13 Computador SISC Von Neumann

13.1 Introducción

En esta introducción vamos ha resumir las ventajas e inconvenientes de los dos computadores que hemos terminado de diseñar en el capítulo anterior y luego comentaremos las del nuevo computador que diseñaremos en este capítulo. Lo llamamos SISC Von Neumann o simplemente SISC, ya que es el último y definitivo que proponemos:

- SISC (*Simple Instruction Set Computer*) porque tiene el mismo lenguaje máquina/ensamblador SISA (*Simple Instruction Set Architecture*) que los dos anteriores excepto la nueva instrucción JALR, *Jump And Link Register*, para llamadas y retorno de subrutinas y saltos incondicionales a la dirección contenida en un registro. Con esta instrucción completamos definitivamente el lenguaje SISA, que con solo 25 instrucciones puede ser usado como lenguaje máquina objeto de cualquier programa escrito en un lenguaje de alto nivel, como el lenguaje C.
- Von Neumann porque así se denominan los computadores que tienen la memoria de instrucciones y de datos juntas en una única memoria (a diferencia de los dos anteriores que tenían estructura Harvard, con dos memorias especializadas, una para almacenar el programa y otra para los datos).

Sólo algunos computadores empotrados (*embedded*), de propósito único, por ejemplo para controlar un electrodoméstico o una parte de un automóvil, tienen estructura Harvard. Todos los computadores de propósito general, como los servidores de un centro de cálculo o los computadores personales, tienen estructura Von Neumann. Por eso no se suelen usar demasiado estos términos. Nosotros los usamos porque estamos llegando al computador de propósito general a partir de procesadores de propósito específico y queremos poner de manifiesto sus diferencias.

Aunque el computador SISC Von Neumann es multiciclo, no le añadimos la palabra multiciclo al nombre porque no puede ser uniciclo un computador Von Neumann: en un ciclo no se pueden hacer dos accesos diferentes a la única memoria existente, uno para leer la instrucción y otro para leer o escribir un dato, si se ejecuta una instrucción de acceso a memoria, a no ser que diseñemos una memoria bastante más compleja que lo permita (una memoria con buses de direcciones y datos independientes: dos buses de direcciones, uno para lectura y otro para escritura, dos buses de datos de lectura y uno de escritura). Pero esto no se suele hacer ya que no tiene ventajas que un computador Von Neumann sea uniciclo: todos son multiciclo.

13.1.1 SISC Harvard Uniciclo

Característica

Computador uniciclo. Todas y cada una de las instrucciones se ejecutan en un único ciclo de reloj (la señal de reloj es periódica: todos los ciclos tardan el mismo tiempo, Tc, y este tiempo es el de la instrucción que más tiempo requiere).

Ventajas

Sencillez de diseño. Por eso es el primer computador que hemos diseñado.

Inconvenientes

Baja eficiencia temporal. Todas las instrucciones tardan un ciclo en ejecutarse, por lo que el tiempo de ciclo no puede ser menor que el tiempo de la instrucción que más tiempo requiere para su ejecución correcta.

Baja eficiencia espacial. Se requieren más unidades funcionales de las estrictamente necesarias ya que no se puede reutilizar ninguna unidad para operar con diferentes datos y/o hacer funciones diferentes en un mismo ciclo. Por ejemplo, no puede haber una sola memoria para almacenar instrucciones y datos, se requieren dos diferentes (de aquí el nombre de computador Harvard).

Baja versatilidad. Al ser un computador uniciclo se genera una única palabra de control para ejecutar cada instrucción y por lo tanto no se puede usar más de una vez cada unidad multifunción durante la ejecución de cada instrucción. Por ello, en este computador no se puedan implementar instrucciones complejas, a no ser que se repliquen las unidades funcionales, lo que aumentaría el coste en hardware (por ejemplo, si para ejecutar una instrucción compleja hay que realizar 5 sumas, no se puede usar un único sumador en cinco momentos distintos, alimentándolo con distintos datos cada vez: hay que tener cinco sumadores distintos que se usarán en el ciclo que dura la ejecución de la instrucción).

Compromiso espacio-tiempo-versatilidad. En general, hay un compromiso entre la cantidad de hardware que tiene el computador (espacio) y el tiempo medio de ejecución por instrucción. Si el diseño es eficiente se puede reducir el tiempo de ejecución aumentando el hardware o reducir el hardware aumentando el tiempo de ejecución. Pero el diseño del SISC Harvard Uniciclo no es muy eficiente respecto a este compromiso espacio-temporal. Además, la versatilidad, importante en un sistema de propósito general, también está relacionada con el compromiso espacio-tiempo. El computador Harvard es poco versátil porque la memoria de instrucciones es una ROM (no existen instrucciones de lenguaje máquina para escribir esta memoria, cosa necesaria si se implementara con una RAM para cargar un nuevo programa desde un disco, por ejemplo. Para conseguir esta versatilidad habría que implementar esta nueva instrucción. Tampoco se pueden implementar, a no ser que se añadiera mucho más hardware, otras instrucciones más complejas.

Se pueden encontrar otros diseños con mejor eficiencia espacio-tiempo como el SISC Harvard multiciclo que reduce el tiempo de ejecución sin apenas incrementar el hardware necesario o el Von Neumann, que diseñaremos a continuación, que requiere bastante menos hardware, aunque más tiempo de ejecución. Además, y tal vez lo más importante, el computador Von Neumann tiene mucha más versatilidad que los dos que hemos diseñado en los capítulos anteriores.

Explicación

El registro PC se carga al inicio de cada ciclo con la dirección de la memoria de la instrucción que se ejecutará en ese ciclo. El flujo de ejecución de la instrucción comienza al estabilizarse la dirección a la salida del PC, luego se accede a la memoria de instrucciones y se obtiene la instrucción, después la lógica de control genera la palabra de control que indica a cada unidad funcional (banco de registros, ALU, memoria de datos, dispositivos de entrada salida) y a los multiplexores de encaminamiento lo que tienen que hacer en ese ciclo, se hace y finalmente, con la llegada del flanco ascendente del reloj que indica el final del ciclo actual y el inicio del siguiente, se modifica el estado del computador con el resultado de la ejecución de la instrucción (se escribe el PC con la dirección de la siguiente instrucción

a ejecutar, se escribe el registro destino en el banco de registros, si este es el caso, se escribe la memoria de datos, si la instrucción es un ST/STB o se escribe un puerto de salida, si la instrucciones un OUT.

- Baja eficiencia temporal.

  El tiempo de ciclo, que es el mismo para todos los ciclos sea cual sea la instrucción a ejecutar en cada ciclo, se debe calcular para que en un ciclo se pueda ejecutar cualquiera de las instrucciones. El flujo de ejecución de la instrucción `LDB Rd 0xN6(Ra)` es el que más tiempo requiere: la suma de los tiempos de

  1. estabilizar el PC,
  2. leer la instrucción de la memoria de instrucciones,
  3. generar la palabra de control, y en concreto el campo/bus N mediante la extensión del bit de signo del campo N6 de la instrucción.
  4. dejar pasar el bus N de la palabra de control a la entrada Y de la ALU
  5. calcular la dirección de la memoria de datos a leer, sumando en la ALU el registro Ra más el desplazamiento N,
  6. leer el byte de la memoria de datos y
  7. llevar el dato leído de la memoria al bus de entrada D del banco de registros y
  8. escribir el registro destino, Rd, del banco de registros.

  En esta secuencia de tiempos, los mayores son los de los puntos 1, 5 y 6. Todas las instrucciones tiene en común los puntos 1, 2 y 3. Otras tienen en común el uso de la ALU para sumar, restar o comparar, tardando un tiempo parecido al del punto 5. Pero el punto 6, que ocupa más de 800 u.t., sólo lo tienen las 4 instrucciones de acceso a memoria. Por eso, a las instrucciones de acceso a memoria las denominamos lentas y al resto rápidas. Las rápidas requieren del orden de 800 u.t. menos que las lentas para ejecutarse. No obstante, el tiempo que tarda cualquier instrucción en ejecutarse, que es el tiempo de ciclo, es el mismo tanto si se ejecuta una instrucción que podría ser "rápida" como si es "lenta". Esto produce una falta de eficiencia importante en el tiempo de ejecución de un programa.

- Baja eficiencia espacial.

  Cada unidad funcional y cada multiplexor de encaminamiento sólo se puede utilizar para una cosa durante la ejecución de cada instrucción, ya que la palabra de control es única para todo el ciclo, es única durante la ejecución completa de cadaa instrucción. Esto hace que si la ejecución de una instrucción requiere, por ejemplo, efectuar una resta y una suma, haya que disponer de un hardware diferenciado para sumar y otro para restar. Por ejemplo, la instrucción SUB usa la ALU para restar y el incrementador (+1) para sumar 1 al PC y calcular la dirección de la siguiente instrucción a ejecutar (de hecho, el sumador de +1 lo usan todas las instrucciones durante su ciclo de ejecución y la ALU también excepto BZ, BNZ, IN y OUT).

  Aunque la ALU puede sumar y restar, como no puede hacer las dos cosas en el mismo ciclo, hay que disponer de más unidades funcionales, más hardware, del estrictamente necesario. Esto produce una baja eficiencia espacial. Otro ejemplo, durante la ejecución de una instrucción LD se debe acceder a la memoria de instrucciones para obtener la instrucción y a la memoria de datos para obtener el dato. En una implementación uniciclo no puede haber una sola memoria (sencilla: con un solo bus de direcciones, como las que usamos aquí) donde estén las instrucciones y los datos ya que solo se puede acceder al contenido de una única dirección, para lectura si Wr-Mem vale 0 o para escritura si vale 1).

- Baja versatilidad. En general se puede ganar versatilidad a base de aumentar el hardware y/o en tiempo de ejecución. Al no poder reutilizar el hardware dentro de la ejecución de una instrucción, para aumentar la versatilidad de este computador se requiere un aumento considerable de hardware. Incluso hay instrucciones complejas que sería impracticable implementarlas en el Harvard uniciclo, como veremos en la siguiente sección.

13.1.2 SISC Harvard Multiciclo

Característica

Computador multiciclo realizado sin apenas cambios en el hardware respecto al uniciclo anterior. Las instrucciones "lentas", las de acceso a memoria de datos, se ejecutan en cuatro ciclos mientras que las "rápidas", el resto de instrucciones, se ejecutan en tres. El tiempo de ciclo se ha reducido a 750 u.t. (frente a los 3.000 del SISC Harvard Uniciclo)

Ventaja

Los programas se ejecutan entre un 0% a un 33,3% más rápidos en el multiciclo que en el uniciclo. Sin cambiar la UPG, sólo cambiando un poco la unidad de control, se consigue un tiempo medio de ejecución de una instrucción menor que en el SISC Harvard Uniciclo. Los cambios en la unidad de control son:

- el registro PC pasa a tener señal de permiso de carga y
- se añade un pequeño circuito secuencial para activar las señales de permiso de modificación del estado del computador solamente en el último ciclo de ejecución de cada instrucción.

Podemos decir que se mejora la eficiencia temporal del SISC Harvard Uniciclo sin empeorar la espacial (sin apenas aumento de hardware). El diseño sigue siendo sencillo.

Desventaja

No se ha mejorado la baja eficiencia espacial y baja versatilidad del SISC Harvard Uniciclo.

Explicación

En el SISC Harvard Multiciclo solo hay que modificar el estado del computador en el último ciclo de ejecución de cada instrucción: en el tercer ciclo cuando es una instrucción rápida y en el cuarto cuando es lenta (de acceso a memoria). Ahora, el PC no se debe cargar en cada ciclo, por lo que hay que usar como PC un registro con señal de permiso de carga, LdPc, y hacer que la lógica de control genere esta señal como parte de la palabra de control. Esta señal se deberá activar solamente en el último ciclo de ejecución de cada instrucción. Como la UPG es la misma que en el SISC Harvard Uniciclo, la UC también lo puede ser, excepto en la generación de las señales de permiso de modificación del estado (LdPc, WrD, Wr-Mem, Rd-In, Wr-Out) que las activa un pequeño circuito secuencial, a partir de las que genera la ROM del decodificador de instrucciones, que espera al tercer ciclo de ejecución de la instrucción o al segundo, según se trate de una instrucción rápida o lenta.

Las instrucciones “lentas” son las de acceso a memoria (LD, LDB, ST y STB) y las “rápidas” el resto. Dado que el tiempo medio de ejecución de una instrucción en el SISC Harvard Uniciclo es de 3000 u.t. mientras que en el SISC Harvard Multiciclo es de $3.000 - 750(1- r)$ o también $750(3+r)$), siendo r el ratio de instrucciones lentas ejecutadas respecto del total, podemos afirmar que:

- Cualquier programa se ejecuta más rápido en el SISC Harvard Multiciclo que en el SISC Harvard Uniciclo (solo si el 0% de las instrucciones ejecutadas son lentas, r=0, cosa que no se dará en ningún programa útil, la velocidad de ejecución de los dos computadores es la misma).
- El SISC Harvard Multiciclo es un $\frac{1-r}{3+r} \times 100$ % más rápido que el SISC Harvard Uniciclo. Este porcentaje va desde un 0%, cuando todas las instrucciones del programa son lentas (r=1), hasta el 33,3% cuando no hay ninguna lenta (r=0). Los valores extremos de r, cercanos al 0 y al 1, no se encuentran en ningún programa útil, siendo usual valores de r del 0,15 al 0,2 que suponen que el Harvard multiciclo es entre un 27 y un 23% más rápido que el Von Neumann (aunque es menos versátil).

13.2 Computador Von Neumann

13.2.1 Una visión de conjunto

Se denomina computador Von Neumann al que tiene una única memoria donde se almacenan tanto las instrucciones del programa que se ejecuta como los datos. Esto no se puede hacer con los computadores anteriores porque generan una única palabra de control para cada instrucción (ni en el caso del uniciclo ni en el del multiciclo).

Reutilización de hardware. Una palabra de control distinta en cada ciclo de ejecución de una instrucción

En el computador hay varias unidades multifunción, como son la memoria, que se puede leer o escribir en una dirección concreta, o la ALU, que puede realizar una de entre 16 funciones distintas. Para conseguir usar una misma unidad multifunción se pueda usar durante la ejecución de una instrucción para varias funciones diferentes es necesario que la instrucción se ejecute en varios ciclos y que en cada uno de ellos se genere una palabra de control diferente. De esta forma se puede cambiar la funcionalidad del hardware en cada ciclo. Lo mismo ocurriría si se tratara de usar más de una vez, durante la ejecución de una instrucción, una unidad que realizara siempre la misma función pero con distintos datos: en cada ciclo de reutilización de la unidad habría que cambiar la palabra de control para redirigir distintos datos fuente a la unidad y/o redirigir el resultado de la unidad a diferentes destinos.

