

8.

La memoria: estructura

En este capítulo se describe la interacción entre el procesador y la memoria a nivel estructural. Es decir, se estudia la forma como se construye la memoria del computador —no los chips—, así como los métodos utilizados para comunicarla y sincronizarla con el procesador. Se empieza por ver la forma de interconectar componentes, para lo cual se introduce el concepto de bus; en seguida, se pasa al caso concreto del procesador y la memoria.

8.1 CHIPS Y BUSES

Los chips¹ se comunican con el mundo exterior por medio de pines o algún otro tipo de contacto. Estos contactos se encargan de transmitir las señales eléctricas al exterior, o de recibirlas del mismo.

Los chips se comunican entre ellos interconectando los contactos de uno con los correspondientes del otro. Es conveniente estructurar esta conexión de alguna manera, de lo contrario se tendría una maraña de conexiones. Una forma de hacerlo es por medio de *buses*; estos son conjuntos de líneas, cada una de las cuales

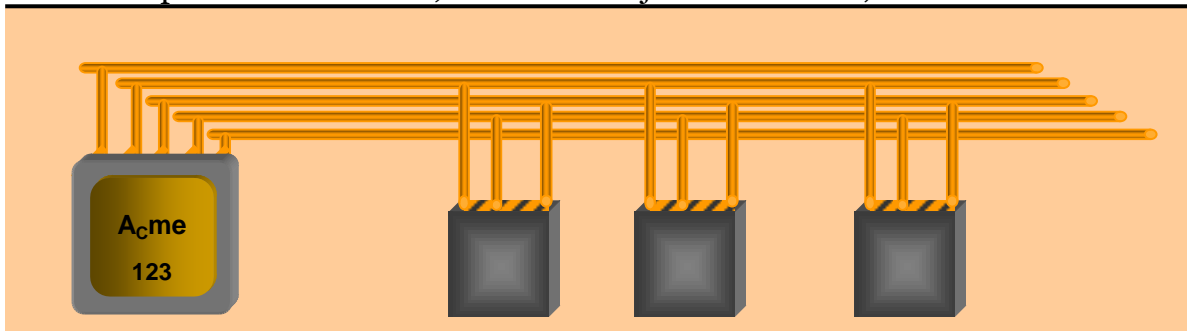


Fig. 8.1. Conexión de chips por medio de un bus

¹ El uso del termino "chip" es abusivo. Estrictamente hablando, "chip" es la pastilla de semiconductor y no el portador en sí. De hecho, hay varios tipos de portadores: DIP, LCC, PGA. Sin embargo, chip se suele utilizar informalmente para designar el conjunto chip-portador.

tiene una función determinada. Los chips se conectan con las líneas que necesitan (fig. 8.1).

Hay cuatro clases de líneas en un bus: *datos*, *direcciones*, *control* y *alimentación*. Es clara la función de las líneas de datos y de direcciones, su nombre lo dice todo. En cuanto a las de control, corresponden a líneas por medio de las cuales se envían órdenes o señales de sincronización. Entre dichas señales de sincronización se puede contar la señal de reloj, la cual sirve para establecer una referencia temporal que permita sincronizar los diferentes componentes del computador. El cuarto grupo de líneas corresponde a las de alimentación y tierra.

Un bus es más que un conjunto de cables, y esto tanto en un sentido físico como lógico. En el sentido físico, porque se requieren varios componentes electrónicos, por ejemplo, para evitar rebotes de la señales al final del cable (*terminadores*). En el sentido lógico, porque es necesario establecer una serie de convenciones para usarlo. Dichas convenciones van desde la forma de usar el bus (para qué sirve cada señal, cómo y cuándo se usa) hasta la forma y tamaño que deben tener los conectores usados por los periféricos. Además, se incluyen restricciones de tiempo del estilo: "la señal A debe ponerse en 1 después de la señal B, y debe hacerlo después de que hayan transcurrido t_{\min} microsegundos pero antes de que trascurren t_{\max} microsegundos".

Dos características importantes de un bus son su *tamaño* y su *velocidad de transferencia*. El tamaño es el número de líneas que componen el bus. La velocidad es la cantidad de bits por unidad de tiempo que el bus está en capacidad de transmitir.

8.2 INTERACCIÓN DEL PROCESADOR Y LA MEMORIA

En la discusión que sigue, vamos a suponer que todos los chips se conectan al bus del sistema, el cual estará compuesto por las líneas antes descritas.

El procesador tiene contactos que corresponden a los cuatro tipos de líneas. Así, un procesador con palabra de 8 bits necesita 8 contactos para conectarse a las líneas de datos, y, si tiene direcciones de 16 bits, debe reservar 16 contactos para éstas.

Los contactos de control varían de procesador a procesador, entre ellos podríamos contar el reloj.² Las líneas de alimentación suelen ser uno o más voltajes —fig. 8.2(a)—.

En el caso, imaginario, de la figura 8.2(a), tenemos un procesador con un contacto para tierra (GND) y uno para el voltaje de alimentación (Vcc); todos ellos se conectan a las líneas correspondientes del bus de alimentación.

Los contactos A (de *Address*) se conectan con las líneas correspondientes en el bus de direcciones. Los contactos D (de *Data*) con las líneas de datos. Solo consideraremos dos señales de control (excluyendo el reloj, CLK): una para lectura (RD) y otra para escritura (WR), estas señales serán descritas en lo que sigue.

² Algunos la consideran como una señal aparte.

Los chips de memoria, ver fig. 8.2(b), también tienen contactos con papeles similares: unos para datos, otros de direcciones y otros de control; en particular, tiene una línea —o líneas— para indicar si se desea leer o escribir. Los chips esperan que les llegue una dirección y una señal de lectura o escritura; si la señal es de lectura, buscan el dato en la dirección indicada y lo retornan por las líneas de datos, si la señal es de escritura, toman el valor que se encuentra en las líneas de datos y lo escriben en la dirección indicada. El número de pines de datos determinan cuál es el ancho de cada posición (en la figura 8.2(b) es 2: cada posición almacena dos bits); el número de pines de dirección determina la cantidad de posiciones del chip (en la figura 8.2(b) son 6, luego tiene $2^6 = 64$ posiciones).

