

10 Unidad de control general

10.1	Introducción	2
10.2	Estructura de la unidad de control general	6
10.3	Creando el lenguaje máquina y ensamblador SISA	18
10.4	Lenguaje máquina y ensamblador SISA	28
10.5	Modificación de la ALU de la UPG para implementar MOVHI	32
10.6	Implementación de la lógica de control con una ROM.....	35

10.1 Introducción

En capítulos anteriores. El procesador de cualquier computador es un procesador de propósito general (el mismo hardware sirve para resolver cualquier problema, es decir, para ejecutar cualquier algoritmo). Un procesador, tanto si es de propósito específico como de propósito general, está formado por una unidad de proceso (UP) y una unidad de control (UC). En el capítulo 7 diseñamos procesadores de propósito específico (PPE) donde tanto la UP como la UC eran específicas para cada problema (no servían para resolver otro). En el capítulo 8 seguimos diseñando PPE pero usando para todos ellos la misma UP, que era de propósito general: la UPG. Para cada problema teníamos que diseñar una unidad de control específica que, como es un circuito secuencial, diseñábamos dibujando el grafo de estados (y especificando, en este caso, la palabra de control con mnemotécnicos). Ya hemos tratado la implementación de una UC específica (UCE) usando el número mínimo de biestables D y, por ejemplo, dos memorias ROM (una para el estado siguiente y otra para las salidas) en el capítulo 6 (al sintetizar circuitos secuenciales) y en los capítulos 7, 8 y 9 (al diseñar procesadores de propósito específico).

En este capítulo. Vamos a dar el último paso hacia el diseño de un procesador de propósito general. Este procesador usará como UP la UPG y como UC la unidad de control general (UCG) que vamos a diseñar ahora. El resultado será un procesador que ejecuta cada instrucción en un ciclo (un procesador uniciclo). Cada nodo de los grafos que dibujábamos en los capítulos 7, 8 y 9 para especificar la UCE ahora se convertirá en una o varias instrucciones del lenguaje máquina del procesador (o lenguaje ensamblador, si usamos mnemotécnicos).

10.1.1 Diseño inicial de la unidad de control general

El diseño de cualquier UCE (implementación de un grafo de estados) para formar un PPE usando la UPG se ha simplificado bastante en el capítulo anterior al utilizar la entrada/salida de datos asíncrona al implementar las señales del protocolo de handshaking de cuatro fases (Req y Ack) a través de los puertos de entrada/salida. Por esto la unidad de control sólo tiene una señal de entrada: el bit Z que le llega de la ALU de la UPG. Por lo tanto, de cada nodo del grafo de estados sólo salen dos arcos, uno para $Z=0$ y otro para $Z=1$. Además, al usar controladores con efecto lateral, como el teclado y la impresora vistos en el capítulo anterior, se ha simplificado mucho el número de nodos para efectuar una entrada/salida, pasando de 7 a 3 nodos.

Si usamos las técnicas de implementación de grafos de estado con el mínimo número de biestables y dos memorias ROM, para resolver un nuevo problema solamente deberíamos cambiar las dos memorias ROM. Esto es así suponiendo que tenemos el suficiente número de biestables para almacenar la codificación del estado actual y las ROM son del tamaño que corresponde al número de biestables: si hay k biestables la ROM del estado siguiente debe tener 2^{k+1} palabras de k bits cada una y la ROM que genera la palabra de control 2^k palabras de 43 bits. La figura 10.1 muestra una primera versión de la unidad de control de nuestro procesador de propósito general usando dos memorias ROM. Hemos dispuesto un registro de 16 bits para poder almacenar el estado de la unidad de control. Hemos elegido 16 bits para codificar el estado porque nuestro procesador es de 16 bits y porque con 16 bits se pueden codificar grafos de hasta 2^{16} estados, que nos parecen suficientemente grandes como para resolver problemas complicados.

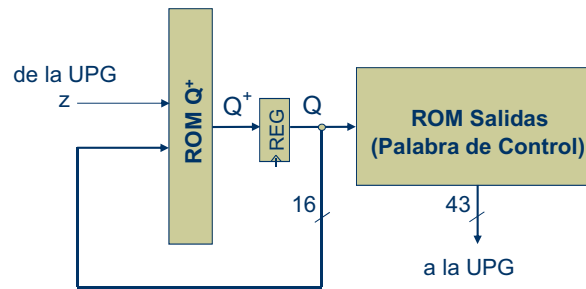


Fig. 10.1 Implementación de la unidad de control con dos ROMs.

Otra posible implementación de la unidad de control es la que usa **una sola ROM y un multiplexor de buses** para seleccionar el estado siguiente en función del bit Z (en el capítulo 6 ya vimos este tipo de implementación de un circuito secuencial). Este diseño, que se muestra en la figura 10.2, es el que usamos de partida para ir transformándolo en este capítulo, hasta llegar a la unidad de control de propósito general que definitivamente formará parte del procesador de nuestro computador.

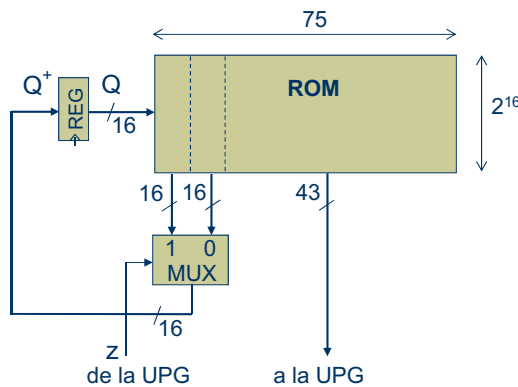


Fig. 10.2 Implementación de la unidad de control con una sola ROM y un multiplexor de buses.

Memoria de Instrucciones. En los computadores de propósito general se usa una **memoria RAM**, cuyo contenido se puede leer y escribir, para no tener que cambiar la ROM cada vez que queremos resolver un problema diferente. A esta memoria, ya sea ROM o RAM, se la denomina **memoria de instrucciones (I-MEM, instruction memory)**, ya que se denomina **instrucción de lenguaje máquina** a cada palabra de esta memoria. A la secuencia de instrucciones que implementa el grafo de esta unidad de control se la denomina **programa en lenguaje máquina** (por ello a la memoria de instrucciones también se le puede llamar memoria de programa).

Estructura a bloques de un computador. Un computador está formado por tres subsistemas interconectados entre sí por buses: el procesador (también llamado CPU, *Central Processing Unit*) formado por la UCG y la UPG, el subsistema de entrada/salida (con los periféricos como el teclado y la

impresora) y el subsistema de memoria (formado por una memoria RAM para almacenar grandes cantidades de datos y poder leerlos y escribirlos). La unidad de control general del procesador la estamos diseñando en este capítulo y el subsistema de memoria lo veremos en el capítulo siguiente; el resto ya lo conocemos.

Modelo Harvard y modelo Von Neumann. Hay dos posibles modelos de computadores: aquellos en los que las memorias de instrucciones y datos son diferentes memorias (modelo Harvard) y aquellos en los que son la misma (modelo Von Neumann). Nosotros diseñaremos primero un computador con el modelo Harvard, que terminaremos en la primera mitad del capítulo 12, (y que además es una implementación unicity: cada instrucción se ejecuta en un ciclo de procesador) y después, en la segunda mitad del capítulo 12, implementaremos el mismo computador Harvard en su versión multiciclo (cada instrucción tarda varios ciclos en ejecutarse y no todas tardan el mismo número de ciclos). El multiciclo resulta más eficiente que el unicity. Por último, en el capítulo 13 diseñaremos una nueva versión de computador que sigue el modelo Von Neumann. Para ello juntamos las dos memorias, la de datos y la de instrucciones del computador Harvard multiciclo, en una sola memoria RAM.

Procesadores de uso general y procesadores empotrados. Los procesadores de uso general (para un PC, por ejemplo) usan el modelo Von Neumann con una memoria RAM como memoria conjunta de instrucciones y datos, pero los de propósito más específico (“empotrados” en un coche, una lavadora, etc.) suelen usar el modelo Harvard con una ROM como memoria de instrucciones y una RAM como memoria de datos (así, cuando se pone en funcionamiento el sistema, el programa, que siempre es el mismo, siempre está cargado en la memoria de instrucciones).

¿ROM o RAM? De momento diseñaremos la unidad de control del procesador con una ROM y después (en el capítulo 13), cuando juntemos la I-MEM con la memoria de datos, que es RAM, ya tendremos el programa en una RAM y sabremos cómo escribir nuevos programas (escribiendo la RAM de la misma forma que se escriben nuevos datos, como veremos en el capítulo siguiente). Así que, de momento, siguiendo con el modelo Harvard y con la I-MEM en ROM, cambiar el grafo de la unidad de control para resolver otro problema en el computador, será equivalente a cambiar la ROM.

Objetivo: reducir el tamaño de las instrucciones. De momento podemos decir que una **instrucción**, (cada palabra de 75 bits de la I-MEM, de la ROM, de la figura 10.2) es la información que necesita el computador durante un ciclo (la información que indica cada nodo del grafo de estados de la unidad de control):

- Cuál es el estado siguiente, la siguiente instrucción a ejecutar, según Z valga 0 o 1. Esto lo indica el grafo con el destino de los arcos que salen de cada nodo y se codifica en la instrucción en 32 bits.
- Qué valor tiene cada bit de la palabra de control (para que la UPG y los subsistemas de entrada/salida sepan qué tienen que hacer durante el ciclo en que se ejecuta la instrucción). Esto lo indican las salidas de cada nodo del grafo con mnemotécnicos y se codifica en la instrucción con 43 bits.

Podemos decir que el esquema de la figura 10.2 ya es una unidad de control general, pero vamos a transformarla bastante todavía, paso a paso, hasta conseguir instrucciones con 16 bits. El primer objetivo del resto del capítulo es este. Será más fácil escribir programas con instrucciones de 16 bits que con las actuales instrucciones de 75 bits.

El coste de este cambio es pequeño. Deberemos añadir un poco más de lógica combinacional (a cambio de una gran reducción en el tamaño de la memoria de instrucciones) y perderemos paralelismo: lo que ahora hacemos en un nodo/instrucción, en un ciclo, necesitará, en algunos casos, más instrucciones de 16 bits, más ciclos.

10.1.2 Del grafo de estados al programa

Tal como tenemos definida la unidad de control, podemos especificar lo que ha de hacer el computador para resolver un problema mediante un grafo de estados (que es lo que hemos hecho hasta ahora) o mediante un programa (que es hacia donde vamos).

Veamos un mismo ejemplo para los dos casos. En la figura 10.3 se ven los tres nodos del grafo que diseñamos en el capítulo anterior para leer un dato de la impresora y dejarlo en R3. Especificamos el estado siguiente mediante los arcos que salen del nodo y la palabra de control mediante mnemotécnicos. Para que se ejecuten las acciones de este grafo deberíamos primero codificar los estados (decidir los 16 bits que codifican cada estado). Esto es lo mismo que decidir en qué dirección de la memoria de instrucciones escribiremos la información de cada nodo, de cada instrucción. Supongamos que asignamos el código (dirección) 1 al nodo 1 del grafo, el 2 al 2 y el 3 al 3. Ahora debemos crear la palabra de 75 bits (instrucción de lenguaje máquina) que escribiremos en cada posición de la memoria de instrucciones. Por ejemplo, la palabra que va en la dirección 2, que es la especificada en el nodo 2, consiste en (de izquierda a derecha).

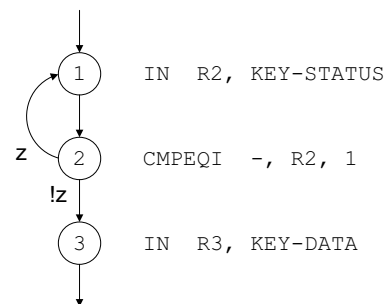


Fig. 10.3 Tres primeros nodos del grafo de la UC para leer un dato del teclado.

- 32 bits para codificar el estado siguiente: 16 bits para la dirección de la siguiente instrucción en caso de que Z valga 1 (en el ejemplo se codifica un 1: 0000000000000001) y 16 bits para cuando Z valga 0 (se codifica un 3: 0000000000000011)
- los 43 bits de la palabra de control, que en el ejemplo son:

@A			@B			Rb/N	OP	F			In/Alu		@D				WrD	Wr-Out	Rd-In	N (hexa)				ADDR-IO (hexa)
0	1	0	x	x	x	0	0	1	0	1	1	x	x	x	x	0	0	0	0	0	0	1	X	X

A continuación expresamos en hexadecimal el contenido de la dirección 2 de la memoria de instrucciones. Codificamos los 75 bits en una palabra de 19 dígitos hexadecimales (76 bits), por lo que el bit de más peso del dígito de más peso de la palabra, que hemos codificado con 0 no es significativo. Además, codificamos los valores que no importan, las x, como ceros:

0x00008001A0580000100

También podemos especificar el programa en **lenguaje ensamblador** que consiste en una secuencia de sentencias, de líneas de código, como se muestra en la figura 10.4 para el mismo ejemplo. De momento, cada línea de código tiene 2 partes:

- Una etiqueta, nombre simbólico de la dirección de la I-MEM donde estará almacenada la instrucción, seguida del símbolo “dos puntos” como separador.
- La instrucción propiamente dicha, que consta de dos partes separadas por el símbolo `| |` indicando que se ejecutan en paralelo (en el mismo ciclo) y por lo tanto que el orden entre ellas es irrelevante:
 - El mnemotécnico de la palabra de control.
 - La siguiente instrucción a ejecutar en función del valor de Z.

También podríamos añadir un comentario a la derecha de cada línea, precedido por el símbolo “punto y coma” (`;`), aunque esto no se ha usado en este ejemplo.

Para ejecutar este programa primero hay que traducirlo a lenguaje máquina, esto es, traducir cada instrucción de ensamblador a lenguaje máquina (a una palabra de 72 bits), asignando valores distintos de 16 bits a cada etiqueta y después se deberá escribir el programa en lenguaje máquina en la memoria de instrucciones, cada instrucción en la dirección asignada. Si asignamos la etiqueta (*label*) L1 a la dirección 1, la L2 a la 2, etc., como hicimos con el grafo, la palabra que debemos escribir en la dirección 2 de la memoria de datos es la misma que antes: 0x00008001A0580000100.

```
L1:      IN R2, KEY-STATUS // goto L2
L2:      CMPEQI -, R2, 1   // if (z) goto L1 else goto L3
L3:      IN R3, KEY-DATA   // goto L4
L4:
```

Fig. 10.4 Especificación de la unidad de control mediante un fragmento de código ensamblador, para el grafo de la figura 10.3.

No obstante, como hemos dicho, este no es ni el lenguaje ensamblador ni la unidad de control definitivos.

10.2 Estructura de la unidad de control general

En las siguientes secciones vamos a ir transformando el lenguaje maquina/ensamblador y la unidad de control hasta conseguir instrucciones con 16 bits. Dos técnicas nos van a permitir la reducción del tamaño de las instrucciones:

- secuenciamiento semiimplícito¹ (absoluto y, posteriormente, relativo), en esta sección y
- codificación de las instrucciones en la sección 10.3

1. Nos inventamos el calificativo de semiimplícito, ya que lo que hacemos en estas dos secciones no es secuenciamiento implícito, todavía. Se llegará al verdadero secuenciamiento implícito en la sección 10.3.1.