Para ello, el circuito encargado de generar la palabra de control ya no puede ser un circuito combinacional (como en el computador Harvard uniciclo) ni un combinacional más un sencillo secuencial para activar las señales de escritura del estado en el último ciclo de ejecución (como en el Harvard multiciclo). El control, ahora, será un circuito secuencial que generará en cada ciclo la palabra de control completa, que podrá ser totalmente diferente de un ciclo a otro dentro de los que dure la ejecución de la instrucción.

Aumento de la eficiencia espacial. Ya que el computador Von Neumann puede reutilizar unidades funcionales, además de reutilizar la memoria para leer cada instrucción y para leer/escribir un dato en las instrucciones de acceso a memoria, vamos a reutilizar también la ALU para modificar PC, tanto en las instrucciones de salto como en el resto. Con esto nos ahorraremos, además de la memoria de instrucciones, el hardware incrementador y sumador encargados de secuenciar la ejecución de las instrucciones, lo que aumenta la eficiencia espacial de este computador frente a los dos anteriores.

Aumento de la versatilidad. El poder generar una palabra de control distinta en cada ciclo de ejecución de una instrucción, y con ello poder reutilizar las unidades funcionales con distintos datos/ resultados parciales y/o para distintas acciones (si son multifunción) en distintas fases de la ejecución de una instrucción, hace que con un hardware reducido se puedan implementar instrucciones complejas El único inconveniente de esto es que tardaran más ciclos en ejecutarse que las instrucciones más sencillas. Además, el hecho de tener las instrucciones en la misma memoria RAM que los datos hace que se puedan escribir, cargar, en la memoria distintos programas (con la misma instrucción que se escriben los datos: LD o LDB, a diferencia de los computadores Harvard anteriores en los que no había ninguna instrucción de escritura de la memoria de instrucciones, que por eso mismo era una ROM. Esto hace que este computador sea realmente de propósito general, mientras que los anteriores eran bastante específicos: cambiar el algoritmo requería cambiar la ROM del programa.

Fases de ejecución de una instrucción

Como acabamos de justificar, el computador Von Neumann que vamos a diseñar tiene que ser multiciclo y para generar la palabra de control de cada ciclo de la ejecución de una instrucción se necesita un circuito secuencial. Descomponemos la ejecución de una instrucción en una secuencia de fases y cada fase se realizará en un ciclo. En cada fase el circuito secuencial de control generará una palabra de control para que la unidad de proceso haga la función correspondiente. En general las fases son:

- Búsqueda de la instrucción. Se lee la instrucción de la memoria (la única que hay y que contiene el programa y los datos) y se escribe, al final de este ciclo, en un registro denominado IR (*Instruction Register*). Esta fase es la misma para cualquier instrucción ya que no puede ser distinta según la instrucción si todavía no se sabe qué instrucción se pretende ejecutar. En la implementación actual esta fase se realiza en un único ciclo, en un único estado del grafo, pero podría no ser así.
- Decodificación. La instrucción ya está en el IR, pero no se puede, en este ciclo, generar una palabra de control diferente según la instrucción porque la unidad de control es un circuito secuencial de Moore, como todos los que diseñamos a partir del grafo de estados. Así que en esta fase se tiene que hacer lo mismo siempre, independientemente de la instrucción que se encuentra en IR. Lo esencial de esta fase es decidir el estado siguiente, el nodo que se ejecutará al ciclo siguiente: ese estado siguiente sí que será diferente según la instrucción que haya en IR. En función del código de operación y del bit de extensión del código de operación de la instrucción que se encuentra en el IR se pasará a un estado concreto u otro de la fase de ejecución propiamente dicha. Esta fase también se implementa en un único ciclo, ya que no parece que pueda requerir varios si no se hace ninguna acción en la unidad de proceso.
- Ejecución y modificación del estado del computador (escritura del resultado en el registro destino, en la memoria o en un puerto del espacio se salida). Las acciones concretas en esta fase depende de la instrucción que se esté ejecutando, y dependiendo de que la instrucción sea más o menos compleja se implementa esta fase en uno o varios ciclos/estados, cada ciclo generando una palabra

de control diferente, que permite reutilizar el hardware dentro de la ejecución propiamente dicha de una instrucción.

El circuito secuencial encargado de implementar las palabras de control para ejecutar cada instrucción SISA de acuerdo a estas fases se especifica mediante el grafo de estados que se muestra en la figura 13.1 y que luego detallaremos en profundidad. Solamente cabe hacer énfasis ahora en que todas las instrucciones pasan por los dos primeros estados/ciclos de Búsqueda (*Fetch*), F, y Decodificación (*Decode*), D, y que después de D se pasa a ejecutar una secuencia de dos nodos/ciclos de ejecución para las instrucciones lentas (las de acceso a memoria) y un nodo/ciclo de ejecución para las rápidas (el resto de instrucciones). Cada nodo de la fase de ejecución propiamente dicha es diferente, dependiendo de la instrucción.

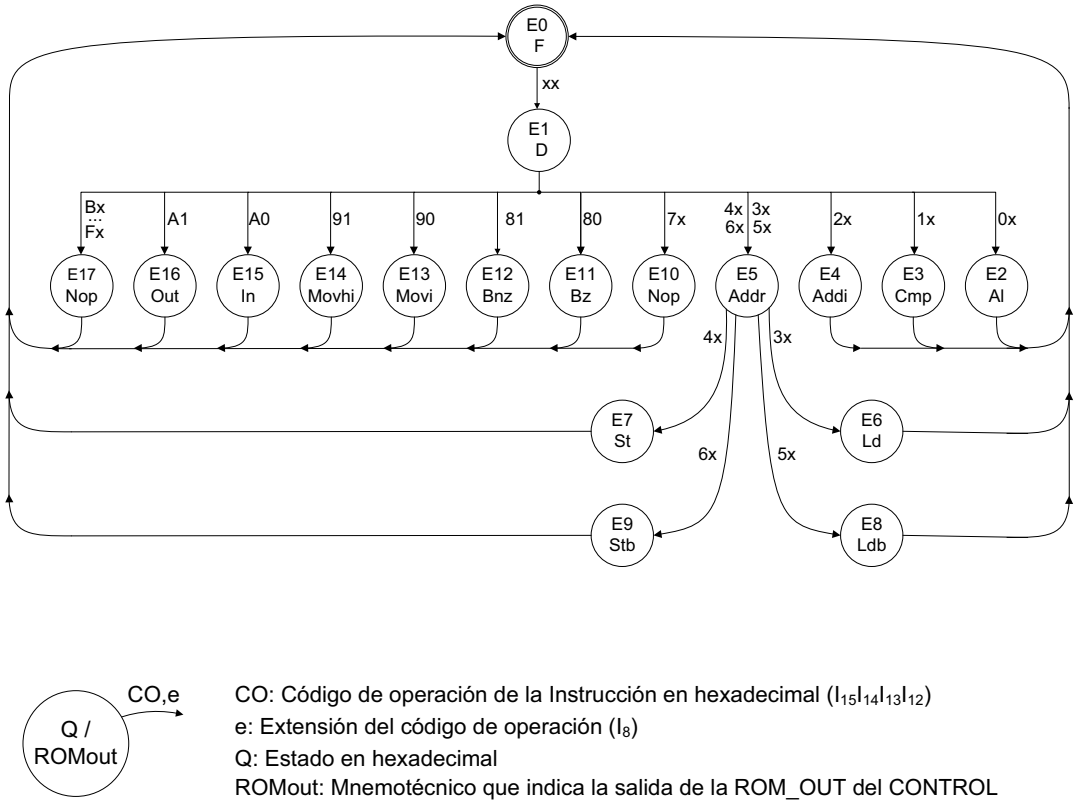


Fig. 13.1 Grafo de estados de la unidad de control del computador SISA (sin JALR).

Después de terminada la fase de ejecución de la instrucción en curso, después del último nodo/ciclo de ejecución de la instrucción actual, la que se acaba de terminar de ejecutar, se pasa siempre la nodo F, para ir a buscar la siguiente instrucción del programa. ¿En qué nodo se incrementa el PC (para todas las instrucciones) y se le suma además el desplazamiento N (para las instrucciones de salto que se debe

romper la secuencia) para que apunte a la siguiente instrucción a ejecutar. Esto debe hacerse durante la ejecución de la instrucción actual, antes de ir a buscar la siguiente instrucción.

Pero ahora el incremento (suma de una constante) del PC y la suma del desplazamiento, N, se tienen que hacer en la ALU (ya no hay hardware específico para ello, a diferencia de los computadores Harvard anteriores) lo que requiere usar la ALU en dos ciclos diferentes, uno para cada suma. Para no añadir más nodos/ciclos en la ejecución de cada instrucción estas dos sumas se hacen en dos nodos en los que, con lo dicho hasta ahora, no se usaba la ALU y por los que cualquier instrucción debe pasar durante su ejecución:

- *Fetch*, F. A la vez que se va a buscar la instrucción actual se incrementa el PC en la ALU, dejándolo ya apuntando a la siguiente instrucción del programa. Si la instrucción actual no es de salto, después del ciclo F ya no debe modificarse más el PC, hasta que se vuelva otra vez a esta fase F cuando se inicie la búsqueda de la siguiente instrucción. Se está aprovechando este ciclo en el que la ALU no se usaba para nada para hacer un trabajo, incremento del PC, que debe hacerse en todas las instrucciones y que en esta máquina requeriría un ciclo extra de ejecución.
- *Decode*, D. A la vez que se decide cuál será el siguiente ciclo/nodo de ejecución en función de la instrucción, en la ALU se suma el PC, que ya se ha dejado apuntando a la siguiente instrucción después del ciclo anterior, más el desplazamiento, N. Pero el valor calculado no debe cargarse en el PC al final de este ciclo ya que solo debe hacerse esto si se trata de una instrucción de salto (y eso no se sabe hasta el siguiente ciclo en el que ya se ha decodificado la instrucción) y si el salto es tomado (lo que se sabrá cuando el ciclo de ejecución propiamente dicho de la instrucción de salto, Bz o Bnz, se deje pasar el contenido del registro Ra por la ALU y la unidad de control sepa si se debe romper la secuencia o no). Por ello, se pone un nuevo registro en la unidad de proceso, que denominamos R@, el cuál se carga al final de D con el valor calculado de la dirección de la instrucción de salto tomado, por si luego hay que usar esta dirección para cargarla en el PC antes de pasar al *Fetch* de la siguiente instrucción. Podemos decir que, aprovechando que la ALU no tiene que hacer nada en el ciclo D, se adelanta el cálculo de la dirección de salto tomado a antes de saber, ni siquiera, si se está ejecutando una instrucción de salto. Si no se hiciera así, las instrucciones de salto necesitarían dos ciclos para su ejecución propiamente dicha, ya que en los dos se necesita la ALU: uno para dejar pasar Ra y saber si es cero o distinto de cero y el otro para calcular la dirección del salto tomado.

La siguiente tabla resume qué nodo concreto del grafo se ejecuta después de la decodificación de la instrucción, después de D, para el caso de las instrucciones rápidas, que están un único ciclo en la fase de ejecución propiamente dicha.

Instrucción rápida	**Nodo de la fase de ejecución**
AND, OR, XOR, NOT, ADD, SUB, SHA, SHL	Al
CMPLT, CMPLE, CMPEQ, CMPLTU, CMPLEU	Cmp
ADDI	Addi
BZ	Bz
BNZ	Bnz
MOVI	Movi

Instrucción rápida	Nodo de la fase de ejecución
MOVHI	Movhi
IN	In
OUT	Out

Las instrucciones lentas, las cuatro instrucciones de acceso a memoria, están dos ciclos en la fase de ejecución. El primero de ellos es común para las cuatro instrucciones, es el ciclo/nodo en el que se calcula la dirección de memoria, Addr, ya que se calcula de la misma manera para las cuatro instrucciones: Ra+SE(N6).

	Nodos fase ejecución	
Instrucción	**Primero**	**Segundo**
LD	Addr	Ld
ST		St
LDB		Ldb
STB		Stb

En resumen, la especificación de este circuito secuencial se hace mediante un grafo de estados que, en función de la instrucción, pasa por unos estados u otros para ejecutar la instrucción en varios ciclos. En cada estado generará una palabra de control diferente.

13.2.2 Del computador Harvard al Von Neumann

La unidad de proceso o camino de datos

A partir del computador Harvard Multiciclo vamos a construir el camino de datos para poder ejecutar las mismas instrucciones SISA pero eliminando la memoria de instrucciones (ahora las instrucciones se irán a buscar a la memoria que antes era de solo datos) y eliminando también el bloque +2 y el sumador encargados de calcular la dirección de la siguiente instrucción a ejecutar (ahora estas operaciones se realizan en la ALU). El camino de datos resultante se muestra en la figura 13.2.

Direccionamiento de las instrucciones inalterado. Al juntar las dos memorias en una, o más exactamente, al desaparecer la memoria de instrucciones del computador Harvard y pasar esa funcionalidad a la memoria de datos, que ahora es la única memoria del computador y que es igual a la de datos del computador Harvard (RAM de 2^16 bytes, que puede accederse para lectura o escritura tanto a byte como a word, y en caso de acceso a word se hace alineado a dirección par en formato little endian) no hay que cambiar nada del secuenciamiento de las instrucciones ya que la I-MEM del SISC Harvard ya tenía la misma unidad de direccionamiento que su memoria de datos (aunque no se podía escribir ni acceder a byte, eso no afecta al secuenciamiento de las instrucciones).

