Capítulo 6

LA RUTA DE DATOS Y LA MICROARQUITECTURA

Ahora veremos que se necesita en un procesador para poder ejecutar las instrucciones de lenguaje de máquina y cuál es el proceso de ejecución de las mismas.

Ejecución de una instrucción.

A fin de cuentas cada instrucción de lenguaje de máquina debe traducirse de alguna manera en una serie de señales que viajan por diferentes cables. Algunas de estas señales le dicen a ciertos circuitos que hacer con otras señales en las que son traducidos los datos. La instrucción de suma que leímos de la memoria debe generar una señal para la ALU que le indica que debe hacer una suma y los otros campos de la instrucción deben ocasionar que los operandos lleguen al ALU por otro conjunto de lineas. La circuitería de la ALU calcula la suma y debe a su vez generar nuevas señales eléctricas que viajan hasta donde sea necesario para almacenar el resultado.

A los distintos subsistemas de un procesador que se encargan de labores específicas se les suele llamar unidades funcionales. Así por ejemplo la ALU es una unidad funcional y la unidad aritmética de punto flotante es otra. Pueden existir otras encargadas de calcular direcciones o de hacer las veces de interfaz con la memoria.

Nuestras computadoras actuales, en su mayoría son de arquitectura de Von Neumann. Eso significa que su arquitectura es, en esencia, la de la máquina IAS que Von Neumann diseño en el Instituto de Estudios Avanzados de Princeton. Se tiene una memoria donde se almacenan los datos y el programa a ejecutar, se tiene una unidad de control que

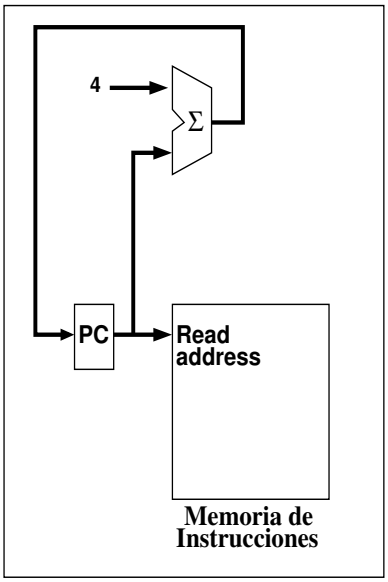

Figura 6.1. El PC y sus conexiones.

secuencia las instrucciones que lee de la memoria, obtiene los datos y le dice a una unidad aritmético-lógica que operación efectuar, se tiene un registro para almacenar los resultados de la ALU (un acumulador a fin de cuentas) y la unidad de control transfiere también los resultados a la memoria. La característica distintiva de las arquitecturas de Von Neumann es que el programa es guardado también en la memoria.

Program Counter.

Para ejecutar un programa es necesario, entonces, tener en algún lugar la dirección en memoria de la siguiente instrucción a ejecutar, para saber que pedirle a la memoria cuando sea necesario traer de ella la siguiente instrucción (*fetch*). Tendremos entonces un registro especial en nuestro procesador que llamaremos PC (*program counter*) que apunta a la siguiente instrucción a a ejecutar.

Para asegurarnos de que el PC siempre apunta a la siguiente instrucción, debemos actualizarlo cada vez que ya hemos hecho el fetch de una instrucción. Como casi siempre la siguiente instrucción a ejecutar será la inmediata siguiente a la actual, cuya dirección está actualmente en PC y como todas nuestras instrucciones en DLX miden 4 bytes, entonces no hay más que incrementar en 4 el valor actual de PC. Además PC necesita decirle a la memoria que dirección quiere así que debe estar conectado con ella. Esto lo podemos poner gráficamente como se muestra en la figura 6.1.

Por supuesto en ocasiones habrá que cambiar PC de otra manera, cuando haya que hacer un salto. Pero de eso nos encargaremos luego.

Instruction Register.

Ya que la memoria nos entrega la instrucción la guardamos en un registro especial al que llamaremos *instruction register* o simplemente IR. Este está capacitado para partir la instrucción en campos, o mejor dicho, es el registro de trabajo del decodificador de instrucciones, una unidad encargada de partir la instrucción y determinar las acciones a tomar con base en el análisis de la instrucción.

Imaginemos que en IR hay una instrucción de suma, algo como `ADD R3, R2, R1`, suma R1 con R2 y guarda el resultado en R3. La instrucción es del tipo R, y los números de registro están en los campos respectivos de la instrucción. Así que IR debe ser capaz de pedirle al banco de registros del CPU los dos operandos necesarios y de informarle que también deberá permitir la escritura en un tercer registro. IR debe estar conectado con el banco de registros mediante tres lineas, una para cada operando.

