Capítulo 5

EL CONJUNTO DE INSTRUCCIONES

Conjunto de instrucciones como interfaz

Hasta hace algún tiempo la arquitectura de computadoras estaba abocada exclusivamente a diseñar el conjunto de instrucciones de una máquina y la manera en que estas instrucciones se implementaban, a lo que se le suele llamar *arquitectura del conjunto de instrucciones* o *Instruction Set Architecture* (ISA). Actualmente el área es mucho más amplia pero el diseño del conjunto de instrucciones sigue siendo parte fundamental de ella. Es fácil ver por qué es tan importante: la arquitectura del conjunto de instrucciones es la parte de la estructura de la computadora, que necesita comprender cabalmente el programador de compiladores para saber como obtener programas correctos y eficientes y, por otra parte, es también el punto de partida del diseñador del hardware, la descripción que el ingeniero debe comprender para desarrollar la implementación correcta de la computadora. La arquitectura del conjunto de instrucciones es lo que está justo entre el hardware y el software, es decir, es una interfaz y como tal debe poseer las cualidades que una buena interfaz posee: proveer de el adecuado nivel de abstracción, la suficiente generalidad y versatilidad para ser útil y permitir una implementación eficiente de los servicios que provee.

Nos dedicaremos ahora a estudiar el diseño del conjunto de instrucciones, las diversas alternativas que existen para ello y las implicaciones que trae consigo cada diseño particular.

5.1 INSTRUCCIONES

Las instrucciones de lenguaje de máquina de una computadora constituyen el catálogo de las operaciones que esa computadora puede hacer y las hay, en general, de cuatro tipos:

- Manipulación de datos y acceso a memoria. Son las que realizan labores como leer o escribir datos desde y hacia la memoria o establecer el contenido de un registro del procesador.
- Instrucciones aritmético-lógicas. Las que, como el nombre lo indica, realizan operaciones aritméticas (suma, resta, multiplicación, corrimientos, inversión de signo, residuo, comparaciones) o lógicas (conjunción, disyunción, disyunción exclusiva, negación).
- Flujo de control. Saltos incondicionales, saltos condicionales, llamadas a subrutinas.
- Punto flotante y cómputo vectorial. Son aquellas cuyos operandos son números en punto flotante, enfocadas a cómputo numérico o a cómputo vectorial, tan usual en el contexto multimedia.
- Instrucciones complejas. Llamadas así genéricamente, constituyen una variopinta colección de instrucciones que realizan labores como mover bloques de un lugar a otro de la memoria o escritura de los registros en una pila o enfocadas a la coordinación de procesos (test and set, por ejemplo) o algunas cosas que bien pudieran considerarse bizarras.

Las instrucciones de lenguaje de máquina poseen cierta estructura a la que suele llamarse *formato de instrucción*. En un conjunto de instrucciones normalmente existen varios formatos en función de otras características de la instrucción: tipo o tamaño de los operandos o manera de accederlos. Claro está que no todos los formatos son iguales ni cuentan con los mismos elementos, pero toda instrucción de lenguaje de máquina tiene un campo para especificar el código que le corresponde a la operación a realizar. A esto se le llama el *código de operación* u *opcode*. Normalmente además cuentan con campos para colocar los operandos o el lugar donde están almacenados. Adicionalmente pueden poseer campos para decir cómo se deben interpretar otros campos: por ejemplo un número cinco en el cuarto campo de una instrucción puede significar que el operando es el número cinco o bien el quinto registro del procesador o que hay que sumar cinco al contenido de un registro para obtener la dirección del operando en memoria. Para saber cómo

interpretar al cinco se puede pensar en incluir un campo que indica cuál de las opciones mencionadas es la verdadera.

5.2 CLASIFICACIÓN DE LA ARQUITECTURA DEL CONJUNTO DE INSTRUCCIONES

Existen varias posibles clasificaciones de los conjuntos de instrucciones porque hay varias características que podemos tomar en consideración y para las que hay varias opciones diferentes.

- De acuerdo con el número de operandos que aparecen en la instrucción.
- De acuerdo con el número de operandos de la instrucción que pueden estar localizados en la memoria.
- De acuerdo con el tamaño y la complejidad del conjunto de instrucciones.
- De acuerdo con la relación existente entre la manera de especificar las operaciones y la manera de decir donde están los operandos.

Por supuesto una clasificación no excluye a otra, es decir, el conjunto de instrucciones de una computadora está ubicado, simultáneamente en varias categorías de acuerdo con los rubros mencionados. A lo largo del capítulo iremos aclarando cada uno de ellos.

De acuerdo con el primer criterio una instrucción puede requerir que se diga explícitamente quienes son los operandos. En una suma, por ejemplo, quienes son los dos sumandos, además de decir donde se debe almacenar el resultado, tres operandos en total, por lo que nuestra operación se podría ver así en lenguaje ensamblador:

```
add r3, r4, r6
```

lo que bien podría significar que en el registro 3 del procesador se debe guardar el resultado de sumar el contenido del registro 4 y el contenido del registro 6. Normalmente al lugar en el que se almacenará el resultado de una operación se le llama el *destino* (*destination*, abreviado como *dst*) y a los operandos propiamente dichos se les llama *fuente* (*source*, abreviado como *src*).

En algunos conjuntos de instrucciones uno de los operandos es también el lugar donde ha de guardarse el resultado. Así por ejemplo, una instrucción como:

```
add r3, r6
```

puede significar: en el registro 3 del procesador guarda el resultado de sumar el contenido previo del propio registro 3 y el contenido del registro 6. En un caso así, el tercer operando, en el que se guarda el resultado, es implícitamente el primero de los operandos que aparecen en la instrucción. Por supuesto tanto en este ejemplo, como en el previo, el operando destino puede ser, no el primero como hemos supuesto, sino el último; la convención adoptada es propia de cada modelo de procesador.