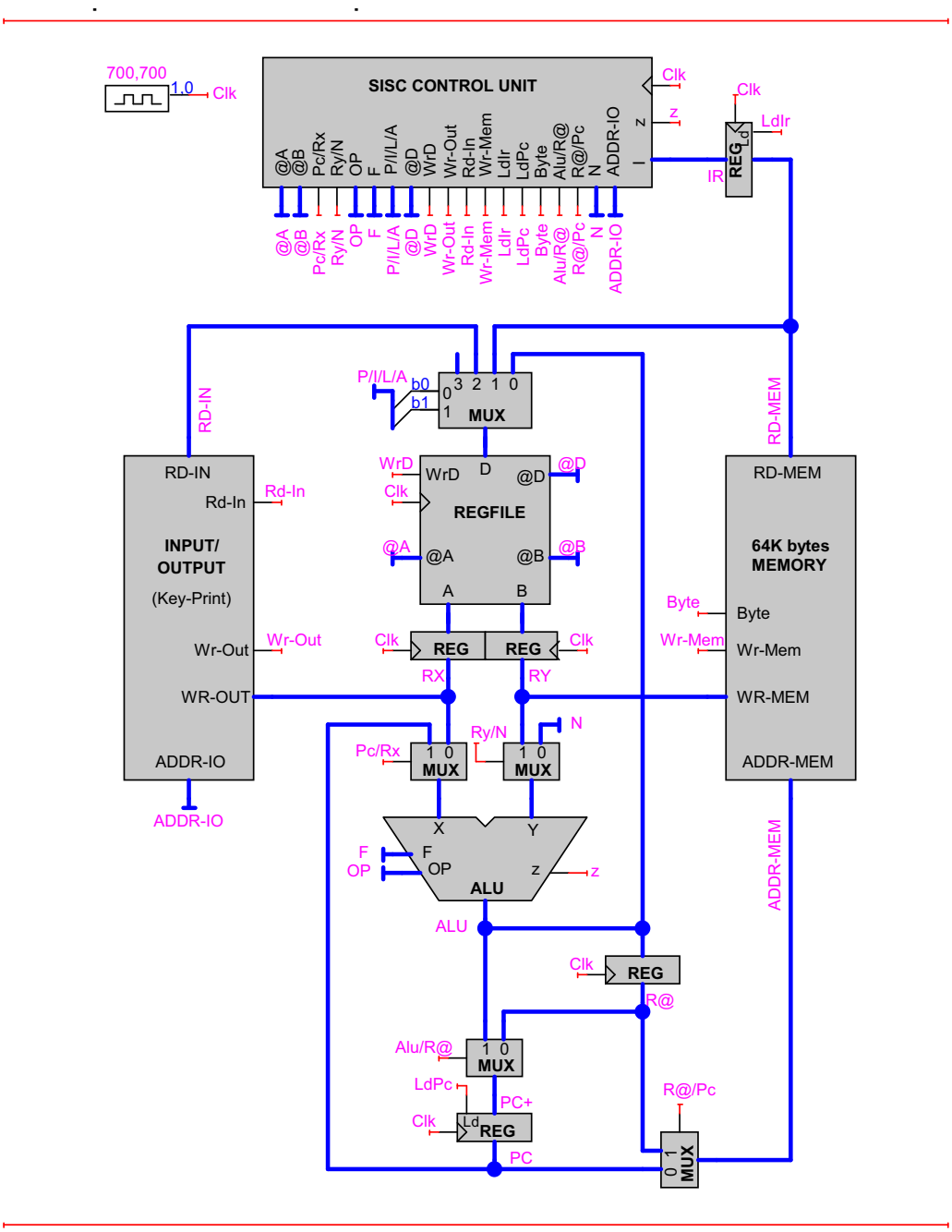


Fig. 13.2 Esquema detallado del computador SISC (sin JALR*)*

Nuevos registros en el camino de datos. Registros temporales. En este camino de datos aparecen algunos registros que en los computadores anteriores no estaban. Son registros para mantener los operandos o resultados parciales durante la ejecución de cada instrucción. Sirven para almacenar la información resultante de cada ciclo de ejecución de una instrucción que se debe usar en los siguientes ciclos. Estos registros no forman parte del estado del computador: el estado del computador es la memoria "visible" desde el lenguaje máquina del computador (la memoria que se puede acceder con una instrucción del lenguaje máquina) y lo forman los registros R0,..., R7, las posiciones de memoria y los puertos. Los nuevos registros son de uso temporal entre los ciclos de ejecución de cada instrucción y no importa el valor que tengan al inicio de la ejecución de cada instrucción, ya que no forman parte del estado del computador. Estos registros son:

- IR, denominado *Instrucction Register*, para almacenar la instrucción leída de la memoria al final del ciclo/estado F. Este registro tiene señal de permiso de escritura, LdIr, ya que se debe escribir el IR solo al final del ciclo F, pero debe mantener la instrucción en IR el resto de ciclos de ejecución ya que el código de operación y el bit e de la instrucción en curso sirven para decidir qué nodos del grafo se ejecutan y, viendo el grafo de la figura 13.1 se ve que en algunos estados, además de en D, se usan estos bits para decidir el siguiente estado. En la implementación actual del grafo esto ocurre en el estado Addr. No obstante, después del ultimo ciclo/nodo de ejecución de cada instrucción se pasa siempre a ejecutar el nodo F, por lo que en estos estados podría ponerse un 0 o un 1 en la señal LdIr.
- RX y RY para almacenar, después de la fase D, el contenido de los registros fuente de la instrucción, Ra y Rb, si es que la instrucción los requiere. En este computador, como se hizo en los anteriores, los campos de la palabra de control @A y @B se generan directamente de la instrucción sin necesidad de decodificar la instrucción, siempre de la misma manera: @A se forma con los tres bits del campo I<11..9> de la instrucción y @B con I<8..6>. Recuérdese, viendo la tabla de codificación y formato de las instrucciones de la figura 13.3 que si la instrucción tiene registro fuente Ra, este siempre se especifica en el mismo campo de la instrucción, I<11..9>, y que si tiene registro fuente Rb siempre está codificado en I<8..6>. (No ocurre lo mismo con el registro Rd: para generar @D la instrucción debe estar decodificada, ya que el campo de la instrucción de donde se obtienen los bits de @D depende de la instrucción, no siempre están en la misma posición dentro de la instrucción. Estos registros temporales, RX y RY, no tienen señal de permiso de carga por lo que se cargan al final de cada ciclo con el contenido de los registros codificados en los bits de la instrucción. Al final de D el contenido de estos registros ya es significativo y correcto para la ejecución de la instrucción y este valor se mantiene (se va cargando el mismo valor en cada ciclo) hasta que termina la ejecución de la instrucción). En F y en D el valor de salida de RX y RY no tiene sentido, pero tampoco va a usarse para nada en estos dos ciclos, sólo se usará en algunos de los ciclos de la fase de ejecución. Aprovechamos ahora para decir que en la fase D además de decodificarse la instrucción y calcularse de forma anticipada la dirección de salto tomado, se leen los registros fuente de la instrucción que se esta ejecutando.
- R@, Registro para almacenar el resultado de la fase Addr, la dirección de memoria obtenida sumando RX+N para las cuatro instrucciones de acceso a memoria. Esta dirección se usará al ciclo siguiente (en el nodo/ciclo Ld, Ldb, St o Stb, dependiendo de la instrucción que se esté ejecutando) para realizar el acceso a memoria.

Puede observarse que existe el camino adecuado para ejecutar todas y cada una de las instrucciones SISA que se ejecutaban en los anteriores computadores, solo es cuestión de que en cada ciclo de

16-bit Instruction																Mnemonic	Format
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
0	0	0	0	a	a	a	b	b	b	d	d	d	f	f	f	AND, OR, XOR, NOT, ADD, SUB, SHA, SHL	3R
0	0	0	1	a	a	a	b	b	b	d	d	d	f	f	f	CMPLT, CMPLE, -,CMPEQ, CMPLTU,CMPLEU, -, -	
0	0	1	0	a	a	a	d	d	d	n	n	n	n	n	n	ADDI	2R
0	0	1	1	a	a	a	d	d	d	n	n	n	n	n	n	LD	
0	1	0	0	a	a	a	b	b	b	n	n	n	n	n	n	ST	
0	1	0	1	a	a	a	d	d	d	n	n	n	n	n	n	LDB	
0	1	1	0	a	a	a	b	b	b	n	n	n	n	n	n	STB	
0	1	1	1	a	a	a	d	d	d	x	x	x	x	x	x	JALR	
1	0	0	0	a	a	a	0	n	n	n	n	n	n	n	n	BZ	1R
							1									BNZ	
1	0	0	1	d	d	d	0	n	n	n	n	n	n	n	n	MOVI	
							1									MOVHI	
1	0	1	0	d	d	d	0	n	n	n	n	n	n	n	n	IN	
				a	a	a	1									OUT	

Fig. 13.3 Formato y codificación de las instrucciones SISA (sin JALR).

ejecución la palabra de control genere las señales adecuadas de los multiplexores. Por ejemplo, para poder leer la instrucción que indica el PC la señal de selección del MUX-2-1 denominada R@/Pc debe valer 0 y 1 para conectar R@ al bus de direcciones ADDR-MEM en el último ciclo de las instrucciones de acceso a memoria. En breve veremos esto en más detalle

¿Qué es ahora la Unidad de Proceso y qué es la Unidad de Control?

En los computadores Harvard, como veníamos de los procesadores de propósito específico, denominábamos unidad de control general, UCG, al conjunto de las tres partes que resultaron de transformar la unidad de control específica en general:

- el secuenciador de instrucciones (formado por el PC, que mantiene el estado de la unidad de control, los bloques combinacionales +2 y ADD, para calcular la dirección de la siguiente instrucción a ejecutar y sus conexiones con el MUX-2-1 con señal de selección TknBr),
- la memoria de instrucciones, de donde se lee la instrucción almacenada en la dirección que indica el PC, y
- el bloque CONTROL LOGIC (en caso del uniciclo) o el bloque CONTROL (para el multiciclo) que genera la palabra de control a partir de la instrucción y de la señal z que viene de la ALU.

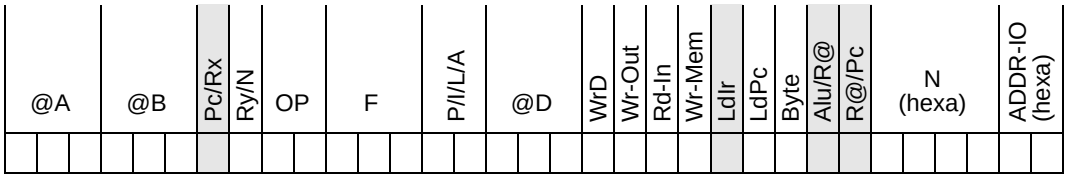
Ahora, estas dos primeras partes se han integrado en el camino de datos: en la unidad de proceso, en lo que antes denominábamos UPG, unidad de proceso general. Sólo ha quedado como Unidad de Control

el bloque encargado de generar la palabra de control a partir de la instrucción, que ahora se denominará unidad de control, CONTROL UNIT. Este bloque se construye con un circuito secuencial que implementa el grafo de estados encargado de ejecutar una instrucción: a partir de la instrucción y de z genera la palabra de control para efectuar en la unidad de proceso un nodo del grafo de ejecución de una instrucción y calcula su estado siguiente: el nodo siguiente dentro de las fases de ejecución de una instrucción. En el ciclo último de ejecución de cada instrucción el siguiente estado es siempre el F, para, así, ir a buscar a memoria la siguiente instrucción.

Desde este punto de vista, podríamos decir que la Unidad de Control del computador Von Neumann es de propósito específico: implementa siempre el mismo grafo, el de las fases de ejecución de una instrucción y después del último nodo de ejecución de la instrucción en curso (ciclo tercero si es una instrucción rápida o cuarto si es lenta) pasa al estado F, a la búsqueda de la siguiente instrucción, repitiendo este proceso indefinidamente. Parece que estamos volviendo al principio...

La Unidad de Proceso en el computador Von Neumann esta formada por la UPG y la parte de la Unidad de Control General de los computadores Harvard que se ha integrado en ella y que es la encargada del secuenciamiento de las instrucciones. El computador Von Neumann esta formado por el procesador (UC+UP), la memoria (que ahora contiene las instrucciones del programa y los datos) y el sistema de entrada/salida, que es como el de los computadores anteriores, con el teclado y la impresora.

Palabra de control. La palabra de control de 51 bits que genera en cada ciclo la unidad de control es la siguiente, donde se han sombreado los bits o campos (buses) nuevos:



Comparando esta palabra de control con la del computador Harvard Multiciclo, además de aparecer la nueva señal de carga del IR, y las señales de selección de los nuevos multiplexores (Pc/Rx, Alu/R@, R@/Pc) ha desaparecido la señal TknBr. Ahora, cuando se tenga que romper la secuencia de ejecución, en caso de instrucción de salto tomado, la función que antes hacía el TknBr ahora la hace la señal de permiso de escritura del PC, LdPc. que si vale 1 en el ciclo/estado Bz o Bnz, permitirá que se cargue el contenido de R@ (que contiene la dirección de salto tomado calculada en D, en el PC (para lo que, además, la señal Alu/R@ deberá valer 1).

13.3 Unidad de Control

La Unidad de Control genera, en cada ciclo, la palabra de control necesaria para implementar cada una de las fases de ejecución de una instrucción, de acuerdo con el grafo de la figura 13.1. En esta sección vamos a especificar la palabra de control que se debe generar en cada estado del grafo, sin tener en cuenta la implementación concreta de la Unidad de Control, que se verá en la siguiente sección.

La tabla de la figura 13.4 muestra para cada nodo/estado del grafo (para cada fila de la tabla (de la columna de la izquierda a la de la derecha):

- la codificación en decimal del estado,
- el mnemotécnico de la salida del nodo, que simboliza la palabra de control para ese estado/nodo,
- la acción que se realiza en el computador en ese ciclo, expresada en un lenguaje de transferencia de registros y
- la palabra de control asociada, expresada de la forma que hemos denominado "compacta". No se expresan explícitamente todos los 51 bits de la palabra de control, sólo los que son significativos, necesarios, para implementar correctamente la acción del nodo. Se asume que los bits que no se especifican valen x (se pueden implementar como 0 o como 1) excepto los bits de permiso de modificación del estado del computador (LdPc, Wr-Mem, Rd-In, Wr-Out y WrD) y el bit LdIr para los nodos D y Addr que valen 0. (Los registros RX, RY y R@ se cargan al final de cada ciclo, no tienen señal de permiso de escritura, por lo que no hace falta especificar esta acción en ningún caso.)

Tal como se ha implementado la memoria, en los accesos a word (Byte=0), tanto para lectura como para escritura, aunque la dirección en ADDR-MEM sea impar la memoria considera que el bit de menor peso de la dirección es 0. Por eso no hace falta poner a 0 por hardware el bit de menor peso de la dirección tal como indica la semántica de las instrucciones LD o ST: Mem_w[(Ra+SE(N6))&(~1)]. Por ello en el nodo Addr se calcula simplemente Ra+SE(N6). Además, la memoria, cuando Wr-Mem=1 y Byte=1 también se encarga de escribir solamente los 8 bits de menor peso de WR-MEM (de RY) en la dirección que indica ADDR-MEM, por lo que en el nodo Stb se hace la acción Mem_b[R@] ← RY que implementará correctamente la semántica de la instrucción STB, Mem_b[R@] = Rb<7..0> (teniendo en cuenta que en el ciclo anterior se escribió en R@ la dirección de memoria donde hay que escribir el byte).