A continuación, pasamos a ver cómo se utilizan las señales para comunicar procesador y memoria en lectura y en escritura.

Acceso de la memoria en lectura y escritura

Para comunicarse con la memoria en escritura, el procesador envía la dirección donde quiere escribir por las líneas A —hay tantas líneas como bits de direcciones, y se envía un bit por línea—. Un poco más tarde —para permitir que A se estabilice—, envía el dato por las líneas D y activa la señal WR (la pone en 1), para informarle a la memoria que se trata de una escritura. La memoria, por su lado, va a recibir la dirección; cuando WR se pone en 1, la memoria sabe que se trata de una escritura, así que toma la información que se encuentra en el bus de datos y la pone en la dirección que recibió al principio. El proceso de escritura, en términos de diagramas de tiempo, se ilustra en la figura 8.3(a).

El proceso de lectura es similar al anterior —fig. 8.3(b)—. La diferencia estriba en que se utiliza la señal RD para indicar que se trata de una lectura. Por otro lado, las líneas de datos, en este caso, son manejadas por la memoria para enviar el dato pedido.

Señales alternativas para controlar la memoria

En una primera aproximación, las señales RD y WR pueden parecer redundantes; en efecto, se puede pensar en tener una sola señal e interpretar el 1 como lectura y el 0 como escritura. Esta señal ha sido utilizada en algunos procesadores, se representa con el símbolo R/\overline{W} .

Este método sufre de un inconveniente: la señal R/\overline{W} debe estar en 1 ó en 0, por lo tanto, indicará continuamente una lectura o una escritura —según esté en 1 ó 0—;

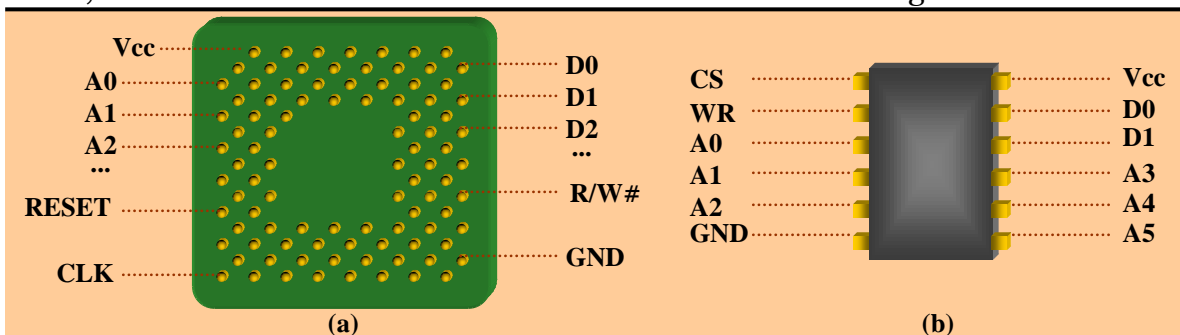





Fig. 8.2. (a) Contactos de un procesador. (b) Contactos de un chip de memoria.

¿Qué es ...

... un diagrama de tiempo? Una representación de cómo evolucionan en el tiempo un conjunto de señales.

Las señales se dibujan horizontalmente (el tiempo transcurre hacia la derecha), y una debajo de otra para apreciar las relaciones entre ellas.

Si la señal está arriba es un uno, si está abajo es un cero. Otras convenciones son:

- ♦ Un conjunto de señales con algún valor no precisado: 
- ♦ Una línea flotante (que no está siendo manejada por nadie): 
- ♦ Cambio en el valor de un conjunto de líneas: 

esto implica que R/\overline{W} no sirve para sincronizar el procesador con la memoria. En efecto, la memoria no puede saber si R/\overline{W} vale 1, ó 0, porque hay efectivamente una lectura en curso, o si es porque no está siendo utilizada por el procesador.

Se debe establecer una señal independiente de activación, usualmente llamada *E* (*Enable*). La memoria sólo se activa si *E* vale 1; en tal caso, mira R/\overline{W} para decidir si se trata de una lectura o de una escritura. Si *E* vale 0, la memoria no obedece la señal de lectura-escritura.

En la discusión anterior, se ha simplificado un poco el panorama, hemos eliminado unos componentes. Por ejemplo, este es el caso de los *buffers* que sirven de intermediarios entre los pines del procesador y las líneas de los buses; dichos *buffers* son necesarios puesto que los pines del procesador no tiene mucha capacidad de salida. La labor del *buffer* es proporcionar una capacidad mayor para manejar las líneas de los buses.

8.3 DISEÑO DE LA MEMORIA

En principio, para diseñar una memoria, se podría pensar en construir la contraparte del procesador; por ejemplo, suponiendo que el procesador de la figura 8.2(a) maneja direcciones de 16 bits y datos de 8, sería un chip de memoria con 16 pines, o contactos, de direcciones, 8 pines de datos, pines para las señales RD y WR y