¿Qué tal si la instrucción es del tipo I, con operando inmediato?, quizás una operación aritmética. Entonces uno de los operandos aparece en la instrucción misma y su valor deberá pasar a la ALU. Así que IR además está conectado con la ALU mediante una linea. Aquí hay un pequeño problema. Decidimos poner 16 bits en el campo inmediato de la instrucción de tipo I porque determinamos que, estadísticamente, en la mayoría de los casos esa longitud basta. Pero nuestra ALU debe poder operar con los registros, que miden 32 bits, así que de alguna manera debemos "agrandar" nuestro campo inmediato para que sea de 32 bits. La manera de hacer eso es meter nuestro inmediato en un registro de 32 bits, uno especial al que llamaremos Imm, concretamente en los 16 bits menos significativos de Imm y luego añadirle a ese registro sus 16 bits más significativos. Como vamos a hacer operaciones aritméticas, es entonces necesario asegurarnos de que no se altere el signo del operando inmediato, así que le pegamos 1's si era negativo y 0's si no lo era.

En síntesis IR tiene cuatro lineas de salida: tres al banco de registros y una indirecta a la ALU, que pasa antes por un dispositivo de extensión de signo (al que llegan 16 bits y salen 32) y luego por un registro de 32 bits llamado Imm, que entra como operando a la ALU.

El banco de registros debe entregar el contenido de los registros que le fueron solicitados por IR y entregarlos como operandos de la ALU. Además, si se ejecuta una instrucción aritmético lógica debe poder recibir el resultado para guardarlo en un registro, así que debe recibir como entrada la salida de la ALU. ¡Oops! pero que tal si se ejecuta un `load`, entonces debe recibir entrada de la memoria también, ambas entradas: la de la ALU y la de memoria, contienen el dato que hay

que escribir en un registro, mismo que está especificado en uno de los campos de IR . ¡Ah! pero no pueden ocurrir simultáneamente ambas cosas, el banco de registros recibe el dato a escribir de una y sólo una de dos fuentes: la ALU y la memoria. Sólo una de las dos fuentes envía algo al banco de registros en un instante determinado, nunca ambas a la vez. Así que básicamente lo que hay que hacer es *multiplexar* la salida de la ALU y la entrada de la memoria para sacar una sola línea de entrada al banco de registros.

Quien hace todo el trabajo interesante es la ALU, que recibe sus dos entradas, a las que llamaremos A y B de:

- Ambas del banco de registros. Operación aritmético-lógica, instrucción tipo R.
- Una del banco de registros, otra del registro Imm . Operación aritmético-lógica, Instrucción tipo I.
- Una del registro Imm y otra del resultado de sumar 4 a PC . En el caso de saltos donde la dirección de salto esta indicada en el campo inmediato. Instrucción tipo I. En este caso el registro que hay que comparar con cero pasa a un comparador, no a la ALU.
- Una de NPC (así llamaremos a un registro donde almacenamos el valor de PC +4) y otra del banco de registros. Esta opción existe como posibilidad, pero no la utiliza DLX. Sería útil añadirla si tuviéramos una instrucción que hiciera algo así como sumar NPC con el registro para obtener la dirección de un salto, habría que distinguirla de la instrucción `JR` actual, donde no se suma NPC al registro.

Con base en este análisis podemos determinar ponerle a la ALU dos entradas, cada una de ellas *multiplexa* dos entradas. Uno de los multiplexores recibe un registro y el resultado de PC luego de sumarle 4, que almacenaremos en un registro llamado NPC . El otro multiplexor recibe como entradas un registro del banco y el registro Imm . Cada multiplexor deja pasar sólo una de sus entradas, así tenemos cubiertas las cuatro posibilidades mencionadas arriba.

Saltos. Conviene ahora discutir acerca de PC nuevamente. Como dijimos antes si la instrucción a ejecutar es un salto entonces el resultado de PC incrementado en 4 no es realmente la dirección de la siguiente instrucción, o mejor dicho, puede no serlo. Lo que el ensamblador debe hacer al encontrarse un salto a una etiqueta del programa es determinar cuál es la distancia del salto, es mucho más barato poner una

distancia que poner una dirección absoluta. Esta distancia es la que se deberá poner en la instrucción y es, de hecho, lo que hay que sumarle al PC ya incrementado para obtener la dirección efectiva de la siguiente instrucción. Esto en algunos casos, en otros se debe tomar el contenido de un registro como el nuevo valor de PC .