10.2.1 Del secuenciamiento explícito absoluto al secuenciamiento semiimplícito absoluto

En la unidad de control actual (ver figura 10.2) y, por lo tanto, en las instrucciones de lenguaje máquina (o ensamblador) actuales, hay que explicitar en cada instrucción cuál es la instrucción que se debe ejecutar al ciclo siguiente (en función de Z). Esto se hace en los dos campos (de 16 bits cada uno) que se encuentran en la parte izquierda de cada instrucción en lenguaje máquina: dirección de la siguiente instrucción en caso de que Z valga 1 (campo de más a la izquierda) y en caso de que valga 0 (siguiente campo hacia la derecha). Por indicar la dirección completa se denomina direccionamiento absoluto. Por ello, a un lenguaje con estas características se le denomina lenguaje con **secuenciamiento explícito absoluto** y a una unidad de control como la de la figura 10.2 se la puede denominar unidad de control con secuenciamiento explícito absoluto.

Ahora vamos a aprovechar una característica típica de los grafos/programas. La figura 10.5 muestra un fragmento de grafo típico en el que se ha omitido las etiquetas con valor x (no importa) de los arcos que van siempre de un nodo a otro independientemente del valor de Z (arcos que salen de un nodo del que sólo sale un arco). Se ha omitido, por no tener interés en este momento, la palabra de control de salida de cada nodo (mnemotécnicos o palabra de 43 bits). En este grafo, los nodos que se han de ejecutar siempre consecutivos (porque el estado siguiente es independiente de Z) se han numerado consecutivamente. La experiencia indica que casi todos los programas que escribimos

- tienen secuencias de nodos/instrucciones que se ejecutan siguiendo siempre el mismo orden con independencia del valor de Z y que
- sólo en algunos nodos/instrucciones hay que indicar dos estados siguientes en función de Z.

El registro de estado de la unidad de control de la figura 10.2 es un puntero a la I-MEM: contiene la dirección de memoria de la instrucción que se está ejecutando en cada ciclo. Por esto, en algunos procesadores, a este registro se le denomina IP (*Instruction Pointer*) o IAR (*Instruction Address Register*), aunque nosotros no usaremos ninguno de estos dos nombres. Si el registro de estado contiene la dirección i, lo dicho en el párrafo anterior quiere decir que uno de los dos campos de la instrucción almacenada en la dirección i de la I-MEM, que indican la dirección de la siguiente instrucción a ejecutar, contiene el valor i+1, mientras que el otro campo contiene cualquier otra dirección (que llamaremos k).

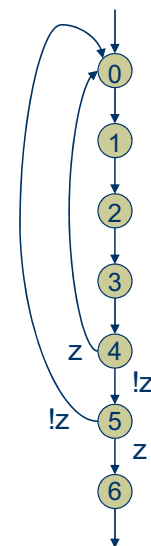


Fig. 10.5 Ejemplo de grafo.

Ejemplo 1

El contenido de la I-MEM (la ROM) de la figura 10.2 para el grafo de la figura 10.5 es el siguiente (sólo para las primeras direcciones y expresando los dos campos de más a la izquierda en hexadecimal y la palabra de control (campo de la derecha) no indicada, puesto que no lo está en el grafo):

Dirección (hexa)	Contenido		
	(hexa)	(hexa)	(no codificado)
0000	0001	0001	Pal. Control nodo 0
0001	0002	0002	Pal. Control nodo 1
0002	0003	0003	Pal. Control nodo 2
0003	0004	0004	Pal. Control nodo 3
0004	0000	0005	Pal. Control nodo 4
0005	0006	0000	Pal. Control nodo 5
0006	0007	0007	Pal. Control nodo 6

La idea, ahora, es no almacenar en cada instrucción (en cada palabra de la I-MEM) el campo de 16 bits con valor $i+1$ y así ahorrarnos 16×2^{16} bits en la I-MEM. Al aplicar esta idea se obtiene la estructura de la nueva unidad de control de la figura 10.6 y que explicamos a continuación.

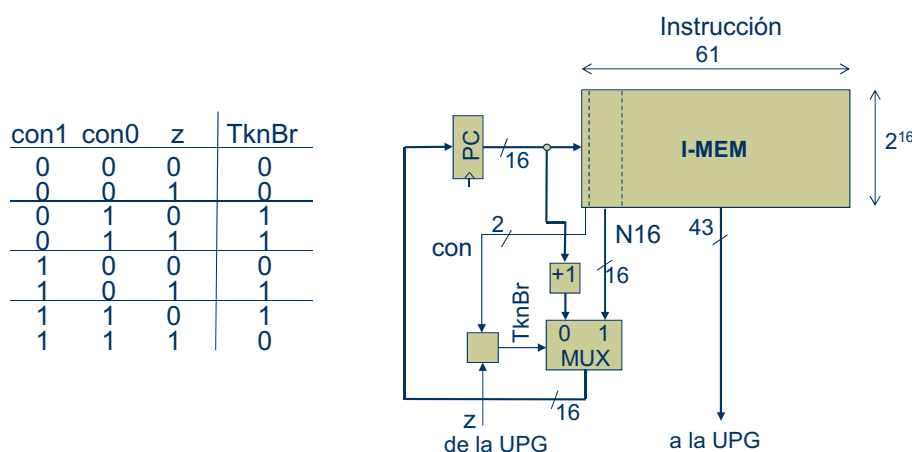


Fig. 10.6 Unidad de control con secuenciamiento semiimplícito absoluto.

Ha sido posible ahorrarnos 14 bits (16 eliminados menos dos añadidos para codificar con) en cada instrucción a costa de añadir el bloque combinacional que incrementa en +1 el valor i contenido en el registro de estado, obteniendo el valor $i+1$.

En la figura 10.6 puede verse el bloque **incrementador** (etiquetado con +1) y la conexión que va de la salida del registro de estado a la entrada del registro de estado pasando por el incrementador y por el multiplexor (gobernado por la señal TknBr). Un registro con esta capacidad de incrementar su valor en cada ciclo se denomina registro **contador** y por eso en muchos procesadores el registro de estado de la unidad de control se denomina registro **contador de programa, PC, Program Counter**. Este será el nombre que nosotros le daremos de ahora en adelante al registro de estado: PC (y así se ha etiquetado en la figura).

En la mayoría de los ciclos el PC se incrementará (contará) y en sólo unos pocos se cargará con otro valor distinto al incrementado. Como se ve en la figura de la nueva unidad de control, la señal TknBr selecciona, a través del multiplexor de buses, cuál es la dirección de la I-MEM donde se encuentra la instrucción/nodo que se ejecutará al siguiente ciclo: dirección $i+1$ (que se calcula) si TknBr vale 0 o dirección k (contenida en un campo de la instrucción actual) si TknBr vale 1. TknBr es el acrónimo de **Taken Branch** que indica, cuando vale 1, que se rompe la ejecución secuencial: que no se pasa del nodo i al $i+1$ sino a otro cualquiera, al k (se *toma la rama* (*Taken Branch*), ver figura 10.5).

Así que en cada palabra de la I-MEM sólo hace falta almacenar un campo de 16 bits, que denominamos N16, para almacenar el valor k . Eliminando el otro campo hemos ahorrado 16 bits por instrucción, a costa del circuito incrementador y de añadir un pequeño campo de dos bits más en cada instrucción: campo con (de condición), como se ve en la figura 10.6. Este nuevo campo se usa para codificar en dos bits cómo se ha de generar la dirección de la siguiente instrucción a ejecutar, que se cargará en el PC al final del ciclo.

La tabla de verdad de la figura 10.6 define el comportamiento del dispositivo combinacional que calcula la señal TknBr en función del código con de la instrucción actual y del valor de z que llega de la ALU a la unidad de control (bloque cuadrado sin etiqueta). Según el valor del campo con el secuenciamiento puede ser uno de los cuatro que se indican (El PC actual contiene el valor i y el campo de 16 bits de la instrucción (N16) contiene el valor k o x (no importa):

- con = 00: PC \leftarrow PC + 1 ;siguiente en secuencia
- con = 01: PC \leftarrow N16 ;saltar incondicional
- con = 10: if (z) PC \leftarrow N16 else PC \leftarrow PC + 1 ;saltar si cero
- con = 11: if ($!z$) PC \leftarrow N16 else PC \leftarrow PC + 1 ;saltar si no cero

Con esto hemos reducido los bits de la I-MEM (cada instrucción ha pasado de 75 a 61 bits), aunque para ello se requiere un circuito incrementador (16 Half-adders) y algunas puertas más. En la nueva unidad de control, si se está ejecutando la instrucción i (instrucción almacenada en la dirección i de la I-MEM) la siguiente instrucción a ejecutar sólo puede ser o la $i+1$ u otra cualquiera, k . Esta limitación no es grave si se ordenan adecuadamente las instrucciones (se numeran adecuadamente los nodos del grafo).

La nueva unidad de control la podemos denominar unidad de control con **secuenciamiento semiimplícito absoluto**, ya que cada instrucción sólo indica explícitamente la dirección **absoluta** de la siguiente instrucción a ejecutar en caso de ruptura de la ejecución en secuencia, mientras que la dirección de la siguiente instrucción en secuencia no se encuentra explícitamente en ningún campo de la instrucción, sino que está **implícita** (es siempre PC+1 para cualquier instrucción). La figura 10.7 muestra otra forma de dibujar el circuito de la nueva unidad de control, equivalente al de la figura 10.6.

Ejemplo 2

El contenido de la I-MEM de la figura 10.7 para el grafo de la figura 10.5 es el siguiente (con las mismas consideraciones que para el Ejemplo 1, excepto que los dos bits del campo con, de más a la izquierda, se expresan en binario):

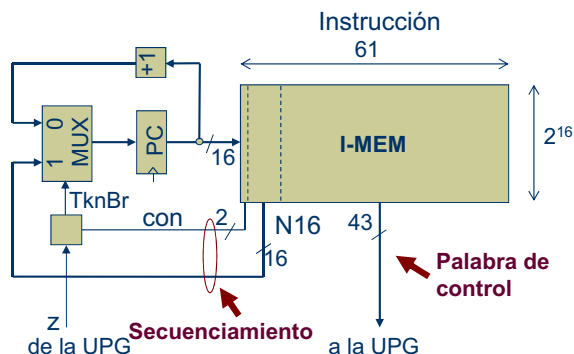


Fig. 10.7 Unidad de control con secuenciamiento semiimplícito absoluto. Otra forma de dibujar el circuito.

Dirección (hexa)	Contenido		
	(bin)	(hexa)	(no codificado)
0000	00	XXXX	Pal. Control nodo 0
0001	00	XXXX	Pal. Control nodo 1
0002	00	XXXX	Pal. Control nodo 2
0003	00	XXXX	Pal. Control nodo 3
0004	10	0 0 0 0	Pal. Control nodo 4
0005	11	0 0 0 0	Pal. Control nodo 5
0006	00	XXXX	Pal. Control nodo 6

10.2.2 Secuenciamiento semiimplícito relativo

Otra característica de los grafos/programas es que usualmente desde un nodo/instrucción se salta a ejecutar o el siguiente nodo u otro que está cerca de él. No es normal escribir un programa/grafos con muchos nodos y que la mayoría de los saltos sean muy lejanos. Esta característica nos va a permitir reducir el campo de 16 bits de la dirección “absoluta” N16 que se usa en caso de ruptura de secuencia.

La figura 10.8 muestra la implementación de la nueva unidad de control. Hemos sustituido el campo N16 de la instrucción por un campo de sólo 8 bits, N8. El campo N8 codifica, en complemento a dos, el número de instrucciones que hay que contar desde la siguiente instrucción a la actual para llegar a la instrucción destino de salto tomado. Si N8 es un valor positivo hay que contar N8 instrucciones hacia adelante y si es negativo se debe contar el valor absoluto de N8 hacia atrás, a partir de la instrucción siguiente a la actual.

. Ahora, el PC se va a incrementar en 1 por si el salto es no tomado y además se le suma a PC+1 el valor codificado en complemento a dos en el campo N8 por si se rompe la secuencia. Para hacer esta segunda suma se extenderá previamente el bit de signo de N8 hasta formar 16 bits. Dado que el campo N8 codifica un número entero en complemento a dos cuyo rango va de -128 a + 127, desde una instrucción (que está en la dirección que indica el PC durante el ciclo en el que se está ejecutando) sólo

podremos saltar, romper la secuencia, a otra que esté como máximo 127 instrucciones antes ($PC+1+(-128)$) o 128 después ($PC+1+127$) de la instrucción actual. El coste en hardware añadido es el de un sumador de 16 bits, ya que el bloque SE que extiende el bit de signo de N8 no requiere ninguna puerta.

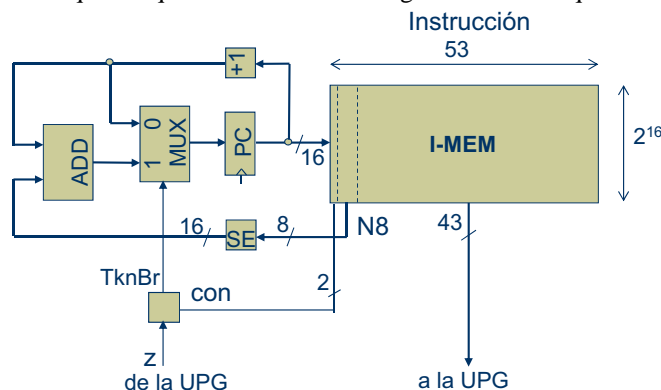


Fig. 10.8 Unidad de control con secuenciamiento semiimplícito relativo.

La dirección destino de salto tomado que calcula el hardware es exactamente $[(PC+1)\%(2^{16}) + SE(N8)]\%(2^{16})$. No obstante, como ya sabemos que al hacer una asignación a un registro de 16 bits se escriben los 16 bits de menor peso de la expresión de la derecha, reescribimos el calculo anterior así:

$$PC \leftarrow (PC + 1) + SE(N8)$$

Hemos mantenido el paréntesis de $PC+1$ porque vamos a hablar ahora de la suma de $PC+1$ con $SE(N8)$. Esta suma, que como se muestra en el esquema de la figura 10.8 se efectúa con el sumador binario de dos vectores de 16 bits, puede parecer un tanto extraña ya que estamos sumando un número natural, la dirección de la siguiente instrucción si el salto fuera no tomado ($PC+1$), más un número entero, denominado desplazamiento, que está codificado en complemento a dos con 8 bits en un campo de la instrucción y que ha sido representado en 16 bits antes de ser sumado mediante la extensión de su bit de signo. Ya vimos en el capítulo 5 que el sumador binario suma correctamente tanto números enteros como naturales, siempre y cuando el resultado sea representable en 16 bits. Ahora lo que hacemos es usar el sumador binario para sumar un número natural con otro entero e interpretar el resultado como natural; y esto es correcto siempre que el resultado pueda representarse en 16 bits.

Esta forma de calcular la dirección destino de una ruptura de secuencia se llama modo **relativo al PC**, y de ahí el nombre que le hemos dado a la unidad de control resultante (y a su lenguaje máquina): **con secuenciamiento semiimplícito relativo**. La codificación/significado del campo **con** en esta implementación de la unidad de control puede ser como en la unidad de control anterior, cambiando N16 por la extensión de signo a 16 bits de N8.

Ejemplo 3

El contenido de la I-MEM de la figura 10.8 para el grafo de la figura 10.5 es el siguiente (con las mismas consideraciones que para el Ejemplo 2). El campo N8 de la instrucción/nodo 4 (la que se accede con $PC=4$) es -5 para que al sumarlo a $PC+1$, que vale 5, se obtenga la dirección 0. En

complemento a dos el -5 se codifica con 8 bits como 1111011, que en notación hexadecimal es FB. De forma equivalente, el campo N8 para la instrucción 5 de la I-MEM es FA.

