11 Memoria

11.1 Introducción

Cuando, en el capítulo 7, diseñábamos unidades de proceso específicas para un problema, lo hacíamos con tantos registros como necesitábamos. Sin embargo ahora, usando el procesador construido con la UPG, el número de registros disponibles está fijado a 8, que es un número pequeño. En los ejemplos de problemas que hemos resuelto hasta ahora no hemos necesitado más registros porque los problemas requerían pocos resultados parciales y en algunos casos tenían además pocos datos o si tenían muchos se usaban sólo una vez y por ello no hacía falta almacenarlos durante todo el proceso de resolución. Pero esto no va a ser así en general. Los problemas para los que se usa un computador suelen tener muchos datos iniciales y muchos resultados parciales y/o finales que tienen que estar almacenados dentro del computador, pues se van a usar varias veces durante la resolución del problema.

Si esto es así, ¿por qué no hemos puesto un banco de registros mucho más grande en la UPG? Hay varias razones para no hacerlo. Una de ellas es que el número de bits necesarios para especificar un registro en el campo asignado para ello dentro de la instrucción de lenguaje máquina crecería mucho (por ejemplo pasando de 3 a 16 bits, si quisiéramos tener 2^{16} registros), y por lo tanto cada instrucción necesitaría más de 48 bits (al menos las que requieren especificar tres registros). Esto requeriría mucho espacio de memoria para almacenar las instrucciones de los programas. Otra razón, más importante que la anterior, se comentara al final del capítulo, cuando conozcamos las diferencias entre el modo de acceder a un dato en el banco de registros o a uno en memoria. Por último, tener un banco de registros muy grande en vez de un banco de registros pequeño y una memoria de datos muy grande, tiene importantes repercusiones en el tiempo de ciclo del procesador y por lo tanto en el tiempo de ejecución de los programas, que no veremos en detalle hasta los capítulos 12 y 13.

La solución adoptada por los arquitectos de computadores, que estudiamos a continuación, consiste en dejar en el procesador un banco de registros pequeño (de 8 registros, como el que nosotros tenemos en nuestra UPG, o de 32 como en muchos de los procesadores RISC comerciales) y conectar el procesador a una memoria externa grande, que pueda almacenar muchos datos.

En este capítulo cuando hablamos de memoria nos referimos a la memoria de datos del computador, que es de lectura y de escritura, a diferencia de la memoria de instrucciones del computador Harvard uniciclo que estamos construyendo ahora, que es de solo lectura. No obstante, en el capítulo 13 diseñaremos el computador SISC von Neumann, en el que solo hay una memoria de lectura/escritura que contiene tanto el programa como los datos.

Al igual que hemos hecho con la entrada/salida de datos en el capítulo 9, ahora vamos a conectar una memoria de datos al procesador (UCG+UPG). Esto es, vamos a dotar al procesador de un nuevo espacio de direcciones, denominado **espacio de direcciones de memoria**, y de cuatro nuevas instrucciones para leer o escribir el contenido de una de estas direcciones de memoria. Dos de estas instrucciones son para acceder, leer/escribir, un byte, 8 bits, (*Load Byte*, `LDB` / *Store Byte*, `STB`) y las otras dos para leer/escribir un word, 16 bits (*Load*, `LD` / *Store*, `ST`).

En primer lugar presentamos el módulo de memoria que vamos a usar para construir, a continuación, el sistema de memoria (`MEM`) que permite la lectura/escritura de bytes y de words. Luego definimos la sintaxis en lenguaje máquina y ensamblador SISA y la semántica de las cuatro nuevas instrucciones de acceso a memoria para, a continuación, conectar el sistema de memoria (MEM) al procesador de

propósito general (UCG+UPG) que junto con el sistema de entrada/salida (IOKey-Print) forman el computador SISC Harvard uniciclo ya prácticamente terminado (excepto el cálculo del tiempo de ciclo y un pequeño ajuste en el tiempo de generación de alguna señal de la palabra de control, que veremos en la primera mitad del capítulo siguiente). Por último presentamos dos ejemplos de programas que hacen uso de la memoria de datos.

11.2 Módulo de memoria RAM

Presentamos el dispositivo denominado memoria RAM[1] como una caja negra: definimos sus conectores de entrada y de salida, la funcionalidad del dispositivo y sus requerimientos temporales para que el funcionamiento sea correcto, pero no vamos a entrar en el detalle de su construcción interna. La función de la memoria es como la del banco de registros, se tiene que poder almacenar información (escribir) para luego poderla recuperar (leer). Podemos pensar que, a nivel lógico, la memoria es parecida a un banco de registros, pero realmente no está construida de la misma forma, sino que usa otros dispositivos a nivel de transistor, que no estudiamos aquí, para conseguir mucha capacidad de almacenamiento en poco espacio y con un retardo temporal aceptable[2].

Lo usual en un computador es procesar datos de varios tamaños como por ejemplo caracteres alfanuméricos (cada uno de ellos codificado en 8 bits: un *byte*), números enteros codificados en complemento a dos en 16 bits (un *word*) y/o 32 bits (un *long word*) o incluso números reales codificados en coma flotante con 64 bits (*quad word)*. Esta nomenclatura de *word* para 16 bits y sus múltiplos es la que usamos aquí y la que usan algunos fabricantes de computadores, pero otros definen el *word* como un vector de 32 bits, siendo el *half word* 16 bits y el *long word* 64.

El sistema de memoria del computador que estamos construyendo, para que resulte sencillo, solo podrá acceder (leer/ escribir), ejecutando una instrucción en un ciclo, a un byte o a un word. Si un programa requiere datos de 32 bits estos deberán almacenarse (y procesarse) cada dato como dos datos consecutivos de 16 bits.

Como el tamaño mínimo de un dato que se puede acceder en nuestra memoria (y en cualquiera de las memorias de los computadores comerciales) es el byte, cada byte de memoria tiene que tener una dirección distinta, para poderlos diferenciar, para poderlos acceder individualmente. Esto es, la **unidad de direccionamiento** de la memoria del computador es el byte.

---

1. Las siglas RAM vienen de *Ramdom Access Memory* que significa memoria de acceso aleatorio, esto es, que al acceder a una palabra cualquiera de la memoria, independientemente de a qué dirección se esté accediendo o cuáles sean las palabras que se hayan accedido con anterioridad, el tiempo de acceso es siempre el mismo. Pero siempre que nos referimos a una memoria RAM nos referimos a una memoria de lectura y escritura, frente a las memorias ROM que son de sólo lectura, aunque este significado no se encuentre en las siglas RAM pero sí en las ROM (*Read Only Memory*).
2. Algunas diferencias de la realización interna entre una memoria RAM (y más concretamente SRAM: RAM estática) y el banco de registros que hemos visto en este curso son: 1) La celda que almacena un bit en una RAM real no está formada por un biestable D activado por flanco, como en nuestro modelo de banco de registros, sino que es una celda activada por nivel, que se construye de forma más simple, usando unos pocos transistores. La celda no se escribe con el flanco ascendente de una señal sino manteniendo a 1 durante un determinado tiempo una señal de permiso de escritura. 2) Para seleccionar qué palabra se conecta al bus de datos de salida, en el banco de registros usamos un multiplexor de buses de 8 a 1, porque tenemos sólo 8 registros. En una memoria de 2^{16} palabras tendríamos que disponer de un multiplexor de 2^{16} a 1 que sería muy costoso en hardware. En vez de multiplexores se usa otra tecnología con unos dispositivos electrónicos denominados *sense amplifiers* que requieren mucho menos espacio.

Como nuestro computador tiene tamaño de palabra de 16 bits (los registros son de 16 bits, opera directamente sobre datos de 16 bits almacenados en los registros, el PC tiene 16 bits,...) decidimos tener un sistema de memoria con capacidad para 2^16 bytes así un registro puede contener directamente una dirección de memoria. Así, cada dirección de memoria tiene 16 bits y contiene un dato de 8 bits (un byte).

Veamos a continuación el funcionamiento, como caja negra, de un módulo de memoria RAM que tiene una capacidad de almacenamiento de 2^{16} bytes y sobre el que se pueden hacer dos acciones leer (load) un byte de una dirección concreta o escribirlo (store). En la siguiente sección diseñaremos el sistema de memoria definitivo de nuestro computador (usando dos módulos de 2^15 bytes) para poder acceder a datos tanto de tamaño byte como de tamaño word.

De momento, a nivel conceptual podemos ver la memoria como un vector o tabla, que denominamos $\texttt{MEM}_\texttt{b}$, de 2^{16} bytes (2^{16} posiciones de memoria donde cada posición tiene 8 bits). Nos referimos al elemento `i` del vector, o mejor, a la posición `i` de memoria, o aún de forma más precisa al contenido de la posición de memoria cuya dirección es `i`, como $\texttt{MEM}_\texttt{b}\texttt{[i]}$, para `i` = 0 hasta 2^{16}-1.

La figura 11.1 muestra las cinco primeras posiciones de la memoria (direcciones de 0 a 4) expresando la dirección, @, en decimal y su contenido, $\texttt{MEM}_\texttt{b}\texttt{[@]}$, a) en binario y b) en hexadecimal. Dibujaremos la dirección 0 de memoria en la parte superior de la tabla, la 1 debajo y así sucesivamente. Además, dibujaremos siempre el bit/dígito 0 de cada byte, el de menor peso, a la derecha y el de más peso a la izquierda, como hacemos cuando el dato está en un registro. Según la figura $\texttt{MEM}_\texttt{b}\texttt{[3]}$ = `0xF7`.

Con este convenio podemos incluso no indicar las direcciones, como el formato que usa LogicWorks, en el que sólo se expresan los contenidos en hexadecimal separados por un espacio o un cambio de línea, comenzando por la dirección 0. Esto se muestra en la caja c) de la figura 11.1 para el mismo contenido que en a) y b).

a)

@	$\texttt{MEM}_\texttt{b}$[@] 7 6 5 4 3 2 1 0
0	0 0 0 0 0 0 1 1
1	0 0 0 0 0 1 1 0
2	0 0 0 1 1 0 1 1
3	1 1 1 1 0 1 1 1
4	0 0 0 0 1 1 0 0
...	...

b)

@	$\texttt{MEM}_\texttt{b}$ [@] (hexa)
0	0 3
1	0 6
2	1 B
3	F 7
4	0 C
...	...

c)

```
0 3   0 6   1 B   F 7
0 C   ...
```

Fig. 11.1 Dibujo de la memoria como un vector o tabla. La figura indica la dirección (en decimal) y el contenido de las primeras 5 posiciones de memoria, a) en binario y b) en hexadecimal. En c) se muestra el mismo contenido en formato LogicWorks.

Sobre esta memoria se puede hacer sólo una de dos posibles acciones en cada ciclo: leer un byte de memoria o escribirlo. La figura 11.2 muestra el símbolo de la este módulo de memoria (bloque MEM-64KB) con los nombres de los buses y señales de entrada y de salida. Tiene un bus de datos de lectura (bus de salida del módulo) y otro de datos de escritura (bus de entrada al módulo), ambos de 8 bits (un byte) que denominamos RD-MEM-8bits y WR-MEM-8bits respectivamente.

En esta memoria, aunque los buses de datos de lectura y escritura están separados, solamente hay un bus de direcciones de 16 bits (bus de entrada a la memoria) denominado ADDR-MEM, que indica la dirección de la palabra a acceder tanto en las operaciones de lectura como de escritura. Por ello, no es posible estar leyendo una posición de memoria y a la vez escribiendo otra, a diferencia de lo que ocurre en el banco de registros de nuestro procesador. Para que la memoria sepa, en cada momento, si queremos leer o escribir la posición de memoria que indica el bus de direcciones, se dispone de la señal de control de 1 bit (de entrada a la memoria) Wr-Mem. Cuando Wr-Mem vale 1 se escribe la memoria. Veamos ahora un modelo temporal simplificado de cómo se efectúan las dos posibles acciones sobre la memoria, primero la lectura de una de sus posiciones y después la escritura.

RD-MEM-8bits: Read Memory. Bus de 8 bits de salida del módulo.
WR-MEM-8bits: Write Memory. Bus de 8 bits de entrada al módulo.
ADDR-MEM: Address Memory. Bus de 16 bits de entrada al módulo.
Wr-Mem: Permiso de escritura. Señal de 1 bit de entrada al módulo.

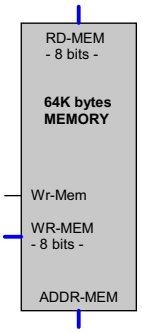


Fig. 11.2 Buses y señales de entrada y salida al módulo de memoria RAM: bloque DMEM-*2^16B.*

Lectura

El cronograma de lectura de memoria se muestra en la figura 11.3 junto con la señal de reloj del procesador. El tiempo de acceso a memoria es el tiempo que transcurre desde que son estables las señales del bus de direcciones, ADDR-MEM, hasta que se estabiliza en el bus de datos de lectura, RD-MEM-8bits, el contenido del byte de memoria que indica el bus de direcciones. El único parámetro temporal para la lectura de la memoria que es importante en el diseño de nuestro computador es el tiempo de acceso, T_{acc}, (*access time*).

