

```
    regrs= rpg[rs][1];  //El registro rs se queda como est
  }
  else if ((regrs==0) && (regrt==0))  //Si los dos registros son positivos
  {
    regrs=rpg[rs][1];  //El registro rs se queda como est
    regrt=rpg[rt][1];  //El registro rt se queda como est
    cambio=0;    //No hace falta hacer un cambio de signo en el resultado.
  }

  if (Estado("UX")==0) //Para 32 bits.
  {
    //Hacemos la multiplicaci¢n y guardamos el resultado en los registros
    rpg[32][1]=multiplicacion(0,regrs,0,regrt,"BAJAMEDIA"); //Reg HI
    rpg[33][1]=multiplicacion(0,regrs,0,regrt,"BAJA");  //Reg LO
    if (cambio==1)   //Vamos a hacer el cambio de signo al resultado
    {  //si la variable cambio es igual a uno.
      rpg[33][1]=(~rpg[33][1]+1);  //Se hace el complemento a dos
      rpg[32][1]=(~rpg[32][1]);  //Se invierten los bits
      if (rpg[33][1]==0)  //Si la parte baja es igual a cero es que hay carry
      {
      rpg[32][1]++;  //y se incrementa en uno la parte alta
      }
    }
  }
  else if (Estado("UX")==1)   //Para 64 bits.
  {
    //Hacemos la multiplicaci¢n y guardamos el resultado en el reg. LO
    rpg[33][1]=multiplicacion(0,regrs,0,regrt,"BAJA");
    if (cambio==1)  //Hacemos el cambio de signo
    {
      rpg[33][1]=(~rpg[33][1]+1);  //Hacemos el complemento a dos
    }
    auxi=rpg[33][1] >> 31;   //Vamos a hacer la extensi¢n de signo de LO
    if (auxi!=0)             //Si auxi es distinto de cero
      rpg[33][0]=0xFFFFFFFF;   //ponemos la parte alta del registro todo a 1
    else if (auxi==0)    //Si auxi es igual a cero
      rpg[33][0]=0x00000000;   //ponemos la parte alta del registro todo a 0

    //Hacemos la multiplicaci¢n y guardamos el resultado en el reg. HI
    rpg[32][1]=multiplicacion(0,regrs,0,regrt,"BAJAMEDIA");
    if (cambio==1)   //Hacemos el cambio de signo.
    {
      rpg[32][1]=(~rpg[32][1]);  //Invertimos los bits
      if (rpg[33][1]==0)   //Si la parte baja del reg. LO es cero hay carry
      {
      rpg[32][1]++;   //incrementamos la parte baja del registro HI
      }
    }
    auxi=rpg[32][1] >> 31;  //Vamos a hacer la extensi¢n de signo de HI
    if (auxi!=0)          //Si auxi es distinto de cero
      rpg[32][0]=0xFFFFFFFF;   //ponemos la parte alta del registro todo a 1
    else if (auxi==0)    //Si auxi es igual a cero
      rpg[32][0]=0x00000000;    //ponemos la parte alta del registro todo a 0
  }
}       //fin de la instrucci¢n MULT
```

```
void MULTU (ULONG instruccion)   //Multiply Unsigned
/*Multiplica sin signo el contenido de los registros rs y rt. Los 32 bits mas
bajos del resultado se guardan en el registro LO y los 32 bits m s altos
del resultado se guardan en el registro HI. En 64 bits se hace una extensi¢n
de signo en el correspondiente registro del resultado*/
{
  UCHAR rt,rs,sa;  //Para coger el n£mero de los registros
  ULONG auxi;   //Variable auxiliar

  rs=Campo ("RS",instruccion);  //Cogemos el campo rs de la instrucci¢n
  rt=Campo ("RT",instruccion);  //Cogemos el campo rt de la instrucci¢n

  if (Estado("UX")==0) //Para 32 bits.