Hay dos tipos de salto, condicional o incondicional. En nuestra arquitectura pondremos de ambos y en el caso de los condicionales pondremos instrucción de "salta si cero" y "salta si no cero" que verifican el contenido de un registro para determinar si saltar o no.

Así que una de las salidas del banco de registros debe pasar por un dispositivo de verificación que determina si es cero o no. La salida de A es la que debe entrar al comparador, es allí donde se guarda el operando `rs1` que se usa para estos saltos (instrucción tipo I). Luego el resultado de esta verificación sirve para determinar si la dirección de la siguiente instrucción a ejecutar será la de NPC (PC +4) o bien la otra alternativa: el resultado, obtenido por la ALU, de sumar una cantidad extra al contenido de NPC .

Sinteticemos. Ya habíamos dicho que PC , luego de traerse la instrucción a ejecutar debía incrementar su valor en 4, este valor nuevo lo guardamos en un registro llamado NPC , así que no tenemos más que copiar NPC a PC y ya. Pero esto no es cierto siempre. Si la instrucción a ejecutar es un salto entonces el contenido de NPC puede o no, ser la dirección de la siguiente instrucción. Si no lo es, se debe a que un registro del procesador cumplió con la condición necesaria para saltar o bien es un salto incondicional y entonces es el resultado de la ALU lo que debe quedar como nuevo PC . Entonces la entrada de PC es, en realidad, la salida de un multiplexor que recibe, por una parte, NPC y por otra la salida de la ALU, el multiplexor decide a quién dejar pasar en función de la salida de un dispositivo de verificación de condición.

Bien, luego de todo este rollo mareador podemos ver el diagrama que resulta en la figura 6.2. A este tipo de diagramas se les conoce como ruta de datos (*datapath*).

Load Memory Data.

Si la instrucción a ejecutar es una de carga de memoria, entonces, luego de calcular la dirección absoluta del dato requerido de la memoria, esta debe ser pasada a la memoria junto con una petición de lectura. Una vez que la memoria trae el dato requerido de regreso debe poder guardarse en algún lugar temporalmente, este registro se llama LMD (*Load Memory Data*).

En las instrucciones de carga y en las aritmético-lógicas debemos escribir el resultado de la operación en un registro del CPU. Así que la entrada de datos del banco de registros proviene de una de dos