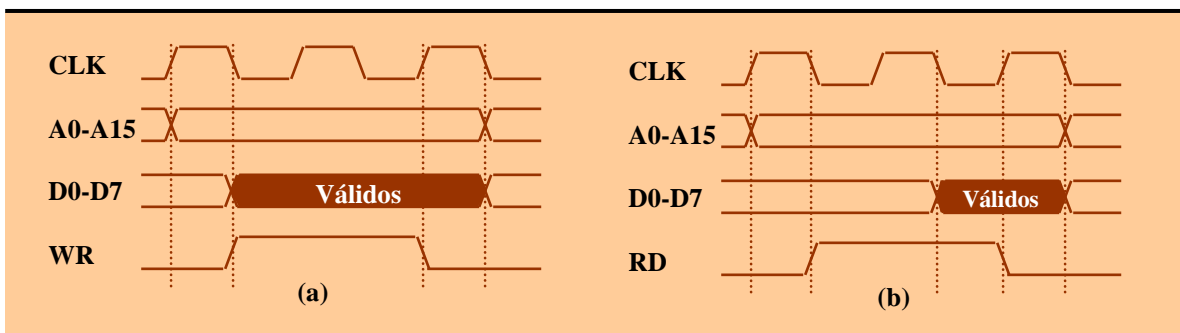


Fig. 8.3. (a) Escritura en memoria. (b) Lectura de memoria

los pines de alimentación. Solamente habría que conectar los pines a las líneas correspondientes del bus —lo mismo que el procesador— y se tendría lista la comunicación entre los dos (ver figura 8.4).

Pero la realidad no es tan simple. En primer lugar, los chips de memoria no se fabrican del tamaño justo y adecuado; por ejemplo, la memoria de un computador de 512 MiB no consta de un solo chip de 512 MiB, sino que tiene varios chips que, entre todos, conforman la capacidad deseada.

Esto es así porque, dada la capacidad de direccionamiento de los procesadores actuales, es complicado construir chips tan densos; además, es poco flexible, ya que los procesadores tienen diferentes capacidades de direccionamiento. En la vida real, un diseñador de computadores elige cuánta memoria le dará al computador, y después decide el número y la capacidad de los chips que utilizará para completarla.

Utilizando de nuevo nuestro procesador imaginario, y dado que tenemos 16 bits de direcciones —64KiB de memoria—, elegimos, arbitrariamente, tener 4 chips de memoria, cada uno de 16KiB. El primero contendrá los bytes con direcciones entre 0 y 16Ki - 1, el segundo se direcciona desde 16Ki hasta 32Ki -1 y así sucesivamente. Cada chip debe tener 14 líneas de direcciones y 8 de datos.

Decodificación de direcciones

La pregunta que sigue es cómo organizar estos 4 chips para que respondan solamente al rango de direcciones que les fue asignado. En la figura 8.5, puesto que los 2 chips son iguales, si el procesador envía una dirección, los dos responderían enviando cada uno su dato por el bus, con lo cual se crearía un conflicto de información en el bus y los datos serían ilegibles.

En consecuencia, se necesitan circuitos adicionales que, al recibir una dirección, decidan a cuál chip le corresponde y sólo le permitan funcionar a ese chip.

El primer paso consiste en agregar un pin a los chips de memoria: *selección de chip* (CS, *Chip Select*);³ mientras dicho pin esté en 0, el chip no responde; cuando se pone en 1 funciona normalmente. Esto nos permite activar o desactivar el chip a

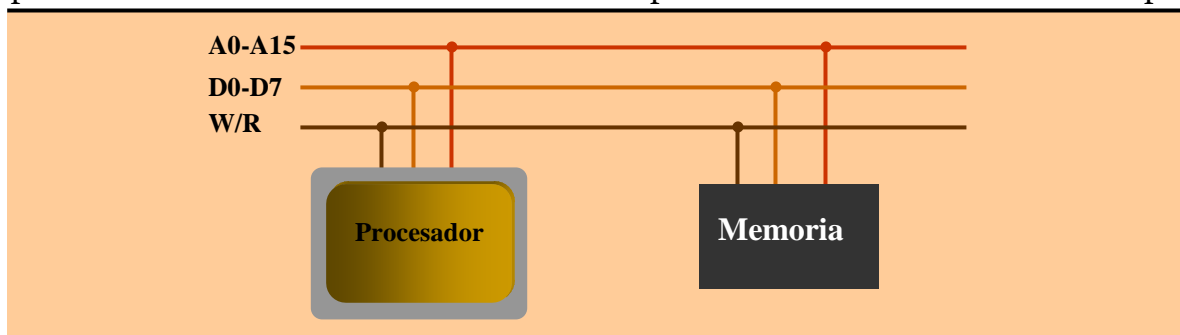


Fig. 8.4. Conexión directa del procesador y la memoria

voluntad. En el caso de la figura 8.5, si se activa uno de los chips y se desactiva el

³ También llamado *Enable*, o *Disable*, o puede estar separado en *WE* (*Write Enable*) y *OE* (*Output Enable*).

otro, solo uno respondería a la dirección enviada por el procesador, con lo cual se evitaría el conflicto en el bus.

En nuestro ejemplo, si numeramos los chips de 0 a 3, los rangos de direcciones, en hexadecimal, serían:

Chip	Rango
0	0000H a 3FFFH
1	4000H a 7FFFH
2	8000H a BFFFH
3	C000H a FFFFH

Si examina los rangos de direcciones de cada chip, en binario, observará que todas las direcciones del chip 0 empiezan por 00, las del chip 1 por 01, las del 2 por 10 y las del chip 3 por 11. Es decir, los dos primeros bits de la dirección identifican el chip. Esto nos proporciona un método para seleccionar los chips: se dividen las líneas de direcciones en dos grupos; unas sirven para seleccionar el chip que se quiere acceder; el resto sirven para seleccionar la dirección deseada dentro del chip. En nuestro ejemplo, el procesador tiene 16 líneas de direcciones: 14 líneas son suficientes para direccionar los 16KiB que contiene cada chip; las dos líneas restantes —los dígitos más significativos— sirven para elegir uno de los cuatro chips: 00 elige el primero, 01 el segundo, etc. Las dos líneas que llevan los bits más significativos permiten decidir cuál es el chip seleccionado —usar los dos bits más significativos es como contar en grupos de 16KiB—.