Arquitecturas de acumulador y de pila

Que haya operandos implícitos significa, claro está, que hay cosas que se establecen por omisión. En el ejemplo del párrafo anterior no se pone un tercer registro porque se ha establecido en la arquitectura del conjunto de instrucciones, que el primer operando es a la vez dato de entrada y lugar de destino del resultado. Estos lugares por omisión pueden definirse no sólo para uno de los operandos. Podríamos pensar en una arquitectura en la que tanto un operando de entrada como el destino son un sitio por omisión. En este caso a ese sitio privilegiado, que es un registro del procesador y frecuentemente el único registro del procesador accesible para los programadores, se le denomina *acumulador*: frente a una operación aritmético-lógica sólo aparece un operando, el otro dato de entrada es el contenido actual del acumulador y el resultado se guarda allí también. En un caso aún más extremo podemos pensar en una arquitectura en la que no debe ser especificado ningún operando, tanto los datos de entrada como el lugar de destino del resultado son definidos por omisión. Tal es el caso de las arquitecturas de pila (stack):ambos operandos son los dos últimos datos almacenados en esta y el resultado se coloca como el nuevo tope.

Ahora bien, respecto a los operandos que aparecen explícitamente en la instrucción podemos preguntarnos ¿Cuántos de ellos pueden ser datos almacenados en la memoria? Podemos pensar en una arquitectura como la X86 de Intel, en la que de dos operandos que aparecen explícitamente en la instrucción, uno de ellos, a lo más, puede estar en memoria. Como son dos operandos los que aparecen en la instrucción y sólo uno puede estar alojado en la memoria, se suele llamar *reg-mem* a este tipo de conjuntos de instrucciones. En contraste si simultáneamente ambos operandos pudieran ser estar en memoria, el conjunto se denominaría *mem-mem*.

Han existido casos extremos en la historia de las arquitecturas de conjuntos de instrucciones en los que se podían poner explícitamente los tres operandos en una instrucción y además los tres podían estar alo-

Stack	Acumulador	Registro-Mem	Mem-Mem	load-store
`push A`	`lda A`	`mov R1,A`	`mov C,A`	`load R1,A`
`push B`	`add B`	`add R1,B`	`add C,B`	`load R2,B`
`add`	`sta C`	`mov C,R1`		`add R3,R1,R2`
`pop C`				`store C,R3`

Tabla 5.1. Ejemplos de realización de una suma en diferentes arquitecturas de conjunto de instrucciones. Se supone que A, B y C son direcciones de memoria en las que se almacenan los operandos.

jados en la memoria. Es el caso de las máquinas de Digital Equipment Corporation (DEC), introducidas a mediados de los 70, con conjunto de instrucciones VAX (*Virtual Address eXtension*). Una instrucción válida en una VAX-11/780 podría ser:

```
addl3 (r2), 230(r3), (r4)[r1]
```

lo que literalmente significa: guarda en la dirección de memoria contenida en el registro r2, el resultado de la suma del contenido de la dirección de memoria que está 230 bytes más abajo de la apuntada por r3 y el de la que resulta de sumar los registros r4 y r1.

Arquitectura *load-store*

El caso extremo contrario es también posible, se puede establecer que toda instrucción aritmético-lógica debe operar sólo con operandos guardados en registros y para traer y llevar los datos desde y hacia la memoria, respectivamente, se proveen de sendas instrucciones conocidas como *load* (carga) y *store* (almacenamiento). En una arquitectura de carga-almacenamiento estas dos instrucciones son las únicas que pueden acceder a la memoria.

En la tabla 5.1 se muestran ejemplos de como se vería la realización de una instrucción suma en máquinas con diferentes tipos de arquitectura de conjunto de instrucciones.

Las máquinas de *stack* han desaparecido, ejemplos de ellas eran las Burroughs B5000 o B6500 y los coprocesadores de punto flotante de Intel. Las arquitecturas de acumulador fueron muy populares al comienzo de la era de las computadoras personales, por ejemplo Mostek 6502, el procesador de 8 bits usado por las Apple II y Commodore 16 y 64. Las únicas arquitecturas conocidas de tres operandos, en las que incluso los tres podían ser direcciones de memoria fueron algunas VAX. La arquitectura de Intel X86 se suele denominar "de acumulador extendida": hay registros que, en principio sirven para todo o casi todo,

pero están mejor provistos para hacer ciertas operaciones e incluso hay operaciones que requieren de ciertos registros para hacerse (por ejemplo el contador de ciclos siempre debía estar en un registro llamado CX).

¿Por qué usar registros?

Nuestras computadoras actuales tienen siempre registros y, de hecho, suelen ser más bien del tipo *load-store*. Esto tiene su razón de ser. Dado que los registros están en el procesador mismo, la velocidad de acceso a ellos es muy superior a la de acceso a memoria, así que almacenar datos en registros favorece el desempeño. Además el "nombre" de un registro es una cadena de unos cuantos bits, mientras que, en general, especificar una dirección de memoria requiere muchos más. Esto además de ser ventajoso por el hecho de hacer más breves las instrucciones, nos lleva a una consideración adicional: si se permite que las instrucciones aritmético-lógicas accedan a memoria entonces habrá una versión de suma, por ejemplo, que opera sólo con registros y otra que opera con uno operando en memoria y el otro en un registro y posiblemente otra más con ambos operandos en memoria, todas ellas de longitudes y formatos diferentes, lo que le hará la vida difícil a la unidad de control para decodificar las instrucciones. Usar registros es bueno porque minimiza los tamaños de las instrucciones, permite acceder más rápido a los operandos y, si además sólo permitimos operar con registros, contribuimos a uniformar los tamaños y los formatos de las instrucciones lo que facilita la tarea de decodificarlas.