En la primera versión de nuestro computador, que finalizaremos en el capítulo siguiente, en un ciclo se debe poder efectuar la lectura de una posición de memoria y escribir el dato leído en uno de los registros del banco de registros de la UPG. Ya veremos más adelante como se conecta la memoria a la UPG. El tiempo de ciclo del procesador (Tc en la figura) debe ser elegido por el diseñador del computador lo suficientemente grande para que, en el ciclo en el que se ejecuta una lectura de la memoria, de tiempo suficiente para

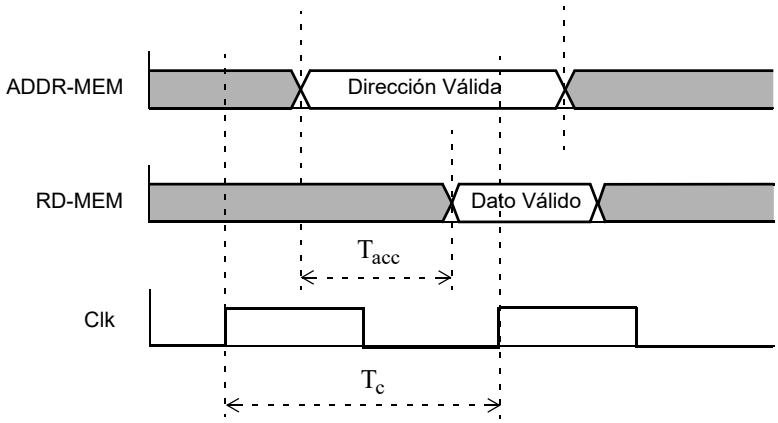


Fig. 11.3 Cronograma de una lectura de memoria.

- que el procesador genere la dirección de memoria a leer y la deje estable en el bus ADDR-MEM,
- que la memoria lea el contenido de la dirección y lo estabilice en el bus RD-MEM-8bits y
- que el dato leído pase del bus RD-MEM-8bits al bus de entrada del registro destino de la UPG y esté estable en la entrada del registro antes de que llegue el flanco ascendente de la señal de reloj, que indica el final del ciclo de reloj del procesador. (Como veremos más adelante, en nuestro computador, los 16 bits del registro destino se escribirán con el byte leído de memoria en los 8 bits de menor peso del registro y con el bit de signo del byte replicado 8 veces en los 8 bits de más peso)

El tiempo de ciclo del computador será calculado en detalle en el capítulo 13 para tener en cuenta las restricciones que impone la memoria de datos, que suponemos tiene un T_{acc} = 800 u.t.

Un último comentario sobre el cronograma. La memoria mantiene estable el dato leído en el bus RD-MEM-8bits durante un cierto tiempo después de que la dirección cambie. Este tiempo no nos interesa ya que no impone ninguna restricción al tiempo de ciclo: la dirección estará estable en el bus ADDR-MEM hasta un cierto tiempo después de que finalice el ciclo de acceso y se pase a ejecutar otra instrucción y por lo tanto, para cuando el dato deje de ser válido en el bus RD-MEM-8bits (que se conectará al bus RD-MEM de entrada a la UPG, previa extensión del bit se signo para formar 16 bits) seguro que el dato leído ya está almacenado en el registro destino del procesador.

Escritura

El cronograma de escritura de un byte en el módulo de memoria se muestra en la figura 11.4 junto con la señal de reloj del procesador. El dato que se encuentra en el bus de escritura, WR-MEM-8bits, se escribe en la posición de memoria que indica el bus de direcciones ADDR-MEM.

Al igual que en la lectura, en la primera versión de nuestro computador, también se podrá efectuar una escritura en memoria en cada ciclo del procesador. Como veremos en detalle en la sección 11.4.1, cuando conectemos la memoria al procesador, en nuestro caso, las señales del bus WR-MEM, que tiene el dato válido que se debe escribir en memoria, están estables en la entrada de la memoria bastante

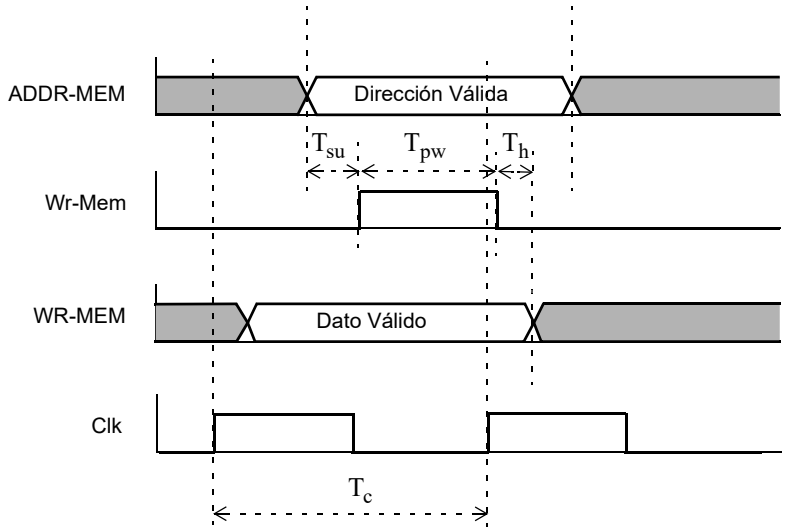


Fig. 11.4 Cronograma de una escritura en memoria.

antes que las señales del bus de direcciones, ADDR-MEM (como muestra el cronograma). Para asegurar que la escritura se efectúa en la posición deseada, la dirección debe estar estable a la entrada de la memoria, en el bus ADDR-MEM, al menos T_{su} (*set up time*) unidades de tiempo antes de que la señal de permiso de escritura Wr-Mem valga 1 (el dato también debe estar estable en la entrada WR-MEM T_{su} u.t. antes del pulso, pero como lo estará antes que ADDR-MEM, no impone ninguna restricción en nuestro caso). Para que la escritura se realice correctamente, la señal Wr-Mem debe permanecer activa durante un pulso de al menos T_{pw} (*pulse width time*) unidades de tiempo mientras la dirección y el dato están estables en sus buses de entrada a memoria T_{su} u.t. antes del flanco ascendente del pulso y T_h (*hold time*) u.t. después del flanco descendente del pulso.

Como se verá en el capítulo siguiente, la lógica de control del procesador pone a 1 la señal Wr-Mem durante la segunda mitad del ciclo de reloj del procesador. Por eso la señal de reloj del procesador debe ser diseñada para que:

- el tiempo de la primera mitad del ciclo de reloj (cuando la señal Clk vale 1) sea superior a la suma del tiempo desde que se inicia el ciclo hasta que se estabilizan las señales del bus ADDR-MEM más el tiempo T_{su} y para que, además,
- la duración de la segunda mitad del ciclo (cuando el reloj vale 0), que coincide con la duración del pulso en que Wr-Mem vale 1, sea superior al tiempo T_{pw} y que
- el dato se mantenga estable en el bus WR-MEM al menos T_h u.t. después de que termine el ciclo del procesador.

En nuestro caso T_{su}, T_{pw} y T_h son los únicos parámetros temporales de la escritura en memoria que pueden afectar al tiempo de ciclo del procesador, que se calculará en detalle en el capítulo 12.

Un último comentario sobre el cronograma de escritura. 30 u.t. después del final del ciclo del procesador y el inicio del siguiente, que es el tiempo que tarda en pasar el valor 1 de Clk a través de la Not, para convertirse en 0 y de la And-1 con WrMem para poner a 0 Wr-Mem, los buses ADDR-MEM y WR-MEM permanecen estables durante un cierto tiempo hasta que el procesador genera las nuevas señales para el siguiente ciclo. Este tiempo en que los buses están estables después de que Wr-Mem pase a ser 0 es muy superior a T_h lo que asegura el buen funcionamiento de la memoria.

11.3 Sistema de memoria con acceso a byte y a word

La **unidad de direccionamiento** de memoria es el byte(8 bits): cada una de las 2^16 direcciones de memoria diferentes (0, …, 2^16-1) que puede haber en el bus de 16 bits ADDR-MEM hace referencia a uno de los 2^16 bytes de la memoria.

Tipos de accesos. Se puede acceder para leer, *load*, (con Wr-Mem = 0) y para escribir, *store*, (con Wr-Mem = 1). Además, se puede acceder (leer o escribir) a uno de los 2^16 bytes (con Byte = 1) o a uno de los 2^15 words (con Byte = 0).

Memoria entrelazada de dos módulos. Se denomina así a la forma de asignar las direcciones de memoria entre los módulos de memoria que la forman. En nuestro caso tenemos una memoria con direcciones de 16 bits (ADDR-MEM tiene 16 bits) y cada dirección contiene un dato de 8 bits (un byte). La memoria está formada por dos módulos cada uno de 2^15 bytes (el bus de dirección de cada módulo tiene 15 bits y cada dirección contiene un dato de tamaño byte). La figura 11.5 muestra la asignación de cada dirección de 16 bits, expresada en decimal, a cada byte de cada módulo. Las direcciones pares se asignan al módulo 0 y las impares al 1. El contenido de cada byte se ha expresado en hexadecimal y los valores concretos son arbitrarios. El byte cuya dirección es ADDR-MEM se encuentra en el módulo 0 si el bit de menor peso de ADDR-MEM es 0 y en el 1 si es 1. Y se encuentra en la dirección (de 15 bits) de ese módulo formada por los 15 bits de más peso de ADDR-MEM. Esto es, el byte con dirección ADDR-MEM se encuentra en la dirección ADDR-MEM<15..1> del módulo ADDR-MEM<0>. Para los datos contenidos en la memoria de la figura 11.5 el byte contenido en la dirección 3 (ADDR-MEM = 3) es 0x32. Esto lo denotamos como $MEM_b[3]$ =0x32.

Acceso a palabra (word). Las palabras están en formato *Little Endian* y alineadas a byte par. Esto es, la dirección de una palabra de memoria tiene que ser par (después veremos qué ocurre si la dirección es impar) y el byte con la dirección par es el byte de menor peso de la palabra mientras que la dirección siguiente (dirección par + 1) contiene el byte de más peso de la palabra. La memoria tiene una entrada nueva, Byte, de un bit, para diferenciar entre acceso a byte, cuando vale 1, y acceso a word, cuando vale 0. Por ejemplo, el word que contiene la dirección 4 de la memoria de la figura 11.5 es el 0xFFF2. Esto lo denotamos como $MEM_w[4]$ = 0xFFF2, ya que $MEM_b[4]$ = 0xF2 y $MEM_b[5]$ = 0xFF.

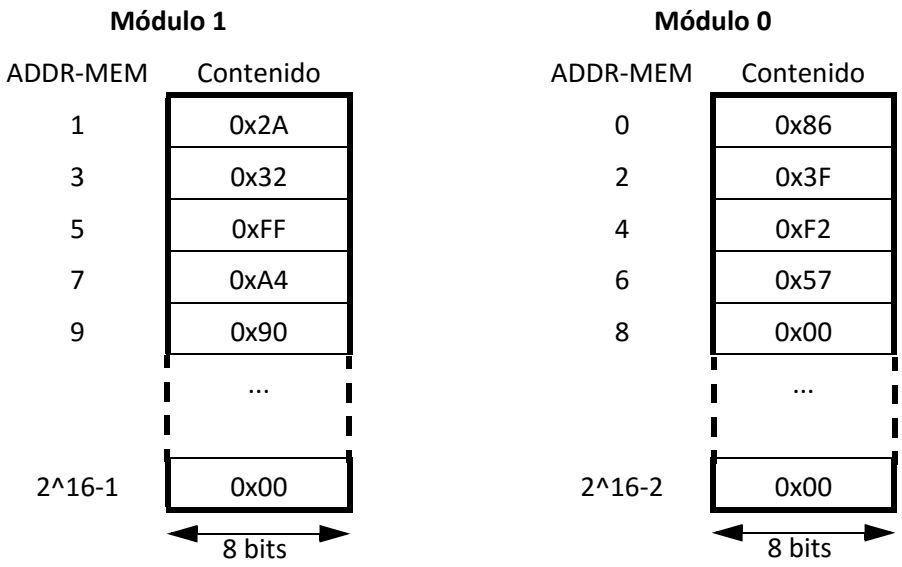


Fig. 11.5 Memoria entrelazada. Asignación de direcciones de memoria (de 16 bits) a dos módulos de memoria. Cada uno con una capacidad de 2^15 bytes.

Circuito interno del sistema de memoria: bloque MEM-64KB-32KW.