Esto se puede lograr por medio de un circuito decodificador, como se muestra en la figura 8.6. La idea es que el circuito decodificador recibe los dos bits más significativos y activa el chip correspondiente —por medio de la señal CS—. Los restantes bits de dirección se utilizan para seleccionar la dirección deseada en el chip. Así, si la dirección es 00100000 00000000, esto corresponde a la posición 100000 00000000, del chip 00; en tanto que 10100000 00000000 es la misma posición pero en el chip 10.

Note que todos los chips están conectados a las mismas líneas de datos en el bus, pero como solo uno de ellos va a funcionar —debido al decodificador—, no se genera conflicto en el bus.

Ancho de palabra

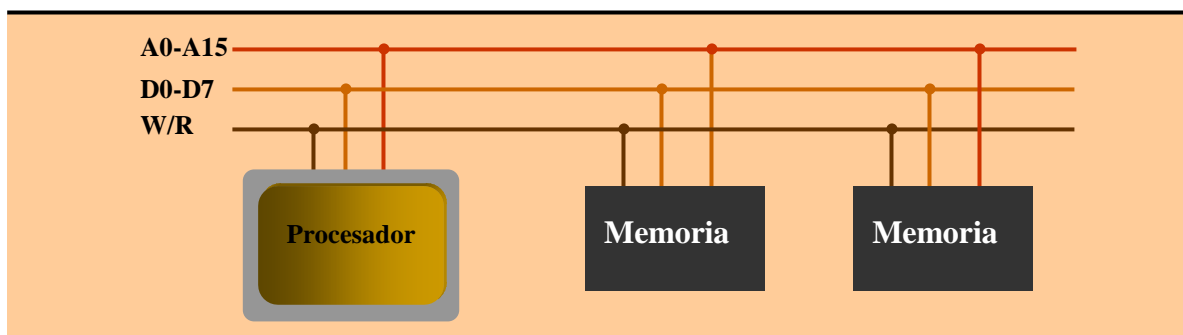


Fig. 8.5. Conexión de varios chips de memoria

¿Qué es ...

... un decodificador? Un circuito con n líneas de entrada y 2^n líneas de salida. Las n líneas de entrada representan un número binario de n bits; las líneas de salida se numeran de 0 a $2^n - 1$.

Si el valor del número binario que viene en las n líneas de entrada es i , la salida i se pone en 1, y las demás en 0. Es decir, el número de entrada indica cuál de las salidas se debe activar, y todas las otras quedan en cero.

Hay otra realidad de diseño que debe tenerse en cuenta. Hasta el momento hemos supuesto que cada uno de los chips de memoria genera 1 byte. Es decir, hemos supuesto que los chips tienen una capacidad de $n \times 8$, donde n es el tamaño del espacio direccionable —16Ki en el ejemplo—. Esto no es necesariamente así; los circuitos de memoria se construyen de diferentes tamaños, de tal manera que puedan almacenar uno, dos, cuatro, ocho o más bits por dirección. Según el caso, se dice que la capacidad es $n \times 1$, $n \times 2$, etc. El diseñador debe completar el tamaño de palabra de memoria deseado con los chips que sean necesarios.

En nuestro ejemplo, vamos a utilizar chips de 4 bits. Cada módulo de la figura 8.6 se convierte en dos chips, como se muestra en la figura 8.7 (solo se muestra uno de los módulos para no complicar la figura). Los pines D0 a D3 del primer chip se conectan a las líneas D0 a D3 del bus de datos, y los pines D0 a D3 del segundo chip se conectan a las líneas D4 a D7. Por supuesto, el número de chips usados, y el número de bits por chip, dependen del ancho de palabra requerido.

A manera de síntesis, y generalizando lo antes presentado, la memoria se implementa con chips de memoria. Los chips, al igual que el total de memoria, se caracterizan por tener una cierta cantidad de líneas de datos —digamos j — y una cierta cantidad de líneas de direcciones —digamos k —. La capacidad del chip, en bits, será: $j \cdot 2^k$.

Excepto para procesadores muy simples, no es razonable pensar que toda la memoria se encuentre en un solo chip. Normalmente la memoria se implementa

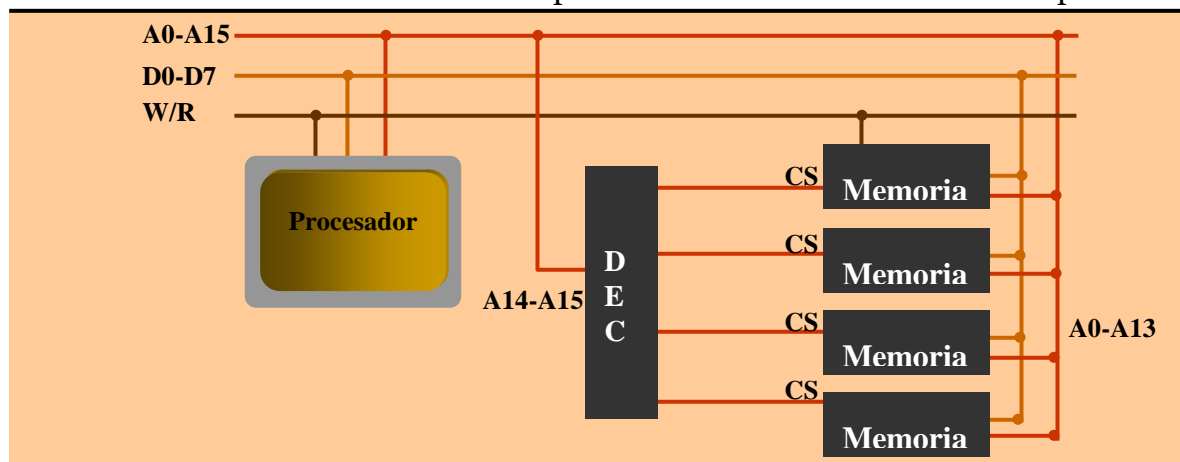


Fig. 8.6. Circuito decodificador para seleccionar chip.

con un conjunto de chips dispuestos en forma de matriz. De manera general, si tenemos una memoria con m bits en cada posición, y la implementamos con una matriz de $a \times b$ chips, se debe cumplir: $b*j = m$.