No siempre ha sido así

Pero entonces ¿Por qué ha habido en el pasado conjuntos de instrucciones como el de VAX con tres operandos en memoria y todas las variantes de registros y memoria posibles? Ciertamente no es porque los arquitectos de hardware del pasado no se hayan dado cuenta de las ventajas de los registros y el *load-store* es que esas ventajas no estaban allí entonces. Recordemos que antes de los 80 del siglo pasado las memorias eran caras y muy lentas y que la prioridad de cualquier arquitecto de hardware era lograr que el programador de lenguaje ensamblador tuviera la vida fácil para lograr programas breves que cupieran en la mínima cantidad de memoria posible. Para este propósito, un conjunto de instrucciones como VAX, es mejor que uno del estilo *load-store*.

5.3 TIPO Y TAMAÑO DE LOS OPERANDOS

Ha llegado el momento de hacer explícito algo que hemos estado suponiendo desde hace un rato. Que las instrucciones de lenguaje de máquina son cadenas de ceros y unos con cierta estructura. Una instrucción está entonces dividida en campos de tamaño diferente y cada

Nombre	Tamaño en bits	Ejemplo en C
Byte	8	char
Halfword	16	short
Word	32	int, float
Doubleword	64	long, double

Tabla 5.2. Los diferentes tamaños de operandos.

campo sirve para especificar ciertas características de la operación que se desea ejecutar. Ya nos hemos referido al código de operación, este es un número que identifica a cada instrucción del repertorio del procesador. Bien pudiera ser que $A8_{16}$ fuera "suma" (`add`) y $B8_{16}$ fuera "resta" (`sub`), por ejemplo. El resto de los campos sirven para identificar a los operandos y en general es posible que existen campos para especificar el tipo de los operandos y el modo de acceder a ellos.

Tener la capacidad de especificar el tipo de los operandos, en particular, es relevante porque aparejado con el tipo hay un tamaño, una longitud determinada de los operandos, lo que es necesario saber cuando se traen de memoria o se transfieren a ella. En general suele incluirse en el código de la instrucción (el *opcode*) el tipo de los operandos con los que se llevará a cabo. Así pues $A8$ podría ser el código de suma (como dijimos antes) pero con enteros de 8 bits, mientras que $A9$ es el de suma con enteros de 16 bits y AA el de suma con 32 bits. Podríamos preguntarnos ahora, ¿a cuáles de estas operaciones les damos soporte en el hardware?, es decir, si el hardware que diseñamos puede hacer "de un tirón" una suma, ¿de qué tamaño hacemos los registros que se suman para garantizar el mejor desempeño. Si nuestra ALU es de 8 bits entonces hacer una suma de 32 bits nos tomará trabajo adicional. Si, en otro caso, estamos preparados para manipular datos de 32 bits y deseamos traer de memoria un operando de 8 bits, estaremos, igual que antes, haciendo trabajo de más para quedarnos sólo con lo necesario.

De acuerdo con la ley de Amdahl debemos diseñar el hardware adecuado al tamaño de operando más usual: *hacer mejor lo más común*. En adelante usaremos la nomenclatura mostrada en la tabla 5.2. Por otra parte en la figura 5.1 se muestra la frecuencia de uso de los diferentes tamaños de datos. De acuerdo con esto deberíamos diseñar teniendo en mente operandos de 32 bits (word).

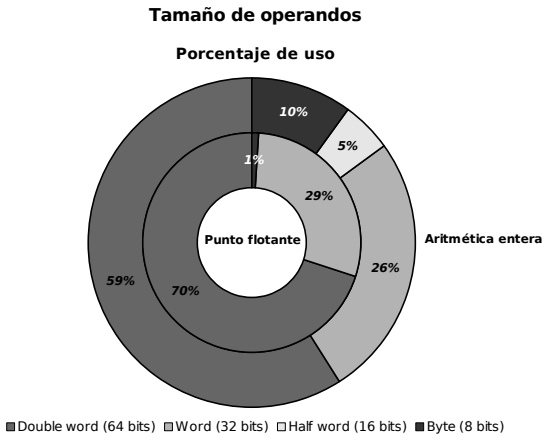

Figura 5.1. Frecuencia de uso de los distintos tamaños de datos.

Nomenclatura

En la tabla se denomina "byte" a un número de 8 bits. A pesar de que puede parecer evidente hoy día, no siempre ha sido así. Los ocho bits (por lo que se le llama a veces y para evitar ambigüedad, *octeto*). En realidad el término se utilizó para designar a la cantidad de bits necesarios para representar un caracter y era dependiente de la implementación. Había máquinas con bytes de 6 bits o de 7. Comúnmente se llama *nibble* a la mitad de un byte o, formalmente, una cadena de 4 bits. Al bit en el extremo izquierdo de un número le llamaremos *más significativo* ya que tendría asociada la mayor potencia de 2 de acuerdo con la representación posicional, por el motivo contrario llamamos *menos significativo* al bit del extremo derecho. Cuando hablemos de tipos de datos de varios bytes de tamaño usaremos una nomenclatura similar llamando byte más significativo al que tenga asociadas las potencias de dos mayores y byte menos significativo al del extremo contrario.

Una anarquía similar a la de los bytes ocurría con los tipos de datos usados para almacenar números en punto flotante. Cada fabricante tenía su propia representación hasta que en 1985 finalmente el IEEE estableció el estándar 754 que es usado por todo mundo hoy día y del que provienen nuestros tipos de precisión simple (como el *float* de C) y de precisión doble (como el *double*).

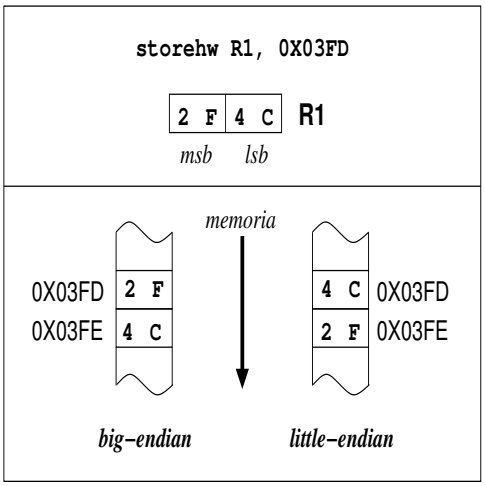

Figura 5.2. Los dos esquemas de orden de almacenamiento para operandos de tamaño mayor a un byte.