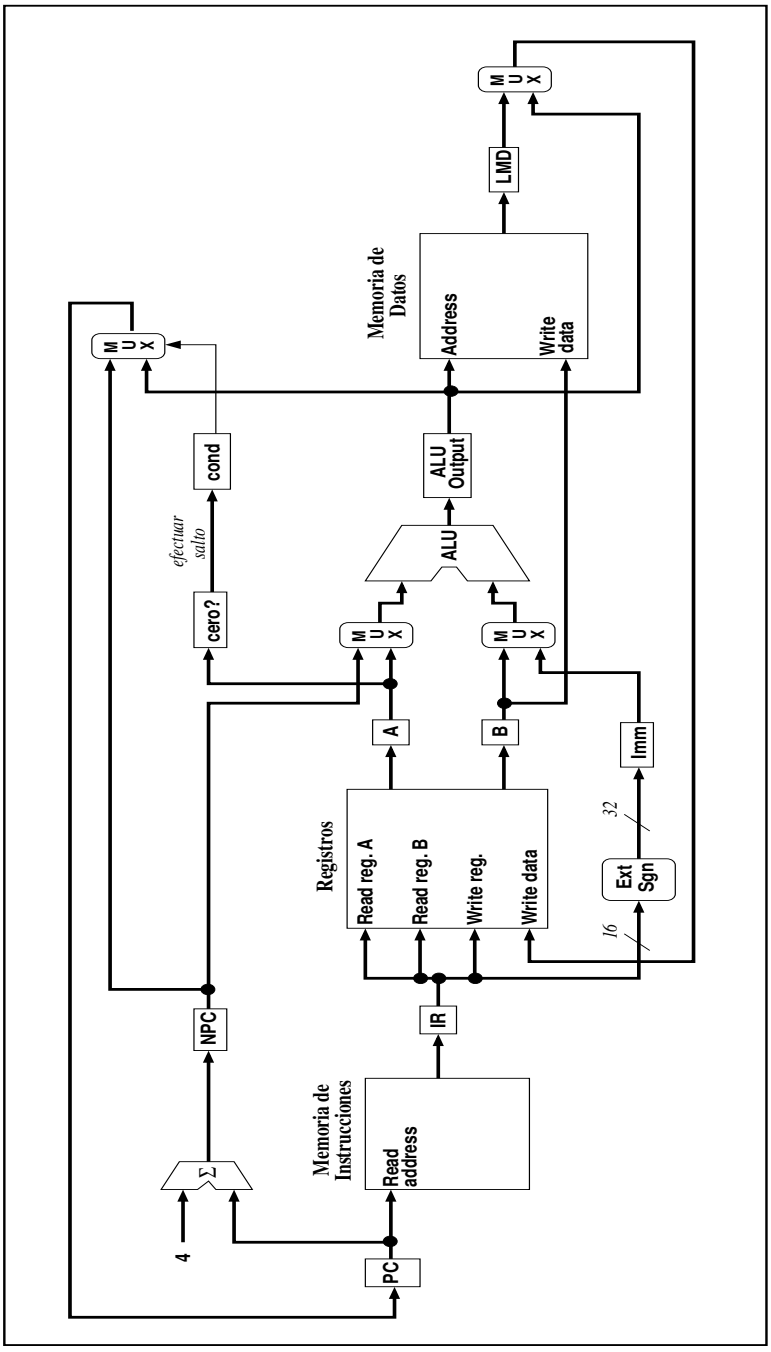

Figura 6.2. La ruta de datos (*datapath*) de DLX.

fuentes: LMD o la salida de la ALU. Otra vez pasamos ambas por un multiplexor del que sólo sale una de las dos para escribirse en el registro del procesador.

6.1 SEÑALES DE CONTROL

Hemos dicho que ejecutar una instrucción significa, esencialmente, distribuir una serie de señales por diferentes lineas de transmisión que se encargan de: habilitar o deshabilitar la entrada a ciertas unidades funcionales de otras lineas (por ejemplo, determinar que debe salir de un multiplexor), indicar a ciertas unidades funcionales que operación realizar (por ejemplo, decirle a la ALU que operación efectuar).

Unidad de control.

La encargada de determinar que señales enviar y en que secuencia es llamada *unidad de control*. Por supuesto su entrada es el IR (puede haber otras entradas más, pero esta es la fundamental) y su salida son todas las lineas transportadoras de señales de control que llegan a las distintas unidades funcionales.

Nos toca decidir ahora cuáles señales son necesarias y posteriormente, bajo que condiciones una señal debe estar prendida.

Señales de control.

En nuestra ruta de datos de DLX se requiere de las siguientes señales de control:

RegWrite Para especificar que el banco de registros debe aceptar la entrada `WriteData` y escribirla en el registro especificado en `RegWrite`. 1=escribir, 0=ignorar.

ALUSrc1 0 significa que la entrada de la ALU debe ser NPC . 1 que la entrada debe ser el registro A .

ALUSrc2 0 significa que la entrada de la ALU debe ser Imm . 1 que la entrada debe ser el registro B .[1]

ALUCtrl Especifica la operación que debe efectuar la ALU. Deben ser unos tres o cuatro bits.

PCsrc Decide quien debe ser el contenido de PC . 0 significa que el contenido debe ser forzosamente NPC , 1 significa que puede ser NPC o la salida de la ALU dependiendo de si la condición de salto se cumple o no.

MemRead Le indica a la memoria que lea la localidad indicada en `Adress` y regrese su contenido en LMD . 1 significa lectura, 0 no lectura.

[1]Recuérdese que no es válida la combinación ALUSrc1=0, ALUSrc2=1.