Cada una de las a filas se conoce como *banco de memoria*; en el ejemplo antes presentado, hay 4 bancos.

Por otro lado, el total de posiciones será: $a*2^k$ (puesto que cada chip tiene 2^k posiciones y hay a filas), y el total de memoria, en bits, es: $a*b*j*2^k$.

Además, si el procesador tiene n líneas de direcciones y la memoria está completa, también se debe cumplir: $a*2^k = 2^n$. De las n líneas de direcciones, $n-k$ se usan para seleccionar el banco, y k se usan para direccionar en el banco. Esto, en particular, implica que $a \leq 2^{n-k}$.

Acceso a los bytes

Implicítamente, la discusión anterior ha estado orientada a una memoria direccionable a palabra; en efecto, todos los chips de un banco se leen o se escriben al mismo tiempo.

Si se desea tener acceso a los bytes, es necesario “partir” el banco en bytes y permitir que cada pedazo pueda funcionar independientemente de los otros.

Lo primero es fácil de hacer: basta con asignar cada byte a un chip (o a varios que entre todos conformen un byte). Por ejemplo, supongamos que nuestro procesador tiene palabra de 32 bits; cada banco podría tener cuatro chips, cada uno de un byte.

Para lo segundo necesitamos ayuda del procesador: es imposible que la memoria pueda saber si el procesador está haciendo un acceso de palabra o byte; el procesador lo sabe debido a las características de la instrucción que está ejecutando. Por ejemplo, en la IA32, si se está ejecutando un `mov a eax`, se sabe que es un acceso de palabra; si es `a ah`, se sabe que es un acceso a byte.

Un posible mecanismo (que ha sido usado por Intel) es el siguiente: supongamos que el procesador tiene direcciones de 32 bits; los 30 más significativos (A2-A31) se usan para identificar la palabra donde se encuentra el byte; los dos menos

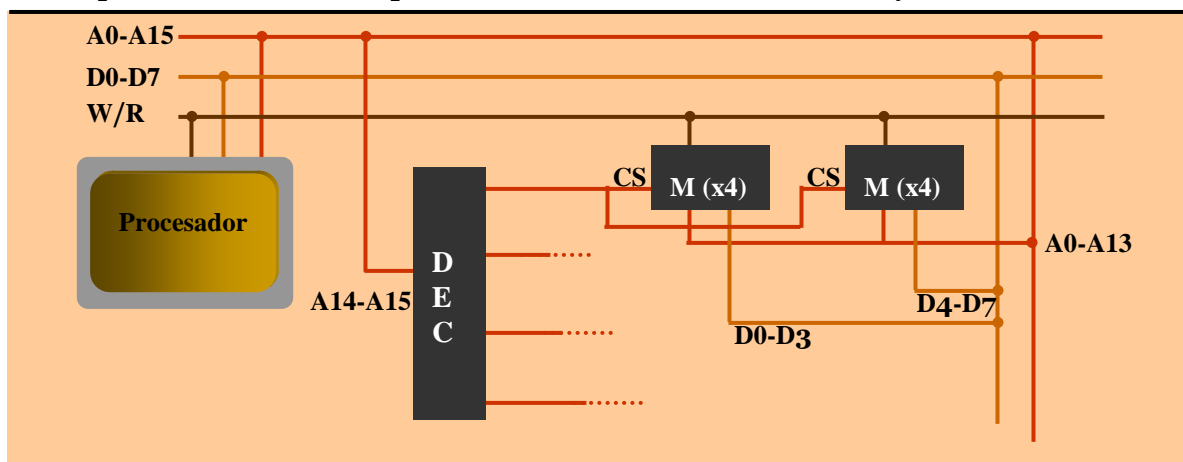


Fig. 8.7. Conformación del ancho de palabra usando varios chips.

significativos identifican el byte en cuestión. El procesador envía al exterior los 30 bits más significativos, y estos son los que se usan para seleccionar el banco y la palabra dentro del banco tal como se explicó anteriormente.

Además de los 30 bits de direcciones, el procesador genera y envía al exterior cuatro señales usadas para seleccionar el byte: B1, B2, B3 y B4; cada una de estas señales se usa para activar un chip distinto, o, lo que es lo mismo, un byte distinto.

El procesador genera estas señales de la siguiente manera: si se trata de un acceso a palabra, se activan todas (se ponen todos en 1); si se trata de un acceso a byte, solo se activa una de ellas: B1, si la dirección acaba en 00; B2, si la dirección acaba en 01; B3, si la dirección acaba en 10; B4, si la dirección acaba en 11.

En la figura 8.8 se ilustra cómo conjugar estas señales con la selección de chip para así lograr que solo funcione el chip del byte correspondiente (solo se muestra para dos chips; los otros dos son similares pero con B3 y B4). El circuito marcado con “&” se encarga de hacer un Y-lógico entre la señal de selección de chip y la respectiva señal B; de esta manera, el chip de memoria solo se activa si el banco está seleccionado y el byte también.

Empaquetamiento

En máquinas antiguas, la memoria se implementaba como la matriz de chips descrita en las secciones anteriores. Hoy día se prefiere agrupar los chips en módulos de memoria.

Estos módulos de memoria son pequeñas tarjetas que agrupan varios chips, y que se instalan en ranuras, con lo cual se simplifica el manejo de los chips de memoria.