A continuación se hacen comentarios sobre algunas señales de la palabra de control:

- **LdIr**. Vale 1 solamente en el estado de búsqueda, F, para escribir, al final del ciclo, en el registro temporal IR la instrucción en curso de ejecución. Vale 0 en los estados que requieran que la instrucción se mantenga en el registro IR durante el siguiente ciclo: estos son todos los estados excepto el último estado de ejecución de cada instrucción (estados D y Addr) y vale x en el resto de estados que son los últimos de la ejecución de la instrucción (ya que aunque al siguiente ciclo aparezca otra instrucción en IR eso no importa ya que en ese ciclo se está buscando la nueva instrucción, independientemente de lo que haya en IR).
- **R@/Pc**. Vale 1 solamente en los nodos en que se accede a memoria para leer o escribir un dato ya que en estos casos la dirección de memoria, que se calculo en la ALU el ciclo anterior (en el estado Addr, se escribió al final de ese ciclo en R@. Esto es, R@/Pc vale 1 en los estados Ld, St, Ldb y Stb. Vale 0 en los accesos a memoria en busca de una instrucción, ya que la dirección se debe coger del PC, y esto ocurre solamente en el estado F. La señal R@/Pc puede valer 0 o 1, esto es, vale x en el resto de estados.
- **Byte**. Vale 1 solamente en los accesos a memoria para leer o escribir un byte (Ldb y Stb), vale 0 en los accesos a word (en el estado F para buscar la instrucción y en Ld y St para acceder a un dato de tamaño word). Vale x en cualquier otro estado (ya que no se efectúan accesos a memoria).
- **LdPc, Wr-Mem, Rd-In, Wr-Out y WrD**. Son las señales de permiso de modificación del estado del computador y nunca pueden valer x, ya que podrían modificar el estado cuando no hay que hacerlo.

Estado		Acciones	Palabra de control compactada
0	F	Búsqueda de la Instr.: IR ← Mem_w[PC] // Incremento del PC: PC ← PC + 2	R@/Pc=0, Byte=0, LdIr=1, Pc/Rx=1, N=0x0002, Ry/N=0, OP=00, F=100, Alu/R@=1, LdPc=1.
1	D	Decodificación. Calculo @ salto tomado: R@ ← PC + SE(N8)*2 // Lectura de registros. (RX ← Ra) // (RY ← Rb)	La decodificación no requiere ninguna acción en la UPG por lo que se usa la UPG en este ciclo para adelantar trabajo que pueda ser útil. Este cálculo solo será util si la instrucción es BZ o BNZ. N=SE(IR<7..0>)*2, Pc/Rx=1, Ry/N=0, OP=00, F=100. Estas acciones se realizan sin tener que especificar nada en la palabra de control ya que RX y RY no tienen señal de permiso de escritura y @A y @B se generan directamente de los campos de bits del registro de instrucción IR<11..9> e IR<8..6>, respectivamente, en cada ciclo del grafo. Por ello las especificamos aquí entre paréntesis y ya no las especificaremos de ninguna forma en el resto de nodos.
2	Al	Rd ← RX Al RY	Pc/Rx=0, Ry/N=1, OP=00, F=IR<2..0>, P/I/L/A=00, WrD=1, @D=IR<5..3>.
3	Cmp	Rd ← RX Cmp RY	Pc/Rx=0, Ry/N=1, OP=01, F=IR<2..0>, P/I/L/A=00, WrD=1, @D=IR<5..3>.
4	Addi	Rd ← RX + SE(N6)	N=SE(IR<5..0>), Pc/Rx=0, Ry/N=0, OP=00, F=100, P/I/L/A=00, WrD=1, @D=IR<8..6>.
5	Addr	R@ ← RX + SE(N6)	N=SE(IR<5..0>), Pc/Rx=0, Ry/N=0, OP=00, F=100.
6	Ld	Rd ← Mem_w[R@]	R@/Pc=1, Byte=0, P/I/L/A=01, WrD=1, @D=IR<8..6>.
7	St	Mem_w[R@] ← RY	R@/Pc=1, Byte=0, Wr-Mem=1.
8	Ldb	Rd ← Mem_b[R@]	R@/Pc=1, Byte=1, P/I/L/A=01, WrD=1, @D=IR<8..6>.
9	Stb	Mem_b[R@] ← RY<7..0>	R@/Pc=1, Byte=1, Wr-Mem=1.
10	Jalr	PC ← RX&(~1) // Rd ← PC	Pc/Rx=0, OP=10, F=011, Alu/R@=1, LdPc=1, P/I/L/A=11, WrD=1, @D=IR<8..6>.
11	Bz	if (RX == 0) PC ← R@	Pc/Rx=0, OP=10, F=000, Alu/R@=0, LdPc=z.
12	Bnz	if (RX != 0) PC ← R@	Pc/Rx=0, OP=10, F=000, Alu/R@=0, LdPc=!z.
13	Movi	Rd ← SE(N8)	N=SE(IR<7..0>), Ry/N=0, OP=10, F=001, P/I/L/A=00, WrD=1, @D=IR<11..9>.
14	Movhi	Rd ← (N*(2^8)) | RX<7..0>	N=SE(IR<7..0>), Pc/Rx=0, Ry/N=0, OP=10, F=010, P/I/L/A=00, WrD=1, @D=IR<11..9>.
15	In	Rd ← Input[N8]	ADDR-IO=IR<7..0>, Rd-In=1, P/I/L/A=10, WrD=1, @D=IR<11..9>
16	Out	Output[N8] ← RX	ADDR-IO=IR<7..0>, Wr-Out=1.
17	Nop		
...	...		
31	Nop		

Fig. 13.4 Palabra de Control compacta para cada nodo del grafo de estados de la UC que implementa las fases de ejecución de cada instrucción.

A continuación se hacen comentarios sobre las tres fases de ejecución de una instrucción, que pueden ayudar a entender la tabla, aunque esto ya se ha comentado anteriormente:

- En el estado F (*Fetch*) se realizan dos acciones. Al final del ciclo se cargan dos registros: el IR con la instrucción que se encuentra en la memoria en la dirección que indica el registro PC y el PC con el valor que tenía en este ciclo incrementado en dos unidades (para que el PC apunte a la siguiente instrucción del programa almacenado en la memoria).
- En el estado D se realizan tres acciones:
  - Se decodifica la instrucción: se decide, en función de los bits del código de operación de la instrucción, IR<15..12>, y del bit de extensión del código de operación, IR<8>, cuál es el siguiente nodo del grafo. No obstante, en cada ciclo se calcula cuál será el nodo del ciclo siguiente: esta acción es propia del grafo de estados para todos sus ciclos. Se enfatiza aquí porque es el primer ciclo en el que se sabe cuál es la instrucción en curso y por ello se puede pasar a su ejecución al ciclo siguiente.
  - Como lo anterior no requiere el uso del camino de datos (UPG), se aprovecha este ciclo para adelantar trabajo de forma especulativa: se calcula la dirección destino de salto tomado, por si la instrucción en curso es BZ o BNZ, y se guarda al final del ciclo en el registro R@: R@<-PC+SE(N8). Es importante resaltar que esta acción no modifica el estado del computador ya que no se carga el PC, sino el registro R@ que no almacenaba información válida para la ejecución de la instrucción en curso: si la instrucción en curso resulta no ser de salto no hay que deshacer este trabajo, simplemente no se usará el contenido de R@ que se acaba de escribir. Hay que resaltar que el registro R@ se carga al final de cualquier ciclo ya que este registro no tiene señal de permiso de escritura, por lo que si la instrucción en curso es de salto y se cumple la condición de ruptura de secuencia, hay que usar al ciclo siguiente el contenido de R@ calculado en este ciclo y escribirlo en el PC.
  - Por último, en este ciclo se leen los registros fuente de la instrucción, Ra y Rb. Esto se puede hacer en este ciclo, en el que aun no se ha decodificado la instrucción, porque todas las instrucciones que tienen como operando fuente Ra codifican el valor a en el mismo campo de la instrucción: IR<11..9>. Pasa lo mismo para Rb, que siempre se codifica en el campo IR<8..6>. De hecho, la lectura de Ra y Rb se efectúa en todos los ciclos ya que los registros RX y RY no tienen señal de carga. Esta acción se enfatiza aquí porque es el primer ciclo en el que la nueva instrucción a ejecutar se encuentra ya en IR y el contenido del banco de registros que se carga en RX y RY es el de los registros que especifica la instrucción en curso. Si la instrucción no tiene registro fuente b (Rb) no pasa nada grave: lo cargado en RY no se utilizará al ciclo siguiente.
- El resto de nodos son los de la ejecución propiamente dicha de la instrucción en curso: según la instrucción de que se trate se ejecuta uno solo de estos nodos y si la instrucción es de acceso a memoria se ejecutan dos nodos, el Addr en primer lugar y en segundo lugar el Ld, St, Ldb o Stb según se esté ejecutando la instrucción LD, ST, LDB o STB, respectivamente.
- Al estado 17 (Nop) van a parar todas las instrucciones que no han sido definidas. En este estado no se modifica el estado del computador: todas las señales de la palabra de control pueden valer x excepto las siguientes que toman valor 0: LdPc, Wr-Mem, Rd-In, Wr-Out y WrD.
- Por último, después de la ejecución de la instrucción se pasa siempre al nodo 0, F, donde se irá a buscar la siguiente instrucción, repitiéndose el proceso de las fases de ejecución de una instrucción indefinidamente.

Así pues, cada instrucción tarda tres ciclos en ejecutarse, excepto las de acceso a memoria (LD, LDB, ST y STB) que tardan cuatro. Todas las instrucciones pasan por los mismos dos primeros ciclos/nodos: F y D. Después, cada instrucción requiere un nodo distinto para su ejecución, excepto en las instrucciones de acceso a memoria que requieren dos nodos, el primero de ellos, Addr, es el mismo para las cuatro instrucciones y el segundo y último diferencia su funcionalidad.

13.4 Esquema lógico interno de la Unidad de Control

La unidad de control, encargada de generar la palabra de control de cada ciclo de la ejecución de una instrucción, que implementa el grafo de la figura 13.1, no se construye haciendo que las entradas al circuito secuencial sean toda la instrucción más el bit z que sale de la ALU. Si se hiciera así la ROM del estado siguiente tendría 22 bits de entrada, de dirección (5 del estado actual, 16 de la instrucción y 1 de z) y 5 bits de salida (la ROM tendría 2^22 palabras de 5 bits cada una). No obstante, sabemos que solo los bits del código de operación y el bit e sirven para decidir el estado siguiente del grafo que implementa la unidad de control. Por eso la ROM que calcula el estado siguiente, Q+, solo tiene 10 bits de dirección (5 del estado actual, 4 del código de operación y 1 del bit e, de extensión del código de operación), lo que hace que tenga 2^10 palabras de 5 bits cada una.

El esquema lógico interno de la Unidad de Control, bloque SISC CONTROL UNIT, se muestra en la figura 13.5. Hay bastante similitud entre esta UC y la Lógica de control del SISC Harvard uniciclo, con la gran diferencia que antes era un circuito combinacional y ahora es un secuencial.

La ROM de las salidas (ROM_OUT), al igual que en los computadores Harvard, no genera toda la palabra de control directamente, solo algunas señales. El resto de señales de la palabra de control se forman directamente de campos de la instrucción y en algunos casos, para generar un campo de la palabra de control, la ROM_OUT genera señales de selección de multiplexores para elegir entre varios campos de la instrucción el que formará el campo de la palabra de control. El campo/bus N se genera manipulando campos de la instrucción (SE(N6), SE(N8), SE(N8)*2) o tomando un valor constante (0x0002) y seleccionando entre estas 4 alternativas mediante un MUX-4-1 cuyos dos bits de selección, MxN1 y MxN0, los genera ROM_OUT.

La señal LdPc de la palabra de control se crea a partir de las señales Bz y Bnz que genera ROM_OUT en función de la instrucción que se esté ejecutando y del valor del bit z que genera la ALU mediante las puertas para implementar la expresión LdPc = Bn·z + Bnz·!z. Por lo tanto, para implementar LdPc=1, en el nodo F para que al final del ciclo se guarde en el PC el PC+2, haremos que ROM_OUT genere Bnz=Bz=1. Para hacer LdPc=z, en el nodo Bz, haremos Bz=1 y Bnz=0, y para hacer LdPc=!z, en el nodo Bnz, haremos que ROM_OUT genere Bz=0 y Bnz=1.

Implementación de la ROM_OUT

La tabla compactada del contenido de la ROM_OUT es la siguiente. Hemos diseñado la UC para que, de forma similar a como hicimos en los SISC Harvard, cuando se ejecuta una instrucción con un código de operación que no coincide con ninguno del SISA se ejecuta una instrucción que denominamos NOP, no operación. Un ejercicio típico consiste en completar algunas filas y/o columnas de esta tabla.

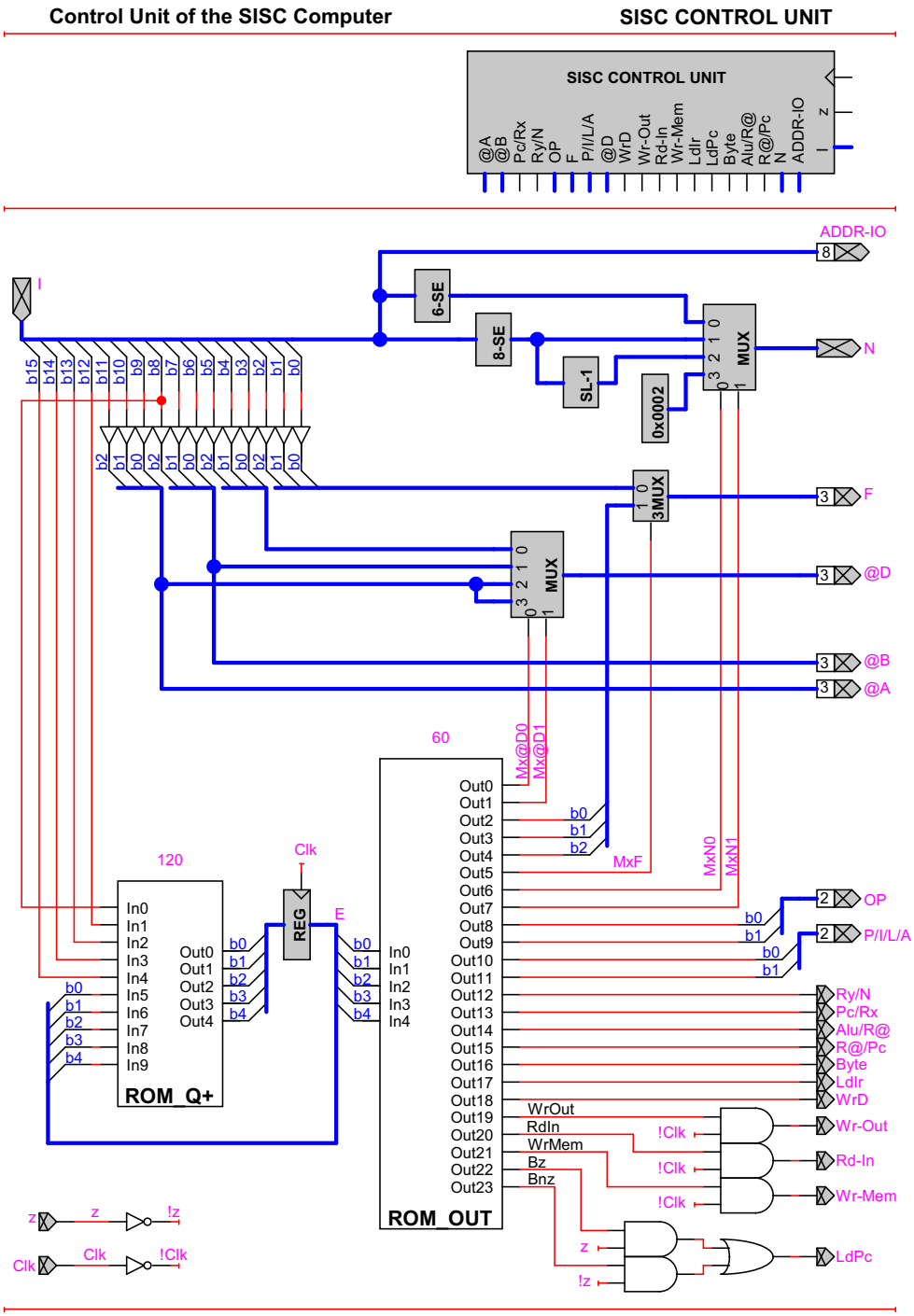


Fig. 13.5 Esquema interno del bloque SISC CONTROL UNIT.