Orden de los bytes

El byte es la unidad de mínima de memoria direccionable, es decir, cada celda de memoria tiene capacidad para almacenar exactamente un byte y posee su propio dirección, una manera de identificarla unívocamente. Si se utilizan operandos de más de un byte, entonces otra de las cosas que hay que decidir es como serán acomodados los bytes del operando en memoria. Así que hay esencialmente dos maneras diferentes de acomodar los bytes, a saber: colocando en la dirección de memoria más baja el byte menos significativo del dato (*little endian*) y la complementaria, que en la dirección de memoria más baja coloca el byte más significativo del dato (*big endian*[1]).

little-endian y big-endian

Supongamos que tenemos una máquina *little endian* con registros de 16 bits. Si una instrucción dice: `storehw R1, 0X03FD` que indica que se almacenará el contenido de los 16 bits (media palabra) del registro R1 en la dirección 03FD en hexadecimal, esto significa que el byte menos significativo de R1 se guarda en 0X3FD y el siguiente byte más significativo en 0X3FE. Si en cambio la máquina fuera *big endian*, entonces el byte menos significativo se almacenaría en 0X3FE y el siguiente más significativo en 0X3FD. En la figura 5.2 se muestran las dos alternativas de orden de almacenamiento.

Aparentemente nadie nota el orden de las cosas y a nadie le importa. En realidad esto se transforma en un dolor de cabeza cuando se pre-

[1]Los términos *little endian* y *big endian* provienen de los viajes de Guliver de Swift, corresponden a una clasificación de acuerdo a el lado del huevo que se rompe para comerlo, el ancho o el angosto

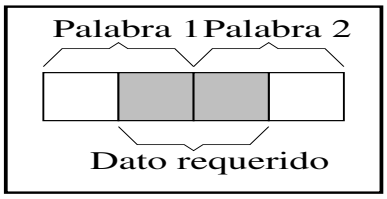

Figura 5.3. Esquema de un acceso no alineado. Se requiere una palabra no alineada que posee sus dos mitades en palabras distintas de una memoria alineada en palabras, el primer acceso hace que se deba desechar la mitad inferior de la palabra accedida, el segundo acceso hace que se deba desechar la mitad más significativa de la segunda palabra accedida.

tende intercambiar información entre diferentes máquinas con diferente convención.

Alineamiento

También en el caso de operandos mayores de un byte es necesario decidir si poner restricciones de alineamiento o no. Las restricciones de alineamiento son aquellas que especifican que un operando de tamaño k bytes debe empezar en una dirección múltiplo de k. Es decir un objeto de tamaño k está alineado en la dirección A si y sólo si

$$A \equiv 0 \pmod{k}$$

Así por ejemplo una palabra (4 bytes) está alineada si comienza en una dirección múltiplo de 4, es decir: $0, 4, 8, 12, \ldots$ o equivalentemente todas las direcciones que tienen ceros en los dos bits menos significativos.

¿Por qué establecer estas restricciones? básicamente por dos razones: los datos no alineados hacen más lentas las operaciones y porque proveer a las máquinas de lo necesario para que no sea relevante la alineación es una complicación de hardware.

Las memorias vienen alineadas típicamente en palabras o dobles palabras así que un acceso no alineado ocasiona múltiples accesos alineados, que son los naturales.

Aun si los datos están alineados no es trivial el acceso a datos de longitud menor al tamaño de palabra de memoria. Si una máquina soporta acceso a bytes o medias palabras requiere de hardware extra. Por ejemplo supóngase que se desea acceder a un byte en la memoria que contiene un $FF = -1$ y que se desea almacenar en un registro del procesador cuya longitud es de una palabra para hacer una operación aritmética. Necesitamos que los ocho bits de memoria se copien con todo y signo al registro de 32 bits que debe resultar con $FFFFFFFF$.

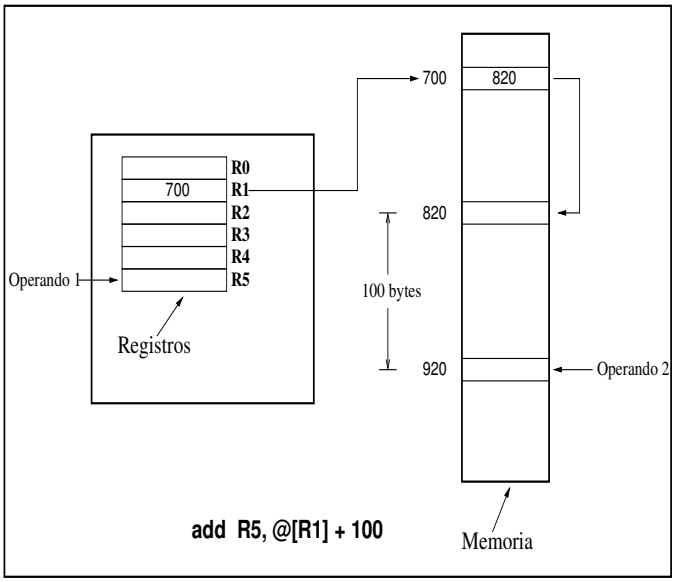

Figura 5.4. Direccionamiento de memoria indirecto con desplazamiento.

5.4 MODOS DE DIRECCIONAMIENTO

Cuando ponemos una operación con sus operandos ¿como especificamos los operandos? ¿como decimos donde exactamente está el operando? A esto es a lo que se refieren los modos de direccionamiento justamente.