Los primeros módulos utilizados se llamaban SIMM (*Single Inline Memory Module*); tenían 30 contactos, y proveían un ancho de palabra de un byte. Posteriormente, impulsados por los procesadores de 32 bits, surgieron los SIMM de 72 contactos; estos proveían 32 bits. Luego, también impulsados por los procesadores del momento, surgieron los llamados DIMM (*Dual Inline Memory Module*); estos tienen 168 contactos y proveen 64 bits.

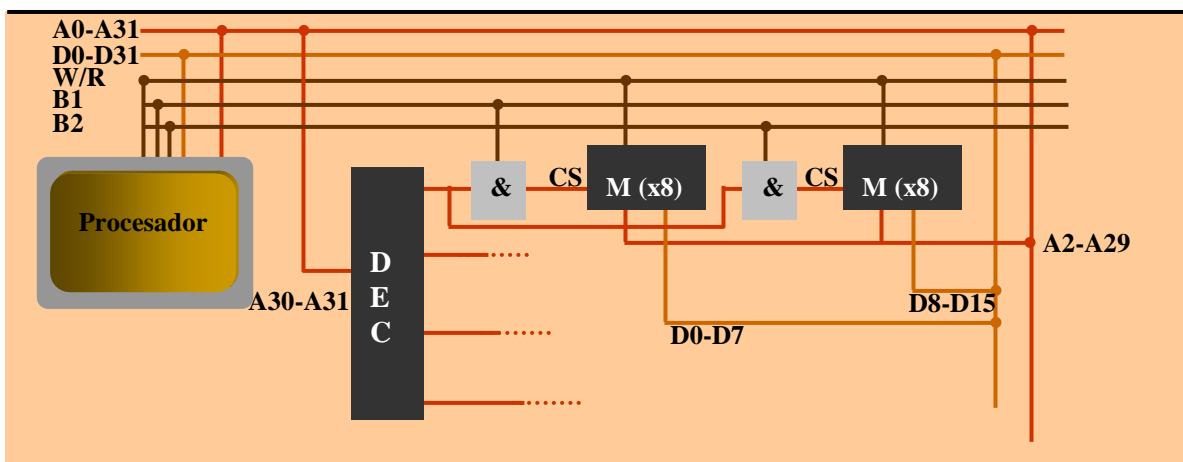


Fig. 8.8. Selección de byte.

Existen otros módulos de memoria, como por ejemplo los SO-DIMM (*Small Outline DIMM*), para computadores portables, y los RIMM (*Rambus Inline Memory Module*).

8.4 CARACTERIZACIÓN DE LA MEMORIA

La memoria tiene unos comportamientos característicos que es conveniente conocer para entender algunas decisiones de diseño y ciertos mecanismos utilizados. En lo que sigue presentaremos los aspectos más destacados de su funcionamiento.

Las instrucciones están en memoria

Que las instrucciones están en memoria es una de esas verdades evidentes que frecuentemente se olvida tener presente. Cuando se habla de la memoria, se suele pensar en los datos, y no en las instrucciones, y resulta que las instrucciones condicionan en buena medida el comportamiento de la memoria. Veamos por qué.

Podemos dividir el total de accesos a la memoria en accesos de datos y accesos de instrucciones, ¿cuál es la relación entre los dos? Del 100% de las instrucciones, es razonable estimar que un 25% de ellas hacen algún acceso a memoria; en consecuencia, si ejecutamos 100 instrucciones, habrá que hacer 100 lecturas de memoria para obtener las 100 instrucciones, y 25 accesos para obtener los datos.⁴ Es decir, de 125 accesos a memoria, 100 son para leer instrucciones; esto quiere decir que el 80% de los accesos son para leer instrucciones. ¡La memoria funciona la mayoría del tiempo para obtener instrucciones y no datos!

Las lecturas priman sobre las escrituras

Si miramos solamente los accesos de datos, aproximadamente las dos terceras parte de los mismos son lecturas, y solo una tercera parte son escrituras.

Esto es razonable intuitivamente; después de todo el computador sirve, en general, para sintetizar; cuando se sintetiza, se “lee mucho, y se escribe poco”. Por otro lado, desde un punto de vista pragmático, muchas operaciones son binarias, lo cual implica leer dos valores y escribir un resultado.

Ahora bien, si consideramos el total de accesos —incluyendo la lectura de instrucciones—, tenemos: 100 lecturas de instrucción, y 25 accesos de datos; estos últimos se dividen (aproximadamente) en 17 lecturas y 8 escrituras. Esto nos lleva a que las lecturas responden por algo del orden del 93% de los accesos a memoria.

Principio de localidad

La experiencia indica que los accesos a la memoria no ocurren de una manera arbitraria. Es decir, si se observa una secuencia de accesos a la memoria, se observará un cierto tipo de patrón.

⁴ Esto es solo un estimado, y en realidad depende del tamaño de las instrucciones. En RISC suelen ser de tamaño palabra, así que efectivamente corresponden con un acceso por instrucción.

En concreto, típicamente se observan dos tipos de comportamiento conocidos como *localidad espacial* y *localidad temporal*:

- La localidad espacial nos dice que si una cierta posición de memoria es referenciada, esto aumenta la probabilidad de que posiciones vecinas sean referenciadas en un futuro cercano.
- La localidad temporal nos dice que si una cierta posición de memoria es referenciada, esto aumenta la probabilidad de que la misma sea referenciada en un futuro cercano.

Aunque estos son comportamientos observados experimentalmente, no por ello dejan de ser razonables. En primer lugar, recordemos que la mayoría de los accesos a memoria son originados por las instrucciones; a esto agreguémosle que:

- Los programas tienden a ejecutar secuencialmente la mayoría del tiempo (excepto cuando toman saltos); esto nos explica buena parte de la localidad espacial.
- Para que un programa se demore ejecutando es necesario que itere; esto nos explica buena parte de la localidad temporal.