@ROM	Bnz	Bz	WrMem	RdIn	WrOut	WrD	LdIr	Byte	R@/Pc	Alu/R@	Pc/Rx	Ry/N	P/I/L/A1	P/I/L/A0	OP1	OP0	MxN1	MxN0	MxF	F2	F1	F0	Mx@D1	Mx@D0	
0	1	1	0	0	0	0	1	0	0	1	1	0	x	x	0	0	1	1	1	1	0	0	x	x	F
1	0	0	0	0	0	0	0	x	x	x	1	0	x	x	0	0	1	0	1	1	0	0	x	x	D
2	0	0	0	0	0	1	x	x	x	x	0	1	0	0	0	0	x	x	0	x	x	x	0	0	AI
3	0	0	0	0	0	1	x	x	x	x	0	1	0	0	0	1	x	x	0	x	x	x	0	0	Cmp
4	0	0	0	0	0	1	x	x	x	x	0	0	0	0	0	0	0	0	1	1	0	0	0	1	Addi
5	0	0	0	0	0	0	0	x	x	x	0	0	x	x	0	0	0	0	1	1	0	0	x	x	Addr
6	0	0	0	0	0	1	x	0	1	x	x	x	0	1	x	x	x	x	x	x	x	x	0	1	Ld
7	0	0	1	0	0	0	x	0	1	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	St
8	0	0	0	0	0	1	x	1	1	x	x	x	0	1	x	x	x	x	x	x	x	x	0	1	Ldb
9	0	0	1	0	0	0	x	1	1	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	Stb
10	1	1	0	0	0	1	x	x	x	1	0	x	1	1	1	0	x	x	1	0	1	1	0	1	Jalr
11	0	1	0	0	0	0	x	x	x	0	0	x	x	x	1	0	x	x	1	0	0	0	x	x	Bz
12	1	0	0	0	0	0	x	x	x	0	0	x	x	x	1	0	x	x	1	0	0	0	x	x	Bnz
13	0	0	0	0	0	1	x	x	x	x	x	0	0	0	1	0	0	1	1	0	0	1	1	0	Movi
14	0	0	0	0	0	1	x	x	x	x	0	0	0	0	1	0	0	1	1	0	1	0	1	0	Movhi
15	0	0	0	1	0	1	x	x	x	x	x	x	1	0	x	x	x	x	x	x	x	x	1	0	In
16	0	0	0	0	1	0	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	Out
17..31	0	0	0	0	0	0	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	Nop

Tabla 13.1 Contenido completo de la ROM_OUT.

Implementación de la ROM del estado siguiente

Como la ROM_Q+ tiene 1024 entradas, vamos a dibujar su tabla de verdad compactando en una fila varias entradas. Y esto lo hacemos en tres pasos, en tres tablas. Primero creamos una tabla simbólica (ver tabla de la izquierda de la siguiente figura). Las dos primeras columnas representan los bits de entrada de la ROM, los bits de dirección: Q, de 5 bits, e I, de 5 bits (4 del código de operación y 1 de e). La tercera columna es la salida de la ROM, Q+ (de 5 bits).

La segunda tabla (centro de la figura) usa bits para las entradas y dígitos hexadecimales para las salidas en vez de nombres simbólicos.

La tercera tabla es una forma compacta de especificar el contenido de la ROM en hexadecimal muy cercana a como lo hace LogicWorks. La primera columna indica, para cada fila, el número de veces que se repiten los dos dígitos hexadecimales de salida de cada fila (que corresponden a contenidos de la ROM en palabras consecutivas). Por ejemplo, lo primero que hay en la ROM a partir de la dirección 0 es 0x01 repetido 32 veces, luego, en la dirección 32 hay un 0x02 y el la 33 lo mismo, etc. Un ejercicio típico consiste en preguntar qué contiene la ROM-OUT para una dirección concreta.

Q	I	Q^+	$q_4q_3q_2q_1q_0$	$I_{15}I_{14}I_{13}I_{12}I_8$	Q^+ (Hexa)	# veces	Q^+ (Hexa)
F	x	D	0 0 0 0 0	x x x x x	01	32	01
D	AL	Al	0 0 0 0 1	0 0 0 0 x	02	2	02
D	CMP	Cmp	0 0 0 0 1	0 0 0 1 x	03	2	03
D	ADDI	Addi	0 0 0 0 1	0 0 1 0 x	04	2	04
D	LD	Addr	0 0 0 0 1	0 0 1 1 x	05	2	05
D	ST	Addr	0 0 0 0 1	0 1 0 0 x	05	2	05
D	LDB	Addr	0 0 0 0 1	0 1 0 1 x	05	2	05
D	STB	Addr	0 0 0 0 1	0 1 1 0 x	05	2	05
D	JALR	Jalr	0 0 0 0 1	0 1 1 1 x	0A	2	0A
D	BZ	Bz	0 0 0 0 1	1 0 0 0 0	0B	1	0B
D	BNZ	Bnz	0 0 0 0 1	1 0 0 0 1	0C	1	0C
D	MOVI	Movi	0 0 0 0 1	1 0 0 1 0	0D	1	0D
D	MOVHI	Movhi	0 0 0 0 1	1 0 0 1 1	0E	1	0E
D	IN	In	0 0 0 0 1	1 0 1 0 0	0F	1	0F
D	OUT	Out	0 0 0 0 1	1 0 1 0 1	10	1	10
D	ilegal	Nop	0 0 0 0 1	1 0 1 1 x	11	2	11
			0 0 0 0 1	1 1 x x x	11	8	11
Al	x	F	0 0 0 1 0	x x x x x	00	32	00
Cmp	x	F	0 0 0 1 1	x x x x x	00	32	00
Addi	x	F	0 0 1 0 0	x x x x x	00	32	00
Addr	!(LD+ST+ LDB+STB)	x	0 0 1 0 1	0 0 0 0 x	xx	2	00
			0 0 1 0 1	0 0 0 1 x	xx	2	00
			0 0 1 0 1	0 0 1 0 x	xx	2	00
Addr	LD	Ld	0 0 1 0 1	0 0 1 1 x	06	2	06
Addr	ST	St	0 0 1 0 1	0 1 0 0 x	07	2	07
Addr	LDB	Ldb	0 0 1 0 1	0 1 0 1 x	08	2	08
Addr	STB	Stb	0 0 1 0 1	0 1 1 0 x	09	2	09
Addr	!(LD+ST+ LDB+STB)	x	0 0 1 0 1	0 1 1 1 x	xx	2	00
			0 0 1 0 1	1 x x x x	xx	16	00
Ld	x	F	0 0 1 1 0	x x x x x	00	32	00
St	x	F	0 0 1 1 1	x x x x x	00	32	00
Ldb	x	F	0 1 0 0 0	x x x x x	00	32	00
Stb	x	F	0 1 0 0 1	x x x x x	00	32	00
Jalr	x	F	0 1 0 1 0	x x x x x	00	32	00
Bz	x	F	0 1 0 1 1	x x x x x	00	32	00
Bnz	x	F	0 1 1 0 0	x x x x x	00	32	00
Movi	x	F	0 1 1 0 1	x x x x x	00	32	00
Movhi	x	F	0 1 1 1 0	x x x x x	00	32	00
In	x	F	0 1 1 1 1	x x x x x	00	32	00
Out	x	F	1 0 0 0 0	x x x x x	00	32	00

Tabla 13.2 Contenido de la ROM_Q+ en tres tablas con formatos diferentes pero la misma información

Q	I	Q^+
Nop	x	F

$q_4q_3q_2q_1q_0$	$I_{15}I_{14}I_{13}I_{12}I_8$	Q^+ (Hexa)
1 0 0 0 1	x x x x x	00
1 0 0 1 x	x x x x x	00
1 0 1 x x	x x x x x	00
1 1 x x x	x x x x x	00

# veces	Q^+ (Hexa)
32	00
64	00
128	00
256	00

Tabla 13.2 Contenido de la ROM_Q+ en tres tablas con formatos diferentes pero la misma información

Ejercicio 1

Cada uno de los apartados pregunta sobre un ciclo concreto de la ejecución de una instrucción en el SISC (No se trata de la ejecución ciclo a ciclo de un programa, sino de ciclos sueltos de instrucciones sueltas). La primera tabla define la situación en la que se encuentra la Unidad de Control (UC) en cada apartado indicando el nodo/estado de la UC en ese ciclo y la instrucción (en ensamblador) que está almacenada en el IR en ese ciclo. Por claridad, nos referimos a cada nodo/estado por el mnemotécnico de su salida (Así, el estado E0 lo denotaríamos como F).

Además, se aprovecha esta primera tabla para que podáis responder, para cada apartado, a la siguiente pregunta: **¿Cuál es el nodo/estado que se ejecutará al ciclo siguiente?** Para responder a esta pregunta podéis ver el grafo de estados de Moore del circuito secuencial de la UC en el anexo.

En la segunda tabla debéis **escribir el valor de los bits de la palabra de control** que genera el bloque SISC CONTROL UNIT durante el ciclo a que hace referencia cada apartado. Poned x siempre que no se pueda saber el valor de un bit (ya que no sabemos cómo se han implementado las x en la ROM_OUT). Suponed, para responder al apartado b, que el contenido de R4 antes de ejecutarse la instrucción `BNZ R4, -1` era 0xFFFF.

Apartado	Nodo/Estado (Mnemo Salida)	Instrucción en el IR (en ensamblador)	Nodo/Estado Siguiente (Mnemo Salida)
a	F	`(no se sabe)`	
b	D	`BNZ R4, -2`	
c	Addr	`STB -3(R1), R7`	
d	AI	`SUB R3, R1, R2`	
e	Movhi	`MOVHI R1, R2, R7`	
f	Bnz	`BNZ R4, -5`	

Apartado	@A	@B	Pc/Rx	Ry/N	OP	F	P/I/L/A	@D	WrD	Wr-Out	Rd-In	Wr-Mem	LdIr	LdPc	Byte	Alu/R@	R@/Pc	N (hexa)	ADDR-IO (hexa)
a																			
b																			
c																			
d																			
e																			
f																			

13.5 Añadiendo la nueva instrucción JALR del lenguaje SISA y al computador SISC

Vamos a añadir la nueva instrucción JALR, *Jump and Link Register*, al lenguaje SISA. Con esto ya tendremos definidas las 25 instrucciones del lenguaje completo SISA. Esta última instrucción permite implementar llamadas y retornos de subrutinas, así como saltos incondicionales a cualquier posición de memoria. No nos interesa aquí el uso de la instrucción en los programas, sólo decir que es imprescindible para poder compilar programas escritos en un lenguaje de alto nivel a lenguaje máquina SISA. Sin esta instrucción no se podrían implementar en SISA funciones y procedimientos, tan importantes en los lenguajes de alto nivel. Sin embargo, ahora, con solo 25 instrucciones, disponemos de un compilador de GNU que genera código SISA a partir de cualquier programa escrito en C, sin limitaciones. Otra cosa es que en SISA, por ejemplo, una multiplicación en coma flotante, que en otros lenguajes máquina se puede realizar con una sola instrucción, e implementarse en pocos ciclos computadores actuales, requiera muchas instrucciones SISA y muchos ciclos de ejecución en el SISC.

La sintaxis ensamblador, la semántica y el formato 2R y codificación de la nueva instrucción es la siguiente (al ser una instrucción de ruptura de secuencia, en la semántica especificamos también el incremento del PC):

Sintaxis ensamblador:	`JALR Rd, Ra`
Semántica:	`PC = PC+2; tmp=Ra&(~1); Rd=PC; PC=tmp;`
Lenguaje máquina:	`0111 aaa ddd xxxxxx`

En la figura 13.6 vemos el formato y la codificación definitiva de las 25 instrucciones SISA.

16-bit Instruction																Mnemonic	Format
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
0	0	0	0	a	a	a	b	b	b	d	d	d	f	f	f	AND, OR, XOR, NOT, ADD, SUB, SHA, SHL	3R
0	0	0	1	a	a	a	b	b	b	d	d	d	f	f	f	CMPLT, CMPLE, -,CMPEQ, CMPLTU,CMPLEU, -, -	
0	0	1	0	a	a	a	d	d	d	n	n	n	n	n	n	ADDI	2R
0	0	1	1	a	a	a	d	d	d	n	n	n	n	n	n	LD	
0	1	0	0	a	a	a	b	b	b	n	n	n	n	n	n	ST	
0	1	0	1	a	a	a	d	d	d	n	n	n	n	n	n	LDB	
0	1	1	0	a	a	a	b	b	b	n	n	n	n	n	n	STB	
0	1	1	1	a	a	a	d	d	d	x	x	x	x	x	x	JALR	
1	0	0	0	a	a	a	0	n	n	n	n	n	n	n	n	BZ	1R
							1									BNZ	
1	0	0	1	d	d	d	0	n	n	n	n	n	n	n	n	MOVI	
							1									MOVHI	
1	0	1	0	d	d	d	0	n	n	n	n	n	n	n	n	IN	
				a	a	a	1									OUT	

Fig. 13.6 Formato y codificación de las 25 instrucciones SISA.