Vamos a diseñar ahora el sistema de memoria de nuestro computador. Tiene que tener una capacidad de 2^16 bytes (64KB), pero tiene que poder leer o escribir en un ciclo un byte o un word, según se desee. El bloque de este sistema de memoria se muestra en la figura 11.6.

RD-MEM: Read Memory. Bus de 16 bits de salida del bloque.

WR-MEM: Write Memory. Bus de 16 bits de entrada al bloque.

ADDR-MEM: Address Memory. Bus de 16 bits de entrada al bloque.

Wr-Mem: Permiso de escritura. Señal de 1 bit de entrada al bloque.

Byte: Control de acceso a Byte o a Word. Señal de 1 bit de entrada al bloque.

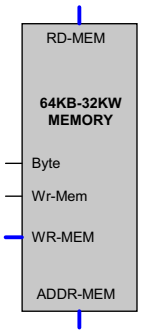


Fig. 11.6 Buses y señales de entrada y salida al módulo de memoria RAM: bloque DMEM.

Vamos a diseñar su circuito interno en tres fases. En la primera nos centraremos en la función de lectura de memoria y en la segunda fase en la escritura y en la tercera juntaremos las dos funciones. En cada una de las dos primeras fases, lectura y escritura, realizaremos el diseño en tres pasos. En el primero diseñaremos el circuito interno del sistema como si sólo tuviera que accederse a byte, en el segundo como si fuera solo a word y en el tercero juntaremos los circuitos para que pueda hacerse a cualquiera de los dos tamaños de datos.

El circuito para la fase 1 (solo lectura) paso 1 (de solo bytes) se ve en la figura 11.7. Se usa un único módulo de memoria de 2^{16} bytes, 64 KB. Ya conocemos del capítulo anterior el bloque 8-SE que representa a su salida en complemento a dos con 16 bits en mismo número entero que hay en su entrada representado en 8 bits, mediante la extensión del bit de signo (bit 7) hasta formar 16 bits.

El circuito para la fase 1 (solo lectura) paso 2 (de solo words) se ve en la figura 11.8 y los esquemas internos de los nuevos bloques 16-to-15H1L y 8L8H-to16 se muestran al final del capítulo (figura 11.32 y 11.33). El bloque 16-to-15H1L pasa de un bus de 16 bits a un bus de 15 bits, con los 15 bits de mayor peso de la entrada y una señal de un bit con el bit de menor peso de la entrada. El bloque 8H8L-to16 pasa de dos buses de entrada de 8 bits a uno de salida de 16 bits formado por la concatenación de los dos buses de 8 bits.

El circuito interno para la fase 1 (solo lectura) paso 3 (de bytes o de words) se ve en la figura 11.9. Los bloques que aparecen en su diseño ya los conocemos.

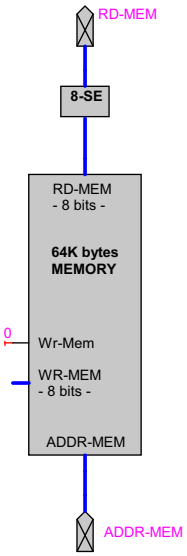


Fig. 11.7
Fase 1 paso 1.
Solo lectura
de solo byte

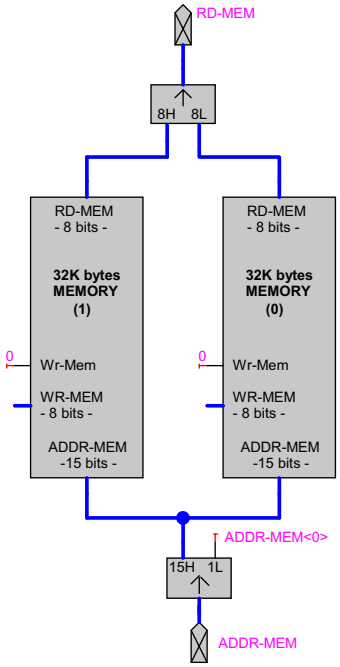


Fig. 11.8
Fase 1 paso 2.
Solo lectura
de solo word

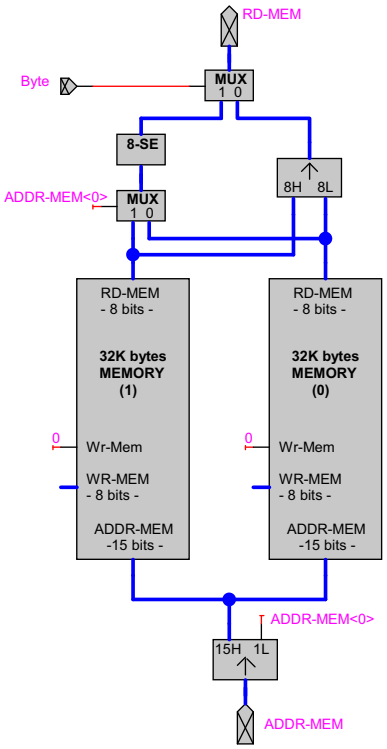


Fig. 11.9
Fase 1 paso 3.
Solo lectura
de byte o de word

El circuito interno para la fase 2 (solo escritura) paso 1 (de solo bytes) se ve en la figura 11.10 y el esquema interno del bloque 16-to-8H8L que se usa en su diseño se muestra en la figura 11.34. Este bloque pasa de un bus de entrada de 16 bits a dos buses de salida de 8 bits, uno de ellos con los 8 bits de menor peso de la entrada y el otro con los 8 de más peso).

El circuito interno para la fase 2 (solo escritura) paso 2 (de solo words) se ve en la figura 11.11

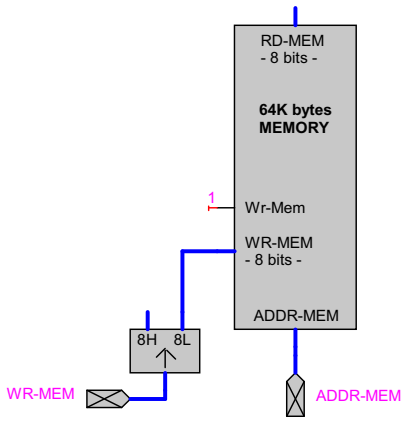


Fig. 11.10

Fase 2 paso 1.
Solo escritura
de solo byte

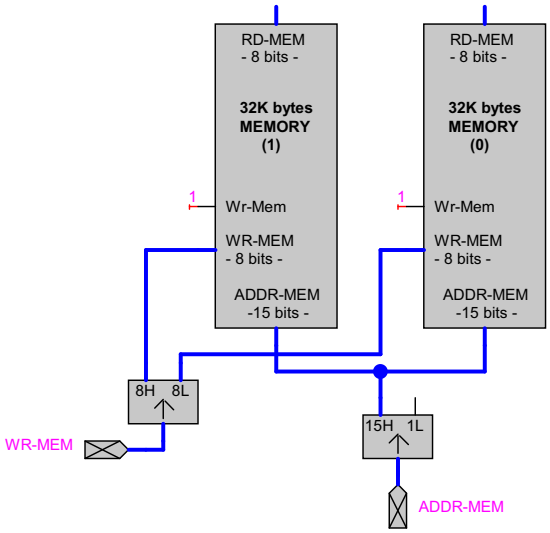


Fig. 11.11

Fase 2 paso 2.
Solo escritura
de solo word

El circuito interno para la fase 2 (solo escritura) paso 3 (de bytes o de words) se ve en la figura 11.12

El circuito de la izquierda, que llamamos "Circuito de control de los accesos", obtiene las señales de permiso de escritura de cada módulo, Wr-Mem-1 y Wr-Mem-0 en función de Wr-Mem, ADDR-MEM<0> y Byte. Se ha implementado minimizando por Karnaugh la tabla de verdad de la figura 11.13. La implementación de las salidas Wr-Mem-1 y Wr-Mem-0 para las dos filas grises (acceso a Word desalineado) que se ha elegido es la que hace que el acceso a Word desalineados se conviertan en un acceso alineado a la dirección que resulta de poner a 0 el bit de menor peso de ADDR-MEM. El resultado es:

Wr-Mem-1 = Wr-Mem · !Byte + Wr-Mem ·ADDR-MEM<0>

Wr-Mem-0 = Wr-Mem · !Byte + Wr-Mem ·!ADDR-MEM<0>

El circuito interno para la fase 3, diseño completo (de lectura o escritura de bytes o de words), se ve en la figura 11.14.

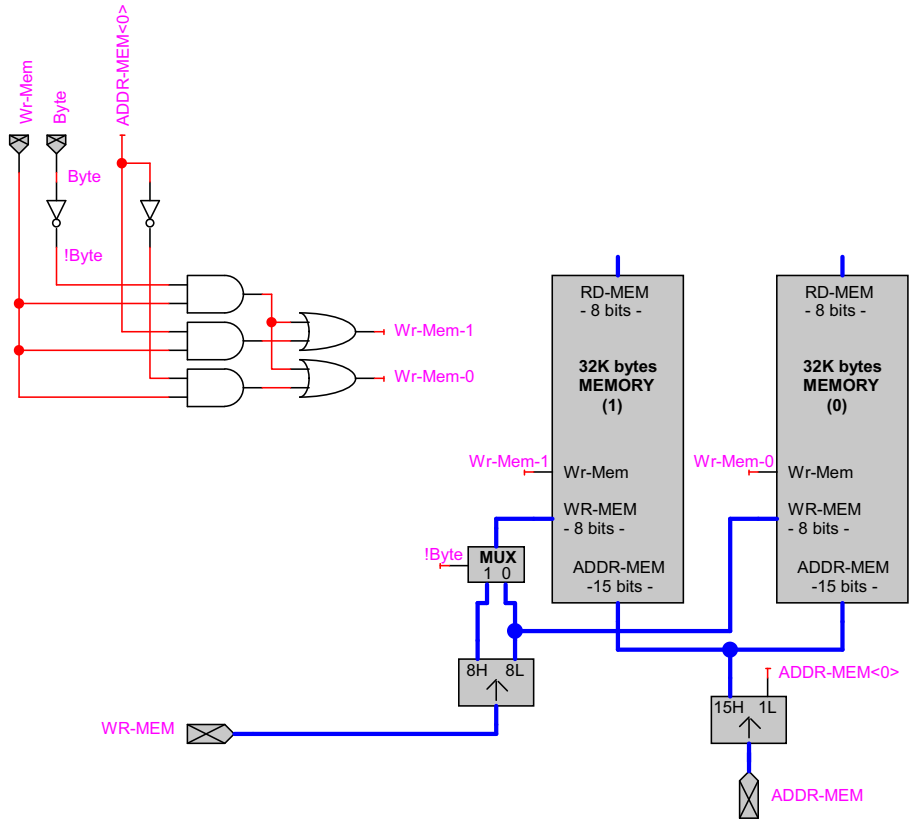


Fig. 11.12 Fase 2 paso 3. Solo escritura de byte o word.

¿Qué ocurre en caso de intento de acceso a Word desalineado? Si en un acceso a Word (Byte = 0) la dirección presente en ADDR-MEM es impar (ADDR-MEM<0> = 1), el acceso podría calificarse de ilegal y el resultado depender de la implementación. En la implementación que hemos realizado (bloque MEM-64KB-32KW), en un acceso a palabra (Byte = 0) si la dirección es impar, tanto en lectura como en escritura, el resultado es el mismo que acceder a la dirección par que se obtiene poniendo a 0 el bit de menor peso de la dirección dada (que inicialmente es impar). Esto es:

La lectura/escritura de Word con dirección impar tiene el mismo resultado que la lectura/escritura del Word con la dirección impar dada menos uno

Wr-Mem	Byte	ADDR-MEM<0>	Wr-Mem-1	Wr-Mem-0
0	0	0	0	0
0	0	1	0	0
0	1	0	0	0
0	1	1	0	0
1	0	0	1	1
1	0	1	1	1
1	1	0	0	1
1	1	1	1	0

Fig. 11.13 Tabla de verdad de las señales de permiso de escritura de cada módulo de memoria.

11.4 Instrucciones de acceso a memoria LD/LDB y ST/STB.

Formato de las instrucciones

Ahora tenemos que conectar la memoria de datos al procesador y dotar a este de nuevas instrucciones que permitan leer y escribir datos de tamaño byte y word. Definimos cuatro instrucciones de acceso a memoria en nuestro computador:

- LD (*Load*) y LDB (*Load Byte*), que leen un word y un byte, respectivamente, de memoria y lo escriben en un registro, Rd. En LDB los 16 bits que se escriben en el registro destino se forman extendiendo el bit de signo (bit 7) del byte leído de memoria.
- ST (*Store*) y STB (*Store Byte*), que escriben el contenido de un registro, Rb, en un word y un byte, respectivamente, de memoria. En STB se escriben los 8 bits de menor peso del contenido de Rb.