Ventaja de usar modos de direccionamiento

La ventaja de utilizar modos de direccionamiento es que en una sola instrucción es posible resumir varios cálculos para determinar la dirección efectiva de un operando. Por ejemplo, si un operando de una suma es el contenido del registro R5 de un procesador de arquitectura *load-store* y el otro está en la dirección de memoria que resulta de sumarle 100 al contenido de la dirección de memoria guardada en R1 (véase figura 5.4), entonces para hacer la suma tendríamos que hacer algo como esto:

```
load R2, [R1]  ; carga en R2 el contenido de la dirección de
               ; memoria apuntada por R1
add  R2, 100   ; suma R2 con 100
load R4, [R2]  ; carga en R4 el operando
add  R5, R4    ; realiza la suma finalmente
```

Pero en alguna otra máquina podría existir una instrucción para acceder al operando que está especificado indirectamente y podría verse tal como se muestra en la figura 5.4.

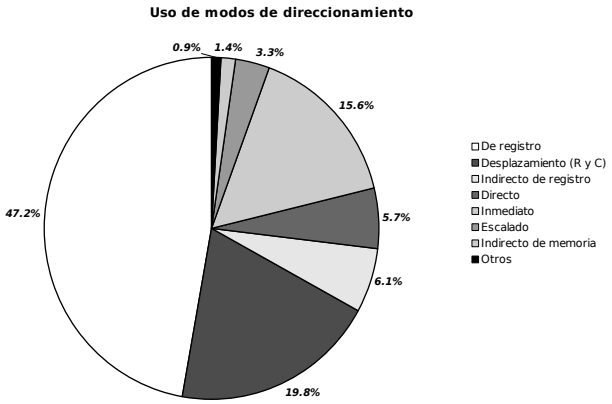

Figura 5.5. Frecuencia de uso de los diferentes modos de direccionamiento.

Así que, en general el usar modos de direccionamiento diferentes reduce el contador de instrucciones (IC) por programa, lo que, como vimos en el capítulo anterior, tiene un impacto favorable en el desempeño.

Pero todo cuesta, el tener modos de direccionamiento diferentes aumenta la complejidad de las instrucciones (a fin de cuentas sí hay que hacer lo que hicimos a pie en el código anterior) y aumenta la dificultad de decodificación. Nuevamente tenemos, como ya es costumbre, un compromiso.

Para decidir que modos de direccionamiento incluir en una arquitectura nos ayuda saber cuáles modos son lo más usuales. Así sólo incluimos aquellos que sean de indispensables a muy usados y desechar los que raramente se usan.

Modos de direccionamiento

Los diferentes modos de direccionamiento con sus frecuencias aproximadas son (véase figura 5.5):

De registro 47.2%.

Desplazamiento 31.6%. Se consideran aquí tres casos particulares:

Desplazamiento (propiamente dicho, tanto el registro como la constante de direccionamiento son distintos de cero). 62.7% relativo a este modo, 19.8% respecto al total.

Modo	Ejemplo	Significado	Comentarios
Registro	`add R4, R3`	regs[4] += regs[3]	Operando en un registro
Inmediato	`add R4, #3`	regs[4] += 3	Operando constante
Desplazamiento	`add R4, 100(R1)`	regs[4] += mem[100+regs[1]]	Acceso a variables locales
Indirecto	`add R4, (R1)`	regs[4] += mem[regs[1]]	El contenido de R1 es la dirección de la celda donde está el operando
Indexado	`add R4, (R1+R2)`	regs[4] += mem[regs[1]+regs[2]]	Arreglos: R1=inicio, R2 = índice (tamaño=1 byte)
Directo	`add R4, (1001)`	regs[4] += mem[1001]	Datos estáticos
Indirecto de memoria	`add R4, @(R3)`	regs[4] += mem[mem[regs[3]]]	R3 contiene la dirección de un apuntador
Autoincremento	`add R4, (R2)+`	regs[4] += mem[regs[2]]; regs[2] += d	Arreglos: cada elemento mide d bytes
Autodecremento	`add R4, -(R2)`	regs[2] -= d; regs[4] += mem[regs[2]]	Arreglos: cada elemento mide d bytes
Escalado	`add R4, 100(R2)[R3]`	regs[4] += mem[100+regs[2]+regs[3]*d]	Arreglos bidimensionales

Tabla 5.3. Modos de direccionamiento.

Indirecto de registro 19.4% relativo a este modo, 6.1% respecto al total.

Directo (17.9% relativo a este modo, 5.7% respecto al total.

Inmediato 15.6%.

Escalado 3.3%.

Indirecto de memoria 1.4%.

Otros 0.9%.

Por supuesto todos los modos presentes en la tabla 5.3 son para direccionar datos. Direccionar código (transferencias de control) es algo que se verá más adelante.

Modos de direccionamiento más usuales

Al parecer el modo de direccionamiento más usual en el que realmente se accede a memoria, es el de desplazamiento, que toma el contenido de un registro y lo suma con una constante para obtener la dirección efectiva de un operando. Ahora bien ¿de que tamaño debe ser el rango de la constante de desplazamiento? a fin de cuentas esta constante debe aparecer en la instrucción así que es recomendable saber de que tamaño debe ser el campo de la instrucción asociado al desplazamiento.

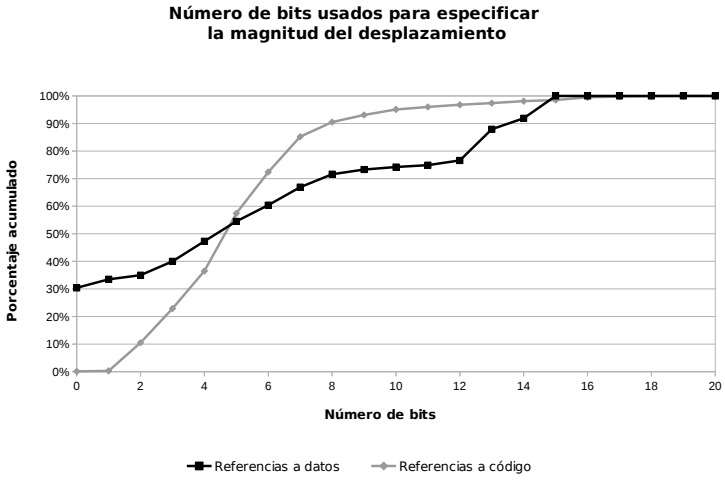

Figura 5.6. Tamaño del desplazamiento en accesos a memoria por datos (es decir, el tamaño en bits de la constante usada en el direccionamiento directo o en el de desplazamiento) y por acceso a código (es decir al especificar la distancia de salto).