  {
    //Hacemos la multiplicaci¢n y guardamos resultado en los registros
    rpg[32][1]=multiplicacion(0,rpg[rs][1],0,rpg[rt][1],"BAJAMEDIA");
    rpg[33][1]=multiplicacion(0,rpg[rs][1],0,rpg[rt][1],"BAJA");
  }
  else if (Estado("UX")==1)   //Para 64 bits.
  {
    //Hacemos la multiplicaci¢n y guardamos resultado en los registros
    rpg[32][1]=multiplicacion(0,rpg[rs][1],0,rpg[rt][1],"BAJAMEDIA");
    auxi=rpg[32][1] >> 31;    //Vamos a hacer la extensi¢n de signo
    if (auxi!=0)   //Si auxi es distinto de cero
      rpg[32][0]=0xFFFFFFFF;  //ponemos la parte alta del registro todo a 1
    else if (auxi==0)    //Si auxi es igual a cero
      rpg[32][0]=0x00000000;  //ponemos la parte alta del registro todo a 0

    //hacemos la multiplicaci¢n y guardamos resultado en los registros
    rpg[33][1]=multiplicacion(0,rpg[rs][1],0,rpg[rt][1],"BAJA");
    auxi=rpg[33][1] >> 31;  //Vamos a hacer la extensi¢n de signo
    if (auxi!=0)  //Si auxi es distinto de cero
      rpg[33][0]=0xFFFFFFFF;  //ponemos la parte alta del registro todo a 1
    else if (auxi==0)   //Si auxi es igual a cero
      rpg[33][0]=0x00000000;  //ponemos la parte alta del registro todo a 0
  }
}    //fin de la instrucci¢n MULTU

void ERET (ULONG instruccion)  //Exception Return
{
  sr = sr & 0xFC;   //Limpiamos los dos bits de menor peso
  sr = sr | 0x02;  //Ponemos en modo usuario
  gotoxy (2,19);
  printf ("¯ Usuario       Ì");
}

/*A partir de aqui voy a implementar todas las excepciones*/
/*******************************************************/
/*********************EXCEPCIONES***********************/
/*******************************************************/

void General_Exception (void)
{
   sr = sr & 0xFC;   //Ponemos en modo Kernel.
```

```
   gotoxy (2,19);
   printf ("¯ Kernel        Ì");
}

/*******************************************************/
/*********************FORMATOS**************************/
/*******************************************************/

void Formato_cero (ULONG instruccion,UCHAR dec)
/*Este formato se utiliza para las instrucciones con mnem¢nicos de la
siguiente manera: NOMBRE_INSTR RD,RS,RT. Las instrucciones que usan este
formato son: ADD, ADDU, AND, DADD, DADDU, DSUB, DSUBU, NOR, OR, SLT,
SLTU, SUB, SUBU, XOR*/
{
  UCHAR rs,rt,rd;  //Para coger el n£mero de los registros

  rs=Campo ("RS",instruccion);  //Cogemos el campo rs de la instrucci¢n
  rt=Campo ("RT",instruccion);  //Cogemos el campo rt de la instrucci¢n
  rd=Campo ("RD",instruccion);  //Cogemos el campo rd de la instrucci¢n
  /*ahora vamos a imprimir el mnem¢nico por pantalla*/
  printf ("%s r%d,r%d,r%d        ",formatos[dec].nombre,rd,rs,rt);
}

void Formato_uno (ULONG instruccion,UCHAR dec)
/*Este formato se utiliza para las instrucciones con mnem¢nicos de la
siguiente manera: NOMBRE_INSTR RD,RT,RS. Las instrucciones que usan este
formato son: DSLLV, DSRAV, DSRLV, SLLV, SRAV, SRLV*/
{
  UCHAR rs,rt,rd;  //Para coger el n£mero de los registros

  rs=Campo ("RS",instruccion);  //Cogemos el campo rs de la instrucci¢n
  rt=Campo ("RT",instruccion);  //Cogemos el campo rt de la instrucci¢n
  rd=Campo ("RD",instruccion);  //Cogemos el campo rd de la instrucci¢n
  /*Ahora vamos a imprimir el mnem¢nico por pantalla*/
  printf ("%s r%d,r%d,r%d        ",formatos[dec].nombre,rd,rt,rs);
}

void Formato_dos (ULONG instruccion,UCHAR dec)
/*Este formato se utiliza para las instrucciones con mnem¢nicos de la