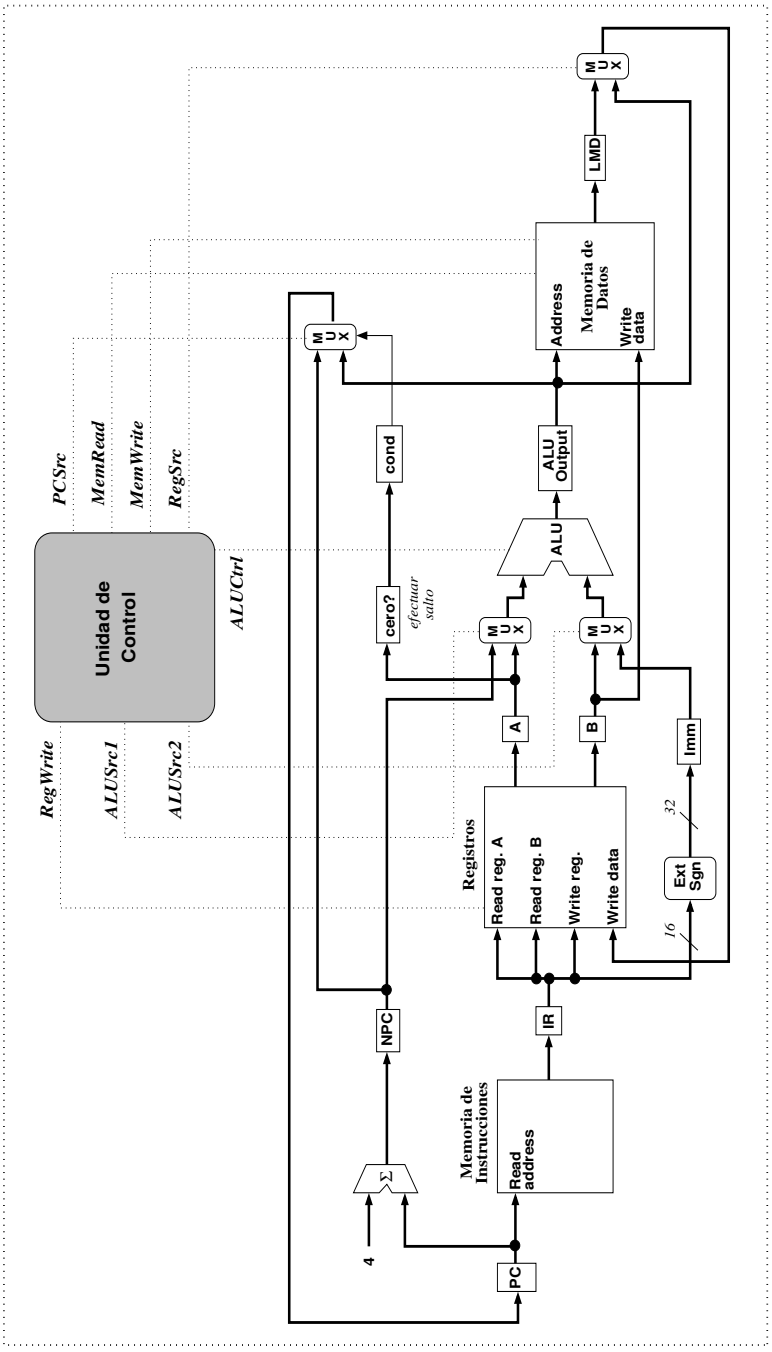

Figura 6.3. La ruta de datos (*datapath*) de DLX con señales de control.

Instr.	RegWr	ALUSr1	ALUSr2	ALUCtrl	PCSr	MRd	MWr	RegSrc
`ADD R3,R2,R1`	1	1	1	Suma	0	0	0	1
`SUB R3,R2,R1`	1	1	0	Resta	0	0	0	1
`LW R3,120(R2)`	1	1	0	Suma	0	1	0	0
`SW R3,120(R2)`	0	1	0	Suma	0	0	1	X
`BEQZ R3,salto1`	0	0	0	Suma	1	0	0	X

Tabla 6.1. Ejemplos de instrucciones y las señales de control que generan.

MemWrite La indica a la memoria si debe o no ser escrito el dato `WriteData` en `Adress`. 1 significa escritura, 0 no escritura[2]

RegSrc Indica quien debe pasar como el dato a escribir en el banco de registros. 0 significa que pasa LMD y 1 que pasa el resultado de la ALU.

¿Como se vería entonces la ejecución de una instrucción? Esencialmente se ve entonces como una secuencia de señales de control. En la tabla 6.1 se muestran las señales de control asociadas con ciertas instrucciones.

6.2 CONTROLADORES, CLASIFICACIÓN

Ejecutar una instrucción es traducirla a señales de control que se generan en un cierto orden. Es decir, de alguna manera se debe secuenciar el envío de estas señales a través de las lineas de salida de la unidad de control. En el caso de DLX no es evidente que se requiere de un orden. En arquitecturas con conjuntos de instrucciones más complejos es más claro.

Imaginemos que tenemos una máquina cuya ruta de datos está determinada por un único canal bidireccional que comparten todas las unidades funcionales, un *bus*, como la mostrada en la figura 6.5 y que es muy similar a la que solían tener las minicomputadoras de la línea PDP de Digital en los 70. Nuestra máquina será de dos direcciones, del tipo reg-mem y, digamos 4 modos de direccionamientos con conjunto

[2]Esto significa que si la instrucción ejecutándose no es `load` o `store` tanto esta señal como la anterior deben ser cero.