Para acceder a una posición de memoria (de tamaño byte o word) hace falta que el procesador especifique los 16 bits de la dirección de la memoria que se desea acceder. Es evidente que en una instrucción de 16 bits, como las que definimos en el capítulo anterior para nuestro procesador, no hay espacio para los 16 bits de la dirección de memoria, los 3 bits del registro (fuente o destino) y de los 4 bits del código de operación. La solución que se adopta para resolver este problema consiste en no especificar en la instrucción los 16 bits de la dirección de memoria, sino en especificar, solamente con 3 bits, un registro del procesador en el que se encuentra contenida la dirección de memoria a la que se desea acceder.

Así pues, cualquiera de estas cuatro instrucciones, deben codificar, además de los cuatro bits del código de operación:

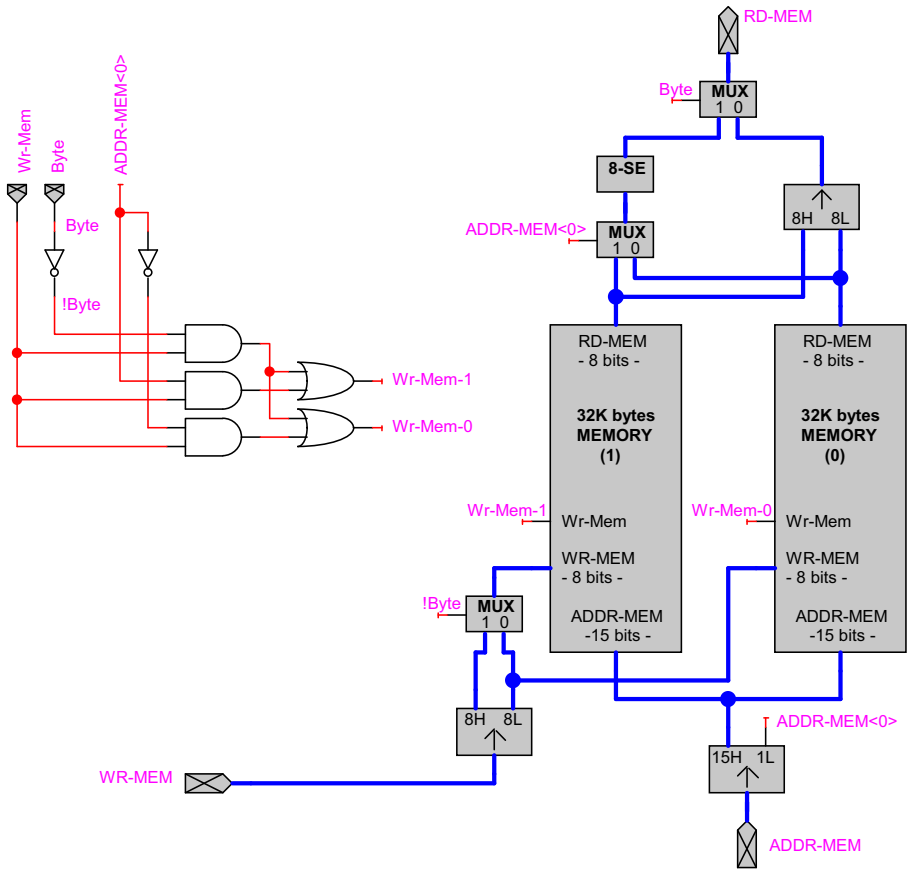


Fig. 11.14 Esquema interno del bloque DMEM *en la fase 3 (de lectura o escritura de byte o de word). Diseño completo del bloque.*

- un campo de 3 bits para especificar el registro destino Rd en caso de las instrucciones LD y LDB (registro donde se almacenará el dato leído de memoria) o para especificar el registro fuente Rb en las instrucciones ST y STB (registro cuyo contenido se escribirá en memoria), siendo d o b un valor entre 0 y 7 y
- un campo de 3 bits para especificar el registro Ra, con a de 0 a 7, (registro que contiene la dirección de memoria a leer)

Para codificar cada una de estas instrucciones podemos usar el formato de instrucción de 2 registros (figura 11.15), el mismo formato que usamos para la instrucción ADDI en el capítulo anterior cuando definimos la primera versión del lenguaje máquina de nuestro computador (salvo que en este caso para ST y STB en el campo que en el ADDI se especifica Rd aquí se especifica Rb.

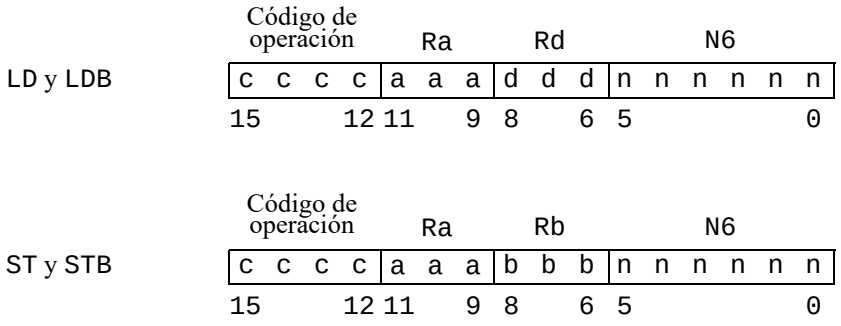


Fig. 11.15 Formato 2-R usado para la instrucciones de acceso a memoria.

Sintaxis en ensamblador y semántica

En principio no tenemos necesidad del campo N6 que nos ofrece el formato 2-R, pero como está en la instrucción lo vamos a usar para codificar un número en complemento a dos con 6 bits que en tiempo de ejecución se sumará al contenido del registro Ra para calcular la dirección de memoria a leer. Este campo en muchos casos valdrá 0 y por lo tanto el contenido de Ra será la dirección a leer. No obstante, en algunas aplicaciones (como se verá en la sección 11.6.2 mediante un ejemplo) es útil que la instrucción especifique un pequeño desplazamiento a sumar sobre la dirección base que contiene Ra para obtener la dirección de memoria a leer. El procesador, cuando ejecute una instrucción de estas deberá generar el campo N de la palabra de control con la extensión de signo del campo N6 y realizar la suma del contenido de Ra más N para formar la dirección de memoria a leer o escribir.

La sintaxis ensamblador y semántica de las cuatro instrucciones de acceso a memoria es:

Sintaxis en ensamblador	Semántica
`LD Rd, N6(Ra)`	`Rd ← MEM`$_w$`[((Ra+SE(N6))&(~1)]`
`LDB Rd, N6(Ra)`	`Rd ← SE(MEM`$_b$`[Ra+SE(N6)])`
`ST N6(Ra), Rb`	`MEM`$_w$`[((Ra+SE(N6))&(~1)] ← Rb`
`STB N6(Ra), Rb`	`MEM`$_b$`[Ra+SE(N6)] ← Rb<7..0>`

Las cuatro instrucciones tienen dos operandos fuente, Ra y N6 que deben ser sumados para obtener la dirección de memoria del operando, que puede ser fuente, en caso de LD y LDB, o destino, en caso de ST o STB. N6(Ra)es una forma de indicar que el operando (fuente o destino) es el contenido de la dirección de memoria que se forma sumando Ra más N6 (y poniendo el bit de menor peso a 0 en caso de acceso a word). N6(Ra) se sitúa en la posición del operando fuente en las instrucciones LD y LDB y en la de operando destino en las instrucciones ST y STB.

La forma de indicar, al especificar la semántica de las instrucciones de acceso a word, que la memoria efectúa un acceso alineado a dirección par (que aunque la dirección sea impar accederá a la dirección par forzando el bit de menor peso de la dirección a cero) es decir que la dirección a la que se accede se calcula sumando Ra más SE(N6) y luego haciendo la And bit a bit de este resultado con ~1 (que en binario es 1111111111111110) .

Codificación en lenguaje máquina

Usamos los cuatro códigos de operación que están libres a continuación del código de `ADDI`, ya que así todas las instrucciones de nuestro lenguaje SISA que tienen el mismo formato van seguidas en la tabla. Los códigos de operación para `LD`, `ST`, `LDB`, `STB` son 0011, 0100, 0101 y 0110, respectivamente. Con esto ya tenemos las 24 instrucciones del lenguaje SISA (solo falta una para completarlo, que se formulará en el capítulo 13), cuyo formato y codificación se muestra en la figura 11.16.

15 14 13 12	11 10 9	8 7 6	5 4 3	2 1 0	Name	Mnemonic	Format
0 0 0 0	a a a	b b b	d d d	f f f	Logic and Aritmetic Operations	AND, OR, XOR, NOT, ADD, SUB, SHA, SHL	3R
0 0 0 1	a a a	b b b	d d d	f f f	Compare Signed and Unsigned	CMPLT, CMPLE, -, CMPEQ, CMPLTU, CMPLEU, -, -	
0 0 1 0	a a a	d d d	n n n n n n		Add Immediate	ADDI	2R
0 0 1 1	a a a	d d d	n n n n n n		Load	LD	
0 1 0 0	a a a	b b b	n n n n n n		Store	ST	
0 1 0 1	a a a	d d d	n n n n n n		Load Byte	LDB	
0 1 1 0	a a a	b b b	n n n n n n		Store Byte	STB	
0 1 1 1					Branch future extension		
1 0 0 0	a a a	0	n n n n n n n n		Branch on Zero	BZ	1R
		1			Branch on Not Zero	BNZ	
1 0 0 1	d d d	0	n n n n n n n n		Move Immediate	MOVI	
	a a a d d d	1			Move Immediate High	MOVHI	
1 0 1 0	d d d	0	n n n n n n n n		Input	IN	
	a a a	1			Output	OUT	
1 0 1 1	x x x x x x x x x x x x				Future extensions		
1 1 x x							

Fig. 11.16 Formato y codificación de las 24 instrucciones SISA.

En la figura 11.17 se muestran algunos ejemplos concretos de instrucciones de acceso a memoria y su codificación en lenguaje máquina SISA.

`LDB  R1, -3(R2)`	0	1	0	1	0	1	0	0	0	1	1	1	1	0	1
`ST  5(R6), R7`	0	1	0	0	1	1	0	1	1	1	0	0	0	1	0	1
`LD R5, -1(R6)`	0	0	1	1	1	1	0	1	0	1	1	1	1	1	1	1
`STB -2(R6), R1`	0	1	1	0	1	1	0	0	0	1	1	1	1	1	1	0

Fig. 11.17 Codificación en lenguaje máquina de cuatro instrucciones SISA.

11.4.1 Conexión de la memoria al procesador y palabra de control para LD y ST

Cálculo de la dirección de memoria. Bus ADDR-MEM

Para cada acceso a memoria, tanto si es de lectura como de escritura, el procesador debe calcular la dirección de memoria como la suma de una dirección base (número natural codificado en binario en 16 bits) que se encuentra en el registro Ra y un pequeño desplazamiento, que se encuentra codificado en complemento a dos en el campo de 6 bits, N6, de la instrucción. Esta suma la realizará el procesador usando la ALU durante el ciclo en que se ejecute lainstrucción de acceso a memoria. La conexión del bus de direcciones de la memoria (ADDR-MEM) con la salida de la ALU se muestra en la figura 11.18.

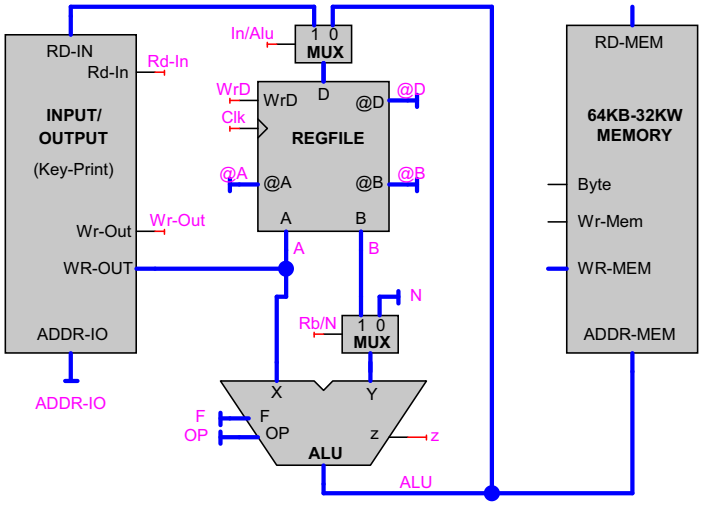


Fig. 11.18 Conexión de la UPG+IO *con el bus de memoria* ADDR-MEM.

En el ciclo en el que se esté ejecutando una instrucción de acceso a memoria la lógica de control del procesador (que hemos construido en el capítulo anterior y que ahora hay que completar con estas

nuevas cuatro instrucciones) debe generar los siguientes campos para que el calculo de la dirección de memoria se realice correctamente:

- el campo N de la palabra de control como la extensión de signo a 16 bits del campo N6 de la instrucción,
- el campo @A con los tres bits (aaa) del campo Ra de la instrucción (esto se hace para cualquier instrucción).
- el bit Rb/N con valor 0 para dejar pasar los 16 bits de N a la entrada Y de la ALU para que se sumen con los 16 bits de Ra que entra a la ALU por el bus X, y
- el campo OP igual a 00 y el F igual a 100 para que la ALU realice una suma.