siguiente manera: NOMBRE_INSTR RS,RT. Las instrucciones que usan este
formato son: DDIV, DDIVU, DIV, DIVU, DMULT, DMULTU, MULT, MULTU, TEQ, TGE,
TGEU, TLT, TLTU, TNE*/
{
  UCHAR rs,rt;  //Para coger el n£mero de los registros

  rs=Campo ("RS",instruccion);  //Cogemos el campo rs de la instrucci¢n
  rt=Campo ("RT",instruccion);  //Cogemos el campo rt de la instrucci¢n
  /*Ahora vamos a imprimir el mnem¢nico por pantalla*/
  printf ("%s r%d,r%d            ",formatos[dec].nombre,rs,rt);
}

void Formato_tres (ULONG instruccion,UCHAR dec)
/*Este formato se utiliza para las instrucciones con mnem¢nicos de la
siguiente manera: NOMBRE_INSTR RD,RT,SA. Las instrucciones que usan este
formato son: DSLL, DSLL32, DSRA, DSRA32, DSRL, DSRL32, SLL, SRA, SRL*/
```

```
{
  UCHAR rd,rt,sa;  //Para coger el n£mero de los registros y desplazamiento

  rd=Campo ("RD",instruccion);  //Cogemos el campo rs de la instrucci¢n
  rt=Campo ("RT",instruccion);  //Cogemos el campo rt de la instrucci¢n
  sa=(instruccion & 0x000007C0)>>6;  //Cogemos el desplazamiento de bits.

  /*Ahora vamos a imprimir el mnem¢nico por pantalla*/
  printf ("%s r%d,r%d,%Xh        ",formatos[dec].nombre,rd,rt,sa);
}

void Formato_cuatro (ULONG instruccion,UCHAR dec)
/*Este formato se utiliza para las instrucciones con mnem¢nicos de la
siguiente manera: NOMBRE_INSTR RS. Las instrucciones que usan este
formato son: JR, MTHI, MTLO*/
{
  UCHAR rs;  //Para coger el n£mero de registro

  rs=Campo ("RS",instruccion);  //Cogemos el campo rs de la instrucci¢n

  /*Ahora vamos a imprimir el mnem¢nico por pantalla*/
  printf ("%s r%d                ",formatos[dec].nombre,rs);
}

void Formato_cinco (ULONG instruccion,UCHAR dec)
/*Este formato se utiliza para las instrucciones con mnem¢nicos de la
siguiente manera: NOMBRE_INSTR RD. Las instrucciones que usan este
formato son: MFHI, MFLO*/
{
  UCHAR rd;  //Para coger el n£mero de registro

  rd=Campo ("RD",instruccion);  //Cogemos el campo rd de la instrucci¢n

  /*Ahora vamos a imprimir el mnem¢nico por pantalla*/
  printf ("%s r%d                ",formatos[dec].nombre,rd);
}

void Formato_seis (ULONG instruccion,UCHAR dec)
/*Este formato se utiliza para las instrucciones con mnem¢nicos de la
siguiente manera: NOMBRE_INSTR RS ¢ NOMBRE_INSTR RD,RS. Las instrucciones
que usan este formato son: JALR RS ¢ JALR RD,RS*/
{
  UCHAR rd,rs;  //Para coger los n£meros de registro

  rd=Campo ("RD",instruccion);  //Cogemos el campo rd de la instrucci¢n
  rs=Campo ("RS",instruccion);  //Cogemos el campo rs de la instrucci¢n

  /*Ahora vamos a imprimir el mnem¢nico por pantalla*/
  if (rd==0)
    printf ("%s r%d                ",formatos[dec].nombre,rs);
  else if (rd!=0)
    printf ("%s r%d,r%d            ",formatos[dec].nombre,rd,rs);
}

void Formato_siete (ULONG instruccion,UCHAR dec)
```

```
/*Este formato se utiliza para las instrucciones con mnem¢nicos de la
siguiente manera: NOMBRE_INSTR RT,RS,INMEDIATO. Las instrucciones
que usan este formato son: ADDI, ADDIU, ANDI, DADDI, DADDIU, ORI, SLTI,
SLTIU, XORI.*/
{
  UCHAR rt,rs;  //Para coger los n£meros de registro
  ULONG inmediato;  //Para coger el dato inmediato

  rt=Campo ("RT",instruccion);  //Cogemos el campo rt de la instrucci¢n
  rs=Campo ("RS",instruccion);  //Cogemos el campo rs de la instrucci¢n
  inmediato=instruccion & 0x0000FFFF; //Cogemos el campo inmediato de la
inst.