Dirección (hexa)	Contenido		
	(bin)	(hexa)	(no codificado)
0000	00	XX	Pal. Control nodo 0
0001	00	XX	Pal. Control nodo 1
0002	00	XX	Pal. Control nodo 2
0003	00	XX	Pal. Control nodo 3
0004	01	FB	Pal. Control nodo 4
0005	10	FA	Pal. Control nodo 5
0006	00	XX	Pal. Control nodo 6

10.2.3 Una vuelta de tuerca más: Unidad de Control General con instrucciones de 16 bits

Ahora vamos a reducir mucho los bits de cada instrucción, pasando de los 53 actuales a únicamente 16 bits ¿Cómo puede ser esto posible?

Por un lado vamos a reducir el tamaño de algunos campos de la instrucción, aunque esto requiera que algunas instrucciones ya no se puedan ejecutar y su funcionalidad ahora requiera la ejecución de una secuencia de varias instrucciones: algunas cosas que antes se hacían en un ciclo ahora requerirán varios. Por ejemplo, es imposible poner toda la información de la acción `MOVEI R1, N` en 16 bits cuando solamente el campo N es actualmente de 16 bits. Ahora vamos a necesitar 2 ciclos para cargar 16 bits en R1, habrá que ejecutar dos nuevas instrucciones que sustituyen a la antigua acción `MOVEI`. La primera instrucción cargará los 8 bits de menor peso y la segunda los 8 de más peso. Van a aparecer varios casos como este.

Por otro lado vamos a codificar las instrucciones usando tres formatos de instrucción diferentes. ¿Qué quiere decir esto? Veamos primero la motivación. Muchos de los campos de la instrucción actual, de 53 bits, no se pueden usar a la vez, debido a las limitaciones del hardware de la UPG: de ahí las muchas x que aparecen en la palabra de control de algunas acciones/instrucciones. Por ello, no todos los campos de la palabra de control son necesarios para ejecutar una instrucción concreta. Por ejemplo, si se está ejecutando `ADD R1, R2, R3`, no se puede estar ejecutando en el mismo ciclo `IN R5, 2` ya que, entre otras cosas, sólo hay un bus de escritura en el banco de registros. Así pues, cuando se ejecuta `ADD R1, R2, R3`, el campo de 8 bits `ADDR-IO`, necesario para ejecutar la instrucción `IN`, no se usa. La idea del formato y codificación de las instrucciones es que algunos bits de las nuevas instrucciones de 16 bits unas veces tengan un significado (por ejemplo, para codificar registros de la instrucción `ADD R1, R2, R3`) y otras veces tengan otro significado (por ejemplo, para alojar los 8 bits de `ADDR-IO` cuando ejecutemos `IN R5, 2`).

En cada instrucción hay un campo fijo, que en nuestro caso son los 4 bits de más peso de la instrucción, que codifica el tipo de instrucción de que se trata (aritmético-lógica, de comparación, de entrada/salida, etc.). Este campo, denominado **código de operación**, informa cómo se deben interpretar el resto de los 12 bits de la instrucción, nos indican el **formato de la instrucción**: qué campos tiene la instrucción y qué codifican.

Con esto vamos a conseguir definir un conjunto de instrucciones, cada una de 16 bits, que denominamos instrucciones del **lenguaje máquina SISA**, (*Simple Instruction Set Architecture*). Este conjunto de instrucciones serán las que va a ejecutar, en un ciclo cada una, nuestra primera versión de computador **SISC** (*Simple Instruction Set Computer*) que es Harvard unicycle.

Tener instrucciones de 16 bits supone un ahorro de bits en la memoria de instrucciones. Pero esto tiene un coste. Por un lado, como hemos dicho, algunas acciones ahora requerirán varias instrucciones, varios ciclos. Esto supone un aumento en el tiempo de resolución de los problemas y también supone que la reducción en bits de la memoria de instrucciones no es tan significativa como se ha dicho. Por otro lado, hay que disponer de un pequeño circuito adicional para decodificar la instrucción de 16 bits (que se lee en cada ciclo de la I-MEM), crear la palabra de control y gestionar el cálculo de la dirección de la siguiente instrucción a ejecutar (secuenciamiento). Denominamos **Lógica de control**, Control Logic, a este nuevo circuito combinacional.

Primera aproximación a la unidad de control general

Una primera aproximación a la estructura a bloques de la nueva unidad de control general de nuestro procesador se muestra en la figura 10.9. Esta unidad de control también calcula la dirección destino de salto tomado como la de la figura 10.8, $PC+1+SE(N8)$, pero ahora los 16 bits de $SE(N8)$ de la figura 10.8 se encuentran en el campo N de la palabra de control que crea la lógica de control a partir del campo N8 que tendrán las instrucciones que puedan romper la ejecución en secuencia, como vemos más adelante.

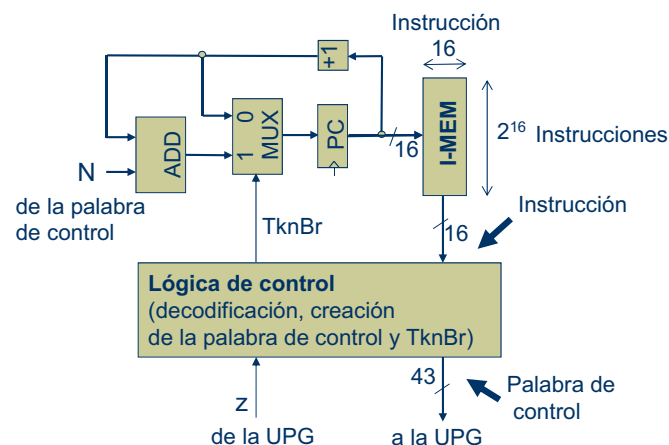


Fig. 10.9 Unidad de control general preliminar.

I-MEM con unidad de direccionamiento de byte

La estructura definitiva de la unidad de control general (UCG) de nuestro procesador se muestra en la figura 10.10. Las diferencias respecto de la propuesta de la figura 10.9 se producen porque ahora usamos una I-MEM cuya unidad de direccionamiento es el byte.

La **unidad de direccionamiento** de una memoria (tanto si es ROM como RAM) es la mínima cantidad de información que se puede direccionar. Dicho de otra forma, es el número de bits que tiene el dato más pequeño que se puede leer o escribir en un solo acceso a memoria. Si, por ejemplo, la unidad de direccionamiento de una memoria es el byte y el bus de direcciones de memoria es de 16 bits, la memoria tiene como mucho una capacidad de 2^{16} bytes. La unidad de direccionamiento de las memorias ROM que hemos usado en cada uno de los pasos anteriores son:

- 75 bits para la figura 10.2,
- 61 bits para las figuras 10.6 y 10.7,
- 53 bits para la figura 10.8 y
- 16 bits para la figura 10.9.

Sin embargo, la I-MEM definitiva (figura 10.10) tiene unidad de direccionamiento de 8 bits, (de un byte) que es la mitad de bits que tiene una instrucción. La dirección de la I-MEM consta de 16 bits (el número de bits del PC) por lo que ahora puede direccionar 2^{16} bytes diferentes, que es la mitad de la capacidad que la I-MEM de la figura 10.9. Esto, de momento, tiene una importante desventaja: solo podemos almacenar 2^{15} instrucciones, ya que cada instrucción ocupa 2 bytes.

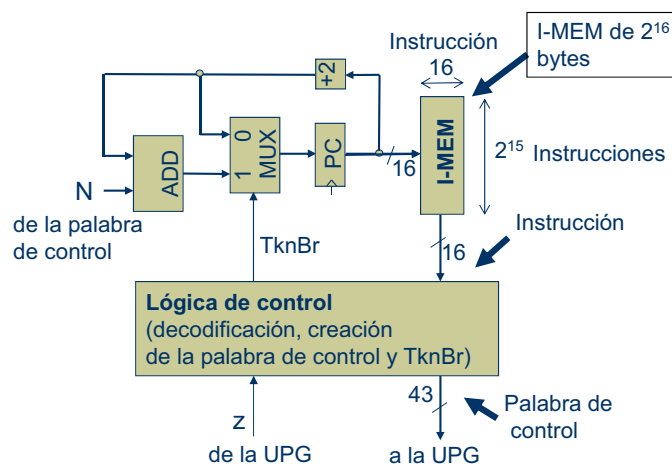


Fig. 10.10 Estructura definitiva de la unidad de control general (UCG)

¿Por qué una I-MEM con unidad de direccionamiento de byte si no parece tener ventajas?

Ciertamente, en este momento sería ventajoso usar una I-MEM con unidad de direccionamiento de word (16 bits): la I-MEM podría almacenar el doble de instrucciones. A pesar de ello usamos una memoria con unidad de direccionamiento de byte para que nos sea más fácil crear el computador SISC von Neumann, que es objetivo final del camino que estamos recorriendo. Lo explicamos a continuación.

Todos los computadores tienen una memoria RAM (de lectura y escritura) para almacenar **datos** ya que con los 8 registros de la UPG es insuficiente para casi todos los problemas que resuelve un computador.

Por ejemplo, con solo 8 posiciones de memoria (o registros) es imposible ordenar alfabéticamente una lista de 1.000 nombres que entran a la UPG por el bus RD-IN. Aunque es en el capítulo siguiente donde añadimos una memoria RAM de datos a nuestro computador, adelantamos ahora que este tipo de memorias siempre tiene unidad de direccionamiento de byte. ¿Por qué?

Aunque el tamaño de la palabra/word de nuestro computador es de 16 bits (que es el tamaño de los registros y de los operandos y resultados que se procesan en la ALU en un ciclo) sería ineficiente tener una memoria de datos con unidad de direccionamiento de 16 bits. Es eficiente poder direccionar (leer o escribir) un único byte en memoria porque hay tipos de datos que no requieren más que 8 bits para codificar cada dato, como ocurre con los caracteres alfanuméricos. Para almacenar un texto (una secuencia de caracteres) en una memoria con unidad de direccionamiento de 16 bits (2 bytes) tenemos dos posibilidades:

- almacenar cada carácter en una palabra de 16 bits (por ejemplo en sus 8 bits de menor peso) o
- almacenar dos caracteres en cada palabra.

La primera solución tiene la ventaja de que el acceso a cada carácter es rápido (se lee o se escribe en la dirección de memoria donde está almacenado el carácter mediante una única instrucción de acceso a memoria) pero la desventaja es que estamos usando el doble de la capacidad de memoria necesaria ya que usamos 2 bytes por carácter y con uno sería suficiente. La segunda opción no tiene esta desventaja, usa solo un byte por carácter, pero tiene la desventaja de requerir mas instrucciones para leer un único carácter ya que hay que acceder a la palabra donde está el carácter y luego extraerlo, ya que hay dos caracteres en cada palabra y solo queremos uno. Para escribirlo aún sería peor, habría que leer la palabra donde deberá escribirse el carácter, escribir el carácter en el byte que le corresponda dentro de la palabra (sin modificar el otro byte) y finalmente escribir la palabra en la memoria en la misma dirección que estaba.

La solución adoptada para facilitar el acceso a bytes y a words es usar memorias RAM con unidad de direccionamiento de byte que permiten en un solo acceso leer o escribir en memoria o bien un byte aislado o dos consecutivos que forman una palabra. Cómo se construyen estas memorias no es difícil y lo veremos en el siguiente capítulo. Los tres computadores que diseñaremos usan la misma memoria RAM de lectura/escritura con unidad de direccionamiento de byte que tiene cuatro instrucciones que mueven datos de tamaño byte o word entre los registros de la UPG y la memoria de datos.

En los dos primeros computadores que diseñamos, que son Harvard, como la memoria de datos y la de instrucciones son distintas no habría ningún problema para hacer que la de datos tuviera unidad de direccionamiento de byte y la de instrucciones de word. Pero como el computador final, SISC von Neumann, tiene una única memoria que sirve para almacenar tanto datos como instrucciones se usa una memoria con unidad de direccionamiento de byte y capacidad para acceder a byte y word, aunque para acceder a las instrucciones siempre se accede a word.

Si usáramos una I-MEM con unidad de direccionamiento de word para los dos primeros computadores y una con unidad de direccionamiento de byte para el tercero al llegar a este tendríamos que cambiar la parte de secuenciamiento y la semántica de las instrucciones de ruptura de secuencia que vamos a definir en este capítulo. Para no hacer esto vamos a empezar por usar una I-MEM con unidad de direccionamiento de byte desde este mismo momento, así ya no tendremos que hacer más cambios:

Esta es la única ventaja de esta decisión: acostumbrarnos al secuenciamiento y la semántica de las instrucciones de salto y no tener que cambiarla más adelante.

Estructura interna de la I-MEM. Como de momento estamos diseñando un computador Harvard, la I-MEM puede construirse con una ROM (ya que solo debe leerse) y no hace falta que se pueda acceder a un byte individual porque siempre se accede a instrucciones: en cada acceso solo se necesita poder leer 16 bits, 2 bytes consecutivos. Así que podemos implementar la I-MEM con una ROM de 2^{15} palabras de 16 bits cada una de forma que los 15 bits de dirección de esta ROM los obtenemos de los 15 bits de más peso del PC, desechando el bit de menor peso (ver figura 10.11 para el circuito interno y 10.12 para el bloque que desecha el bit 0 de la dirección). Así pues, los dos bytes de cada instrucción tienen direcciones @ y @+1 siendo @ una dirección par. Cuando hablemos de la dirección de una instrucción nos referiremos a su dirección par (aunque con la implementación de la figura 10.11 se accede a la misma instrucción tanto si el PC contiene una dirección par, @, como si contiene @+1).

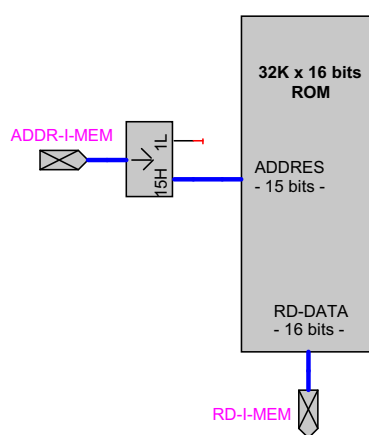


Fig. 10.11 Circuito interno de la I-MEM

Tener la I-MEM con unidad de direccionamiento de byte implica dos cambios más en la UCG de la figura 10.10 respecto a la de la de la de la figura 10.9 que comentamos a continuación.

Ejecución secuencial: $PC = PC+2$

Si en el ciclo actual el PC contiene la dirección @ (que consideramos par) se está ejecutando la instrucción almacenada en la I-MEM en los dos bytes consecutivos con direcciones @ y @+1. Si al ciclo siguiente se debe ejecutar la siguiente instrucción en secuencia, esto es la que se encuentra en los bytes con direcciones @+2 y @+3, al final del ciclo actual hay que cargar el PC con el valor del PC actual incrementado en 2 unidades. Por eso la UPG definitiva (figura 10.10) tiene un bloque +2 en vez del bloque +1 de la primera versión (figura 10.9).

16-bit to 15-bit High and 1-bit Low**16-to-15H1L****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:

$$1L = X(b0)$$

$$15H(bi) = X(bi+1) \text{ for } i = 0 \text{ to } 7$$

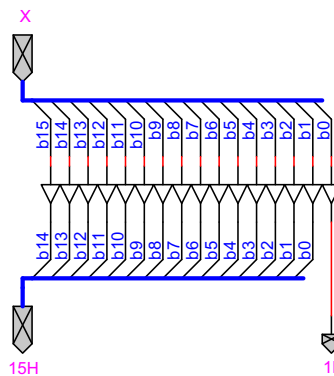
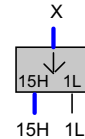


Fig. 10.12 Circuito interno del bloque 16-to-15H1L

Ruptura de secuencia: $PC = PC+2+SE(N8)*2$

Por otro lado, si en el ciclo actual se debe romper la ejecución secuencial y pasar a ejecutar otra instrucción la dirección destino de salto tomado ya no se calcula como en la primera aproximación de la unidad de control, $PC+1+SE(N8)$, sino que se calcula sumando a $PC+2$ el valor de N que genera la Lógica de control a partir de ahora como $N = SE(N8) * 2$. Se multiplica N8 por 2 porque cada instrucción ahora ocupa dos bytes consecutivos, dos direcciones consecutivas, y de esta forma se puede saltar a ejecutar el mismo rango de instrucciones que antes: hasta 128 instrucciones antes de la siguiente a la actual o hasta 127 después. Este cambio no se ve en la estructura a bloques de la figura 10.10 porque afecta al diseño interno del bloque Lógica de control que genera N.