A esta forma de calcular la dirección de memoria en las instrucciones de acceso a memoria se le denomina modo de direccionamiento "registro base más desplazamiento" y es el único modo de direccionamiento de memoria de nuestro procesador.

Instrucción *Load*, LD. Bus RD-MEM

Al ejecutarse una instrucción LD el dato presente en el bus de salida de la memoria, RD-MEM, debe escribirse, al final del ciclo, en uno de los registros del procesador. Para que ello sea posible, se debe conectar el bus RD-MEM con el bus D de entrada al banco de registros. Lo vamos a hacer sustituyendo el MUX-2-1 con señal de selección In/Alu por un multiplexor de 4 buses de entrada a uno de salida, MUX-4-1, que seleccione, en cada ciclo, entre el bus de salida de la ALU, el bus de lectura de memoria, RD-MEM y el de lectura de un puerto de entrada, RD-IN. Las dos señales de selección de este multiplexor se han juntado en un bus de 2 bits denominado -/i/l/a. ¿De donde viene este nombre tan extraño? Los cuatro símbolos (-, i, l y a) separados por la barra inclinada indican qué entrada se conecta a la salida del multiplexor cuando esta señal de selección vale 3, 2, 1, y 0 respectivamente. La entrada 3 no se usa, aunque se ha conectado a la 2 por no dejarla al aire, y por eso el símbolo "-". La entrada 2 se selecciona cuando se ejecuta una instrucción IN, la letra "l", de *load*, porque la entrada 1 se selecciona cuando se ejecuta una instrucción LD o LDB y "a" porque la entrada 0 se selecciona cuando se quiere escribir la salida de la ALU.

En la figura 11.19 se muestra la nueva UPG, con el multiplexor que acabamos de poner, y su conexión con el sistema de entrada/salida (IOKey-Print) y con los buses de memoria ADDR-MEM y RD-MEM.

Los dos bits de selección del MUX-4-1 (bus -/i/l/a) deben ser generados por la lógica de control en cada ciclo, en sustitución del bit Ld/Alu de selección del antiguo MUX-2-1, formando parte de la palabra de control que ahora tiene 45 bits (contando el TknBr). Así, en un ciclo de lectura de memoria (instrucción *load*, LD) el procesador debe generar los valores de los bits de la palabra de control que acabamos de ver para calcular la dirección de memoria y además

- el campo -/i/l/a debe valer 01 cuando se ejecuta la instrucción LD y LDB (mientras que debe valer 00 para las instrucciones aritmético lógicas, de comparación, ADDI, MOVI, MOVHI y 10 para IN (ver figura 11.19).
- el bit Byte igual a 1 para LDB e igual a 0 para LD para diferenciar entre los accesos a byte y a word.

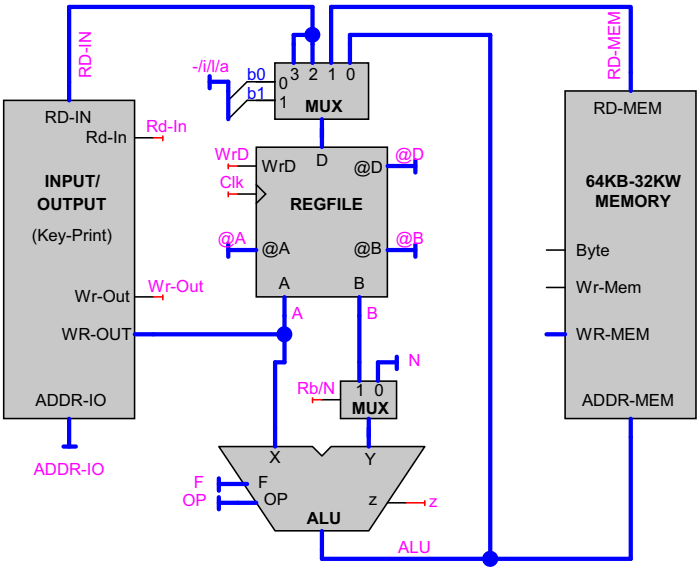


Fig. 11.19 Conexión de la UPG+IO *con los buses de memoria* ADDR-MEM *y* RD-MEM.

Instrucción *Store*, ST. Bus WR-MEM

En una instrucción de escritura a memoria, *store*, ST o STB, el contenido del registro fuente Rb debe conectarse al bus WR-MEM que entra en la memoria. Con la conexión del bus B de salida del banco de registros con el bus WR-MEM se consigue esto. La figura 11.20. muestra la conexión completa de la memoria de datos a la UPG del procesador, con lo que ya se pueden ejecutar las cuatro instrucciones de acceso a memoria que hemos introducido en este capítulo, ademas de las anteriores instrucciones SISA.

En un ciclo de escritura de memoria (instrucciónes *store*, ST y STB) la lógica de control debe generar los valores de los bits de la palabra de control que ya vimos para calcular la dirección de memoria y además

- el bit Wr-Mem debe valer 1 para que se escriban los bits del bus WR-MEM en la dirección de memoria que indican los bits de ADDR-MEM (mientras que para cualquiera otra instrucción este bit debe valer siempre 0) y
- el campo @B debe ser igual a los tres bits (bbb) del campo Rb de la instrucción, para que se lea el registro Rb y su contenido se escriba en memoria.
- el bit Byte igual a 1 para STB e igual a 0 para ST para, al igual que en las instrucciones load, diferenciar entre los accesos a Byte y a Word.

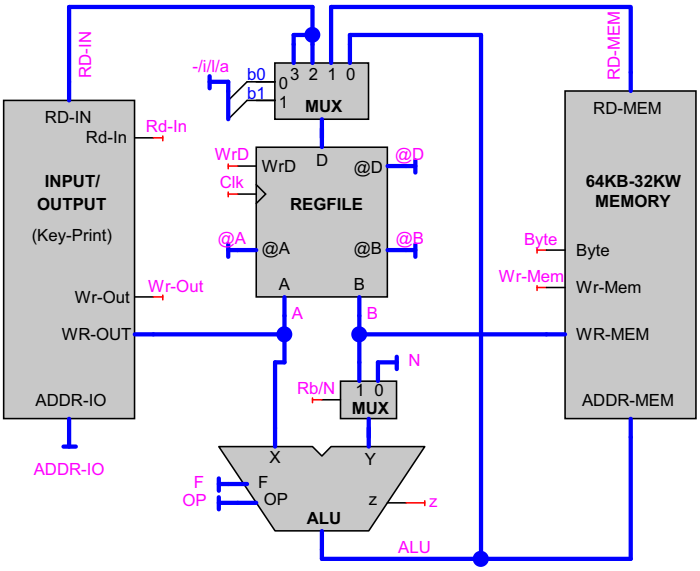


Fig. 11.20 Conexión completa de la UPG+IO+MEM

11.5 El computador SISC Harvard uniciclo

11.5.1 Estructura a bloques del computador

En este momento ya hemos especificado completamente el computador que estamos construyendo, SISC (*Simple Instruction Set Computer*) en su versión Harvard uniciclo y su lenguaje máquina SISA de 24 instrucciones. La figura 11.21 muestra el esquema a nivel de bloques del procesador (con la UPG con su nuevo multiplexor como unidad de proceso y con el PC, el circuito secuenciador, la memoria de instrucciones y la lógica de control como unidad de control) y su interconexión con el bloque de entrada/salida (IOKey-Print) y con el de la memoria de datos (MEM).

11.5.2 Palabra de control completa de 47 bits

La palabra de control completa (de 47 bits), con el nuevo campo de dos bits -/i/l/a en sustitución del bit In/Alu y los nuevos bits Wr-Mem y Byte, marcados en fondo gris, se muestra en la figura 11.22.

Un ejemplo de instrucción `LD` es la que denotamos en ensamblador como (el campo N6 se puede especificar en decimal cuando programamos en ensamblador, aunque también se podría haber especificado en hexadecimal poniendo 0xA en vez de 10):

```
LD R2, 10(R6)
```

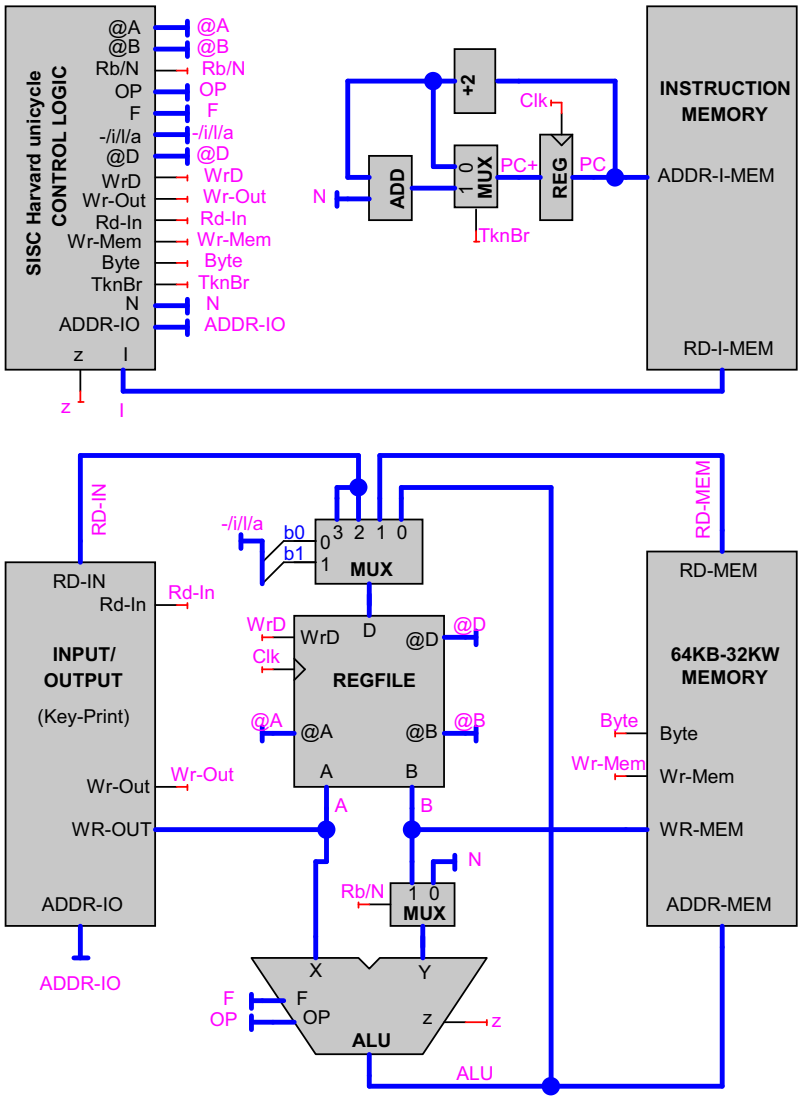


Fig. 11.21 Estructura a bloques del computador (Prodesador+Entrada/salida+Memoria) SISC Harvard Multicycle (UCG+UPG+IO+MEM).

Al final del ciclo en el que se ejecuta esta acción se escribe el registro R2 con el contenido de la posición de memoria cuya dirección es la suma del valor 10 más el contenido del registro R6. En la figura 11.23 se muestra la palabra de control para esta instrucción.

Un ejemplo de mnemotécnico usado para una acción de tipo ST es:

```
STB -3(R4), R7
```

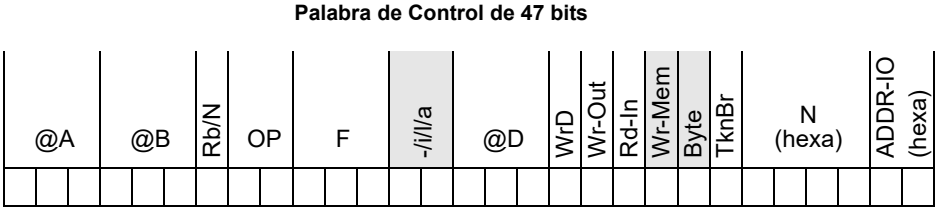


Fig. 11.22 Palabra de control completa (de 47 bits) que genera la lógica de control del computador SISC Harvard uniciclo.

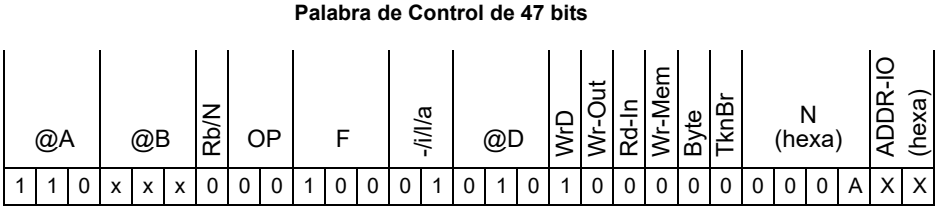


Fig. 11.23 Palabra de control al ejecutar la instrucción `LD R2, 10(R6).`

Pasado el ciclo de ejecución de la instrucción se habrá escrito el byte de memoria cuya dirección es el resultado de sumar `-3` más el contenido de `R4`, con el valor contenido en el registro `R7`. La palabra de control para esta acción se ve en la figura 11.24.