  /*Ahora vamos a imprimir el mnem¢nico por pantalla*/
  printf ("%s r%d,r%d,%04Xh      ",formatos[dec].nombre,rt,rs,inmediato);
}

void Formato_ocho (ULONG instruccion,UCHAR dec)
/*Este formato se utiliza para las instrucciones con mnem¢nicos de la
siguiente manera: NOMBRE_INSTR RS,RT,OFFSET. Las instrucciones
que usan este formato son: BEQ, BEQL, BNE, BNEL*/
{
  UCHAR rt,rs;  //Para coger los n£meros de registro
  ULONG offset;  //Para coger el dato offset

  rt=Campo ("RT",instruccion);  //Cogemos el campo rt de la instrucci¢n
  rs=Campo ("RS",instruccion);  //Cogemos el campo rs de la instrucci¢n
  offset=instruccion & 0x0000FFFF; //Cogemos el campo offset de la inst.

  /*Ahora vamos a imprimir el mnem¢nico por pantalla*/
  printf ("%s r%d,r%d,%04Xh      ",formatos[dec].nombre,rs,rt,offset);
}

void Formato_nueve (ULONG instruccion,UCHAR dec)
/*Este formato se utiliza para las instrucciones con mnem¢nicos de la
siguiente manera: NOMBRE_INSTR RS,OFFSET. Las instrucciones
que usan este formato son: BGTZ, BGTZL, BLEZ, BLEZL*/
{
  UCHAR rs;  //Para coger los n£meros de registro
  ULONG offset;  //Para coger el dato offset

  rs=Campo ("RS",instruccion);  //Cogemos el campo rs de la instrucci¢n
  offset=instruccion & 0x0000FFFF; //Cogemos el campo offset de la inst.

  /*Ahora vamos a imprimir el mnem¢nico por pantalla*/
  printf ("%s r%d,%04Xh          ",formatos[dec].nombre,rs,offset);
}

void Formato_diez (ULONG instruccion,UCHAR dec)
/*Este formato se utiliza para las instrucciones con mnem¢nicos de la
siguiente manera: NOMBRE_INSTR RT,INMEDIATO. Las instrucciones
que usan este formato son: LUI*/
{
  UCHAR rt;  //Para coger los n£meros de registro
  ULONG inmediato;  //Para coger el dato inmediato
```

```
  rt=Campo ("RT",instruccion);  //Cogemos el campo rt de la instrucci¢n
  inmediato=instruccion & 0x0000FFFF; //Cogemos el campo inmediato de la
inst.