En el resto de este capítulo creamos las instrucciones SISA de 16 bits y finalmente implementaremos el circuito interno del bloque combinacional Lógica de control que, a partir de la instrucción de 16 bits, genera los 43 bits de la palabra de control y el bit encargado del secuenciamiento, TknBr.

10.3 Creando el lenguaje máquina y ensamblador SISA

Antes de pasar a dar formato y codificar las instrucciones SISA vamos a modificar, por última vez, el secuenciamiento de las instrucciones.

10.3.1 Secuenciamiento implícito

El secuenciamiento de las instrucciones en el lenguaje SISA va a ser implícito ya que es imposible codificar en los 16 bits de una instrucción SISA la información necesaria para

- generar la palabra de control que le indique a la UPG que realice una de las posibles acciones que puede realizar en un ciclo (operación aritmético-lógica, comparación, IN y OUT) como, por ejemplo, `SUB -, R2, R3`) y para
- romper la secuencia de ejecución en función del bit z que llega de la UPG a la unidad de control, para hacer, por ejemplo, `if (z) PC <- PC + 2 + SE(N8)*2 else PC <- PC + 2`.

En el ejemplo se requieren al menos 6 bits para codificar con 3 bits cada uno de los 2 registros de la instrucción `SUB`, más los 8 bits de `N8` para indicar el desplazamiento a sumar al PC en caso de ruptura de secuencia. Esto solo ya supone 14 bits, sin contar los bits necesarios para indicar si se rompe la secuencia siempre, nunca, en caso de z o de !z, así como los bits necesarios para codificar la acción `SUB` y diferenciarla de las otras acciones posibles que pueden hacerse en la UPG. Y la situación aún sería peor si la acción `SUB` tuviera registro destino.

Por ello, lo que se podría haber hecho en un ciclo si las instrucciones tuvieran suficientes bits,

```
SUB -, R2, R3 // if (z) PC <- PC + 2 + SE(N8)*2 else PC <- PC + 2
```

ahora necesitará dos ciclos ya que vamos a codificar esa información en dos instrucciones de 16 bits cada una. Estas dos instrucciones se ejecutarán en secuencia: la primera hará la parte de la acción en la UPG y la segunda será del tipo “ruptura de secuencia condicional”.

A partir de ahora, en los ciclos en que se ejecute una acción en la UPG, como `SUB`, la instrucción que se ejecutará después es siempre la siguiente en secuencia. Esto es, en los dos ciclos del ejemplo se hará:

```
PC <- PC + 2 // SUB -, R2, R3
if (z) PC <- PC + 2 + SE(N8)*2 else PC <- PC + 2
```

Pero ahora resulta que la instrucción de ruptura de secuencia se ejecuta al ciclo siguiente del ciclo en el que se ejecuta la instrucción aritmética que genera el valor de z según el cual se desea romper o no la secuencia. Pero este valor de z ya no se encuentra en el cable z que sale de la ALU y llega a la unidad de control en el ciclo en que se ejecuta la instrucción de ruptura de secuencia. Por eso ahora la instrucción aritmética debe almacenar el resultado de su ejecución en un registro, por ejemplo `R1`, para que la instrucción de ruptura de secuencia actúe en función del valor contenido en el registro `R1`. El código queda como:

```
PC <- PC + 2 // SUB R1, R2, R3
if (R1 == 0) PC <- PC + 2 + SE(N8)*2 else PC <- PC + 2
```

Así, las instrucciones de ruptura de secuencia necesitan para su ejecución que en la UPG se lea el registro del banco de registros sobre el que se ha de evaluar la condición de la ruptura de secuencia, en nuestro caso el R1, y se pase el contenido del registro por la ALU sin ser modificado para que la ALU genere el bit z según el contenido del registro sea cero o distinto de cero. La unidad de control ya puede decidir en este ciclo si rompe la secuencia o no en función del bit z que le llega de la UPG.

Como en todas las instrucciones, excepto las de ruptura de secuencia, el secuenciamiento es el mismo, se omite éste en la notación de la instrucción. El ejemplo anterior queda como:

```
SUB R1, R2, R3
if (R1 == 0) PC <- PC + 2 + SE(N8)*2 else PC <- PC + 2
```

A esta forma de secuenciamiento se le denomina **secuenciamiento implícito**. Esto es, si no se indica lo contrario mediante una instrucción de ruptura de secuencia, la siguiente instrucción a ejecutar es la siguiente en secuencia: el secuenciamiento es: $PC \leftarrow PC + 2$.

A una instrucción de ruptura de secuencia se le denomina de forma coloquial **instrucción de salto** (cuando se rompe la secuencia se da un salto para ejecutar la siguiente instrucción) y en la literatura anglosajona se denomina **branch instruction** (como si, cuando se rompe la secuencia, se saliera por una rama del tronco que suponen los nodos que se ejecutan con el secuenciamiento $PC=PC+2$).

En resumen, nuestro lenguaje SISA sólo tendrá instrucciones de salto condicional según que el contenido de un registro sea cero o distinto de cero. Estas instrucciones se indican con un mnemotécnico más sencillo que la sentencia `if then else` usada hasta ahora:

Saltar si cero

```
BZ Ra, N8
```

(*Branch on Zero*) para indicar

```
if (Ra == 0) PC <- PC + 2 + SE(N8)*2 else PC <- PC + 2
```

y Saltar si no cero

```
BNZ Ra, N8
```

(*Branch on No Zero*) para indicar

```
if (Ra != 0) PC <- PC + 2 + SE(N8)*2 else PC <- PC + 2
```

Una vez añadido a nuestro lenguaje el secuenciamiento implícito ya no vamos a dibujar grafos de estado, con sus nodos y sus flechas para indicar el secuenciamiento, sino que vamos a escribir programas, secuencias de instrucciones SISA. Cuando las instrucciones estén escritas con mnemotécnicos diremos que el programa está escrito en el **lenguaje ensamblador SISA** y cuando las instrucciones estén codificadas cada una de ellas en 16 bits (o en su notación hexadecimal con cuatro dígitos), diremos que el programa está escrito en **lenguaje máquina SISA**. Por ejemplo, el fragmento de grafo de, por ejemplo, la figura 10.3 se especifica con el siguiente fragmento de programa ensamblador SISA, en el que ya no aparece ningún grafo para indicar el secuenciamiento:

```

LOOP:  IN      R2, KEY-STATUS
        CMPEQI R7, R2, 1
        BZ     R7, -3
        IN     R3, DATA-IN

```

Es importante notar que ahora ya no se puede hacer `CMPEQI -, R2, 1`, como sí hicimos el grafo de la figura 10.3. Ahora es necesario almacenar el resultado de la comparación en un registro (en el ejemplo R7) para poder ver, al ciclo siguiente, si el resultado de la comparación fue verdadero ($R7==1$ y, por lo tanto, $z=0$) o falso ($R7!=1$ y, por ello, $z=1$).

Ejemplo 4 Otra forma de entrar un dato

Vamos a escribir el código SISA para leer un dato del teclado y guardarlo en R3. Esta es la misma funcionalidad que hace el código anterior pero lo escribiremos de otra forma: traduciremos a ensamblador el siguiente código en alto nivel (recordamos que el puerto/registro de estado de la impresora solo puede tener el valor 0 o 1):

```

LOOP: if (KEY-STATUS==0) goto LOOP;
      R3=INPUT[KEY-DATA];

```

Como R3 quedará modificado al final del código, lo usamos como registro temporal antes de asignarle el dato leído; de esta forma, no usamos más registros de los estrictamente necesarios. La traducción directa a SISA es:

```

IN      R3, KEY-STATUS
CMPEQI  R3, R3, 0
BNZ     R3, -3
IN      R3, DATA-IN

```

Si cuando se ejecuta la instrucción BNZ el PC contiene la dirección par @ quiere decir que la primera instrucción IN está almacenada en la dirección @-4 (y @-3). Por el campo N8 de BNZ vale -3 para que al ejecutarse se calcule la dirección $@-4 = @+2+(-3)*2$. Otra forma de verlo es que -3 es el número de instrucciones que hay desde la siguiente del salto hasta la instrucción destino del salto (desde la última instrucción IN hasta la primera).

Podemos optimizar este código sabiendo que KEY-STATUS solo puede valer 0 o 1. No hace falta la comparación de R3 con 0 para saber si es verdadero o falso, podemos preguntar directamente si R3 es cero o distinto de cero con la instrucción BZ o BNZ respectivamente. El código optimizado queda así, que es como escribiremos usualmente la entrada de un dato del teclado por encuesta:

```

IN      R3, KEY-STATUS
BZ      R3, -2
IN      R3, DATA-IN

```

10.3.2 Formato y codificación de las instrucciones SISA

Veamos ahora cómo se codifican las instrucciones, comenzando por las aritméticas, lógicas y de comparación, que usan tres registros, y siguiendo por las de ruptura de secuencia y, finalmente, el resto.

Instrucciones aritmético-lógicas y de comparación. Formato de tres registros: 3-R

Comencemos por las instrucciones aritmético-lógicas y de comparación que usan tres registros para su especificación, dos registros fuente y uno destino. Estas instrucciones requieren 3 campos de 3 bits para codificar los registros (total 9 bits). Hay 13 instrucciones diferentes de este tipo: las 8 aritmético-lógicas, que se realizan cuando la ALU recibe la señal de control $OP=00$, según los 3 bits de F, y las 5 de comparación, cuando OP vale 01, para las 5 combinaciones válidas de F). Así que necesitamos al menos 4 bits para diferenciar entre estas 13 operaciones.

Vamos a usar el siguiente formato de instrucción para codificar estas instrucciones. Lo denominamos formato de las instrucciones de 3 registros: 3-R (ver figura 10.13).

	Código de operación				Ra			Rb			Rd			F		
Formato 3-Reg	c	c	c	c	a	a	a	b	b	b	d	d	d	f	f	f
	15				12	11		9	8		6	5		3	2	0

Fig. 10.13 Formato de las instrucciones con tres registros

Si denotamos por I al vector de 16 bits que representa a la instrucción podemos decir que el campo $\langle 11..9 \rangle$ (bits 11, 10 y 9 de la instrucción) codifica en binario con tres bits el registro fuente Ra. Estos tres bits son $a_2a_1a_0$ pero, por sencillez, se han omitido los subíndices en el campo de la instrucción, etiquetándose como aaa. Esta simplificación de la notación se usa en el resto de campos de la instrucción. El campo $\langle 8..6 \rangle$ indica el registro fuente Rb y el campo $\langle 5..3 \rangle$ el registro destino Rd. Además, el campo $\langle 2..0 \rangle$ codifica qué instrucción de entre las 8 aritmético-lógicas se debe ejecutar o cuál de entre las 5 de válidas de comparación. La codificación usada para diferenciar estas instrucciones es la misma que se usa en la ALU para diferenciarlas con los tres bits de F. Así por ejemplo la instrucción

OR R5, R2, R7

se codifica, de momento, como indica la figura 10.14 (ver la tabla de la ALU, donde se muestra que el código $F=001$ se usa para designar la operación OR, cuando OP vale 00).

Formato 3-R	c	c	c	c	0	1	0	1	1	1	1	0	1	0	0	1
-------------	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Fig. 10.14 Formato de las instrucciones con tres registros. Ejemplo de codificación parcial de la instrucción OR R5, R2, R7

Pero la instrucción de la figura 10.14, cuando OP vale 01, también puede ser la instrucción

CMPLE R5, R2, R7

En este formato de instrucción quedan los 4 bits de más peso de la instrucción (campo $\langle 15..12 \rangle$) para indicar el tipo de instrucción de que se trata: una de las 8 operaciones aritmético-lógicas o una de las 5 operaciones de comparación. Este campo, como se dijo antes, se denomina **código de operación**.

Tenemos 16 posibles combinaciones de códigos de operación. Tomamos la decisión de usar el código 0000 para codificar las instrucciones aritmético-lógicas y el 0001 para las de comparación, en ambos casos usando el formato 3-R. Esta decisión es arbitraria: no importa qué código de los 16 demos a cada tipo de instrucción.

La figura 10.15 muestra la codificación en lenguaje máquina de dos instrucciones con el formato de 3 registros, una aritmético-lógica y la otra de comparación.

SHA	R7, R4, R1	0	0	0	0	1	0	0	0	0	1	1	1	1	1	0
CMPLEU	R3, R1, R2	0	0	0	1	0	0	1	0	1	0	0	1	1	1	0

Fig. 10.15 Ejemplo de codificación de dos instrucciones con el formato de 3 registros.

Resto de códigos de operación. Nos quedan 14 códigos de operación para codificar el resto de instrucciones de nuestro procesador. No obstante, vamos a usar solamente 3 códigos más para codificar las instrucciones de esta primera versión del lenguaje SISA, que definimos en este capítulo. En el siguiente capítulo añadiremos la memoria de datos al computador que estamos construyendo y aparecerán cuatro instrucciones más para leer y escribir datos, de tamaño byte (8 bits) y word (16 bits). Cada una de estas cuatro nuevas instrucciones usará un código de operación diferente. Por último, en el capítulo 13 añadiremos la última instrucción al lenguaje SISA, que usará otro código de operación. Con esto usaremos 11 códigos diferentes de los 16 disponibles. Los 5 códigos restantes los dejamos sin asignar para hacerlo en futuras ampliaciones del lenguaje SISA. Veamos a continuación el formato y la codificación de las dos instrucciones de salto.

Instrucciones de ruptura de secuencia. Formato de un registro: 1-R

Tenemos dos instrucciones de ruptura de secuencia (salto) condicional (en función de z),

BZ Ra, N8
BNZ Ra, N8

Estas dos instrucciones se van a poder codificar usando un sólo código de operación ya que con 16 bits se puede codificar toda la información necesaria:

- Código de operación (4 bits) que diferencia estas dos instrucciones del resto. Asignamos para ello el código 1000 (aunque podría ser otro).
- Campo de 8 bits, nnnnnnnn, denominado N8, para codificar el desplazamiento codificado en complemento a dos. En tiempo de ejecución, este campo, previa extensión del bit de signo, se multiplicará por 2 y se sumará a PC+2 para obtener la dirección de salto tomado (y así pasar a ejecutar la instrucción que se encuentra en la dirección PC+2+SE(N8)*2 de la memoria de instrucciones, si se ha de romper la secuencia).
- Campo de 3 bits, aaa, para codificar el registro Ra sobre el que hay que evaluar la condición de cero o de distinto de cero.

- El bit que sobra para ocupar los 16 de la instrucción lo usamos para diferenciar las dos instrucciones: 0 para BZ y 1 para BNZ. Este campo lo denominamos e (extensión de código de operación).

Estos campos ocupan justo los 16 bits de la instrucción. El formato de las instrucciones de salto, esto es, la división de los 16 bits de la instrucción en distintos campos y su significado, debe ser distinto del de 3-R usado antes, ya que requiere otros campos. Este formato, que denominamos de un registro (abreviadamente 1-R) se muestra en la figura 10.16.