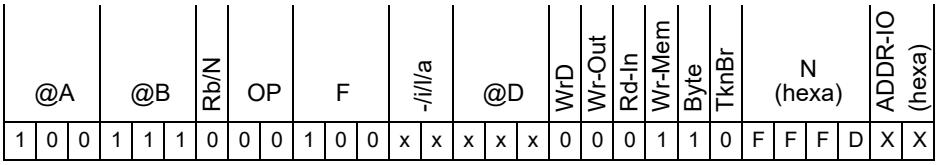


Fig. 11.24 Palabra de control al ejecutar la instrucción `STB -3(R4), R7.`

11.5.3 Completando la lógica de control

La figura 11.25 muestra la logica de control completa

La figura 11.26 muestra la tabla compacta del contenido de la ROM de la lógica de control.

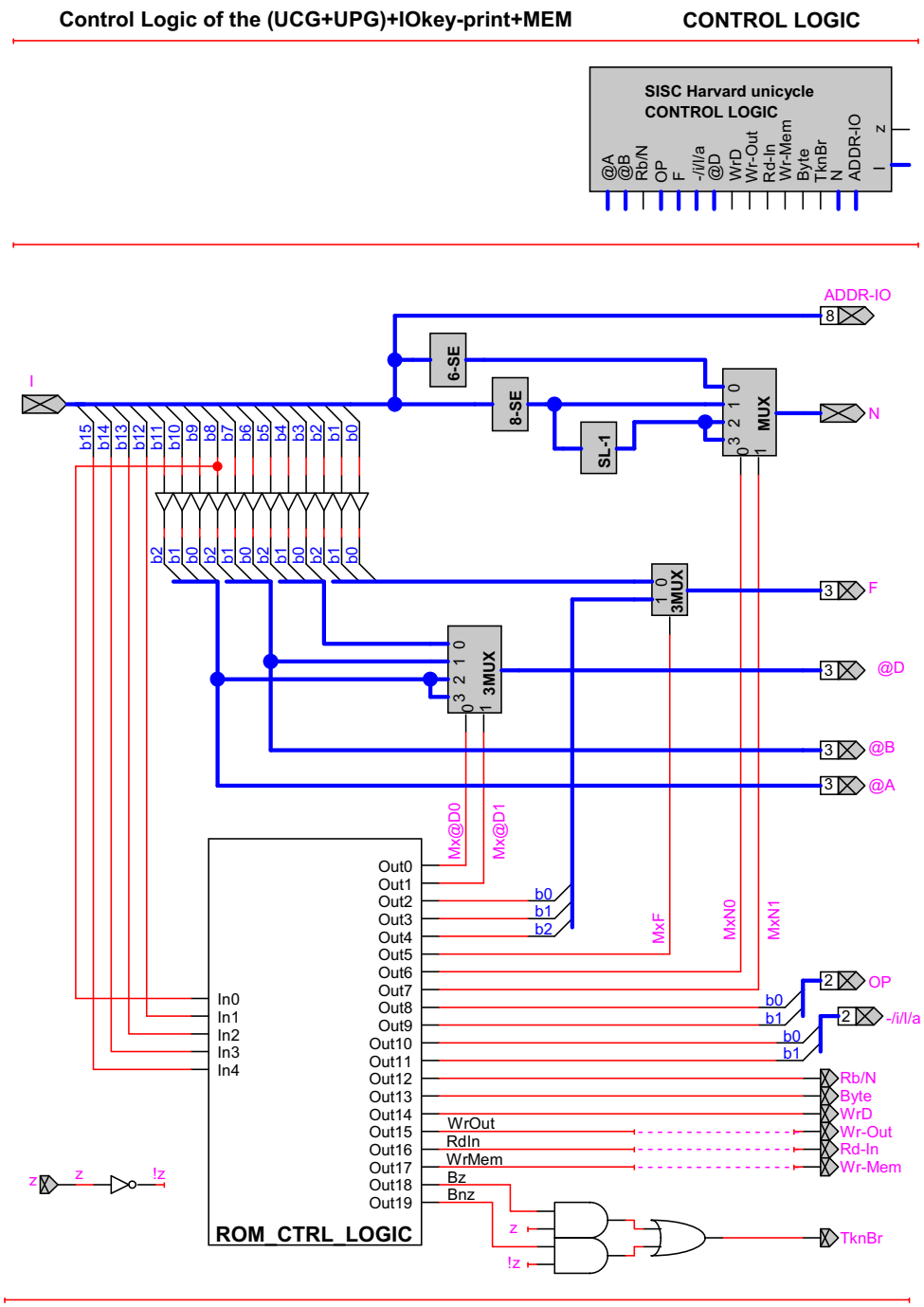


Fig. 11.25 Esquema de la lógica de control del SISC Harvard Unicycle usando una ROM.

Dirección					Contenido																					
In4	In3	In2	In1	In0	Out19	Out18	Out17	Out16	Out15	Out14	Out13	Out12	Out11	Out10	Out9	Out8	Out7	Out6	Out5	Out4	Out3	Out2	Out1	Out0	
I<15>	**I<14>**	**I<13>**	**I<12>**	**I<8>**	Bnz	Bz	Wr-Mem	RdIn	WrOut	WrD	Byte	Rb/N	-/i/I/a1	-/i/I/a0	OP1	OP0	MxN1	MxN0	MxF	F2	F1	F0	Mx@D1	Mx@D0	
0	0	0	0	x	0	0	0	0	0	1	x	1	0	0	0	0	x	x	0	x	x	x	0	0	AL
0	0	0	1	x	0	0	0	0	0	1	x	1	0	0	0	1	x	x	0	x	x	x	0	0	CMP
0	0	1	0	x	0	0	0	0	0	1	x	0	0	0	0	0	0	0	1	1	0	0	0	1	ADDI
0	0	1	1	x	0	0	0	0	0	1	0	0	0	1	0	0	0	0	1	1	0	0	0	1	LD
0	1	0	0	x	0	0	1	0	0	0	0	0	x	x	0	0	0	0	1	1	0	0	x	x	ST
0	1	0	1	x	0	0	0	0	0	1	1	0	0	1	0	0	0	0	1	1	0	0	0	1	LDB
0	1	1	0	x	0	0	1	0	0	0	1	0	x	x	0	0	0	0	1	1	0	0	x	x	STB
0	1	1	1	x	0	0	0	0	0	0	x	x	x	x	x	x	x	x	x	x	x	x	x	x	(NOP)
1	0	0	0	0	0	1	0	0	0	0	x	x	x	x	1	0	1	0	1	0	0	0	x	x	BZ
1	0	0	0	1	1	0	0	0	0	0	x	x	x	x	1	0	1	0	1	0	0	0	x	x	BNZ
1	0	0	1	0	0	0	0	0	0	1	x	0	0	0	1	0	0	1	1	0	0	1	1	0	MOVI
1	0	0	1	1	0	0	0	0	0	1	x	0	0	0	1	0	0	1	1	0	1	0	1	0	MOVHI
1	0	1	0	0	0	0	0	1	0	1	x	x	1	0	x	x	x	x	x	x	x	x	1	0	IN
1	0	1	0	1	0	0	0	0	1	0	x	x	x	x	x	x	x	x	x	x	x	x	x	x	OUT
1	0	1	1	x	0	0	0	0	0	0	x	x	x	x	x	x	x	x	x	x	x	x	x	x	(NOP)
1	1	x	x	x	0	0	0	0	0	0	x	x	x	x	x	x	x	x	x	x	x	x	x	x	(NOP)

Fig. 11.26 Tabla de verdad compacta del contenido de la ROM-CTRL-LOGIC de la lógica de control.

11.6 Ejemplos de programación.

A modo de ejemplo vamos a enunciar y resolver dos problemas: el cálculo del histograma de un conjunto grande de datos y el filtrado paso bajo de una señal digital almacenada en memoria. La resolución consistirá en escribir dos programas en el lenguaje ensamblador de nuestro computador Estos ejemplos nos sirven, por un lado, para justificar la necesidad de una memoria grande de datos (sin ella es imposible resolverlos) y, por otro, para iniciarnos en la programación en ensamblador, haciendo énfasis en

- la entrada/salida de datos por el teclado y la impresora de nuestro computador y en
- el acceso a estructuras de datos tipo vector (o tabla) que por su tamaño deben almacenarse necesariamente en la memoria de datos.

En el primer ejemplo usaremos las instrucciones de acceso a memoria con el desplazamiento igual a 0 (como `LD Rd, 0(Ra)`): el registro `Ra` contiene la dirección de memoria a la que se accede. En el segundo ejemplo usamos el campo del desplazamiento de las instrucciones de acceso a memoria (como `LD Rd, -2(Ra)`): la dirección de memoria a la que se accede se obtiene sumando un pequeño desplazamiento (un número entero en el rango -32, ..., 31) a la dirección base almacenada en `Ra`. Esto ejemplificará los dos usos típicos del único modo de direccionar la memoria (**modo de direccionamiento**[1]) de nuestro procesador: registro base más desplazamiento.

11.6.1 Histograma

Enunciado

Se desea obtener el histograma de un conjunto de 1024 datos, donde cada dato es un número natural representado en binario por el vector `X` de 16 bits. Se sabe que el rango de los posibles valores de cada dato X_u es $0 \leq X_u \leq 255$. Los 1024 operandos se deben entrar al procesador, de uno en uno, desde el teclado del computador. El procesador debe calcular el histograma de esos datos. Esto es, debe contar cuántos datos tienen valor 0, cuántos valor 1,..., y cuántos tienen valor 255. Después, el procesador debe enviar a la impresora del computador los 256 resultados que forman el histograma, empezando por el valor que indica el número de ceros y finalizando por el número de operandos con valor 255. El teclado y la impresora son los que hemos visto en capítulo 9 formando el sistema de entrada salida $IO_{KeyPrint}$. Todos los puertos y por lo tanto los datos de entrada y salida son de 16 bits. Por ello, como los datos de entrada sólo codifican valores de 0 a 255, los 8 bits de mayor peso de los operandos valen todos 0.

Solución

Como el rango de los posibles valores de cada operando es de 0 a 255, el resultado del problema consiste en un vector (o tabla) de 256 números naturales. Denominamos `HISTO` al vector de resultados, siendo `HISTO[i]` la variable que cuenta el número de operandos con valor `i`, para `i`=0, ..., 255. Este vector de resultados deberá estar almacenado en memoria, pues no tenemos suficientes registros para mantenerlo. Almacenaremos este vector a partir de la dirección simbólica de memoria `HISTO` (el valor que realmente tenga esta dirección de memoria no es importante de momento). El valor de cada resultado está en el rango de 0 a 1024, ambos incluidos, por lo que con 11 bits es suficiente para codificarlo. No obstante, usaremos una palabra de 16 bits para cada elemento del vector `HISTO`, ya que en nuestra memoria solamente se puede leer y escribir palabras de 16 bits.

1. En los procesadores RISC, como el nuestro, el tamaño de todas las instrucciones es el mismo (en nuestro caso 16 bits y en otros muchos 32) y todas las operaciones aritméticas, lógicas y de comparación se realizan con instrucciones cuyos operandos están en registros (modo de direccionamiento registro) o en la propia instrucción (usando para uno de los operandos el modo de direccionamiento inmediato). Por ello, para poder operar los datos se requieren instrucciones de *load* y *store* para moverlos entre memoria y registros. En los procesadores RISC el único modo de direccionamiento de la memoria es el mismo que el nuestro: registro base más desplazamiento. Sin embargo, en los procesadores CISC las instrucciones no son de tamaño fijo: el rango de tamaños puede ir, por ejemplo, de 8 bits a 64). Además, en los CISC las instrucciones que procesan datos pueden obtener sus operandos y dejar el resultado directamente en memoria. Para acceder a los datos en memoria pueden usar diferentes modos de direccionamiento, además del registro base más desplazamiento. Uno de estos modos, muy cómodo para escribir algunos programas, es el modo indexado. El modo indexado requiere que en la instrucción se codifique una dirección completa de memoria (en 16 bits en nuestro caso y en 32 en otros muchos) y un registro. En tiempo de ejecución se calcula la dirección de memoria sumando la dirección base codificada en la instrucción más el contenido del registro especificado (que se interpreta como un desplazamiento entorno a la dirección base). Según qué modos de direccionamiento use una instrucción requerirá más o menos bits para su codificación.