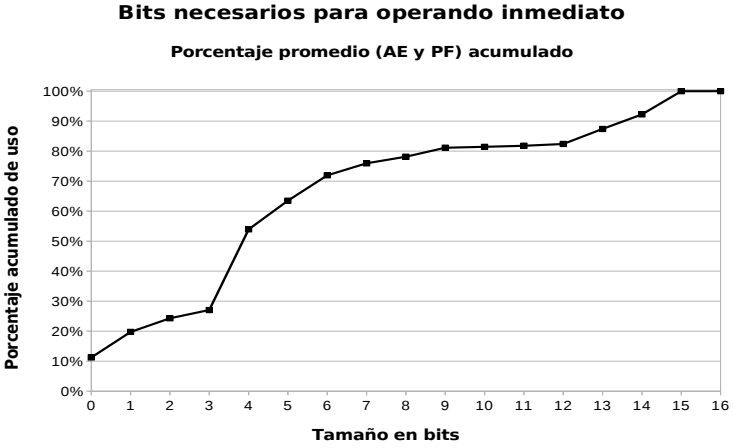

Figura 5.7. Tamaño en bits del operando inmediato.

En la figura figura 5.6 se muestra la frecuencia relativa acumulada del tamaño en bits del desplazamiento. En las referencias a datos vemos que con 16 bits están cubiertos prácticamente todos los casos.

El siguiente modo más usado es el inmediato, en el que el operando aparece en la instrucción. ¿Que tipo de instrucciones usan este modo de direccionamiento? se utiliza generalmente en comparaciones, en operaciones aritméticas con constantes y en operaciones de carga (para establecer con una constante el contenido de un registro). Cuando la constante es usada para operaciones en general es pequeña, cuando será utilizada para especificar direcciones de memoria en el futuro, tiende a ser grande. En el libro ya mencionado se observa que en general las instrucciones de aritmética entera usan este direccionamiento un 35% del tiempo mientras que las de punto flotante los usan un 10%. Las instrucciones que más lo usan son las comparaciones (87% aritmética entera, 77% punto flotante) y las operaciones de ALU (58% y 78% aritmética entera y punto flotante respectivamente).

Debemos preguntarnos ahora ¿de que tamaño (en bits) debe ser el campo de la instrucción asociado al operando? En la figura 5.7 se observa la "distribución acumulativa" para esta "variable aleatoria" tomando como muestra los programas de la *suite* de SPEC. Nuevamente con 16 bits están cubiertos la gran mayoría de los casos.

Conjuntos de instrucciones ortogonales

Cuando en un conjunto de instrucciones el código de instrucción cambia cuando lo hace el modo de direccionamiento se dice que el con-

junto de instrucciones es *no ortogonal*. Por supuesto si el opcode es independiente del modo de direccionamiento el conjunto se denomina *ortogonal*. Otra de las cosas que debe decidir el diseñador de un procesador es, claro, si el conjunto de instrucciones debe o no ser ortogonal. ¿De qué depende esta decisión?, ¿cuál es la óptima? Analicemos. Supongamos que n es el número total de diferentes instrucciones que queremos que la máquina posea y que m es el número de modos de direccionamiento diferentes. Pensemos en lo que habría que hacer a un nivel muy abstracto para decodificar una instrucción que accede a un operando en memoria, supongamos, por ejemplo, lo que habría que hacer al programar un emulador de nuestra arquitectura en lenguaje C.

Si el conjunto de instrucciones es ortogonal entonces, decodificar (determinar qué operación se debe hacer y obtener los operandos) lo podemos hacer con un par de construcciones *switch*: el primero verificando en qué caso, de los n posibles, se debe caer dado el opcode; el segundo estableciendo en qué caso de los m posibles modos de direccionamiento se debe caer. El resultado se ejecuta revisando los primeros n casos y los segundos m casos, es decir, resolvemos en $n + m$ verificaciones a lo más. Si en cambio el conjunto es no ortogonal, entonces habría que entrar a un *switch* con $n \cdot m$ casos. Si hay 256 instrucciones posibles y 16 modos de direccionamiento, entonces, en el primer caso habría que hacer $256 + 16 = 272$ verificaciones, en el segundo caso habría que hacer: $256 \cdot 16 = 4096$ verificaciones, obviamente la opción ortogonal resulta conveniente. Bueno, claro, si consideramos que están uniformemente distribuidas las probabilidades de uso tanto de las instrucciones como de los modos de direccionamiento; si realmente sólo se usan 32 instrucciones y tres modos de direccionamiento, entonces la mayor parte del tiempo estoy revisando condiciones inútiles que nunca se cumplen.

Si pensamos en cambio en una máquina *load-store* con sólo 3 modos de direccionamiento entonces el valor de n efectivo para considerar en el switch es 2 (load o store) y el de m es 3, sólo habría que verificar 6 condiciones diferentes. La no-ortogonalidad no luce mal.

5.5 INSTRUCCIONES DE CONTROL DE FLUJO

Estadísticamente las frecuencias de los distintos tipos de instrucciones de control de flujo son las siguientes:

- Saltos condicionales de 81 a 86% de las veces.

- Llamadas a subrutinas de un 11 a 13%.
- Saltos incondicionales de un 4 a 6%.