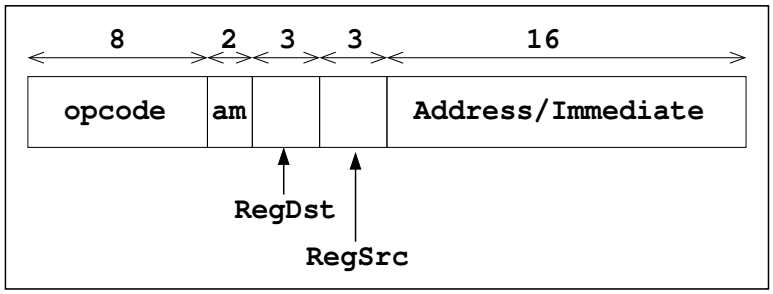

Figura 6.4. Posible formato de instrucción para la arquitectura de canal único.

de instrucciones ortogonal. Digamos que el opcode mide 8 bits, luego sigue el código de modo de direccionamiento (dos bits), luego el operando destino que siempre es un registro codificable en tres bits, luego el operando inmediato o la dirección de memoria a acceder (16 bits) y luego tres bits más para el registro origen. Con esto las instrucciones de la máquina medirán 32 bits, como las de DLX (ver figura 6.4).

En este caso:

- IR: Instruction Register. Igual que en nuestro caso es el registro en el que se guarda la instrucción a ejecutar.
- PC: Program Counter. También como nuestro PC, sirve para almacenar la dirección en memoria de la siguiente instrucción a ejecutar.
- MDR: Memory Data Register. Es un registro que sirve para almacenar el dato que se ha de suministrar a la memoria o bien el lugar en el que esta entrega el dato que se le solicitó.
- MAR: Memory Address Register. El registro en el que se le indica a memoria la dirección que debe acceder.
- Y. Es el registro en el que se almacena uno de los operandos de la ALU.
- Z. Registro en el que se almacena temporalmente el resultado producido por la ALU.
- MUX. Es un multiplexor que deja pasar a Y o bien a la constante 4 como operando de la ALU.
- Rn. Los registros de propósito general.

Los registros están conectados al canal, algunos para escribir en él, otros para leer de él y otros para ambas cosas según el sentido de la flecha. Las señales de control que se requieren son las siguientes:

- Rn_{in} y Rn_{out} para que el n-ésimo registro sea capaz de recibir como nuevo contenido lo que se encuentre en el canal o escribir su contenido en este, respectivamente.
- Señales de entrada y salida al canal para los registros PC y MDR.
- Señales de entrada para IR, MAR, Y y la ALU.
- Señales de salida para Z y la dirección contenida en la instrucción: $Addr$ (que también puede ser, a pesar de su nombre, un operando inmediato).
- Señales para indicarle a la memoria que lea ($ReadMem$) o que escribe ($WriteMem$) y para que el procesador espere a que la función de memoria se complete ($WMFC$ *wait for Memory Function Complete*).
- Señales para establecer la selección del mutiplexor $YTrue$ o $YFalse$ (equivalente a decir que pasa la constante 4 a la ALU).
- Señales que indiquen a la ALU la operación a realizar: add, sub, mul, and, or, xor, shl, shr etc.

Hacer fetch, por ejemplo consiste en:

1. PC_{out}, MAR_{in}, $ReadMem$
2. $YFalse$, add
3. Z_{out}, PC_{in}, $WMFC$
4. MDR_{out}, IR_{in}

lo que, por cierto, hay que hacer siempre sin importar la instrucción que se ejecute. Llevar a cabo una instrucción como `ADD R6, 320(R2)` (sumar en el registro 6 su contenido previo con el de la dirección de memoria 320 bytes después de la indicada por R2):

5. $R2_{out}$, Y_{in}, $YTrue$; R2 al bus y de allí a Y
6. $Addr_{out}$, add ; 320 al bus, suma, cálculo de la dirección
7. Z_{out}, MAR_{in}, $ReadMem$; lectura del operando en memoria
8. $R6_{out}$, Y_{in}, $YTrue$, $WMFC$; R6 al bus y de allí a Y
9. MDR_{out}, add ; cuando llegue el operando de mem, al bus y sumar
10. Z_{out}, $R6_{in}$; escritura de resultado o write back

En cambio `XOR R7, R3` consiste en:

5. $R3_{out}$, Y_{in}
6. $R7_{out}$, xor
7. Z_{out}, $R7_{in}$

Las señales de control, como puede verse, deben seguir un orden. En esta arquitectura no puede haber más de una transferencia de datos al canal y eso impone necesariamente una secuencia en el acceso al mismo.