Descomponemos el programa para resolver este problema en tres fragmentos de código que se ejecutan en secuencia:

1. Inicialización
2. Entrada de datos y cálculo del histograma
3. Salida de resultados.

1. Inicialización

En primer lugar se deben inicializar a 0 los 256 elementos del vector `HISTO` que se encuentran en memoria, pues inicialmente no ha llegado ningún operando todavía. Para ello vamos a programar un bucle que ejecute 256 iteraciones con una instrucción store en el cuerpo del bucle que se encargue de poner a cero una posición distinta del vector cada vez que se ejecute (en cada iteración). Esta instrucción `ST` deberá usar como registro fuente un registro inicializado previamente (fuera del bucle) al valor 0. Usamos, por ejemplo, `R0` para este registro que mantiene la constante 0. Si elegimos, por ejemplo, el registro `R1` para mantener la dirección de memoria en la que se escribirá el contenido de `R0`, el 0, la instrucción `ST` del cuerpo del bucle es:

```
ST   0(R1), R0
```

Así pues, antes de entrar en el cuerpo del bucle debemos inicializar `R1` con el valor `HISTO`, la dirección de memoria donde se almacenará el elemento cero del vector (elemento `HISTO[0]`). Supongamos que, por ejemplo, `HISTO` vale `0x3AF0` o lo que es lo mismo el vector `HISTO` se almacena a partir de la dirección de memoria `0x3AF0`: `HISTO[0]` se almacena en la dirección `0x3AF0`, `HISTO[1]` en la dirección `0x3AF2` (ya que cada elemento de `HISTO` es un word, 2 bytes), ..., `HISTO[255]` en la `0x3CEE`. Con esta suposición el código para inicializar R1 antes de entrar en el bucle es:

```
MOVI   R1,0xF0
MOVHI  R1,0x3A
```

Antes de terminar el cuerpo del bucle deberemos incrementar en una unidad la dirección de memoria contenida en `R1` para dejarlo apuntando al siguiente elemento del vector a tratar (si se va ha dar otra pasada por el bucle). Por ello se dice que `R1` es un puntero (contiene una dirección de memoria) que "apunta" a un elemento del vector `HISTO`. Se debe incrementar `R1` en 2 para que pase a apuntar a la siguiente palabra de memoria. Para ello usamos la instrucción:

```
ADDI R1,R1,2
```

Para controlar el número de pasadas por el bucle (iteraciones) hay varias alternativas. En general, si el cuerpo del bucle se tiene que ejecutar un número fijo de veces (`256` en nuestro caso) una primera alternativa consiste en usar un registro contador de iteraciones (por ejemplo `R7`) que, inicializado a cero antes de entrar en el bucle, se incrementará en el cuerpo del bucle y se compara al final del bucle con el número máximo de iteraciones para saber si la actual es al última iteración o no. Como la comparación con una constante no puede hacerse directamente con una instrucción `CMPEQ` ya que ningún operando puede ser inmediato, deberemos inicializar un registro (por ejemplo `R6`) con la constante antes de entrar en el bucle. Recordad que la instrucción `CMPEQ` genera a la salida de la ALU un 1, y por tanto `Z` = 0, cuando los dos operandos son iguales, mientras que genera un 0, y `Z` = 1, cuando

no lo son): esto explica el uso de la instrucción BZ al final del bucle. El esquema de un bucle así se muestra a continuación:

```
        MOVI   R7,0          ;contador iteraciones a 0
        MOVI   R6,0x00       ;R6 = 0x0000 (Ext. Signo)
        MOVHI  R6,0x01       ;R6 = 0x0100 constante 256
Loop:
        ... ;cuerpo del bucle
        ADDI   R7,R7,1       ;incrementa contador iteraciones
        CMPEQ  R5,R7,R6      ;¿otra iteracion?
        BZ     R5,Loop
```

En este caso, como no hemos definido cuantas instrucciones hay en el cuerpo del bucle no podemos saber el desplazamiento que hay que sumar la PC (al ejecutar la instrucción BZ) para pasar a ejecutar el inicio del cuerpo del bucle, hemos puesto la dirección simbólica del inicio del cuerpo del bucle (Loop) en el campo del desplazamiento de la instrucción BZ. Al traducir a lenguaje máquina, y sabiendo cuantas instrucciones hay en el cuerpo del bucle, se debería codificar este desplazamiento, que por ejemplo si hay 2 instrucciones en el cuerpo del bucle valdría -5.

Una forma mejor de implementar un bucle como este consiste en inicializar fuera del bucle un contador (que ahora indica cuantas iteraciones faltan por realizar) a 256 e ir decrementando este contador en una unidad dentro del cuerpo del bucle para que cuando valga 0 sepamos que ya no hay que volver a iterar. Este código no necesita usar el registro R6 para mantener ninguna constante y requiere ejecutar una instrucción menos que el anterior por iteración del bucle, ya que no hace falta comparar el contador con 0, la instrucción de salto si no cero sobre el contador es suficiente para controlar el fin del bucle:

```
        MOVI   R7,0x00       ;R7 = 0x0000 (Ext. Signo)
        MOVHI  R7,0x01       ;R7 = 0x0100 constante 256
Loop:
        ... ;cuerpo del bucle
        ADDI   R7,R7,-1      ;decrementa contador iter. restantes
        BNZ    R7,Loop       ;¿otra iteracion?
```

El código de inicialización a cero del vector HISTO que usaremos en nuestro programa tiene esta estructura y se muestra en la caja de la figura 11.27.

```
;***** Inicialización ****************************************
        MOVI   R1,0xF0
        MOVHI  R1,0x3A     ;R1 apunta a HISTO[0]
        MOVI   R7,0x00       ;R7 = 0x0000 (Ext. signo)
        MOVHI  R7,0x01       ;R7 = 0x0100 (cte. 256 en decimal)
        MOVI   R0,0          ;constante 0
Loop:   ST     0(R1), R0     ;pone a 0 un elemento de HISTO
        ADDI   R1,R1,2       ;incrementa puntero
        ADDI   R7,R7,-1      ;decrementa contador iter. restantes
        BNZ    R7,-4         ;¿otra iteracion?
```

Fig. 11.27 Fragmento de programa para inicializar el vector HISTO.

En códigos como este, en los que se hace un acceso secuencial a los elementos de un vector, uno en cada iteración, se puede usar el registro puntero para detectar que ya no hay que iterar más veces. Con esta solución nos ahorramos la instrucción para incrementar el contador, que ya no existe, dentro del cuerpo del bucle, pero tenemos que añadir una instrucción de comparación con la dirección de memoria siguiente a la del último elemento del vector. Fuera del bucle ya no hay que inicializar el contador, pero hay que inicializar un registro (por ejemplo, R6) que usaremos como segundo operando de la comparación, que no puede ser un valor inmediato, con valor `0x3CF0 = HISTO+256*2`. El código resultante con esta alternativa es tan eficiente como el de la figura 11.27:

```
        MOVI   R1,0xF0
        MOVHI  R1,0x3A       ; R1 apunta a HISTO[0]
        MOVI   R6,0xF0
        MOVHI  R6,0x3C       ; R6 apunta a HISTO[256]
Loop:   ST     0(R1), R0     ;pone a 0 un elemento de HISTO
        ADDI   R1,R1,2       ;incrementa puntero
        CMPEQ  R5,R1,R6      ;¿otra iteracion?
        BZ     R5,-4
```

Hemos dedicado un tiempo a optimizar este código no porque consideremos que en este momento es importante ahorrarnos algunos ciclos de ejecución sino para mostrar distintas alternativas de implementar un bucle que accede secuencialmente a los elementos de un vector, ya que estamos empezando a programar en ensamblador y analizar distintas alternativas ayuda a comprender el lenguaje y sus posibilidades. Lo importante ahora es que los programas funcionen correctamente, que hagan lo que tienen que hacer, sin importarnos la eficiencia.

2. Entrada de operandos y cálculo del histograma

Para cada uno de los 1024 operandos que entran desde el teclado al procesador se deberá incrementar el valor de `HISTO[Xu]`, siendo `Xu` el valor codificado en los 16 bits de la palabra que envía el teclado (ya que `HISTO[Xu]` es el contador del número de veces que llegó un operando con valor `Xu`).

En nuestro procesador no podemos operar directamente con un operando en memoria, por eso hay que leer la posición de memoria a modificar y copiar su contenido en un registro para después operarlo y por último volverlo a escribir en la misma posición de memoria. Cada vez que llega un operando por el teclado se incrementa el elemento de `HISTO` pertinente y se repite la acción hasta que han llegado los 1024 operandos. Esto también lo efectuaremos mediante un bucle que se ejecutará 1024 veces, una por cada operando.

La figura 11.28 muestra el fragmento de código resultante, que comentamos a continuación. Para referirnos a cada instrucción cómodamente, hemos etiquetado cada instrucción con su dirección simbólica o etiqueta, que hemos denominado `Li`, tal como suelen permitir los lenguajes ensamblador[1].

1. En nuestro lenguaje ensamblador, como en todos, las etiquetas consisten en una secuencia de caracteres alfanuméricos (excluyendo las palabras clave del lenguaje, como son los mnemotécnicos de las instrucciones). Una etiqueta se puede poner opcionalmente al principio de una línea de código ensamblador terminada con el símbolo dos puntos (:). Después de los dos puntos se pone el mnemotécnico de la instrucción con sus operandos y finalmente, también opcionalmente, se puede poner un comentario precedido por el símbolo punto y coma (;).

```
;***** Entrada de operandos y calculo del histograma ************
L1:  MOVI   R6,0xF0
L2:  MOVHI  R6,0X3A       ;R6, constante HISTO

L3:  MOVI   R7,0x00
L4:  MOVHI  R7,0x04       ;R7, contador iter. restantes a 1024

L5:  IN     R1,KEY-STATUS
L6:  BZ     R1,-2
L7:  IN     R1,KEY-DATA   ;entra Xu del teclado

L8:  ADD    R1,R1,R1      ; multiplica R1 por 2
L9:  ADD    R1,R1,R6      ;calcula direccion de HISTO[Xu]
L10  LD     R5,0(R1)      ;lee HISTO[Xu]
L11: ADDI   R5,R5,1       ;incrementa valor
L12: ST     0(R1),R5      ;actualiza HISTO[Xu]

L13: ADDI   R7,R7,-1      ;decrementa contador iter. restantes
L14: BNZ    R7,-10      ;¿otra iteracion?
```

Fig. 11.28 Fragmento de programa para entrar del teclado y procesar cada operando.

Ahora el registro `R7` indica las iteraciones por el bucle que faltan por hacer. Se inicializa a 1024 con las instrucciones `L1` y `L2`. En cada iteración del bucle (formado por las instrucciones `L5` a `L13`) se decrementa este contador en `L12` y se salta en `L3` al inicio del bucle, si no se ha hecho la última iteración. El registro `R6` mantendrá durante todo el código la dirección de memoria donde comienza el vector `HISTO`: la dirección de `HISTO[0]`, que hemos supuesto que vale `0x3AF0` (se inicializa en `L3` y `L4`).

En `L5` comienza el cuerpo del bucle. Las instrucciones `L5`, `L6` y `L7` realizan la entrada de un dato del teclado por encuesta, preguntando primero por el estado del teclado (en `L5` y `L6`) y entrando el dato, `Xu`, después de que ha sido tecleado para dejarlo en `R1` (en `L7`).

En `L8` el dato `Xu` se multiplica por 2, ya que cada elemento de `HISTO` es un word, ocupa dos bytes, dos direcciones de memoria consecutivas. En `L9` se suma a la dirección inicial del vector, `HISTO`, para obtener la dirección de memoria del elemento `HISTO[Xu]`, que es `HISTO+Xu*2`, donde se encuentra almacenado el número de veces que ha llegado `Xu`. Este número se lee de la memoria en `L10`, se incrementa en `L11` y se vuelve a escribir en la misma posición de memoria en `L12`. Con esto se ha actualizado el histograma con el número de veces que ha aparecido un operando con el valor entrado desde el teclado. Una vez analizados los 1024 datos, se pasa al último fragmento de código.

3. Salida de resultados

Por último el procesador envía los resultados del histograma a la impresora, empezando por `HISTO[0]` y terminando por `HISTO[255]`. Esto puede verse en el fragmento de código de la figura 11.29 en el que usamos `R7` como contador de iteraciones que faltan por realizar (que se inicializa a `256` antes del bucle) y `R6` como puntero a `HISTO`, que se encuentra ya apuntando a `HISTO[0]` desde el fragmento de programa 2 donde se uso como constante.

```
;***** Salida de resultados ************************************
     MOVI   R7,0x00
     MOVHI  R7,0x01       ;R7, contador iter. restantes a 256

L:   IN     R0,PRINT-STATUS
     BZ     R0,-2
     LD     R0,0(R6)
     OUT    PRINT-DATA,R0 ;imprime elemento del histograma

     ADDI   R6,R6,2       ;incrementa puntero a HISTO
     ADDI   R7,R7,-1      ;decrementa contador iter. restantes
     BNZ    R7,-7         ;otra iteracion?
```

Fig. 11.29 Fragmento de programa para imprimir los resultados que forman el histograma.