Además resulta más económico poner, en una instrucción de salto un desplazamiento relativo a la instrucción actual (mejor dicho la que sería la siguiente en caso de no saltar) que poner la dirección absoluta de memoria de la instrucción a la que se salta. Porque en general las distancias de salto son expresables en pocos bits (figura 5.6). Aproximadamente un 50% de los saltos son a distancias menores de las expresables en 6 bits, con 16 basta y sobra para expresar casi cualquier salto.

Tamaño de salto

Del mismo modo, en aritmética entera el 86% de los saltos son usando la condición igual o no igual, el restante 14% se divide en partes iguales entre dos categorías: (a) menor que, mayor o igual que y (b) mayor que, menor o igual que. En aritmética de punto flotante (a) tiene el 40% del uso, el 23% para (b) y el 37% para igual o no igual.

5.6 FORMATOS DE INSTRUCCIÓN DE DLX

Registros en DLX

Habiendo hecho el análisis anterior podemos diseñar el conjunto de instrucciones de una computadora ficticia. Usaremos la arquitectura DLX del libro de Hennessy y Patterson[2] a su vez basada en la de MIPS.

Nuestra máquina tendrá 32 registros que llamaremos R0, R0, ..., R31. 32 es una buena cantidad de registros. Además estos registros serán de propósito general, lo que como hemos visto, gusta a los programadores de compiladores, salvo R0 que siempre contendrá la constante cero, lo que resultara conveniente.

Registros de punto flotante

Adicionalmente incluiremos 32 registros de propósito general pero de uso exclusivo de una unidad de punto flotante. Será posible aparearlos de tal forma que un registro par y uno impar formen uno de 64 bits para números de punto flotante de doble precisión. Los llamaremos F0, ... F31, cuando estén apareados sólo nos referiremos a los pares.

Modos de direccionamiento en DLX

Dado nuestro análisis de modos de direccionamiento sólo proveeremos de dos modos, inmediato y de desplazamiento, ambos con campos de 16 bits. Nótese que los modos de direccionamiento indirecto de registro y directo son casos particulares del modo de desplazamiento. En el modo de direccionamiento de desplazamiento se suma una constante al contenido de un registro y eso es usado como la dirección del ope-

[2]Hennessy, J. y D. Patterson *Computer architecture: A quantitative approach*, Morgan Kaufmann, 1990. Desde esta primera edición de la obra se usó la arquitectura DLX. Este libro se ha seguido actualizando a lo largo del tiempo en 2011 la edición más reciente es la quinta.

rando. Así que si la constante es cero la dirección del operando es el contenido del registro (direccionamiento indirecto de registro). Si en cambio el registro tiene cero (para eso esta R0), entonces el operando está en la dirección de memoria indicada por la constante (direccionamiento directo). Así que con sólo proveer de dos modos, obtenemos otros dos gratuitamente

Además nuestra máquina será del tipo *load-store*, las únicas instrucciones que pueden acceder a memoria son `load` y `store`, las demás están obligadas a usar los registros para los operandos o a ponerlos inmediatos. Será de tres "direcciones" es decir en cada instrucción se dicen explícitamente los operandos y el lugar para guardar el resultado.

Formatos de instrucción en DLX

Los formatos de las instrucciones se pueden ver en la figura 5.8. cada formato será usado como se describe a continuación:

Tipo I Para `load` y `store` de bytes, halfword y word. Para todas las instrucciones con operando inmediato (rd = rs1 op inmediato). Instrucciones de salto condicional, rs1 es el registro cuyo valor decide el salto, rd no se usa.

Tipo R Para instrucciones de la ALU registro-registro rd = rs1 func rs2. ¿Y entonces para que sirve el código de operación? en este caso el opcode sólo dice que es una operación de ALU[3].

Tipo J Saltos

5.6.1 Ejemplos

Cuando aparece un número como subíndice de un "=" representa el número de bits transferidos. Cuando el subíndice aparece en una expresión denota el o los bits que intervienen en la operación, 0 es el más significativo. Cuando aparece un número como superíndice de una expresión, denota cuantos bits de la expresión son establecidos, comenzando con el más significativo. el signo $\uplus$ denota yuxtaposición o concatenación de las cadenas binarias que intervienen en la operación.

Tipo I.

1. `LW R7, 120(R9)`
   $\mathsf{regs}[7] = \mathsf{mem}[120 + \mathsf{regs}[9]] \uplus \mathsf{mem}[121 + \mathsf{regs}[9]] \uplus$

[3]En MIPS las instrucciones tipo R, si tienen el opcode igual a 0 significa que esa operación corresponde a la ALU. Eso hubiera sido bueno hacerlo con todas las de la ALU, pero las que tenían operandos inmediatos no cabían en 32 bits, así que esas tuvieron que codificarse de otro modo.

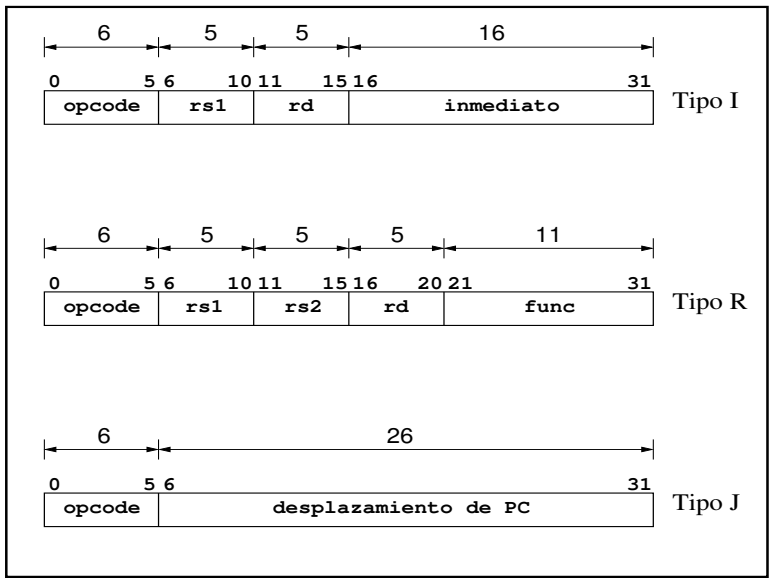

Figura 5.8. Formatos de instrucción de DLX.