Adicionalmente, la experiencia indica que, en general, una pequeña porción de un programa es responsable por la mayor parte del tiempo de ejecución, lo cual hace que las referencias a la memoria se concentren en la zona donde está dicha porción del programa, con lo cual se favorecen los dos tipos de localidad.

Aunque la mayor parte de la localidad es originada por las instrucciones, los datos también presentan algún grado de localidad. Por ejemplo, frecuentemente se recorren los vectores en orden; esto produce localidad espacial. Hay casos menos evidentes; por ejemplo en el siguiente código de búsqueda en una lista encadenada:

```
while ( p->informacion != buscado ) p = p->siguiente;
```

Puesto que se trata de apuntadores, las estructuras apuntadas se pueden encontrar en cualquier sitio de la memoria; se podría pensar que las referencias de este programa “saltan” de un sitio de la memoria a otro sin relación aparente. Y si bien así es, notemos que, en cada iteración, se referencia 3 veces la variable `p`, una vez la variable `buscado`, y una vez los campos `informacion` y `siguiente`; es decir, en cada iteración se referencian 5 variables, pero 3 de ellas son la misma y son iguales de iteración en iteración (`p`), otra es la misma en todas las iteraciones (`buscado`) y solo dos de ellas variarán en cada iteración (`informacion` y `siguiente`). Esto nos generará localidad temporal.

8.5 JERARQUÍA DE MEMORIA

Conceptos generales

Debido a las diferentes técnicas y mecanismos involucrados en la construcción de las memorias, se presenta el siguiente fenómeno: hay técnicas que permiten crear memorias de gran capacidad a bajo costo, y hay técnicas que permiten crear memorias de alta velocidad a un costo alto, pero no hay técnicas para crear

memorias masivas, rápidas y baratas. Esta situación genera una tensión entre aspectos técnicos y económicos.

Aunque no ha sido posible resolver el problema, sí se han desarrollado técnicas para paliar el problema. La idea básica consiste en ver la memoria como una jerarquía de niveles, como un conjunto de memorias “apiladas” la una encima de la otra. En los niveles superiores tenemos memorias rápidas y costosas, y, en consecuencia, pequeñas; en la medida en que se desciende en la jerarquía, encontramos memorias más baratas —por ende más grandes— pero más lentas.

El propósito de este diseño es intentar tener los datos e instrucciones más utilizados (confiando en que sean pocos) en los niveles superiores, con lo cual el procesador, al acceder a ellos, percibe que está trabajando con una memoria veloz. Los datos menos usados estarán en niveles inferiores, en consecuencia el acceso a ellos será más lento. Cuando un dato se encuentra en un nivel superior, es utilizado inmediatamente; de lo contrario se le solicita al nivel inferior que envíe la información.

Ahora, esta idea parte de dos supuestos:

- Hay entidades —datos e instrucciones— más usadas que otras.
- El número de entidades más usadas es relativamente pequeño; la zona que ocupan en memoria es relativamente pequeña (comparada con el tamaño total de la memoria).

El principio de localidad garantiza —estadísticamente— que estos dos supuestos se cumplen.

En general, la jerarquía de memoria está compuesta por cuatro tipos de memoria:⁵

- Registros. Los registros del procesador son, en la práctica, un pequeño banco de memoria incorporado al procesador, y, como se explicará más adelante, se trata de la memoria más rápida disponible.
- Cache de memoria. Se trata de una memoria con una velocidad intermedia entre la de los registros y la memoria principal.
- Memoria principal. Es el almacenamiento primario: la que se ha estado estudiando a lo largo de este capítulo. Es una memoria muy veloz, si se compara con la velocidad del almacenamiento secundario —discos—, pero es relativamente lenta comparada con la velocidad de un procesador. Puede impactar bastante el desempeño del procesador; por esto es necesario recurrir al uso del cache de memoria.
- Memoria virtual. Es una técnica para usar el almacenamiento secundario como si se tratara de memoria principal. Se debe aclarar que no se trata del manejo de archivos como se acostumbra en los lenguajes de alto nivel: el

⁵ Sin embargo, como veremos más adelante, puede haber más de cuatro niveles. Aquí hacemos referencia, más bien, a cuatro “tipos” de niveles.

programa opera como si estuviera todo en la memoria principal, pero parte está en el disco.

En las secciones anteriores hemos tratado extensivamente el caso de la memoria principal, a continuación trataremos el de los registros, y en las dos secciones siguientes, el cache de memoria y la memoria virtual.

Los registros

Los registros son la memoria más rápida por varias razones:

- Se diseñan usando técnicas para aumentar su velocidad, y no para ahorrar espacio.
- Son una memoria pequeña, lo que disminuye el tiempo de respuesta de sus circuitos.
- Están en el mismo chip que la ALU. La comunicación de dos elementos que se encuentran en el mismo chip es más rápida que entre chips.
- Se diseñan para agilizar su interacción con la ALU. Casi podríamos decir que están hechos para servir a la ALU.

El propósito de los registros es aprovechar la localidad en datos escalares. La idea es destinarlos —sea permanente o temporalmente— para mantener las variables más utilizadas en un cierto momento del tiempo; típicamente, las variables más utilizadas en un ciclo.

Los registros son completamente manejados por el software; es el programador —o el compilador— quien decide cuáles variables deben estar en los registros, y el programa debe tener instrucciones para cargar explícitamente dichas variables en los registros —y también para copiar de vuelta los registros en las variables—.

Si en algún momento se desea cargar una nueva variable y no hay registros disponibles, el programa debe descargar explícitamente un registro. Por supuesto es el programador o el compilador quien decide cuál variable descargar.

Para mantener la velocidad de los registros, es necesario que esta memoria sea pequeña —típicamente unas centenas de bytes—, por lo cual se debe seleccionar cuidadosamente cuáles variables van en ellos. Esto no es inconveniente puesto que se trata de un análisis que se hace en compilación, no en ejecución, así que no es problemático si se requiere invertir un cierto tiempo en el estudio.