Control microprogramado.

Lo primero que nos viene a la mente es entonces poner un área de memoria ROM dentro del CPU. Esa memoria está constituida de palabras de una longitud tal que permite poner todas las señales de control en una palabra. Con este esquema tendríamos entonces una serie de programas, secuencias de señales de control, que al ejecutarse hacen realidad las instrucciones de lenguaje de máquina. Tendríamos entonces un pequeño programa que ejecutar para llevar a cabo un `ADD` con direccionamiento registro-inmediato, por ejemplo, uno para hacer cada posible `LW` dependiendo de el modo de direccionamiento usado, etc.

A esta memoria donde se almacenan las secuencias de señales de control se le conoce como micromemoria y a los programas almacenados allí microprogramas.

Horizontal Vs. vertical.

Hay dos posibilidades para almacenar microprogramas. Una es hacer la palabra de micromemoria tan larga como sea necesario para que quepan, bit a bit, todas las señales de control en cada palabra de la micromemoria. Otra es codificar estas señales para abreviar la expresión de todas las posibles señales. Algo así estamos haciendo al decir que con tres bits podemos decir la operación que la ALU debe llevar a cabo. En principio podríamos, si quisiéramos, poner tantas señales como operaciones tiene la ALU en vez de codificarlas. Si tuviéramos 16 operaciones tenemos la opción de enviar un uno por exactamente una de 16 lineas de un bit, una por cada operación, o bien enviar sólo 4 bits expresando un número entre 0 y 15 y que indica inequívocamente, la operación a realizar.

Si se ponen todas las señales de control sin codificación se dice que el controlador microprogramado tiene *organización horizontal*. Si se codifican las señales de control agrupándolas, entonces se dice que tiene *organización vertical*.En nuestra máquina hipotética de bus único, la organización horizontal de la micromemoria significaría que cada palabra de ella sería una larga cadena de bits¿ uno para IR_{in}, otro para