13.5.1 Implementación de JALR en el SISC

Según se especifica en la semántica del JALR, el registro Ra contiene la dirección destino del salto, que se cargará en el PC. Como en SISA cada instrucción tiene 16 bits y debe almacenarse en memoria alineado a dirección par, como cualquier word, antes de cargarse en el PC se fuerza a 0 el bit de menor peso del contenido de Ra haciendo `Ra&(~1)`.

La actual ALU no puede realizar la acción R@ ← RX & (~1), que debería dejar pasar la entrada X forzando a 0 el bit de menor peso. Aunque tal como se ha implementado la memoria del SISC no pasaría nada si el PC quedará con una dirección impar, ya que al hacerse la búsqueda de la siguiente instrucción la memoria accedería al word con la dirección PC & (~1), vamos a añadir a la ALU la funcionalidad X & (~1) para poder implementar la instrucción JALR como indica su semántica.

Añadimos la nueva funcionalidad de la ALU, W= X & (~1), en el bloque MISC cuando OP=10 y F=011. Esto no requiere puertas lógicas, solo debe dejar pasar a la salida los 15 bits de mayor peso de la entrada y poner a 0 el bit de menor peso, el bit 0, de la salida. Esto es:

W<15..1> = X<15..1>

W<0> = 0.

Las figuras 13.7, 13.8 y 13.9 muestran, respectivamente, la tabla de funcionalidades definitiva de la ALU, el circuito interno del bloque MISC con la nueva funcionalidad X&(~1) y su circuito interno.

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	CMPLE (X, Y)	OR (X, Y)
0	1	0	---	MOVHI(X, Y)	---	XOR(X, Y)
0	1	1	---	X&(~1)	CMPEQ (X, Y)	NOT (X)
1	0	0	---	---	CMPLTU (X, Y)	ADD (X, Y)
1	0	1	---	---	CMPLEU (X, Y)	SUB (X, Y)
1	1	0	---	---	---	SHA(X, Y)
1	1	1	---	---	---	SHL(X, Y)

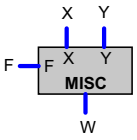
Fig. 13.7 Funciones de la ALU del SISC Von Neumann según OP y F

¿Cómo implementamos la JALR en el SISC, ahora que hemos modificado la ALU? La semántica se ha especificado con una secuencia de 4 sentencias, que se ejecutarían una después de otra, ya que el lenguaje de alto nivel que usamos para especificar la semántica de las instrucciones es secuencial (similar al C). De entrada podríamos pensar que la ejecución de esta instrucción requiere 5 ciclos: PC=PC+2 en F, Decodificación y 3 ciclos para cada una de las tres sentencias siguientes (usando R@ como variable temporal. Pero no tiene porque ser así. En nuestra UP se pueden escribir a final del ciclo más de un registro a la vez. En la figura 13.10 se indican las acciones a realizar en cada uno de los 4 ciclos de ejecución de la instrucción aprovechando que en un ciclo se puede cargar en paralelo Rd con el PC y el PC con R@.

16 Bit-wide Miscellaneous Operators **MISC**

Description:
The 16-bit output W is the function chosen by the
3-bit selection input F, applied to the 16-bit inputs X and Y.
The functions are:

F	W
000	X
001	Y
010	MOVHI(X, Y)
011	X&(~1)
100	---
101	---
110	---
111	---



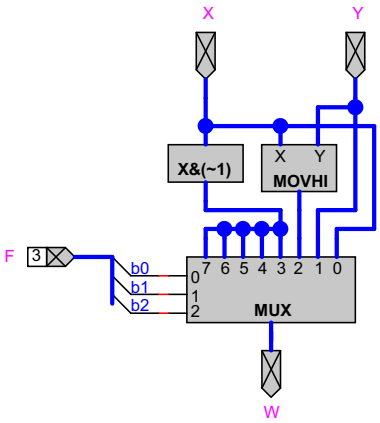


Fig. 13.8 Descripción, símbolo y realización interna del bloque MISC de la ALU del SISC

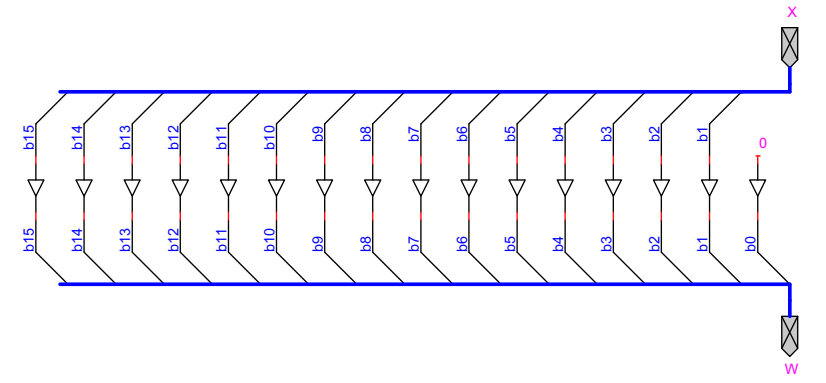


Fig. 13.9 Nuevo bloque W=X&(~1) *y su circuito interno.*

Estado	Acciones	Palabra de control compactada
F	IR ← Memw[PC] // PC ← PC + 2	R@/Pc=0, Byte=0, LdIr=1, Pc/Rx=1, N=0x0002, Ry/N=0, OP=00, F=100, Alu/R@=1, LdPc=1.
D	R@ ← PC + SE(N8)*2 // (RX ← Ra) // (RY ← Rb)	N=SE(IR<7..0>)*2, Pc/Rx=1, Ry/N=0, OP=00, F=100.
Jalr1	R@ ← RX & (~1)	Pc/Rx=0, OP=10, F=011.
Jalr2	PC ← R@ // Rd ← PC	Alu/R@=0, LdPc=1, Pc/Rx=1, OP=10, F=000, P/I/L/A=00, WrD=1, @D=IR<8..6>.

Fig. 13.10 Estados para la ejecución completa de JALR *con dos ciclos en la fase de ejecución sin necesidad de modificar la UP (camino de datos). Acciones y Palabra de Control compacta para cada nodo del grafo de estados de la UC.* F *y* D *son los mismos para todas las instrucciones.*

No obstante, todavía podemos juntar las acciones Jalr1 y Jalr2 en un único ciclo, siempre y cuando añadamos un camino directo de PC a Rd que no pase por la ALU, para así poder hacer en la ALU lo que se hacía en Jalr1 y cargarlo al final del ciclo en PC. Este nuevo camino apenas supone coste en hardware (usamos la entrada 3 del MUX-4-1 para llevar el PC al bus D del banco de registros, que hasta ahora no se usaba) y tampoco requiere aumentar el tiempo de ciclo del procesador. La figura 13.11 muestra el nuevo camino en la UP del SISC, que queda así completamente definido. Las dos señales de selección del MUX-4-1 que antes se denominaban -/I/L/A ahora se llamarán P/I/L/A, ya que con valor 3, 2, 1 y 0 seleccionan, respectivamente, la información del Pc, In, Ld y Alu. Así pues, implementamos la JALR con un único nodo/ciclo en la fase de ejecución, como se indica en la tabla la figura 13.12.

Estado	Acciones	Palabra de control compactada
F	IR ← Memw[PC] // PC ← PC + 2	R@/Pc=0, Byte=0, LdIr=1, Pc/Rx=1, N=0x0002, Ry/N=0, F=100, OP=00, Alu/R@=1, LdPc=1.
D	R@ ← PC + SE(N8)*2 // (RX ← Ra) // (RY ← Rb)	N=SE(IR<7..0>)*2, Pc/Rx=1, Ry/N=0, F=100, OP=00.
Jalr	PC ← RX & (~1) // Rd ← PC	Pc/Rx=0, OP=10, F=011, Alu/R@=1, LdPc=1, P/I/L/A=11, WrD=1, @D=IR<8..6>.

Fig. 13.12 Estados para la ejecución completa de JALR *con un solo ciclo en la fase de ejecución añadiendo una nueva conexión en la UP (camino de datos). Acciones y Palabra de Control compacta para cada nodo del grafo de estados de la UC.* F *y* D *son los mismos para todas las instrucciones.*

La palabra de control que debería generar la UC para realizar las acciones del estado/ciclo Jalr (que especifica en forma compactada en la figura 13.12), cuando se ejecuta la instrucción, por ejemplo, `JALR R7, R0`, es la siguiente (con el máximo número de x):

@A			@B			Pc/Rx	Ry/N	OP		F			P/I/L/A		@D			WrD	Wr-Out	Rd-In	Wr-Mem	LdIr	LdPc	Byte	Alu/R@	R@/Pc	N (hexa)				ADDR-IO (hexa)	
x	x	x	x	x	x	0	x	1	0	0	1	1	1	1	1	1	1	1	0	0	0	x	1	x	1	x	X	X	X	X	X	X

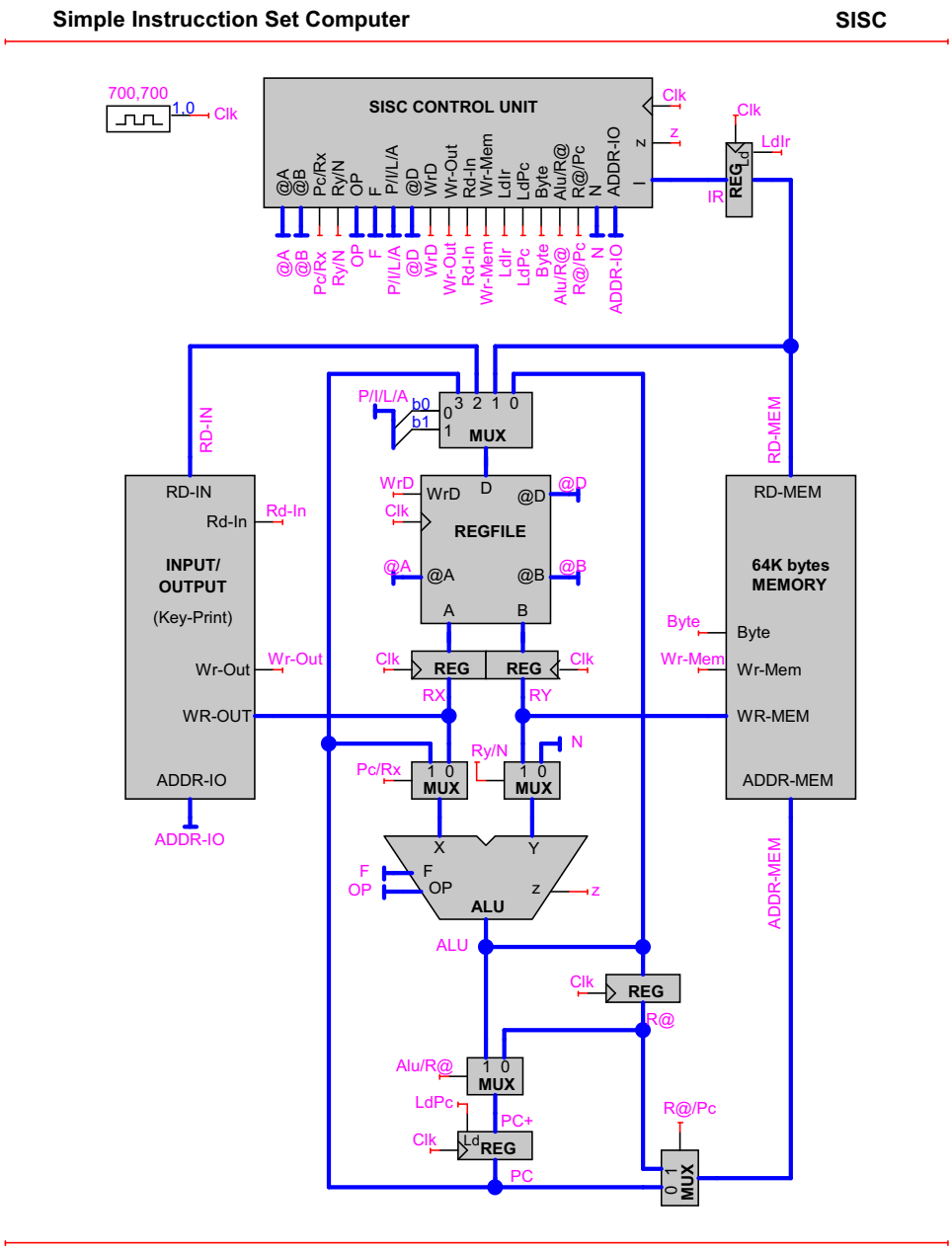
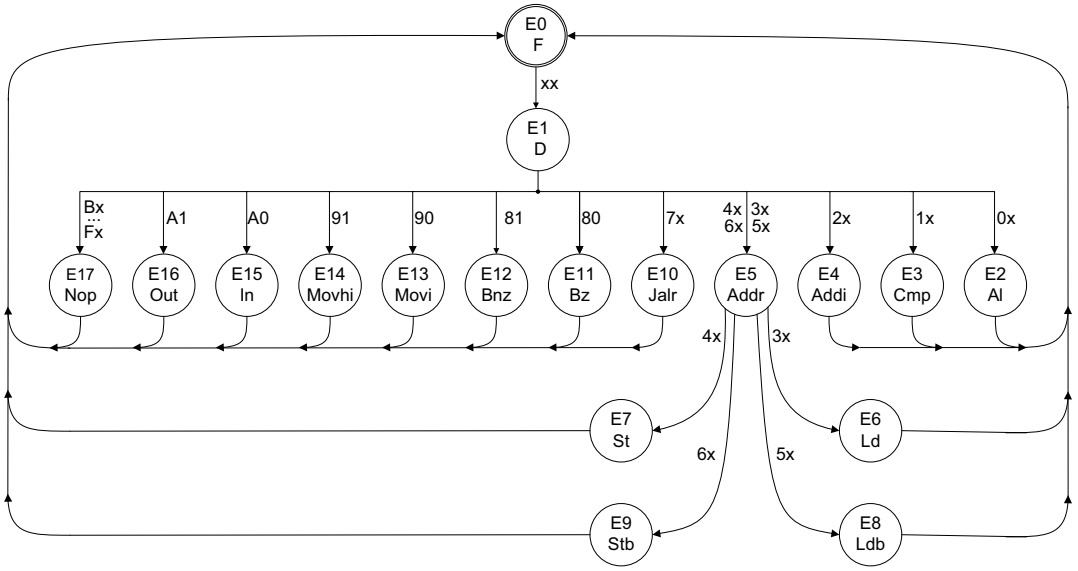


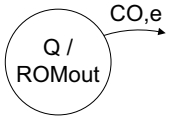
Fig. 13.11 Esquema detallado del computador SISC (definitivo, incluyendo JALR*)*

La palabra de control que genera nuestra implementación de la UC, considerando las x de la ROM_OUT, para el mismo ejemplo anterior, cuando el IR contiene 0111 000 111 000000, es la siguiente (ahora los campos @A, @B y ADDR-IO no tienen x):

@A			@B			Pc/Rx	Ry/N	OP		F			P/I/L/A		@D			WrD	Wr-Out	Rd-In	Wr-Mem	LdIr	LdPc	Byte	Alu/R@	R@/Pc	N (hexa)				ADDR-IO (hexa)	
0	0	0	1	1	1	0	x	1	0	0	1	1	1	1	1	1	1	1	0	0	0	x	1	x	1	x	X	X	X	X	C	0

El grafo de estados del circuito secuencial de la UC definitivo, después de añadir la instrucción JALR en el nodo E10 (que antes implementaba una NOP), es el siguiente.