El código completo

En la figura 11.30 juntamos los tres fragmentos de código en un solo programa ensamblador para el cálculo del histograma.

```
;*****************************************************************
;*****                        HISTOGRAMA                      *****
;*****************************************************************

;***** Inicialización ***************************************
       MOVI   R1,0xF0
       MOVHI  R1,0x3A       ;R1 apunta a HISTO[0]
       MOVI   R7,0x00
       MOVHI  R7,0x01       ;R7 contador iter. restantes a 256
       MOVI   R0,0          ;constante 0
L1:    ST     0(R1), R0     ;pone a 0 un elemento de HISTO
       ADDI   R1,R1,2       ;incrementa puntero a HISTO
       ADDI   R7,R7,-1      ;decrementa contador iter. restantes
       BNZ    R7,-4         ;¿otra iteracion?

;***** Entrada de operandos y calculo del histograma ************
       MOVI   R6,0xF0
       MOVHI  R6,0x3A       ;R6, constante HISTO
       MOVI   R7,0x00
       MOVHI  R7,Ox04       ;R7, contador iter. restantes a 1024

L2:    IN     R1,KEY-STATUS
       BZ     R1,-2
       IN     R1,KEY-DATA   ;entra Xu del teclado
       ADD    R1,R1,R1      ;multiplica R1 por 2
       ADD    R1,R1,R6      ;calcula direccion de HISTO[Xu]
       LD     R5,0(R1)      ;lee HISTO[Xu]
       ADDI   R5,R5,1       ;incrementa valor
       ST     0(R1),R5      ;actualiza HISTO[Xu]
       ADDI   R7,R7,-1      ;decrementa contador iter. restantes
       BNZ    R7,-10      ;¿otra iteracion?

;***** Salida de resultados *************************************
       MOVI   R7,0x00
       MOVHI  R7,0x01     ;R7 = 0x0100 (cte. 256 en decimal)
L3:    IN     R0,PRINT-STATUS
       BZ     R0,-2
       LD     R0,0(R6)
       OUT    PRINT-DATA,R0 ;imprime elemento del histograma
       ADDI   R6,R6,2       ;incrementa puntero a HISTO
       ADDI   R7,R7,-1      ;decrementa contador iter. restantes
       BNZ    R7,-7         ;¿otra iteracion?
```

Fig. 11.30 Programa completo del histograma.

11.6.2 Filtro paso bajo

Enunciado

Se desea efectuar el filtrado paso bajo de una señal digitalizada que se encuentra almacenada en memoria. Asumimos que en el vector `X`, de `n` números naturales, `X[i] para i=0` hasta `n-1`, se encuentra la secuencia de muestras de la señal analógica original una vez digitalizada. La señal filtrada, procesada, se debe almacenar en memoria en el vector `W` del mismo número de elementos. El cálculo que requiere el filtrado es el siguiente:

```
W[i]= (X[i]+X[i-1]+X[i-2]+X[i-3])/4   para i = 3 hasta n-1
```

El cálculo comienza en `i`=3 para que no se produzcan accesos fuera del vector `X`, no importándonos el valor de `W[0]` a `w[2]`.

Solución

El programa ensamblador para realizar estos cálculos se muestra en la figura 11.31 para `n`=10.000, suponiendo que el vector X y el W se encuentran almacenados en memoria a partir de las direcciones 0x0100 y 0x1000 respectivamente. Hemos etiquetado todas las instrucciones para referirnos cómodamente a cada una de ellas. Las instrucciones `L8` a `L20` forman el bucle, sobre el que se pasa 9.997 veces (iteraciones). Cada vez que se ejecuta una pasada por el bucle se efectúa el cálculo para un valor de `i`: se calcula `W[i]`. Antes de entrar en el bucle, en las instrucciones `L1` a `L7`, se inicializa

- `R0` con el número de veces que se debe ejecutar el bucle (9.997),
- `R1` y `R2` con la dirección del elemento 3 del vector de datos original (`X`) y del resultado (`W`) respectivamente y
- `R7` con la constante `-2`, que será usado como segundo operando de la instrucción `SHL` para dividir por 4 (multiplicar por 2^{-2}) en la instrucción `L15`.

El acceso a memoria para leer los 4 elementos del vector `X` que intervienen en el cálculo del elemento `w[i]` (elementos `X[i]`, `X[i-1]`, `X[i-2]` y `X[i-3]`) se realiza con las 4 instrucciones `LD`, de `L8` a `L11`, usando como registro base `R1` y como desplazamiento[1] las constantes 0, -1, -2 y -3 respectivamente.

La suma de los elementos de `X` se realiza en las instrucciones `L12` a `L14` y la división por 4 se realiza en `L15` mediante el desplazamiento de dos bits a la derecha: división por 2^2. El resultado se escribe en `W[i]` en la instrucción `L16` usando el registro `R2` como puntero: que contiene la dirección de memoria a acceder (con desplazamiento 0).

Antes de volver a ejecutar otra vez las instrucciones del bucle, en `L17` y `L18`, se incrementa el registro base que apuntaba a `X[i]` en esta iteración del bucle y se incrementa también el puntero a `W[i]`, para

1. En la sección 11.4, subsección “Así pues, cualquiera de estas cuatro instrucciones, deben codificar, además de los cuatro bits del código de operación:” se hizo referencia a este ejemplo del uso de un pequeño desplazamiento en las instrucciones de acceso a memoria. Otros usos típicos del desplazamiento son para acceder a un elemento concreto dentro de estructuras de datos tipo *struct* en lenguaje C y tipo pila.

dejar los registros `R1` y `R2` preparados para iniciar la siguiente iteración, apuntando a `X[i+1]` y `w[i+1]`. Por último, se decrementa el contador de iteraciones con la instrucción `L19` y si no se han ejecutado ya las 9.997 pasadas por el bucle, se va a ejecutar otra iteración comenzando saltando a la instrucción `L8`.

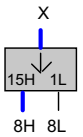
```
;*****************************************************************
;*****                    FILTRO PASO BAJO                       *****
;*****************************************************************

;***** Inicialización ****************************************
L1:     MOVI    R0,0x0D
L2:     MOVHI   R0,0x27         ;R0 numero de iteraciones = 9997
L3:     MOVI    R1,0x03
L4:     MOVHI   R1,0x01         ;R1 apunta a X[3]
L5:     MOVI    R2,0x03
L6:     MOVHI   R2,0x10         ;R2 apunta a W[3]
L7:     MOVI    R7,-2           ;R7 constante -2
;
;***** Bucle para el calculo de W[i] *****************************
L8:     LD      R3,0(R1)        ;R3 = X[i]
L9:     LD      R4,-1(R1)       ;R4 = X[i-1]
L10:    LD      R5,-2(R1)       ;R5 = X[i-2]
L11:    LD      R6,-3(R1)       ;R6 = X[i-3]
L12:    ADD     R3,R3,R4        ;X[i] + X[i-1]
L13:    ADD     R3,R3,R5        ;X[i] + X[i-1] + X[i-2]
L14:    ADD     R3,R3,R6        ;X[i] + X[i-1] + X[i-2]+ X[i-3]
L15:    SHL     R3,R3,R7        ;(X[i] + X[i-1] + X[i-2]+ X[i-3])/4
L16:    ST      0(R2),R3        ;almacena resultado W[i]
L17:    ADDI    R1,R1,2         ;incrementa puntero a X
L18:    ADDI    R2,R2,2         ;incrementa puntero a W
L19:    ADDI    R0,R0,-1        ;decrementa contador iter. restantes
L20:    BNZ     R0,-13          ;¿otra iteracion?
```

Fig. 11.31 Programa ensamblador que realiza el filtrado paso bajo de una señal.

Description:
Split the 16-bit input bus X to a 15-bit output buse AH and a 1-bit signal 1L.

Bit-level Logical Operation:
1L = X(b0)
15H(bi) = X(bi+1) for i = 0 to 7



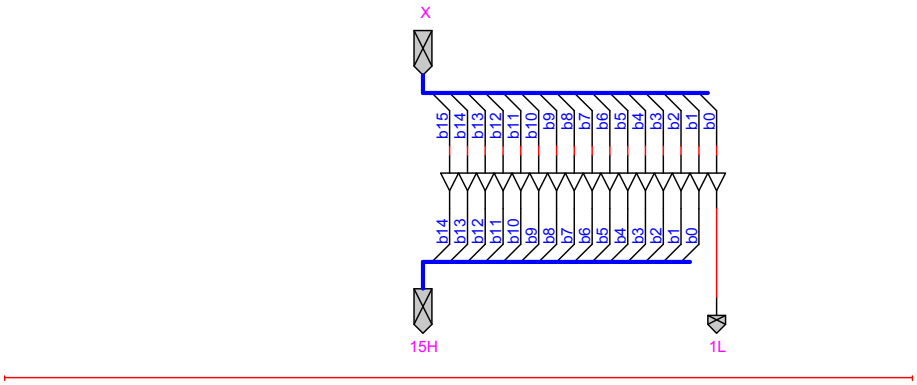
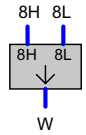


Fig. 11.32 Definición, símbolo y esquema interno del bloque 16-to-15H1L.

Description:
Concatenate two 8-bit buses to form a 16-bit one.

Bit-level Logical Operation:
W(bi) = 8L(bi) for i = 0 to 7
W(bi) = 8H(bi) for i = 8 to 15



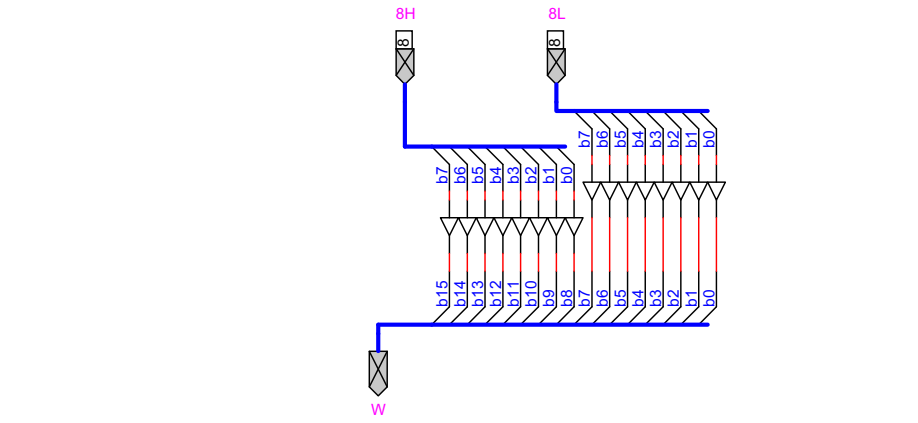
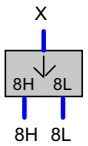


Fig. 11.33 Definición, símbolo y esquema interno del bloque 8H8L-to-16H1L.

Description:
Split the 16-bit input bus X to two 8-bit output buses AH and 8L.

Bit-level Logical Operation:
8L(bi) = X(bi) for i = 0 to 7
8H(bi) = X(bi+8) for i = 0 to 7



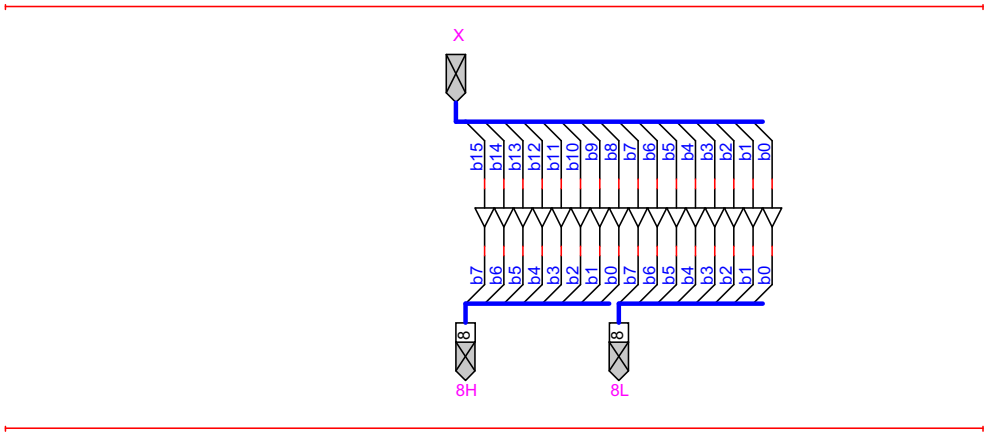


Fig. 11.34 Definición, símbolo y esquema interno del bloque 16-to-8H8L.