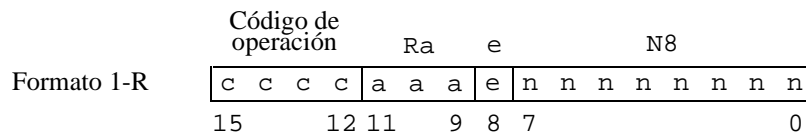


Fig. 10.16 Formato de las instrucciones de salto.

El campo del código de operación debe estar siempre en la misma posición: ya hemos decidido que serán los cuatro bits de la izquierda de la instrucción. Esto es así porque para decodificar la instrucción (extraer toda la información que define la instrucción) el hardware debe comenzar “mirando” el código de operación y cómo antes de esto no sabe nada de la instrucción debe mirar siempre en el mismo sitio. Después, según el código de operación, ya sabe qué campos tiene la instrucción (qué representa cada campo) y dónde se encuentra cada campo dentro de los 12 bits de más a la derecha de la instrucción.

La ordenación de los distintos campos dentro de la instrucción podría ser diferente a la utilizada aquí. El criterio usado se comentará más adelante, cuando se hayan definido las instrucciones y creado el circuito decodificador de instrucciones (que hemos denominado lógica de control).

La figura 10.17 muestra la codificación de las dos instrucciones de salto. La elección del código de operación 1000 es puramente estética y se observará mejor al final de la sección, cuando tengamos codificadas las 20 instrucciones de este capítulo. Dos ejemplos concretos de instrucciones de salto y su codificación se indican en la figura 10.18.

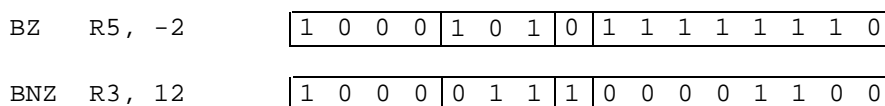


Fig. 10.17 Ejemplo de codificación de dos instrucciones de salto.

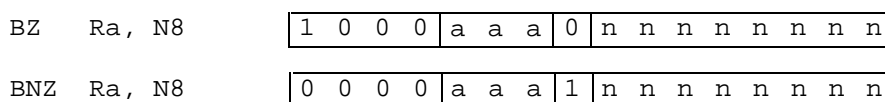


Fig. 10.18 Formato y codificación de las instrucciones de salto.

Instrucciones de entrada/salida (Formato de 1 registro)

Como hemos visto en el capítulo anterior, tenemos dos acciones de entrada salida que denotamos en mnemotécnicos como:

```
IN Rd, AddrPortIn
OUT AddrPortOut, Rb
```

Estas acciones necesitan un campo de 8 bits para codificar la dirección del puerto de entrada o de salida al que se accede, más 3 bits para codificar el registro destino (Rd) en caso de entrada o registro fuente (Rb) en caso de salida. Esto hace que podamos usar el formato 1-R con un solo código de operación para codificar las dos instrucciones (el bit *e* las diferenciará). La figura 10.19 muestra su codificación con el código de operación 1010. Asignamos el valor 0 del bit *e* para la instrucción IN y con valor 1 para la OUT. El campo de 8 bits *nnnnnnnn*, N8, codifica en binario la dirección del puerto del espacio de entrada en el caso de la instrucción IN, o del espacio de salida en el caso de OUT (ahora, para el código de operación 1010 el campo N8 codifica un número natural en binario, no un entero en complemento a dos, como es el caso del código de operación 1000).

IN	Rd N8	<table><tr><td>1</td><td>0</td><td>1</td><td>0</td><td>d</td><td>d</td><td>d</td><td>0</td><td>n</td><td>n</td><td>n</td><td>n</td><td>n</td><td>n</td><td>n</td><td>n</td></tr></table>	1	0	1	0	d	d	d	0	n	n	n	n	n	n	n	n
1	0	1	0	d	d	d	0	n	n	n	n	n	n	n	n			
OUT	N8, Ra	<table><tr><td>1</td><td>0</td><td>1</td><td>0</td><td>a</td><td>a</td><td>a</td><td>1</td><td>n</td><td>n</td><td>n</td><td>n</td><td>n</td><td>n</td><td>n</td><td>n</td></tr></table>	1	0	1	0	a	a	a	1	n	n	n	n	n	n	n	n
1	0	1	0	a	a	a	1	n	n	n	n	n	n	n	n			

Fig. 10.19 Formato y codificación de las instrucciones de entrada y salida.

Dos ejemplos concretos de instrucciones de entrada y de salida se indican en la figura 10.20.

IN	R1 2	<table><tr><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td></tr></table>	1	0	1	0	0	0	1	0	0	0	0	0	0	1	0
1	0	1	0	0	0	1	0	0	0	0	0	0	1	0			
OUT	1, R0	<table><tr><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td></tr></table>	1	0	1	0	0	0	0	1	0	0	0	0	0	0	1
1	0	1	0	0	0	0	1	0	0	0	0	0	0	1			

Fig. 10.20 Ejemplo de codificación de una instrucción de entrada y otra de salida.

Instrucciones con un operando inmediato

Las acciones aritmético-lógicas y de comparación con el segundo operando inmediato, que antes se ejecutaban en un ciclo, no se van a poder codificar en una instrucción de 16 bits: solo el campo N (los 16 bits del valor inmediato) ya ocupan toda la instrucción. Este es el caso de todas las acciones a las que les añadíamos una *I*, de inmediato, al final del mnemotécnico que indica la operación, por ejemplo: `ADDI R1, R1, 23`; `CMPEQI R7, R5, 0xFFAF...`

Hay dos soluciones a este problema. La primera consiste en definir la versión con un operando inmediato de las instrucciones aritmético-lógicas y de comparación pero con un campo N de menos bits. El problema es que sólo podremos tener valores inmediatos muy pequeños, pues sólo van a quedar 3 bits para este campo ya que tenemos que codificar el código de operación (4 bits), dos registros (6 bits) y la operación en el campo F (3 bits). Para esto no vale la pena. Así que, en general, en el juego de

instrucciones SISA no va a haber instrucciones aritmético-lógicas y de comparación con un operando inmediato. Solamente crearemos una instrucción, que llamaremos ADDI, en la que el segundo operando es un inmediato de 6 bits y que es útil para incrementar o decrementar el contador de un bucle y en general incrementar o decrementar en pocas unidades un registro. Esto lo haremos después de implementar la segunda solución al problema.

La segunda solución al problema, la más general, será

- cargar primero un registro con un valor inmediato (lo que requerirá, en principio, la ejecución de dos instrucciones de movimiento inmediato, una para los 8 bits de menor peso y la otra para los de más peso, como las que vamos a definir a continuación), y luego
- ejecutar la instrucción de operación aritmético lógica o de comparación con dos registros fuente y uno destino, como las que ya hemos definido.

Esto requerirá tres ciclos, tres instrucciones SISA, en vez de 1 ciclo, que es lo que se tardaba antes cuando especificábamos directamente la palabra de control.

a) Instrucciones de movimiento inmediato, **MOVI**, **MOVHI** (Formato de 1 registro)

Veamos cómo sustituimos la acción `MOVEI R1, N`, con N de 16 bits, por las dos nuevas instrucciones

```
MOVI  Rd, N8
MOVHI Rd, N8
```

Hemos quitado la E del mnemotécnico de la acción `MOVEI` para formar el mnemotécnico de la instrucción `MOVI` para que no sean iguales ya que la acción y la instrucción hacen cosas diferentes.

Denominamos N8 al campo de 8 bits que sustituye al antiguo campo de 16 bits en las nuevas instrucciones. Este campo puede ser de 8 bits ya que nos quedan 8 bits después de usar 4 para el código de operación, 3 para el registro Rd y aún tenemos 1 bit para diferenciar entre estas dos instrucciones. Vamos a codificar estas dos nuevas instrucciones usando el formato de 1-R, que tiene los campos que necesitamos, como se muestra en la figura 10.21. (se ha elegido el código de operación 1001 para las dos instrucciones con e=0 para `MOVI` y e=1 para `MOVHI`).

MOVI	Rd 0xN8	1	0	0	1	d	d	d	0	n	n	n	n	n	n	n
MOVHI	Rd, 0xN8	1	0	0	1	d	d	d	1	n	n	n	n	n	n	n

Fig. 10.21 Formato y codificación de las instrucciones `MOVI` y `MOVHI`.

Definimos ahora más precisamente qué hacen cada una de estas nuevas instrucciones al ejecutarse. Suponemos que lo más usual en nuestros programas es que queramos almacenar en un registro el valor de un número entero, Ns, codificado en complemento a dos con 16 bits. Si el valor Ns que queremos asignar a Rd está en el rango de los números enteros representables en complemento a dos en 8 bits, de -128 a 127, el problema es sencillo de resolver con una sola instrucción. Para ello, se codifica Ns con 8

bits en el campo N8 de la nueva instrucción `MOVI Rd, N8`, y en tiempo de ejecución se cogen los 8 bits del campo N8 de la instrucción, se extiende el bit de signo hasta crear un vector de 16 bits que representa a N_8 y al final del ciclo se escribe N_8 en R_d .

No obstante, si el valor que queremos almacenar en R_d requiere más de 8 bits para representarse en Ca_2 el problema es diferente. Lo que vamos a hacer es que la nueva instrucción `MOVHI` (*Move High Immediate*) escriba los 8 bits de su campo N8 en los 8 bits de más peso del registro destino, conservando inalterados los 8 bits de menor peso del registro destino. Así, para escribir el valor $0x3FA6$ en R_1 necesitaremos 2 ciclos, uno para ejecutar cada una de las dos instrucciones siguientes:

```
MOVI  R1, 0xA6
MOVHI R1, 0x3F
```

El precio que hemos pagado para conseguir instrucciones de 16 bits es que ahora necesitamos dos ciclos/instrucciones para asignar un número entero a un registro si el número está comprendido entre -2^{15} y $-(2^7+1)$ o de 2^7 a $2^{15}-1$, mientras que antes se hacía en un ciclo/instrucción. No obstante, para valores entre -2^7 a 2^7-1 se puede seguir haciendo en un ciclo. La figura 10.22 muestra la codificación de las instrucciones del ejemplo anterior a lenguaje máquina.

MOVI	R3 0xA6	<table><tr><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td></tr></table>	1	0	0	1	0	1	1	0	1	0	1	0	0	1	1	0
1	0	0	1	0	1	1	0	1	0	1	0	0	1	1	0			
MOVHI	R3, 0x3F	<table><tr><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td></tr></table>	1	0	0	1	0	1	1	1	0	0	1	1	1	1	1	1
1	0	0	1	0	1	1	1	0	0	1	1	1	1	1	1			

Fig. 10.22 Ejemplo de codificación de instrucciones `MOVI` y `MOVHI`.

A modo de resumen, se indica a continuación la semántica de las dos nuevas instrucciones de movimiento inmediato, lo que hacen al ejecutarse:

```
MOVI:      Rd ← SE(N8)
MOVHI:     Rd<15..8> ← N8
```

Tal como hemos definido la nueva instrucción `MOVI`, que extiende el bit de signo de N_8 , si queremos asignar a un registro un número natural podemos hacerlo en una sola instrucción `MOVI` solamente si el número se puede codificar en 7 bits, si está en el rango de 0 a 127. Si requiere 8 o más bits harán falta las dos instrucciones `MOVI` y `MOVHI`. Por ejemplo, para cargar en R_3 el número natural 129, que se codifica en 16 bits como: 0000000010000001 hay que ejecutar:

```
MOVI  R3, 0x81
MOVHI R3, 0x00
```

o lo que es lo mismo

```
MOVI  R3, 129
MOVHI R3, 0
```

Es importante notar que si no ejecutáramos la segunda instrucción, dado que la primera extiende el bit de más peso a los 8 bits de más peso del registro destino, habríamos cargado R3 con 111111110000001, y esto no es lo que queríamos.

Por último, hablemos de la implementación en la UPG de estas dos nuevas instrucciones. La instrucción **MOVI** se puede implementar con la UPG tal como está definida actualmente generando la palabra de control adecuada:

- el campo N de 16 bits debe ser el campo N8 de la instrucción previa extensión del bit de signo,
- Rb/N debe valer 0 para que el multiplexor deje pasar N a la entrada Y de la ALU,
- OP debe valer 10 y F=001 para que la ALU deje pasar la entrada Y a su salida,
- In/Alu debe valer 0 para llevar la salida de la ALU a la entrada D del banco de registros,
- WrD debe valer 1 y @D debe obtenerse del campo l<11..9> de la instrucción para que se escriba, al final del ciclo, el registro destino con el valor N y por último,
- el resto de bits de la palabra de control pueden valer 0 o 1 (pondremos x) excepto los de permiso para modificar los puertos de entrada/salida, por lo que Wr-Out = 0 y Rd-In = 0 (si Rd-In valiera x y se implementara como 1, al ejecutar la instrucción se podría poner a cero un registro de estado de algún controlador de un periférico de entrada).

Sin embargo, la instrucción **MOVHI** no se puede implementar directamente en la actual UPG. La implementación más directa requeriría implementarla como la **MOVI** pero que el campo N de la palabra de control contenga N8 en sus 8 bits de más peso y cambiando el banco de registros de la UPG para que, añadiendo una nueva señal de control que se active al ejecutar esta instrucción, sólo se escriban los 8 bits de más peso en el registro destino y queden inalterados los 8 de menor peso. No obstante, en la sección siguiente planteamos una pequeña ampliación de las funcionalidades de la ALU para poder ejecutar esta instrucción sin tener que modificar el banco de registros.

b) Instrucción de suma con un inmediato, ADDI. Formato de 2 registros

Además de las instrucciones que acabamos de ver para mover, en dos ciclos, un valor inmediato de 16 bits a un registro, vamos a crear ahora una única instrucción más en la que el segundo operando es un valor inmediato, que necesariamente tiene que ser de pocos bits, como vamos a ver. La instrucción suma un pequeño número entero codificado en la propia instrucción más el contenido de un registro fuente y escribe el resultado en un registro destino. La información a codificar en la instrucción y los bits del campo para codificarla es:

- código de operación de la instrucción (4 bits)
- registro fuente, Ra (3 bits) y
- registro destino, Rd (3 bits),
- valor inmediato en complemento a dos.

En esta instrucción la operación **ADD** está indicada implícitamente en el código de operación, ya que al ser la única operación con inmediato, no necesitamos el campo F de tres bits que tienen las instrucciones aritmético-lógicas y de comparación. Aun así, solo quedan 6 bits para codificar en complemento a dos el valor del número entero que se suma al registro fuente Ra para obtener el

resultado que se escribe en el registro destino Rd. Este campo lo denotamos como N6. Desafortunadamente, el rango de valores inmediatos que puede sumar la instrucción es pequeño: solamente de -32 a 31, que es el rango del complemento a dos con 6 bits.

Para codificar esta instrucción creamos un nuevo formato: de dos registros (que se muestra en la figura 10.23).

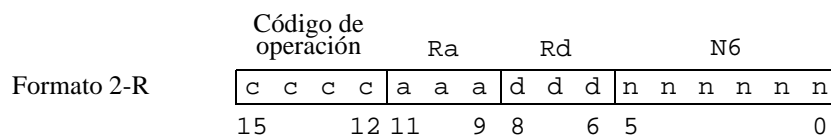


Fig. 10.23 Formato 2-R usado para la instrucción ADDI.

La sintaxis en ensamblador y la semántica de esta última instrucción es:

ADDI Rd, Ra, 0xN6

$Rd \leftarrow Ra + SE(N6)$

Para la codificación en lenguaje máquina usamos uno de los códigos de operación libres, en concreto el 0010. La figura 10.23, muestra el código en lenguaje máquina de una instrucción concreta de este tipo.