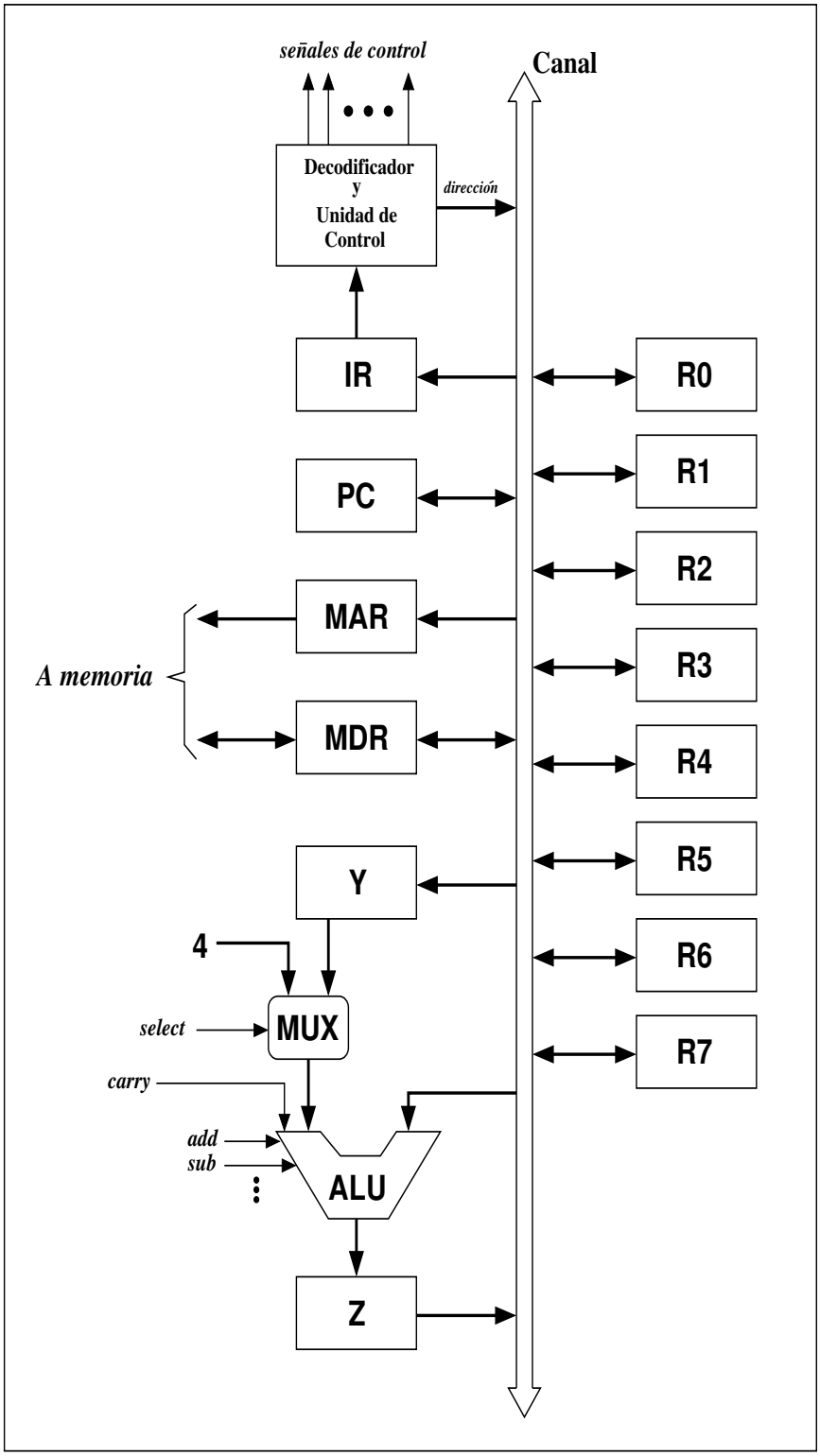

Figura 6.5. La ruta de datos de una arquitectura con un único canal.

PC_{in}, otro para $PCout$, MDR_{in}, MDR_{out}, etc. Esto tiene la ventaja de que ejecutar el microprograma significa tomar esta larga palabra y vaciarla en las líneas de control.

La cantidad de memoria usada en esta alternativa puede ser excesiva, sobre todo si consideramos que hay varias señales que no pueden estar en 1 al mismo tiempo, a saber, todas las *out* son mutuamente excluyentes. Así que podríamos codificarlas. En nuestro caso son 12 señales, así que podríamos usar, en vez de 12 bits, sólo $\lceil \log_2(12) \rceil = 4$ bits.

Por supuesto hay un compromiso. Mucha horizontalidad hace que se necesiten palabras más largas de memoria, es decir, más memoria a fin de cuentas; pero hace trivial el envío de las señales de control, sólo hay que sacar bit por bit la palabra de memoria a las lineas de transmisión de señales. Si posee mucha verticalidad entonces hay que decodificar cada subconjunto de bits y luego traducir eso a las señales sobre las lineas de transmisión, eso es más tardado, pero por supuesto se requiere de menos memoria.

Controladores alambrados (*hardwired*).

Los controladores microprogramados son sólo una posibilidad. Con el advenimiento del paradigma RISC, los conjuntos de instrucciones se han simplificado: poca diversidad de formatos, longitud pequeña y fija, catálogo pequeño, operaciones sencillas, pocos modos de direccionamiento. Esto hace posible que, en vez de tener una micromemoria con una multitud de microprogramas diferentes, se puedan hacer circuitos combinacionales temporizados o secuenciales (máquina de estados finitos) que implementen el envío de señales de control. Es decir, si hay pocas señales, pocas combinaciones válidas y pocas secuencias, como en una máquina RISC, entonces es posible saltarse la microprogramación y hacer un controlador alambrado (*hardwired controller*). Esto, por supuesto, acelera la ejecución de las instrucciones y mejora notablemente el desempeño.

Pensemos por un momento otra vez en nuestra arquitectura de único canal. Si quisieramos construir un circuito combinacional para enviar las señales de control tendríamos que determinar, para cada señal, cuando debemos ponerla en 1. Por lo que sabemos de nuestros ejemplos:

$$Z_{out} = t_3 + t_7 \cdot ADD + t_{10} \cdot ADD + t_7 \cdot XOR + \ldots$$

donde t_i indica que se está en el paso i del microprograma.

Es claro por qué las unidades de contro microprogramadas adquirieron preponderancia sobre las alambradas: en las arquitecturas CISC, en las que el número de modos de direccionamiento y el número de

instrucciones son grandes, una expresión booleana como la de arriba tendría miles de términos, es impensable diseñar los circuitos, aûn cuando cupieran en el área de un procesador de la década de los 70.