$\mathsf{mem}[122 + \mathsf{regs}[9]] \uplus \mathsf{mem}[123 + \mathsf{regs}[9]]$
`rs1=9 (R9)`, `rd=7 (R7)`, `inmediato=120`
direccionamiento de desplazamiento (Fig. 5.9). Se toman 4 bytes a partir de la dirección especificada.

2. `LB R1, 1000(R2)`
   $\mathsf{regs}[1] = \mathsf{mem}[1000 + \mathsf{regs}[2]]_0^{24} \uplus \mathsf{mem}[1000 + \mathsf{regs}[2]]$
   `rs1=2 (R2)`, `rd=1 (R1)`, `inmediato=1000`
   direccionamiento de desplazamiento. Se toma sólo el byte en la dirección especificada.

3. `LH R1, 1000(R2)`
   $\mathsf{regs}[1] = \mathsf{mem}[1000 + \mathsf{regs}[2]]_0^{16} \uplus \mathsf{mem}[1000 + \mathsf{regs}[2]] \uplus$
   $\mathsf{mem}[1001 + \mathsf{regs}[2]]$
   `rs1=2 (R2)`, `rd=1 (R1)`, `inmediato=1000`
   direccionamiento de desplazamiento.

4. `LW R1, 120(R0)`
   $\mathsf{regs}[1] = \mathsf{mem}[120] \uplus \mathsf{mem}[121] \uplus \mathsf{mem}[122] \uplus \mathsf{mem}[123]$
   `rs1=0 (R0)`, `rd=1 (R1)`, `inmediato=120`
   dado que `R0 = 0` esto es direccionamiento directo o absuluto.

5. `LW R1, (R6)`
   $\mathsf{regs}[1] = \mathsf{mem}[\mathsf{regs}[6]] \uplus \mathsf{mem}[\mathsf{regs}[6] + 1] \uplus$
   $\mathsf{mem}[\mathsf{regs}[6] + 2] \uplus \mathsf{mem}[\mathsf{regs}[6] + 3]$

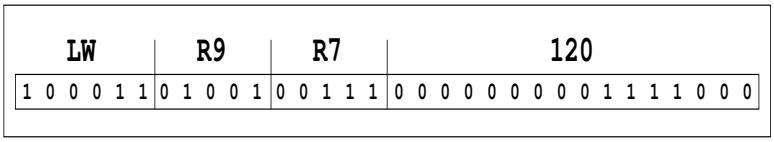

Figura 5.9. Ejemplo de instrucción en formato I: `LW R7, 120(R9)`. En hexadecimal, el opcode de LW es 23.

`rs1=6 (R6)`, `rd=1 (R1)`, `inmediato=0`
direccionamiento indirecto o diferido con registro.

6. `SH R3, 512(R2)`
   $\mathsf{mem}[512 + \mathsf{regs}[2]] =_{16} \mathsf{regs}[3]_{16,\ldots,31}$
   `rs1=2 (R2)`, `rd=3 (R3)`, `inmediato=512`
   direccionamiento de desplazamiento.

7. `ADD R1, R2, 320`
   $\mathsf{regs}[1] = \mathsf{regs}[2] + 320$
   `rs1=2 (R2)`, `rd=1 (R1)`, `inmediato=320`
   Direccionamiento inmediato, el `opcode` especifica suma (suma con inmediato).

8. `JR R4`
   $\mathsf{PC} = \mathsf{regs}[4]$
   `rs1=4 (R4)`, `rd=inmediato=0`.

9. `BEQZ R4, etiqueta`
   Si $\mathsf{regs}[4] = 0$ entonces
   $\mathsf{PC} = \mathsf{PC} + 4 + \mathsf{distancia}(\mathsf{PC} + 4, \mathsf{etiqueta})$
   si no $\mathsf{PC} = \mathsf{PC} + 4$
   `rs1=4 (R4)` al comparador en vez de a la ALU,
   `rd=0`, `inmediato=distancia(PC+4, etiqueta)`.

10. `JALR R2`, "Jump and Link Register"
    $\mathsf{regs}[31] = \mathsf{PC} + 4$, $\mathsf{PC} = \mathsf{regs}[2]$
    `rs1=2 (R2)`, `rd=0`, `inmediato=0`
    nótese que `R31` se usa implícitamente como dirección de retorno.

Tipo R.

1. `ADD R2, R3, R1`
   $\mathsf{regs}[2] = \mathsf{regs}[3] + \mathsf{regs}[1]$
   `rd=2 (R2)`, `rs1=3 (R3)`, `rs2=1 (R1)`

direccionamiento de registro, el campo `func` especifica la operación a realizar, en este caso el `opcode` es cero (MIPS), significa operación de ALU con registros.

Tipo J.

1. `J etiqueta`
   $\mathsf{PC} = \mathsf{PC} + 4 + \mathsf{distancia}(\mathsf{PC} + 4, \mathsf{etiqueta})$
   `desplazamiento=distancia(PC+4,etiqueta)`
   salto incondicional, en MIPS se utilizan estos 26 bits para pegarlos a los 4 bits más significativos de PC+4 y pegar luego dos bits menos significativos en cero (i.e. las direcciones de salto, de hecho todas las instrucciones, están alineadas, las direcciones son múltiplos de 4).

2. `JAL etiqueta`
   $\mathsf{regs}[31] = \mathsf{PC} + 4$, $\mathsf{PC} = \mathsf{PC} + 4 + \mathsf{distancia}(\mathsf{PC} + 4, \mathsf{etiqueta})$
   `desplazamiento=distancia (PC+4,etiqueta)`
   llamada a subrutina.