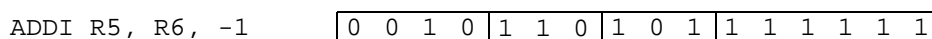


Fig. 10.24 Codificación de una instrucción ADDI.

10.4 Lenguaje máquina y ensamblador SISA

10.4.1 Resumen del formato y codificación de las instrucciones SISA

En la figura 10.25 se muestran los 3 formatos que hemos usado para codificar las instrucciones SISA con 16 bits; en la 10.26, las 20 instrucciones que acabamos de crear, ordenadas por código de operación. En la columna Name se indica el nombre genérico del tipo de instrucciones a las que les corresponde el código de operación de esa fila de la tabla. En las filas en las que a un código de operación le corresponden varias instrucciones, según el campo F de 3 bits, hemos ordenado estas según el código F sea 0, 1, 2,... (en la columna Mnemonic). Así, por ejemplo, la instrucción XOR tiene el código F = 010. Por último, en la columna Format se indica el formato de cada tipo de instrucción.

Hemos elegido los códigos de operación para que las instrucciones en esta tabla estén ordenadas según su formato (3-R, 2-R y 1-R) y además queden ordenadas por tipo de instrucción: aritmético-lógicas, de comparación, operación con inmediato, saltos (ruptura de secuencia), movimiento y entrada/salida. Por eso dijimos que el criterio de elección del código de operación para cada tipo de instrucción era

Formato 3-R	c c c c	a a a	b b b	d d d	f f f
Formato 2-R	c c c c	a a a	d d d	n n n n n n n	
Formato 1-R	c c c c	a a a d d d	e	n n n n n n n n	

Fig. 10.25 Formatos actuales de las instrucciones SISA.

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 Arithmetic Operations	3R
0 0 0 1	a a a	b b b	d d d	f f f		Compare Signed and Unsigned	
0 0 1 0	a a a	d d d	n n n n n n n			Add Immediate	2R
0 0 1 1							
0 1 0 0							
0 1 0 1							
0 1 1 0							
0 1 1 1							
1 0 0 0	a a a	0 1	n n n n n n n n			Branch on Zero Branch on Not Zero	1R
	d d d	0				Move Immediate	
1 0 0 1	a a a	1	n n n n n n n n			Move Immediate High	
	d d d						
1 0 1 0	d d d	0	n n n n n n n n			Input Output	
	a a a	1					
1 0 1 1	x x x x x x x x x x x x						
1 1 x x						Future extensions	





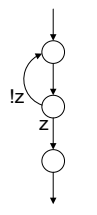
Fig. 10.26 Tabla resumen del formato y codificación de las 20 instrucciones SISA

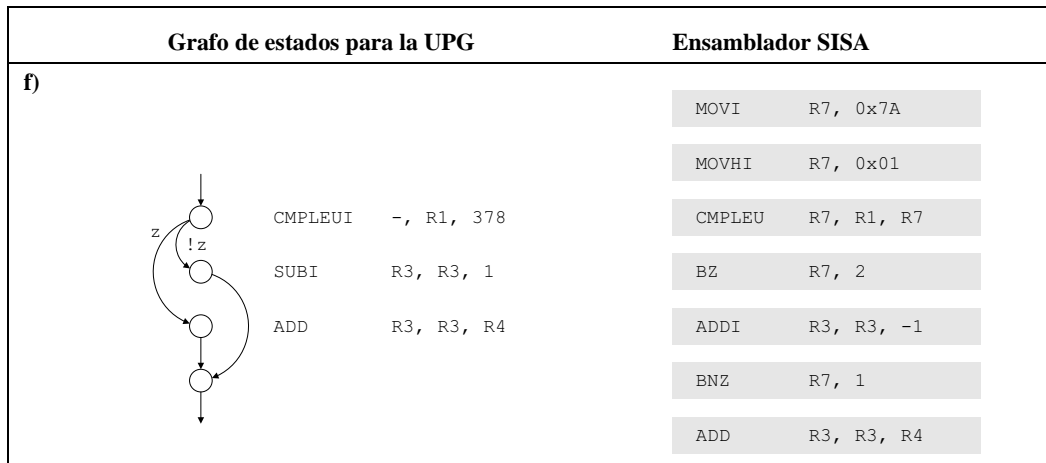
puramente estético: la estética de esta tabla. Se reservan cuatro códigos que se usarán en el siguiente capítulo para codificar las cuatro nuevas instrucciones de acceso a memoria, otro código para una nueva instrucción de ruptura de secuencia que se definirá en el capítulo 13 y los cinco códigos finales que se usarán para hacer ejercicios de añadir nuevas instrucciones al lenguaje SISA e implementarlas en los computadores que terminaremos de diseñar en los capítulos 12 y 13.

10.4.2 Del grafo de estados al programa SISA

La siguiente tabla muestra varios ejemplos de fragmentos de grafos que especifican la unidad de control (UC) de propósito específico de un PPE que tiene como unidad de proceso la UPG. A la derecha de cada grafo se muestra el fragmento de código en lenguaje ensamblador SISA para implementar la misma funcionalidad que el PPE, pero usando el procesador de propósito general que estamos construyendo (UC General con lenguaje SISA y la UPG).

La gran diferencia entre los dos sistemas es que con el procesador se puede implementar cualquier funcionalidad con solo cambiar el programa (es de propósito general) aunque se paga un coste: en muchos casos se tardan más ciclos para hacer lo mismo que con una UC específica. A continuación comentamos cada ejemplo. Cuando en SISA hace falta usar un registro que no se usa en el grafo (para almacenar información temporalmente) usaremos R7, suponiendo que no mantiene información válida.

Grafo de estados para la UPG	Ensamblador SISA
a)  ADD R3, R4, R5	ADD R3, R4, R5
b)  SHAI R4, R3, -3	MOVI R7, -3 SHA R4, R3, R7
c)  MOVEI R5, 0x3AF6	MOVI R5, 0xF6 MOVHI R5, 0x3A
d)  MOVEI R3, -16	MOVI R3, 0xF0
e)  ADD R0, R0, R1 SUBI R2, R2, 1 MOVE R6, R0	ADD R0, R0, R1 ADDI R2, R2, -1 BNZ R2, -3 ADDI R6, R0, 0



a. En este caso la acción del nodo del grafo se hace exactamente igual con una instrucción SISA.

b. Como la instrucción SHA no puede tener un operando inmediato hemos cargado la constante -3 en un registro de uso temporal, R7. Esta carga se puede hacer con una sola instrucción MOVI ya que -3 se puede codificar en complemento a 2 en 8 bits (en el campo N8 de la instrucción) y cuando se ejecute se extenderá el bit de signo de N8 para escribir en R7 el mismo valor, -3, con 16 bits en complemento a dos. La acción que en el grafo se ejecutaba en un ciclo en el procesador requiere dos ciclos.

c. Para cargar la constante 0x3AF6, que requiere 16 bits, en el registro R5 se necesitan las dos instrucciones de movimiento inmediato: primero la MOVI, que carga los 8 bits de menor peso (y extiende el bit de signo a los 16, aunque esto ahora no es útil) y después la MOVHI, que carga los 8 bits de más peso (0x3A) en el registro, dejando los 8 de menor peso (0xF6) inalterados).

d. En este caso solo hace falta la instrucción MOVI ya que -16 se puede codificar en complemento a dos en 8 bits (0xF0). La instrucción extiende el bit de signo de N8 y crea la constante -16 en 16 bits, 0xFFFF0, que carga en el registro destino.

e. El primer nodo se implementa con una instrucción similar a la acción del grafo. La acción del segundo nodo se implementa en un solo ciclo pero usando la instrucción ADDI con la constante -1 (que puede codificarse perfectamente en complemento a dos en 6 bits en el campo N6 de la instrucción. Pero lo que no puede hacerse en el SISA es romper la secuencia según el resultado de z en el mismo ciclo que se realiza la acción. Para ello hay que ejecutar al ciclo siguiente una instrucción de salto condicional según el contenido de R2, por lo que este segundo nodo del grafo requiere dos instrucciones SISA. El tercer nodo se implementa con una sola instrucción. Como no existe una instrucción SISA con la misma semántica que la acción MOVE hemos usado una instrucción que haga lo mismo mediante una operación que no modifica el valor del primer operando (sumando cero al registro R0 para escribirlo en R6). Otras instrucciones que podrían haberse usado para lo mismo son: AND R6, R0, R0; también OR R6, R0, R0 y algunas más.

f. El primer nodo de este grafo requiere cuatro instrucciones/ciclos por dos motivos. En primer lugar, hacen falta las dos instrucciones MOVI y MOVHI para guardar el valor 378, que no se puede codificar

en complemento a dos con 8 bits, en el registro temporal R7, ya que los operandos de la instrucción de comparación tienen que ser dos registros. En segundo lugar, hace falta almacenar el resultado de la comparación en un registro temporal (todas las instrucciones SISA de comparación escriben el resultado en un registro): usaremos R7 para ello, ya que no hay ningún problema de que sea fuente y destino en la misma instrucción. Además, como no se puede ejecutar a la vez una instrucción de comparación y una de salto, tenemos que implementar este primer nodo mediante dos instrucciones más, una que compara y deja el resultado en R7 y la siguiente que rompe la secuencia si el contenido de R7 es 0. Los dos nodos siguientes requieren una sola instrucción por nodo.

10.5 Modificación de la ALU de la UPG para implementar MOVHI

Vamos a realizar las modificaciones necesarias en el procesador (UCG+UPG) que estamos diseñando para poder ejecutar la nueva instrucción MOVHI. Primero recordemos la sintaxis en lenguaje ensamblador (mnemotécnicos) y lenguaje máquina (codificación) y la semántica de la instrucción (cómo modifica el estado del computador al ser ejecutada):

Ensamblador:	MOVHI	Rd, N8
Codificación:	1001	ddd 1 nnnnnnnn
Semántica:	Rd<15..8>	← N8

Dicho con palabras, como resultado de la ejecución de la instrucción se habrán modificado los 8 bits de más peso de Rd con los 8 bits del campo N8 de la instrucción, mientras que los 8 bits de menos peso de Rd quedarán inalterados.

En la sección 10.3.2 planteamos la posibilidad de implementar esta instrucción modificando el banco de registros para que se pueda elegir si se escriben los 16 bits del registro destino (caso general) o solo los 8 bits de más peso (en caso de ejecutar MOVHI). Ahora veamos una forma más sencilla de implementar esta instrucción que solo requiere introducir un pequeño cambio en la ALU para que pueda generar adecuadamente los 16 bits que se escribirán al final del ciclo en el registro destino. Los cambios en la ALU se muestran en la figura 10.28, 10.27, 10.29 y 10.30 muestran, respectivamente,

- la tabla de funciones de la ALU según OP y F que incluye la nueva funcionalidad MOVHI(X, Y),
- el bloque combinacional $W = \text{MOVHI}(X, Y)$, que no requiere ninguna puerta en su circuito interno. Este bloque compone los 16 bits de la salida W poniendo los 8 bits de menor peso de su entrada X en los 8 bits de menor peso de W y los 8 bits de menor peso de Y en los 8 de más peso de W; esto es:

$$W<7..0> = X<7..0>$$

$$W<15..8> = Y<7..0>$$

- la implementación interna de la ALU, que incluye al bloque MISC y
- la implementación interna de MISC, que incluye al nuevo bloque MOVHI(X, Y).

La UPG, con esta ALU, ya puede ejecutar todas las instrucciones SISA que hemos definido en este capítulo, es cuestión de que la unidad de control genere la palabra de control adecuada para cada instrucción. Veamos qué se debe hacer para ejecutar MOVHI Rd, N8:

Funcionalidad de la ALU						
F			OP			
b ₂	b ₁	b ₀	1 1	1 0	0 1	0 0
0	0	0	---	X	CMPLT (X, Y)	AND (X, Y)
0	0	1	---	Y	CMPLT (X, Y)	OR (X, Y)
0	1	0	---	MOVHI(X, Y)	---	XOR(X, Y)
0	1	1	---	---	CMPEQ (X, Y)	NOT (X)
1	0	0	---	---	CMPLTU (X, Y)	ADD (X, Y)
1	0	1	---	---	CMPLTU (X, Y)	SUB (X, Y)
1	1	0	---	---	---	SHA(X, Y)
1	1	1	---	---	---	SHL(X, Y)

Fig. 10.27 Funciones de la ALU según OP y F después de incluir MOVHI.

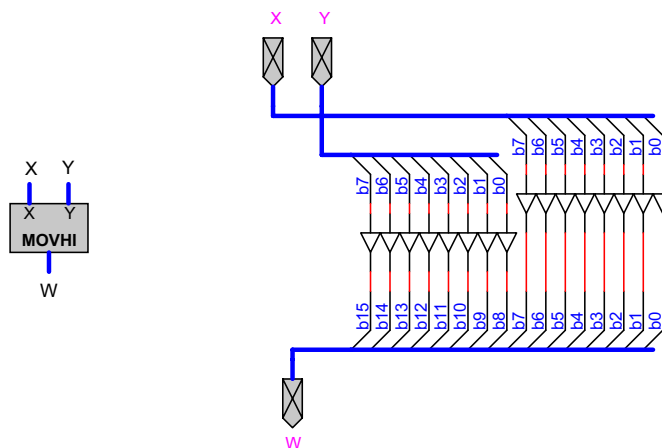


Fig. 10.28 Nuevo bloque W=MOVHI(X,Y) y su circuito interno.

- Leer por el bus A del banco de registros el registro destino como si fuera también registro fuente (para ello la Lógica de Control deberá generar el campo @A de la palabra de control con los tres bits ddd de la instrucción (campo l<11..9>)).
- Generar los 8 bits de menor peso del campo N de la palabra de control con el campo N8 de la instrucción (l<7..0>). El valor de los 8 bits de más peso de N (que tal como implementaremos en la siguiente sección la Lógica de control tendrán la extensión del bit de signo de N8).
- Hacer que la ALU haga la nueva función que le hemos añadido generando OP=10 y F=011 (codificación que antes no se usaba). Para que a la salida de la ALU aparezcan los 16 bits correctos que se escribirán en Rd la entrada X de la ALU debe tener el contenido de Rd antes de ejecutarse la instrucción (esto ya se ha indicado en el primer punto de esta lista) y en los 8 bits de menor peso de la entrada Y debe tener N8. Para esto segundo solamente hay que dejar pasar N a la entrada Y de la ALU haciendo que la señal de selección del multiplexor Rb/N valga 0 (ya que en los 8 bits de menor peso de N está el campo N8 de la instrucción, como se ha dicho en el segundo punto de esta lista).

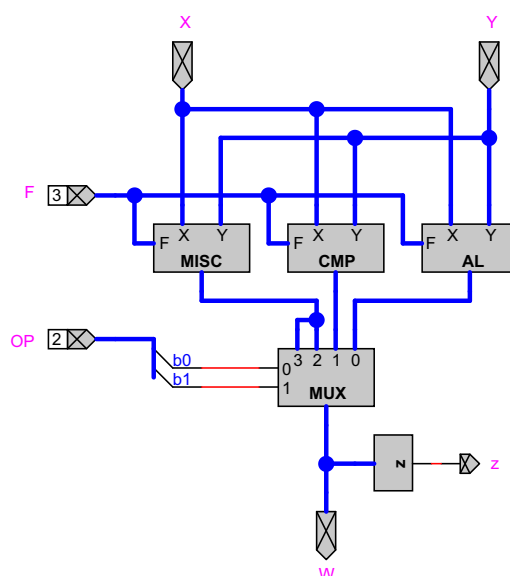


Fig. 10.29 Realización interna de la unidad aritmético-lógica (ALU)

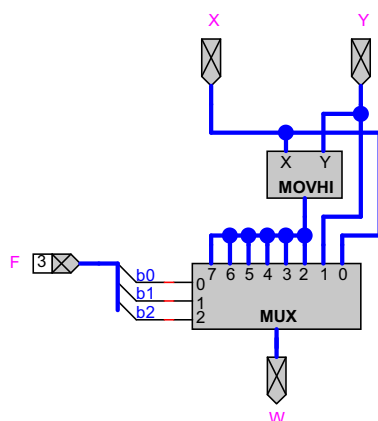


Fig. 10.30 Descripción, símbolo y realización interna del bloque MISC modificado para que contenga al nuevo bloque MOVHI

- Por último, se deberá dejar pasar la salida de la ALU al bus D del banco de registros (haciendo $In/Alu=0$) y ordenar la escritura del registro destino ($WrD=1$ y $@D=l<11..9>$).