CO: Código de operación de la Instrucción en hexadecimal ($I_{15}I_{14}I_{13}I_{12}$)
e: Extensión del código de operación (I_8)
Q: Estado en hexadecimal
ROMout: Mnemotécnico que indica la salida de la ROM_OUT del CONTROL

Por último, veamos los cambios que hay que hacer en la UC para implementar la nueva instrucción. Sólo hay que modificar el contenido de las dos ROMs usadas para implementar el grafo de estados. Empezamos por la ROM_Q+. La ejecución, propiamente dicha, de JALR se realiza en el estado E10 (Jalr), que codificado con los 5 bits de estado es 01010. En este ciclo el IR contiene la instrucción JALR cuyo código de operación es el 7, en binario con 4 bits 0111 y el bit e puede ser 0 o 1 ya que no se usa como bit de extensión del código de operación. Al ciclo siguiente de Jalr se debe ir al ciclo/nodo F, al

estado E0. Esto es, los 5 bits de salida de la ROM_Q+ para las direcciones, en binario, 0101001110 y 0101001111 es 00000. Esto se puede ver con fondo gris en la tabla de la tabla 13.2 en la página 21.

En la ROM_OUT hay que modificar el contenido de la dirección 01010 que corresponde al estado E10, Jalr. El contenido de esta dirección de la ROM es el siguiente y se puede ver con fondo gris en la tabla de la tabla 13.1 en la página 20.

@ROM	Bnz	Bz	WrMem	RdIn	WrOut	WrD	LdIr	Byte	R@/Pc	Alu/R@	Pc/Rx	Ry/N	P/I/L/A1	P/I/L/A0	OP1	OP0	MxN1	MxN0	MxF	F2	F1	F0	Mx@D1	Mx@D0
10	1	1	0	0	0	1	x	x	x	1	0	x	1	1	1	0	x	x	1	0	1	1	0	1

13.6 Tiempo de ciclo mínimo

Para calcular el tiempo de ciclo mínimo vamos a encontrar el camino crítico para cada ciclo/nodo de las fases de ejecución de una instrucción. El nodo que más tiempo requiera marcará el tiempo de ciclo mínimo del computador. Solo buscaremos el camino crítico de los nodos/ciclos que son candidatos a tardar más tiempo: F y D (por los que pasan todas las instrucciones) y, de los estados de ejecución, propiamente dichos, los de las instrucciones que, presuntamente, más tardan, que son:

- Ldb, último estado de la instrucción LDB (el que más tarda de entre los que acceden a memoria, como vimos en el capítulo anterior) y
- Cmp, el estado de ejecución de las instrucciones de comparación (el que más tarda de entre los que usan la ALU) y en particular analizaremos el caso de comparación que más tarda, el de la instrucción CMPLE.

Parámetros de tiempo usados en el análisis:

Repasemos primero los tiempos de los dispositivos básicos que usaremos en el análisis.

- Tiempos de propagación de las puertas. Tp(And-2) = Tp(Or-2) = **20** u.t., Tp(Not)=**10** u.t.
- Tiempo de propagación del biestable D activado por flanco, Tp(FF) = **100** u.t.
- Tiempos de propagación de las ROM, Tp(ROM_Q+) = **120** u.t. y Tp(ROM_OUT) = **60** u.t.
- Tiempo de propagación de un módulo de memoria RAM de 32KB, Tp(RAM) = **800** u.t.

En este capítulo tenemos todas las figuras de los esquemas lógicos internos de los bloques que necesitamos (SISC Von Neumann en la figura 13.2 y el del SISC CONTROL UNIT en la figura 13.5) excepto para los siguientes bloques, que deberemos ir a capítulos anteriores:

- MUX-2-1 (ver Cap. 3). Tiempo de propagación de la entrada de selección a la salida de datos $Tp(\text{MUX-2-1}_{Sel})$ = 50 u.t. (Tp(Not)+Tp(And-2)+Tp(Or-2)) y de una de las entradas de datos a la salida $Tp(\text{MUX-2-1}_{Data})$=40 u.t. (Tp(And-2)+Tp(Or-2)).
- ALU_{ADD} (ver Cap. 8). Tiempo de propagación la ALU desde los buses de entrada X e Y hasta la salida, cuando se realiza una operación ADD (OP=00 y F=100), $Tp(\text{ALU}_{ADD})$ = **860** u.t. (660 de sumar en el bloque ADD, ya que el acarreo de entrada vale 0 siempre, más 120 de atravesar el MUX-8-1 con

entrada de selección por F desde sus buses de datos de entrada a su salida más 80 de atravesar el MUX-4-1 con entrada de selección OP desde buses de datos de entrada a su salida)

- 64KB-32KW MEMORY (ver Cap. 11). Tiempo de acceso a una lectura de Byte, Tacc(MEM_{Ldb}) = 880 u.t. (Tp(RAM)+Tp($MUX\text{-}2\text{-}1_{Data}$)+Tp($MUX\text{-}2\text{-}1_{Data}$)) y el de lectura de word, Tacc(MEM_{Ld}) = 840 u.t. (Tp(RAM)+Tp($MUX\text{-}2\text{-}1_{Data}$)).

Búsqueda de la instrucción (estado F)

En este estado se realizan dos acciones en paralelo: la búsqueda de la instrucción y el incremento del PC en PC+2. Del análisis que detallamos a continuación se desprende que el camino de mayor tiempo es el de la segunda acción: **1.230 u.t.**

Búsqueda de la instrucción. En esta acción el camino sale de los biestables de estado de la UC y termina en entrada de los biestables del IR tardando 1.090 u.t. Los tiempos a los que se van estabilizando las señales del camino crítico son los siguientes:

- A las **100** u.t. desde el flanco ascendente de reloj de inicio de ciclo se estabiliza la salida Q de los biestables de estado de la unidad de control,
- A las160 u.t. se estabiliza R@/Pc (100 anteriores más **60** de la ROM_OUT).
- A las 210 u.t. se estabiliza ADDR-MEM, ya que la entrada 0 del MUX-2-1 ya estaba estable con PC a las 100 u.t. (160 + **50** del MUX-2-1 entrando por la señal de selección R@/Pc).
- A las 1.050 u.t. se estabiliza la nueva instrucción en RD-MEM (210 + **840** de acceso a la RAM en lectura de word)
- A las 1090 u.t. la instrucción está estable a la entrada de los biestables que forman el registro IR (1050 + **40** de pasar por el MUX-2-1 interno del registro con señal de carga IR, desde la entrada de datos, ya que la señal de permiso de escritura LdIr, señal de selección del multiplexor interno de IR. ya estaba estable desde pasadas 160 u.t. del inicio de ciclo). Así, respecto a la acción "búsqueda de la instrucción", a las 1.090 u.t. ya puede llegar el flanco ascendente de final de ciclo.

Incremento del PC. El camino más largo para esta acción comienza en la salida de los biestables de estado de la UC y termina en entrada de los biestables del PC tardando en total **1.230** u.t. Los tiempos acumulados en el camino son:

- A las **100** u.t. se estabilizan las salidas de los biestables de estado de la UC.
- A las160 u.t. se estabiliza MxN0 y MxN1 (100 anteriores más **60** de la ROM_OUT).
- A las 250 u.t. se estabiliza la salida N del MUX-4-1 (160 + **90** desde que la señal MxN0 está estable hasta que lo está la salida del MUX-2-1 del MUX-4-1 que genera N con el valor 0x002, que ya estaba estable en la entrada 3 del multiplexor desde las 100 u.t. desde que se puso en marcha el computador, ya que es un valor constante).
- A las 290 u.t. se estabiliza la entrada Y de la ALU con el valor de N (250 + **40** del MUX-2-1 con señal de selección Ry/N, que ya esta estable a 0 desde las 160 u.t. del inicio del ciclo).
- A las 1.150 u.t. se estabiliza la salida de la ALU (290 anteriores más **860** de Tp(ALU_{ADD})).

- A las 1.190 u.t. se estabiliza la salida del MUX-2-1 con señal de selección Alu/R@ (1.150 anteriores más **40** de atravesar este multiplexor desde los datos, ya que la señal de selección estaba estable mucho antes)
- A las 1.230 u.t. se estabilizan las entradas de los biestables internos del registro PC tras atravesar el MUX-2-1 interno del REGwLd con señal de permiso de escritura LdPc que ya estaba estable mucho antes (1.190 anteriores más **40** del MUX-2-1 desde la entrada de datos.

Decodificación (estado D)

Vamos a analizar los tiempos de los caminos críticos para realizar cada una de las tres funcionalidades de esta fase que se realizan en paralelo, luego nos quedaremos con el camino con mayor tiempo, el del cálculo de la dirección de salto tomado y escritura en R@: **1.150 u.t.**

Calculo del estado siguiente. El camino sale de las salidas Q de los biestables de estado de la UC y termina en la entrada D de estos mismos biestables, Q+, requiriendo **220 u.t.** En detalle:

- A las **100** u.t. se estabilizan las salidas de los biestables de estado de la UC.
- A las 220 u.t. se estabilizan las entradas D de los biestables de estado de la UC (100 + **120** de atravesar la ROM_Q+, cuyas salidas están conectadas a las entradas D de los biestables de estado de la UC.

Lectura de registros. Lectura de Ra y Rb y carga de su contenido en los registros RX y RY que hay a la salida del banco de registros. El camino se inicia en la salida Q los biestables del registro IR y termina en la entrada D de lso registros RX y RY, requiriendo **230 u.t.** En detalle:

- A las **100** u.t. se estabilizan la instrucción a la salida del registro IR y por lo tanto las señales @A y @B, que son los campos IR<11:9> e IR<8:9> de la instrucción respectivamente.
- A las 230 u.t. se estabilizan las entradas D de los registros RX y RY con el contenido de los registros Ra y Rb que indica la instrucción (100 + **130** desde la señal de selección @A del multiplexor de buses MUX-8-1 del banco de registros que selecciona qué registro se lee por el bus A y lo mismo para el bus B desde @B.

Cálculo de la dirección de salto tomado. El camino para la acción R@ ← PC + SE(N8)*2 comienza en la salida Q de los biestables de estado de la UC, pasa por la generación de N, la suma en la ALU y termina en la entrada del registro R@, necesitando **1.150** u.t. En detalle:

- A las **100** u.t. se estabilizan las salidas Q de los biestables de estado de la UC.
- A las160 u.t. se estabiliza MxN0 (100 anteriores más **60** de la ROM_OUT).
- A las 250 u.t. se estabiliza la salida N del MUX-4-1 (160 anteriores más **50** de que la señal MxN0 atraviese el primer nivel de MUX-2-1 con el que se implementa el MUX-4-1 más **40** de que los datos atraviesen el MUX-2-1 que genera la salida N del MUX-4-1).
- A las 290 u.t. se estabiliza la entrada Y de la ALU con el valor de N (250 + **40** de que el dato N atraviese el MUX-2-1 con señal de selección Ry/N, que ya esta estable).
- A las 1.150 u.t. se estabiliza la salida de la ALU, y por lo tanto la entrada D de los biestables de R@ (290 anteriores más **860** de $Tp(ALU_{ADD})$).

Lectura de Memoria (estado Ldb)

De todos los estados del grafo que acceden a memoria a leer o escribir un dato el Ldb (estado 8) es el que más tiempo requiere. El camino crítico se inicia en los biestables de estado de la unidad de control y termina en la entrada D del registro destino, Rd, del Banco de Registros, requiriendo **1.210 u.t**.:

- A las **100** u.t. se estabiliza el estado Q de la UC,
- A las160 u.t. se estabiliza R@/Pc (100 + **60** de atravesar la ROM_OUT).
- A las 210 u.t. se estabiliza ADDR-MEM (160 + **50** del MUX-2-1 entrando por la selección R@/Pc), ya que la entrada 1 del MUX-2-1 ya estaba estable con R@ a las 100 u.t.).
- A las 1.090 u.t. se estabiliza RD-MEM (210 + **880** de acceso a la RAM en lectura de Byte).
- A las 1.170 u.t. se estabiliza la salida del MUX-4-1, que es la entrada D del Banco de Registros (1.090 + **80** del MUX-4-1 con señal de selección -/i/l/a entrando por el dato 0).
- A las 1.210 u.t. se estabiliza la entrada D de Rd (1.170+**40** del MUX-2-1 del REGwLd que es el registro destino de la instrucción).

Comparación (estado Cmp)

De todos los estados del grafo de usan la ALU para hacer alguna operación las que requieren más tiempo son las comparaciones ya que la resta requiere algo más tiempo que la suma y además debe calcularse el resultado de la comparación, verdadero o falso, a partir del resultado de la resta. De entre todas las comparaciones, que se implementan en el estado Cmp, la que requiere más tiempo es la CMPLE. El camino crítico se inicia en los biestables de estado de la unidad de control y termina en la entrada D del registro destino, Rd, del Banco de Registros, requiriendo **1.350 u.t**.:

- A las **100** u.t. se estabiliza el estado Q de la UC,
- A las 160 u.t. se estabilizan Pc/Rx y Ry/N (100 + **60** de atravesar la ROM_OUT).
- A las 210 u.t. se estabilizan los buses X e Y de entrada en la ALU tras pasar el camino crítico por los dos MUX-2-1 entrando por las señales de selección Pc/Rx y Ry/N (160 anteriores más **50** de atravesar un MUX-2-1 desde la señal de selección).
- A las 1.230 u.t. se estabiliza la salida de la ALU con el resultado de la comparación (210 anteriores más **1.020** de atravesar la ALU: 940 del bloque CMP más 80 del MUX-4-1 controlado por OP que selecciona entre cuatro diferentes tipos de operaciones de la ALU: Aritmético lógicas, de comparación o miscelánea). Las 940 u.t. del bloque CMP se obtienen de sumar 710 para generar la salida W del bloque SUB más 90 del bloque z (o el camino equivalente de 750 de la señal de overflow v, más 50 de la Xor-2) más 20 de la Or-2 más 120 del Mx-8-1 de salida del bloque CMP. Los 710 del SUB se obtienen de sumar 10 de la puerta Not más 700 del ADD cuando el acarreo de entrada es 1. Estos 700 se obtienen de sumar 90 (del acarreo de salida del Full-adder 0 cuando el acarreo de entrada es 1) más 14 por 40 (14 eses el número de Full-adders intermedios y 40 es el tiempo de propagar el acarreo en cada Full-adder ya que pasa por una puerta And-2 y una Or-2) más 50 de la Xor-2 de salida del bit de más peso del ADD.
- A las 1310 u.t. se estabiliza el bus de entrada D del banco de registros, tras atravesar el camino crítico por el MUX-4-1 controlado por la señal P/i/l/a, para encaminar el resultado hasta el bus de escritura D del banco de registros (1230 anteriores más **80** del MUX-4-1 desde el bus de entrada).