  /*Ahora vamos a imprimir el mnem¢nico por pantalla*/
  printf ("%s r%d,%04Xh          ",formatos[dec].nombre,rt,inmediato);
}

void Formato_once (ULONG instruccion,UCHAR dec)
/*Este formato se utiliza para las instrucciones con mnem¢nicos de la
siguiente manera: NOMBRE_INSTR RT,OFFSET(BASE). Las instrucciones
que usan este formato son: LB, LBU, LD, LDL, LDR, LH, LHU, LW, LWL, LWR, LWU,
SB, SD, SDL, SDR, SH, SW, SWL, SWR*/
{
  UCHAR rt, base;  //Para coger los n£meros de registro
  ULONG offset;  //Para coger el offset

  rt=Campo ("RT",instruccion);  //Cogemos el campo rt de la instrucci¢n
  base=Campo ("RS",instruccion);  //Cogemos el campo rs de la instrucci¢n
  offset=instruccion & 0x0000FFFF; //Cogemos el campo offset de la inst.

  /*Ahora vamos a imprimir el mnem¢nico por pantalla*/
  printf ("%s r%d,%04Xh",formatos[dec].nombre,rt,offset);
  printf ("(r%d)      ",base);
}

void Formato_doce (ULONG instruccion,UCHAR dec)
/*Este formato se utiliza para las instrucciones con mnem¢nicos de la
siguiente manera: NOMBRE_INSTR RS,OFFSET. Las instrucciones
que usan este formato son: BGEZ, BGEZAL, BGEZALL, BGEZL, BLTZ, BLTZAL,
BLTZALL, BLTZL*/
{
  UCHAR rs;  //Para coger los n£meros de registro
  ULONG offset;  //Para coger el offset

  rs=Campo ("RS",instruccion);  //Cogemos el campo rs de la instrucci¢n
  offset=instruccion & 0x0000FFFF; //Cogemos el campo offset de la inst.

  /*Ahora vamos a imprimir el mnem¢nico por pantalla*/
  printf ("%s r%d,%04Xh          ",formatos[dec].nombre,rs,offset);
}

void Formato_trece (ULONG instruccion,UCHAR dec)
/*Este formato se utiliza para las instrucciones con mnem¢nicos de la
siguiente manera: NOMBRE_INSTR RS,IMMEDIATO. Las instrucciones
que usan este formato son: TEQI, TGEI, TGEIU, TLTI, TLTIU, TNEI*/
{
  UCHAR rs;  //Para coger los n£meros de registro
  ULONG immediato;  //Para coger el offset

  rs=Campo ("RS",instruccion);  //Cogemos el campo rs de la instrucci¢n
  immediato=instruccion & 0x0000FFFF; //Cogemos el campo inmediato de la
inst.
```

```
  /*Ahora vamos a imprimir el mnem¢nico por pantalla*/
  printf ("%s r%d,%04Xh          ",formatos[dec].nombre,rs,immediato);
}

void Formato_catorce (ULONG instruccion,UCHAR dec)
/*Este formato se utiliza para las instrucciones con mnem¢nicos de la
siguiente manera: NOMBRE_INSTR TARGET. Las instrucciones
que usan este formato son: J, JAL*/
{
  ULONG target;  //Para coger el target

  target=instruccion & 0x03FFFFFF; //Cogemos el campo target de la inst.

  /*Ahora vamos a imprimir el mnem¢nico por pantalla*/
  printf ("%s %lXh               ",formatos[dec].nombre,target);
}

void Formato_quince (ULONG instruccion,UCHAR dec)
/*Este formato se utiliza para las instrucciones con mnem¢nicos de la
siguiente manera: NOMBRE_INSTR. Las instrucciones
que usan este formato son: ERET*/
{
  /*Ahora vamos a imprimir el mnem¢nico por pantalla*/
  printf ("%s                    ",formatos[dec].nombre);
}
```

BIBLIOGRAFÍA

- **MANUAL DE USUARIO DEL MICROPROCESADOR MIPS R4000**

  Autor: Joe Heinrich

  Título: MIPS R4000 Microprocessor User's Manual

  Año Publicación: 1994