Esta forma de implementar la instrucción MOVHI hace que el campo $l<11..9>$ (etiquetado como ddd o Rd) en realidad codifique también, además del registro destino, el registro fuente que se leerá por el bus A. Por ello decimos que esta instrucción tiene un **operando fuente implícito**, que es igual al destino: Rd actúa como fuente (sus 8 bits de menor peso) y como destino (sus 16 bits). Esto hay que tenerlo en cuenta cuando construyamos, en la sección siguiente, la lógica de control. Este es el único

caso en la arquitectura SISA en la que un operando es implícito (no se especifica explícitamente en un campo de la instrucción), aunque siendo rigurosos podemos decir que todas las instrucciones tienen un operando fuente y destino implícito, el registro PC, que se modifica en todas las instrucciones para apuntar a la siguiente instrucción a ejecutar ($PC=PC+2$), aunque esto solo se explicita al expresar la semántica de las instrucciones de salto.

La figura 10.31 muestra, a modo de ejemplo, la palabra de control concreta que debe generar la lógica de control para ejecutar la instrucción `MOVHI R1, 0xA5`. Es interesante notar que R1 se lee por el bus A como operando fuente y al final del ciclo se escribe desde el bus D como destino. También que los 8 bits de más peso del campo N pueden tener cualquier valor ya que el bloque `MOVHI` solo usa los 8 bits de menor peso de la entrada Y de la ALU por donde llega N.

@A			@B			Rb/N	OP	F			In/Alu	@D			WrD	Wr-Out	Rd-In	N (hexa)	ADDR-IO (hexa)				
0	0	1	x	x	x	0	1	0	0	1	0	0	0	1	1	0	0	X	X	A	5	X	X

Fig. 10.31 Palabra de control para ejecutar la instrucción `MOVHI R1, 0xA5`.

10.6 Implementación de la lógica de control con una ROM

Vamos a diseñar la Lógica de Control de la UCG que junto con la UPG forman el procesador de propósito general uniciclo para que pueda ejecutar las instrucciones SISA que hemos definido. La Lógica de Control (bloque `CONTROL LOGIC`) es un circuito combinacional y lo vamos a diseñar eficientemente usando unos pocos bloques combinacionales y una pequeña memoria ROM. Para efectuar el diseño necesitamos tener clara la siguiente información:

- Esquema a bloques del computador uniciclo que estamos construyendo (ver figura 10.32 en la siguiente página): el procesador (UCG+UPG) formado por la Unidad de Control General (que estamos diseñando en este capítulo) y la Unidad de Proceso General (que conocemos del capítulo 8) conectado al sistema de entrada/salida $IO_{key-Print}$ (diseñado al final del capítulo 9). La Unidad de Control General tiene la estructura de la figura 10.10 y, como se ve en la figura 10.32, está formada por el bloque `CONTROL LOGIC`, que vamos a diseñar ahora, la parte encargada del secuenciamiento de las instrucciones (bloques: PC, +2, ADD y MUX-2-1) y la memoria de instrucciones I-MEM (bloque `INSTRUCTION MEMORY`).
- Palabra de control de 44 bits que deberá generar la Lógica de Control que vamos a diseñar (ver figura 10.33 al pie de esta página). Para simplificar la nomenclatura, hemos incluido el bit `TknBr` dentro de la palabra de control. Este bit, que genera la lógica de control en cada ciclo, es el encargado del secuenciamiento de las instrucciones.
- La codificación de las instrucciones SISA que hemos definido en este capítulo (ver figura 10.26).
- La operación que realiza la ALU en función de los dos bits `OP` y de los tres bits `F` (ver figura 10.27).

A la lógica de control le llegan (ver figura 10.32)

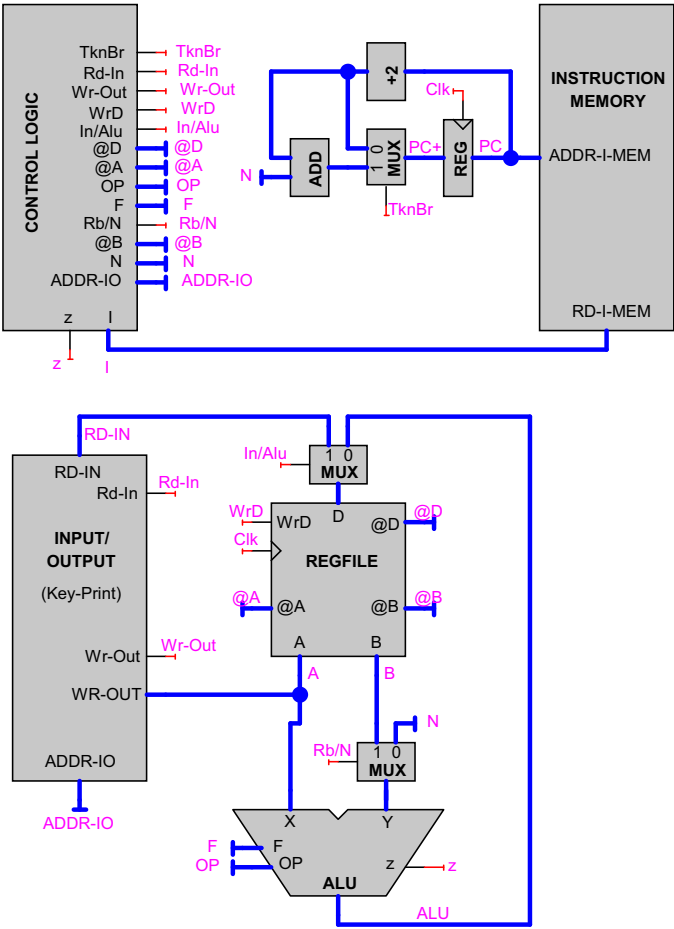


Fig. 10.32 Procesador de propósito general formado por la UCG+UPG conectado al sistema de entrada/salida IO_{Key-Print}

Palabra de Control de 44 bits

@A	@B	Rb/N	OP	F	In/Alu	@D	WrD	Wr-Out	Rd-In	TknBr	N (hexa)	ADDR-IO (hexa)

Fig. 10.33 Palabra de control a la que se le ha añadido la señal TknBr que sale de la lógica de control y va a la parte del secuenciamiento de la Unidad de Control General.

- los 16 bits de la instrucción que se está ejecutando en este ciclo, l, y
- el bit z que sale de la ALU.

Con estos 17 bits la lógica de control, que es un circuito combinacional, debe generar, en cada ciclo, los 44 bits de la palabra de control para gobernar al computador durante ese ciclo. Si construyéramos la lógica de control con una sola ROM necesitaríamos una de 2^{17} palabras de 44 bits por palabra. Este diseño no nos interesa, esta ROM sería casi 16 veces más grande que la memoria de instrucciones, que tiene 2^{16} palabras de 16 bits cada palabra. Todo el esfuerzo que hicimos al inicio del capítulo para reducir la memoria de instrucciones se perdería al diseñar la lógica de control de esta manera.

Vamos a realizar un diseño más eficiente que requiere una ROM mucho más pequeña y un poco de hardware adicional. Resulta que muchos de los bits de la palabra de control se pueden generar casi directamente de los bits de la instrucción, sin que los tenga que generar una gran ROM como la comentada en el párrafo anterior.

Así, el circuito que vamos a crear, cuyo resultado final se muestra en la figura 10.34, obtiene bits/campos de la palabra de control de la siguiente forma:

- ADDR-IO, @A y @B los obtiene directamente de los mismos bits/campos de la instrucción, con independencia de qué instrucción se esté ejecutando.
- OP, Rb/N, ln/Alu y WrD los genera directamente la pequeña ROM (denominada ROM-CTRL-LOGIC, o ROM de la lógica de control) en función de la instrucción que se está ejecutando en ese ciclo.
- @D, N y F los genera usando multiplexores para seleccionar de entre tres (caso de @D y N) o dos (caso de F) alternativas, dependiendo del tipo de instrucción.
- Wr-Out y Rd-In de momento los genera directamente la ROM en función de la instrucción pero como estos bits tienen requerimientos temporales especiales nos eremos obligados a poner una puerta en entre cada una de estas salidas de la ROM y las señales correspondientes de la palabra de control, como se verá en el capítulo 12. Por eso, de momento hemos denominado WrOut y RdIn (sin guión) a las señales que genera la ROM y hemos conectado cada una de ellas mediante un cable o una línea discontinua a los bits de la palabra de control Wr-Out y Rd-In para indicar que esa conexión no es definitiva (ver figura 10.34).
- TknBr, encargada del secuenciamiento, se obtiene en función de la señal z que llega de la UPG y Bz y Bnz que genera la ROM en función de la instrucción que se está ejecutando. Bz y Bnz valen 1 cuando se ejecuta la instrucción BZ o la BNZ respectivamente.

Veamos en detalle cómo se generan todos los bits de la palabra de control, de acuerdo con la clasificación que acabamos de hacer.

10.6.1 Bits generados directamente de la instrucción

Los 8 bits del bus ADDR-IO y los 3 bits de cada uno de los buses @A y @B de la palabra de control se generan directamente de los bits de la instrucción, sin necesidad de más hardware, como puede verse en el esquema de la lógica de control que estamos diseñando, figura 10.34.

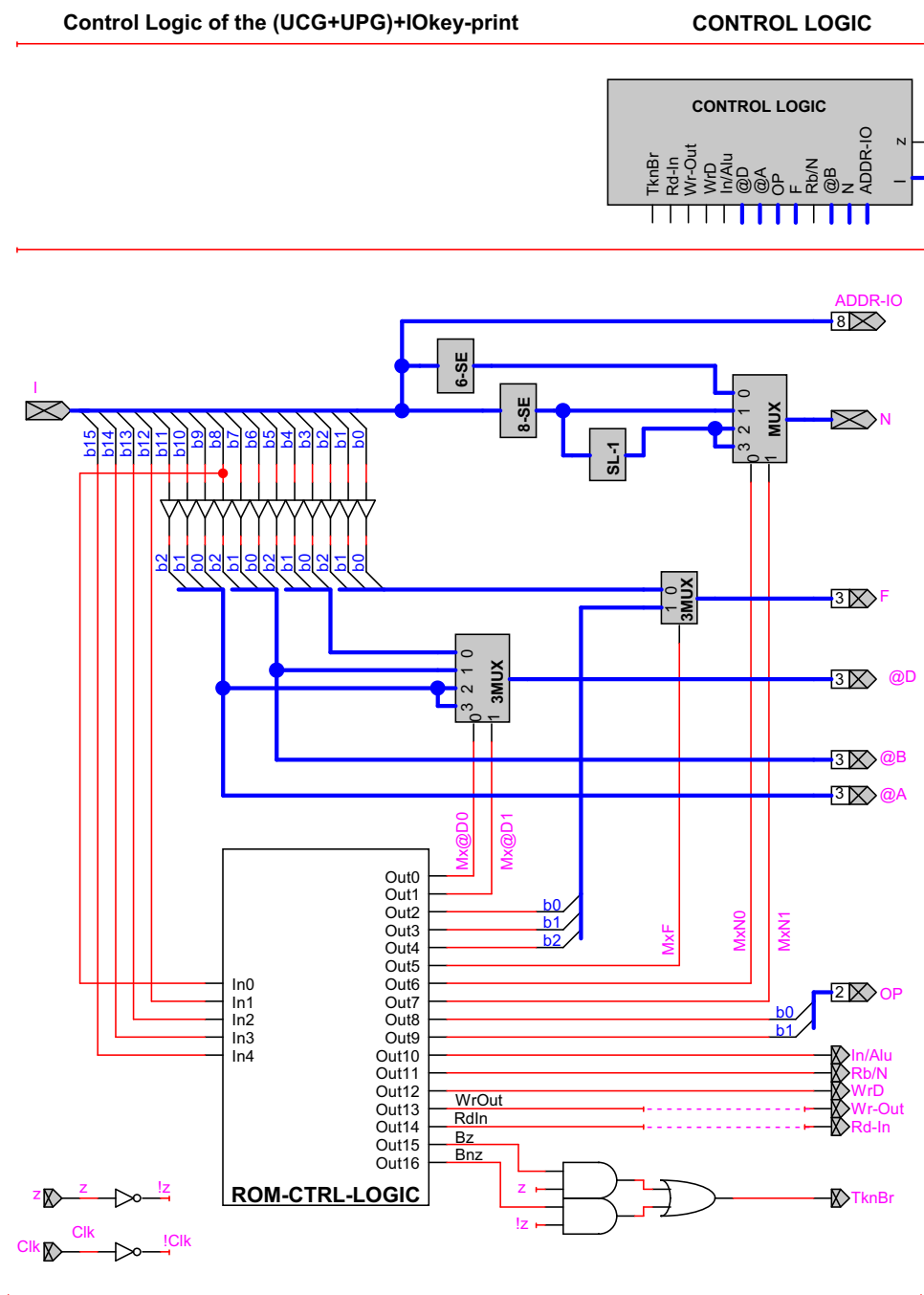


Fig. 10.34 Esquema de la lógica de control de la UCG+UPG+IOkey-print usando una ROM.

ADDR-IO

Los 8 bits del bus de direcciones de entrada/salida, ADDR-IO, se deben generar correctamente en caso de estar ejecutando una instrucción de entrada/salida, IN o OUT. En caso de ejecutar otras instrucciones no importa el valor de ADDR-IO. En la codificación de ambas instrucciones (ver figura 10.26), las dos instrucciones destinan los 8 bits de menor peso de la instrucción a codificar el puerto de entrada/salida que se debe utilizar. Por lo tanto, el bus ADDR-IO se puede obtener directamente de los 8 bits de menor peso de la instrucción, I<7..0>. En el esquema de la figura 10.34 esto se indica con el número 8 en el conector del bus de salida etiquetado como ADDR-IO, al que le llega el bus I (de la instrucción completa): los conectores de buses que no llevan asociado ningún número quiere decir que son de 16 bits, por defecto. Este conector, que lleva un 8, quiere decir que solo conecta los 8 bits de menor peso del bus I, que es de 16 bits.

@A y @B

Los tres bits del bus (campo de la palabra de control) @A y los tres de @B se pueden obtener directamente del campo de la instrucción I<11..9> e I<8..6> respectivamente. Esto es así porque todas las instrucciones con un registro operando fuente, Ra, (que son todas excepto MOVI e IN) y todas con dos operandos fuente, Ra y Rb, (las aritmético-lógicas y las de comparación) tienen los tres bits aaa y los tres bbb siempre en los mismos campos de la instrucción: I<11..9> e I<8..6>, respectivamente (ver la tabla de codificación de las instrucciones, figura 10.26). Una mención especial requiere la instrucción MOVHI que como se ha dicho, tiene que leer por el bus A el registro destino ddd que esta codificado en el mismo campo que siempre está el registro fuente Ra: I<11..9>. Así que en esta instrucción también se puede obtener @A directamente de la instrucción. En la figura 10.34 se ve cómo se ha creado cada uno de los dos buses que van a los conectores de salida de tres bits @A y @B.