- A las 1350 u.t. se estabilizan las entradas D de los biestables del registro destino tras atravesar el MUX-2-1 que implementa la señal de permiso de carga de los registros del banco (1310 más **40** u.t. del MUX-2-1 desde los datos ya que @D y WrD ya están estables, por lo que la señal Ld del registro destino Rd ya es estable cuando llega el dato a escribirse).

Por ello, el tiempo del camino crítico en el ciclo Cmp es de **1.350** u.t. Por último, comentar que en ejercicios posteriores se pedirá calcular el camino crítico de todos los estados de ejecución de todas las instrucciones. De esta forma, si detectamos que si algunos estados del grafo requieren, por ejemplo, aproximadamente el doble de tiempo que los estados que requieren menos tiempo, podríamos proponer otros diseños del grafo para hacer el muevo Tc la mitad del original y que cada estado de los que requieren el doble de tiempo se implemente, en el nuevo grafo de la UC, sustituyéndose por dos nuevos estados.

Tiempo de ciclo del computador SISC. Con los datos obtenidos el tiempo de ciclo tiene que ser mayor de 1.350 u.t. Así, para el diseño actual del computador y sumando 50 u.t. más por seguridad y para redondear, usaremos un tiempo de ciclo de:

Tc (SISC) = 1.400 u.t.

13.7 Restricciones temporales de Wr-Mem, Rd-In y Wr-Out

Los parámetros temporales de cada uno de los dos módulos de memoria RAM de 32 KB usados en la implementación de la memoria de datos del SISC Harvard uniciclo, multiciclo y ahora en la memoria del Von-Neumann que almacena las instrucciones y los datos son: T_{acc}= 800 u.t., T_{su} = 60 u.t., T_{pw} = 600 u.t y T_h= 40 u.t.

Wr-Mem. *Escritura de memoria (estado* St y Stb*)*

Veamos si con los parámetros temporales dados para los módulos de memoria y la generación de Wr-Mem como indica la figura 13.5 (la salida de una And-2 a la que le entran la señal WrMem producida por la ROM_OUT y la señal de reloj negada, !Clk) produce una correcta escritura de la memoria en los ciclos/estados St y Stb. Con este análisis determinaremos el T1 (primera mitad del ciclo, cuando el reloj está a 1) y T0 (segunda mitad, con valor 0) del reloj para que funcione correctamente la escritura de memoria. Debemos tener presente el esquema lógico interno de la memoria construida con los dos módulos de 32 KB tal como vimos en el capítulo 11 y que reproducimos aquí en la figura 13.13.

¿Cuál es el T1 mínimo para que T_{su} u.t. antes de que Wr-Mem-1 y Wr-Mem-0 pase de 0 a 1 ya estén estables la dirección en ADDR-MEM-15 bits- (el bus de entrada de dirección de15 bits a cada uno de los dos módulos de memoria y el dato a escribir en WR-MEM-8 bits- (el bus de entrada del dato a escribir de 8 bits de cada módulo de memoria)?

WR-MEM-8 bits- del módulo 1 tarda mucho más en estabilizarse que su equivalente del módulo 0 ya que el camino crítico para el módulo 1 llega por la señal de selección !Byte del MUX-2-1 cuya salida está conectada a WR_MEM-8 bits-(figura 13.13). La señal !Byte esta estable a las 170 u.t. del inicio del ciclo St o Stb (100 de los biestables de estado, 60 de la ROM_OUT y 10 de la Not). Por lo que la salida del MUX, la entrada WR-MEM-8 bits- del módulo 1 de la memoria, está estable a las **220 u.t. del inicio**

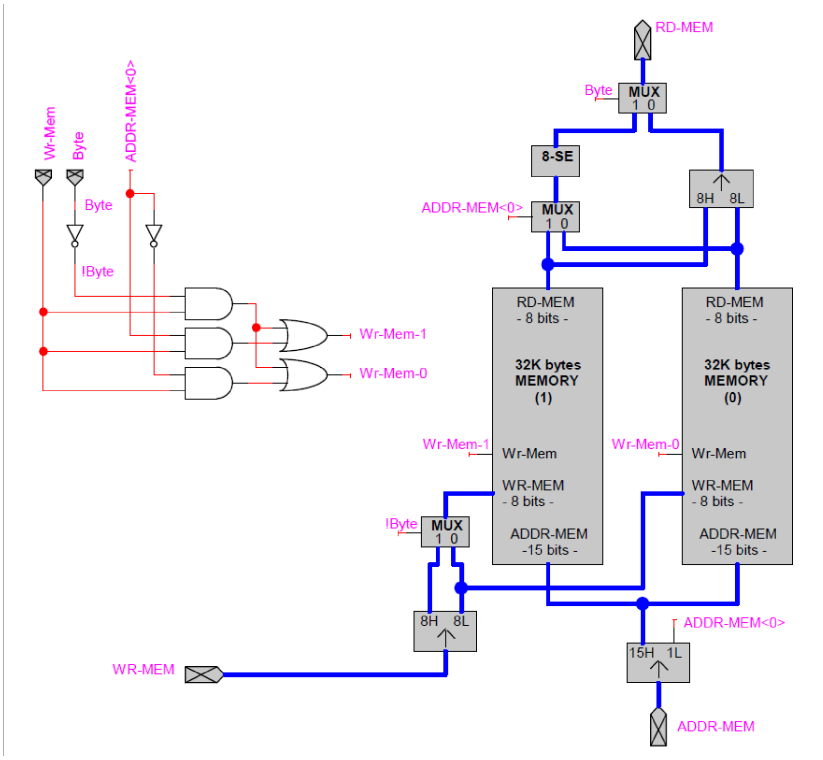


Fig. 13.13 Esquema interno de la memoria del SISC

del ciclo St o Stb (170 más 50 del multiplexor llegando por la entrada de selección). Sin embargo, WR-MEM está estable a la entrada de este multiplexor y a la entrada del módulo 0, ya que no existe multiplexor en el módulo 0, a las 100 u.t. del inicio del ciclo St y Stb, que es el Tp del registro RY (y un análisis más detallado nos dice que ya está estable un ciclo antes).

ADDR-MEM-15 bits- de cada módulo se estabiliza a la vez que el bus ADDR-MEM (ver figura 13.13) ya que el bloque 16-to-15H1L no contiene puertas en su circuito interno y ADDR-MEM se estabiliza a los **210 u.t. del inicio del ciclo St o Stb** (100 del Tp de los biestables de estado + 60 de la ROM_OUT para generar R@/Pc a 1 + 50 para estabilizar a la salida del MUX-2-1 su entrada 1 (poner la salida de R@ en el bus ADDR-MEM)).

Como WR-MEM -8 bits- se estabiliza a 220 u.t del inicio de ciclo, Wr-Mem-1 y Wr-Mem-0 no pueden pasar de 0 a 1 antes de las 280 u.t. del inicio de ciclo (220 + 60 del Tsu). Como el tiempo de propagación del circuito que genera Wr-Mem-1 y Wr-Mem-0 (ver figura 13.13) a partir de Wr-Mem, Byte y ADDR-MEM<0> desde la entrada Wr-Mem a cualquiera de las dos salidas es 40 u.t Wr-Mem no puede pasar de 0 a 1 antes de 240 u.t. (280 - 40 del tiempo de propagación de Wr-Mem a Wr-Mem-1 o a Wr-Mem-0). Así que **T1 mínimo es 210** u.t. (240 menos 20 de la And entre !Clk y WrMem generada por ROM_OUT menos 10 de negar Clk).

Fijándonos en los tiempos de propagación del circuito que genera Wr-Mem-1 y Wr-Mem-0 (ver figura 13.13) a partir de Wr-Mem, Byte y ADDR-MEM<0> y con los datos obtenidos en los párrafos anteriores podemos concluir que el camino crítico para generar Wr_Mem-1 y Wr-Mem-0 si T1 es 210 llega a este circuito por Wr-Mem, que se estabiliza a los 240 u.t., mientras que Byte esta estable antes (a las 160 u.t. del inicio de ciclo) y ADDR-MEM<0> también (a las 210 u.t.). Así que el T1 mínimo de 210 que hemos calculado es correcto.

¿Cuál es el T0 para Wr-Mem?

El T0, que es la anchura del pulso de escritura, que tiene que ser como mínimo de 600 u.t., que es el T_{pw} del módulo de memoria. Además, T0 es lo que tarda la segunda mitad del tiempo de ciclo, por lo que la suma de T1 y T0 debe ser 1400 u.t. para no tener un tiempo de ciclo innecesariamente grande. Por lo tanto, el T0 máximo es 1.190 u.t. (1400 - 210).

Por lo tanto, una señal de reloj simétrica con **T1 = T0 = 700 u.t.** es, de momento, una opción sobradamente correcta.

Rd-In y Wr-Out. *Input (estado* In*) y Output (estado* Out*)*

No vamos a entrar en el detalle de las restricciones temporales de estas señales. Solo decir que imponen menos restricciones que para el acceso a memoria que acabamos de tratar. Afirmamos que generar Rd-In y Wr-Out como se ha hecho para la Wr-Mem (de la salida de ROM_OUT a través de una And-2 con !Clk) usando un reloj simétrico de Tc = 1.400 u.t. es también una buena solución para estas señales.

Conclusión

La señal de reloj del SISC es **simétrica** con **Tc = 1.400** u.t. (**T1 = 700** y **T1 = 700**). Esta decisión y la implementación de Wr-Mem, Rd-In y Wr-Out con una And-2 de WrMem, RdIn y WrOut respectivamente y de !Clk cumple con las restricciones temporales de los módulos de memoria y de las entradas/salidas.

13.8 Evaluación del rendimiento

Veamos primero un ejemplo de evaluación del rendimiento comparando los tres computadores SISC. El Tiempo Medio por Instrucción, Tmpi, del SISC Von Neumann, suponiendo que (para un conjunto de programas en lenguaje máquina SISA) un 30% de las instrucciones ejecutadas son lentas (del tipo acceso a memoria) y un 70% son rápidas (del resto de tipos) es 4.620 u.t. (0,7*3+0,3*4)*1.400 = (2,1+1,2)*1.400 = 3,3*1.400 = 4.620 u.t.

En general, si definimos, como se hizo en la Introducción, el ratio r como el cociente entre el número de instrucciones lentas ejecutadas y el número total de instrucciones ejecutadas tenemos que Tpmi(r) = ((1-r)*3+r*4)*1.400 = 1.400(r+3) u.t.

Tmpi (SISC Von Neumann, r) = 1.400(r+3) u.t.

Si calculamos el Tmpi en los dos computadores Harvard tenemos:

Tmpi (SISC Harvard Uniciclo, r) = 3.000 u.t.

Tmpi (SISC Harvard Multiciclo, r) = 750(r+3) u.t.

Comparando estos tiempos de ejecución, se puede afirmar que para cualquier programa que se ejecute en los tres computadores:

- El computador SISC Harvard Multiciclo es el más rápido de los tres.
- El computador SISC Von Neumann es el más lento de los tres.

Afinemos un poco más estas afirmaciones. Vamos a obtener x para que sea cierta la siguiente frase: Un programa con ratio r de instrucciones lentas ejecutadas sobre el total de instrucciones ejecutadas tarda en ejecutarse en el SISC Von Neumann un x% más de lo que tarda en el SISC Harvard Multiciclo. Como los Tmpi de los dos computadores que vamos a comparar tienen en común el factor multiplicativo (r+3) el valor de x queda independiente de r: x=(650/750)100 = 86,6. Así, podemos afirmar que:

> *Cualquier programa* ***tarda un 86,6% más*** *en ejecutarse* ***en el SISC Von Neumann que en SISC Harvard Multiciclo***.

Obtengamos ahora x cambiando el SISC Harvard Multiciclo por el SISC Harvard Uniciclo, en función de r (ya que el tiempo de ejecución del uniciclo no depende de r y el del Von Neumann sí).

> *Un programa con un ratio r de instrucciones lentas ejecutadas sobre el total de instrucciones ejecutadas* ***tarda un*** $\frac{1400 \times r + 1200}{3000} \times 100$ ***% más*** *en ejecutarse* ***en el SISC Von Neumann que en SISC Harvard Uniciclo***. *Este porcentaje va del 40% cuando todas las instrucciones ejecutadas son rápidas (r=0) al 86,6% cuando todas son lentas (r=1).*

Conclusión:

La ventaja del computador Von Neumann frente a los dos anteriores no es la velocidad de ejecución de programas, sino la mayor eficiencia espacial y, lo que es más importante, la mayor versatilidad.

- **Eficiencia espacial**. Usa bastante menos hardware: no requiere la memoria ROM de instrucciones ni el +2 ni el sumador del desplazamiento al PC, que sí tienen los otros dos computadores, y solo se le añaden 4 registros de 16 bits, dos multiplexores 2-1 y poco más).
- **Versatilidad**. Esta ventaja la valoramos en lo que sigue:
  - Los computadores SISC Harvard (tanto el uniciclo como el multiciclo) no tenían instrucciones de escritura (ni lectura) a la memoria de instrucciones por lo que el programa no se podía cargar (escribir) mediante un programa (del sistema operativo) que ejecutara el propio computador. La forma de cargar un programa era cambiando la ROM de la memoria de instrucciones. Este limita estos computadores a propósitos específicos, que ejecutan siempre el mismo programa una vez que el hardvare está montado. Sin embargo el SISC Von Neumann puede ejecutar un programa que lea de un disco otro programa, lo cargue en memoria y pase a ejecutarlo. Esto es lo que hacen todos los computadores que se usan en entornos generales, como un computador personal.
  - Es fácil añadir una nueva instrucción al SISC Von Neumann sin modificar el camino de datos (o apenas modificarlo, según la instrucción que se añada). Solo hay que cambiar el contenido de las dos memorias ROM que implementan el grafo de estados de la Unidad de Control. Por ejemplo, podemos añadir una instrucción que copie un vector de N bytes de una zona de memoria a otra, VMOVE, y que, claro está, tardará varios ciclos para mover cada byte.