10.6.2 @D. Campo generado de entre tres campos de la instrucción

Los tres bits @D, que codifican el registro destino en la palabra de control, no pueden cogerse siempre del mismo campo de la instrucción sino que se deben seleccionar de entre tres campos, dependiendo de la instrucción:

- I<5..3> para las instrucciones aritmético-lógicas y de comparación.
- I<8..6> para la instrucción ADDI.
- I<11..9> para las instrucciones MOVI, MOVHI e IN.

Esta selección se hace mediante el 3MUX-4-1 (multiplexor de 4 buses de 3 bits cada bus) de la figura 10.34 cuya salida se conecta al conector de salida de tres bits @D. Las señales de selección de este multiplexor (Mx@D1, Mx@D0) las genera la ROM, que hace las funciones de decodificación del código de operación y del bit e de la instrucción, codificando en los dos bits Mx@D1 Mx@D0:

- 0 cuando se ejecute una instrucción aritmético lógica o de comparación.
- 1 cuando se ejecute la instrucción ADDI.
- 2 cuando lo haga MOVI, MOVHI e IN.
- X, cualquier valor, cuando se ejecute cualquier otra instrucción distinta de las indicadas.

La tabla de la figura 10.35 muestra el contenido de la ROM completo en función de los bits de dirección (In4, In3..., In0), que son las entradas de la ROM. Para aumentar la claridad en la interpretación de la tabla, estos bits se han etiquetado también con el nombre de los bits de la instrucción que alimentan estas entradas de la ROM: I<15>, I<14>..., I<12> e I<8>. Los bits de más peso son el código de operación y el bit de menor peso es e, (el bit de extensión del código de operación).

La palabra de salida de la ROM, de 17 bits (Out16, Out15..., Out0) se ha etiquetado con los nombres de las señales: Bnz, Bz, RdIn, WrOut, WrD, In/Alu, Rb/N, OP1, OP0, MxN1, MxN0, MxF, F2, F1, F0, Mx@D1, Mx@D0.

Para especificar el contenido de la ROM no vamos a escribir una tabla de 32 filas, ya que muchas de estas filas consecutivas tienen el mismo contenido. Esto es así porque hemos puesto el bit e de la instrucción como bit de menor peso de la dirección de la ROM y las instrucciones con un mismo código de operación, que deben generar la misma palabra de control independientemente del bit e, aparecen en direcciones consecutivas de la ROM. Por ello, vamos a escribir una tabla con filas

Dirección					Contenido																
In4	In3	In2	In1	In0	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	RdIn	WrOut	WrD	Rb/N	In/Alu	OP1	OP0	MxN1	MxN0	MxF	F2	F1	F0	Mx@D1	Mx@D0
0	0	0	0	x	0	0	0	0	1	1	0	0	0	x	x	0	x	x	x	0	0
0	0	0	1	x	0	0	0	0	1	1	0	0	1	x	x	0	x	x	x	0	0
0	0	1	0	x	0	0	0	0	1	0	0	0	0	0	0	1	1	0	0	0	1
0	0	1	1	x																	
0	1	0	0	x																	
0	1	0	1	x																	
0	1	1	0	x																	
0	1	1	1	x																	
1	0	0	0	0	0	1	0	0	0	x	x	1	0	1	0	1	0	0	0	x	x
1	0	0	0	1	1	0	0	0	0	x	x	1	0	1	0	1	0	0	0	x	x
1	0	0	1	0	0	0	0	0	1	0	0	1	0	0	1	1	0	0	1	1	0
1	0	0	1	1	0	0	0	0	1	0	0	1	0	0	1	1	0	1	0	1	0
1	0	1	0	0	0	0	1	0	1	x	1	x	x	x	x	x	x	x	x	1	0
1	0	1	0	1	0	0	0	1	0	x	x	x	x	x	x	x	x	x	x	x	x
1	0	1	1	x																	
1	1	x	x	x	0	0	0	0	0	x	x	x	x	x	x	x	x	x	x	x	x

AL
CMP
ADDI
-
-
-
-
-
BZ
BNZ
MOVI
MOVHI
IN
OUT
-
NOP

Fig. 10.35 Tabla de verdad compacta del contenido de la ROM-CTRL-LOGIC de la lógica de control.

comprimidas. Por ejemplo, como el contenido de las direcciones 0, y 1 son el mismo, sólo usaremos una fila de la tabla para expresar el contenido de la ROM de esas 2 direcciones consecutivas. Para saber qué direcciones de memoria representa cada fila de la tabla, escribiremos en cada fila la dirección o direcciones en binario a la izquierda y el contenido a la derecha (ver figura 10.35). En el ejemplo anterior, las dos direcciones de la fila 0 se indican como 0000x (ya que el bit x puede valer 0 o 1). Además, a modo de comentario, a la derecha de la tabla se ha indicado el mnemotécnico de la instrucción que produce cada fila de la tabla o el mnemotécnico AL para el tipo de instrucciones aritmético-lógicas y CMP para las comparaciones.

De momento se debe observar solamente el valor las dos columnas que acabamos de justificar que corresponden a los dos bits de menor peso de la palabra de salida de la ROM (Out1, Out0). Para más claridad se han etiquetado los bits de dirección (de entrada de la ROM) con los bits de la instrucción a que están conectados y se han escrito los nombres de las señales para cada bit de salida de la ROM ya que resulta más claro, por ejemplo, Mx@D1 y Mx@D0 que Out1 y Out0.

La entrada 3 del MUX-4-1 no se usa (nunca valdrán 11 las señales de selección del multiplexor), pero por no dejar el bus de la entrada 3 al aire lo hemos conectado a la entrada 2. Esto daría como correcto poner 1x en la columna Mx@D para las filas en que hay que seleccionar la entrada 2 (filas MOVI, MOVHI e IN), pero preferimos que no se codifique nunca el 3 en estos dos bits, por lo que hemos puesto 10 en esas filas de la ROM (ver tabla en figura 10.35).

10.6.3 N. Generado de tres formas diferentes

Como se muestra en el esquema a bloques de la figura 10.34, los 16 bits N se pueden obtener de tres formas diferentes, dependiendo de la instrucción que se esté ejecutando. Por eso N es la salida de un MUX-4-1 gobernado por las señales de selección MxN1 y MxN0 generadas por la ROM. Para cada uno de los posibles cuatro valores codificados en MxN1 y MxN0 los 16 bits de N se forman:

0. Extendiendo el bit de más peso del campo l<5..0> de la instrucción, denominado N6, para que N6 y N codifiquen el mismo número natural en complemento a dos. Para ello la ROM genera un 0 en MxN para que el multiplexor lleve a su salida N la salida del bloque 6-SE (ver el circuito interno en la figura 10.36). Esto debe hacerse cuando se esté ejecutando la instrucción ADDI.
1. Extendiendo el bit de más peso del campo l<7..0>, denominado N8. El bloque encargado de ello es el 8-SE (ver su circuito interno en la figura 10.37) que es seleccionado por la ROM generando NxN=1 cuando se ejecuta la instrucción MOVI, MOVHI. Aunque en el caso de MOVHI no importan los 8 bits de más peso de N, vamos a replicar el bit de signo de N8 en estos bits ya que esto es sencillo y no es incorrecto.
2. Multiplicando por 2 el número entero codificado en Ca2 en el campo N8 de la instrucción. Para ello la ROM genera MxN=3 para que el MUX-4-1 lleve a N la salida del bloque SL-1 que multiplica por 2 la salida del bloque 8-SE que contiene codificado en 16 bits el campo N8 de la instrucción. Esto se hace cuando se está ejecutando una instrucción de ruptura de secuencia BZ o BNZ.
3. La entrada 3 del MUX-4-1 no se usa, y se ha conectado a la entrada 2 por no dejarla en el aire.

Cuando se ejecuta cualquier instrucción distinta a las mencionadas (ADDI, MOVI, MOVHI, BZ o BNZ) es indiferente lo que codifique el campo N de la palabra de control, por lo que la ROM generará MxN1=x y MxN0=x en estos casos, que de momento corresponden a las instrucciones de tipo AL, CMP,

IN y OUT. La figura 10.35 muestra el contenido de la ROM para las dos columnas MxN1 y MxN0 (Out7 y Out6) que acabamos de justificar. Cuando se ejecutan BZ o BNZ se ha codificado 10 en las señales de selección del MUX-4-1 en vez de 1x, porque no queremos que se use nunca la entrada 3.

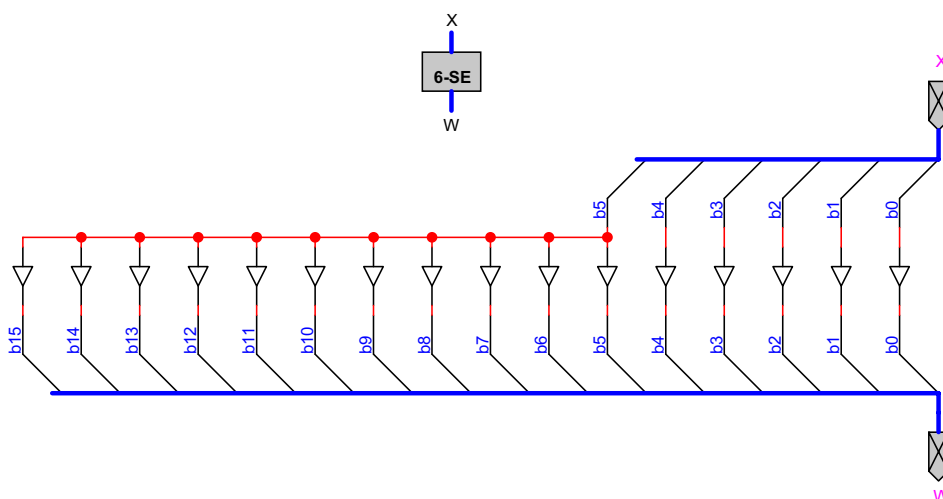


Fig. 10.36 bloque 6-SE y su implementación interna.

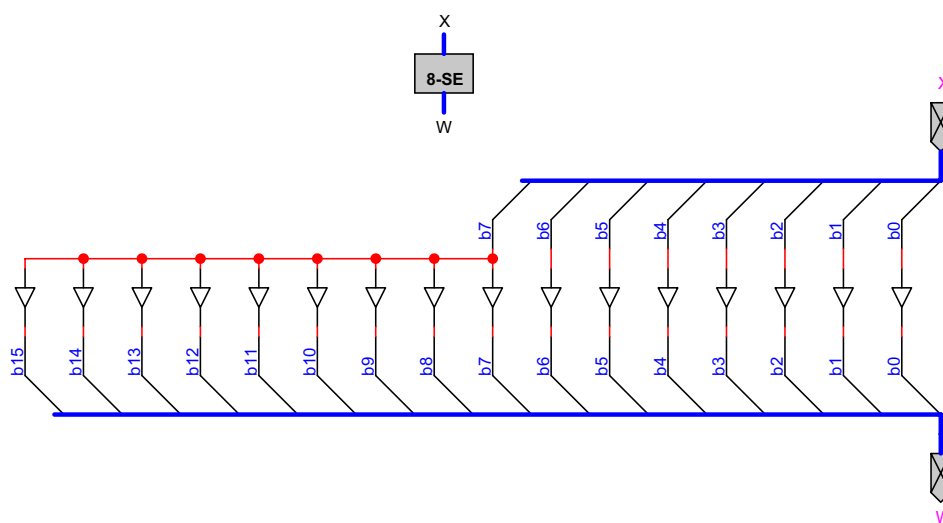


Fig. 10.37 Bloque 8-SE y su implementación interna.

10.6.4 F. Generado de entre dos posibilidades

El campo F de la palabra de control, junto con el campo OP, se usa para indicarle a la ALU la operación concreta que debe realizar en caso de ejecutarse una instrucción aritmético-lógica, incluida la ADDI (usando el bloque AL cuando OP=00), una instrucción de comparación (usando el bloque CMP cuando OP=01) o una de las instrucciones BZ, BNZ, MOVI o MOVHI (usando el bloque MISC cuando OP=10). Ver figura 10.27 y 10.29.

Como se muestra en la figura 10.34, el campo F se forma seleccionando, mediante un 3MUX-2-1, sus tres bits de dos procedencias posibles:

- del campo I<2..0> de la instrucción o
- de los bits F2, F1 y F0 de la salida la ROM.

El bit de selección del multiplexor, MxF, lo genera la ROM en su salida Out5. A continuación se explica el valor de las salidas de la ROM MxF, F2, F1 y F0 según la instrucción o tipo de instrucción que se esta ejecutando (ver figura 10.35):

- Instrucciones aritmético-lógicas (AL) y de comparación (CMP) (primeras dos filas de la tabla del contenido de la ROM). En estos dos casos el campo F se debe obtener del campo I<2..0> de la instrucción, como se ve en la tabla de codificación de las instrucciones, figura 10.26. Para ello MxF debe valer 0 y F2, F1 y F0 no importa, xxx, ya que estos bits no son seleccionados por el multiplexor.
- ADDI. La señal MxF debe valer 1 para generar el campo F con los tres bits F2, F1 y F0 de la ROM que deben valer 100 (y OP=00) para que la ALU realice la suma de sus dos entradas (ver figura 10.26).
- Instrucciones de salto BZ y BNZ. La ALU debe dejar pasar su entrada X (por donde llega el contenido de Ra) a su salida para generar el bit z en función del valor contenido en Ra. Para ello F debe valer 000 (y OP=10). Esto se consigue haciendo que la ROM genere MxF=1 y F2 F1F0 = 000.
- Instrucciones de movimiento MOVI y MOVHI. En ambas instrucciones MxF debe valer 1 para que la ALU haga lo que digan los tres bits F2, F1 y F0 de la ALU (además de que OP debe valer 10). En caso de la instrucción MOVI, el campo F2F1F0 debe valer 001 para que la ALU deje pasar su entrada Y, por la que llegará N (la extensión de signo de N8) para que se escriba al final del ciclo en el registro destino. En el caso de la instrucción MOVHI, F2F1F0 debe valer 010 para que la ALU haga la funcionalidad del bloque MOVHI.
- Resto de instrucciones (IN y OUT). En estos casos MxF y F2, F1 y F0 no importa el valor que tengan porque la ALU no importa lo que haga. Por eso ponemos x en estos bits y filas de la tabla de la ROM.

10.6.5 Resto de bits de la palabra de control generados directamente por la ROM

La ROM genera, además, el resto de bits de la palabra de control directamente:

- OP, Rb/N, In/Alu y WrD.
- los bits WrOut y RdIn (sin guión) y que, de momento, vamos a conectar directamente a los bits de la palabra de control Wr-Out y Rd-In. Cuando analicemos las restricciones temporales de estas señales

colocaremos una puerta entre cada una de estas salidas de la ROM y el conector binario de salida de la lógica de control.

- los bits Bz y Bnz que se activan cuando se ejecuta la instrucción BZ y BNZ respectivamente. Estas señales Bz y Bnz se usan para generar, en función de la señal de entrada z, la señal de la palabra de control TknBr, encargada del secuenciamiento. Cuando TknBr vale 1 se rompe la ejecución secuencial (ver el multiplexor que gestiona la carga del PC en la figura 10.32). TknBr ha de valer 1 cuando se ejecute BZ y z valga 1 o cuando se ejecute BNZ y z valga 0. La expresión lógica en función de los bits que genera la ROM es: $TknBr = Bz \cdot z + Bnz \cdot !z$. La figura 10.34 muestra el circuito que implementa esta expresión.

El valor de estos bits para cada fila de la tabla se muestra en la tabla de la figura 10.35 y su justificación se deja como ejercicio al lector.