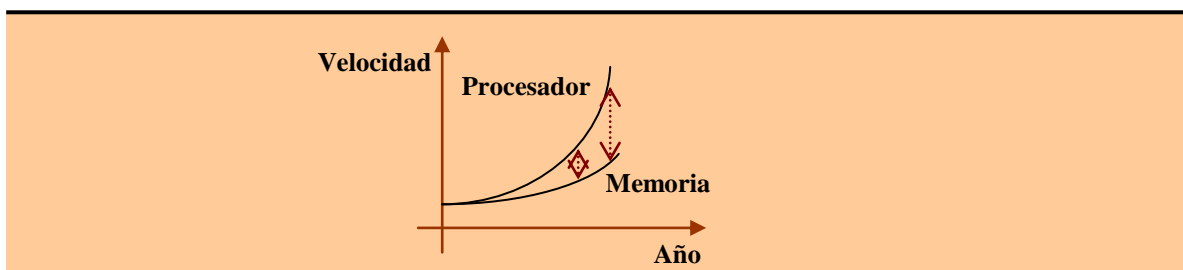


Fig. 8.9. Aumento en velocidad del procesador y la memoria

8.6 CACHE DE MEMORIA

Desde su invención, los microprocesadores han ido aumentando en velocidad exponencialmente. Las memorias también, pero su velocidad aumenta a un ritmo más lento (ver fig. 8.9). Esto causa que, a medida que pasa el tiempo, la diferencia de velocidad entre los procesadores y la memoria principal se incrementa; digamos que cada año que pasa la memoria se hace un poco más lenta con respecto al procesador.

Esta es una situación altamente inconveniente; las instrucciones que guían la ejecución del procesador están en memoria, y si no pueden ser leídas al ritmo que requiere el procesador, de nada, o de muy poco, sirve tener un procesador más veloz.

En realidad sí es posible construir memoria tan veloz como el procesador, pero es más costosa. Esto nos pone a elegir entre: desperdiciar velocidad en el procesador, tener memorias pequeñas o grandes pero costosas, ... o buscar una solución de compromiso.

Según el tipo de máquina, se puede pensar en una alternativa u otra; por ejemplo, las máquinas de muy alto desempeño pueden tener memorias grandes y costosas, pero esto hace que sean utilizadas en mercados mas restringidos; solo por aquellos que definitivamente necesitan esa velocidad, que tienen los medios económicos y que están dispuestos a pagar.

Para la gran masa del mercado es mejor recurrir a una solución de compromiso; dicha solución es el cache de memoria. El cache es una memoria de una velocidad cercana a la del procesador en la cual se mantienen las entidades más utilizadas de la memoria principal.

Al contrario de los registros, el cache se maneja enteramente por hardware. Se trata de una *memoria asociativa*. Una memoria asociativa guarda en cada posición dos valores: el valor almacenado en sí, y la dirección que este valor tiene en la memoria principal (ver fig. 8.10).

El funcionamiento es el siguiente: cuando el procesador solicita un dato a la memoria, envía la dirección al cache; este mira si la dirección coincide con alguna de las almacenadas, si es así, retorna el dato correspondiente. Por ejemplo, en la figura 8.10, si el procesador pide la posición 2, el cache retornará el valor 78.

Cache	Dirección	Dato	Dirección	Dato	Memoria
0	2	78	0	15	
1	4	14	1	24	
2	0	15	2	78	
			3	45	
			4	14	

Fig. 8.10. Relación entre el cache y la memoria principal

Ahora, si no coincide con ninguna de las direcciones almacenadas, el cache pedirá el dato a la memoria, y lo instalará en alguna posición. En la figura 8.10, si el procesador pide la dirección 3, esta no coincide con ninguna de las almacenadas, así que remitirá el pedido a la memoria principal; esta retornará el valor 45, y el cache guardará la pareja 3-45 en alguna posición del cache (como está lleno, le toca sobrescribir alguna de las parejas existentes), y retornará el valor 45 al procesador.

Tipos de cache

La descripción anterior es conceptual, pero tiene problemas en la práctica; si el cache tuviera que comparar la dirección con todas las almacenadas, le tomaría más tiempo que hacer la lectura directamente de memoria. Es necesario buscar cómo agilizar esta comparación.

Cache asociativo

Una primera solución es comparar simultáneamente la dirección con todas las que se tienen almacenadas; para esto es necesario tener un comparador asociado a cada posición del cache (ver fig. 8.11). En la figura, la dirección buscada (4) se compara con todas; la segunda entrada del cache tiene un 4, así que el respectivo comparador reporta que es igual (1); los demás reportan que no lo son (0).

Esta solución permite que las parejas dirección-dato sean almacenadas en cualquier posición del cache; cuando se necesita cargar una pareja, se puede hacer en cualquier posición disponible. Solo si el cache está completamente lleno, puede ser necesario sacar otra pareja del cache (sobrescribirla con la nueva entrada).

Como desventaja tiene que los comparadores ocupan espacio en el circuito, con lo cual dejan menos espacio para la memoria en sí.

Este tipo de memoria asociativa es muy poco utilizada para caches, aunque se usa para otros propósitos.

Cache de proyección directa

Los caches de proyección directa usan una función de *hashing* para establecer una relación entre las direcciones de memoria y las posiciones en el caché; esto quiere decir que una determinada posición de memoria solo puede cargarse en un único sitio del cache. Por ende, puede ocurrir que al cargar una entrada toque sobrescribir otra así se disponga de más entradas libres.

La función de *hashing* se establece de la siguiente manera. Supongamos que la memoria tiene direcciones de n bits, y el cache tiene 2^k posiciones; 2^k posiciones se

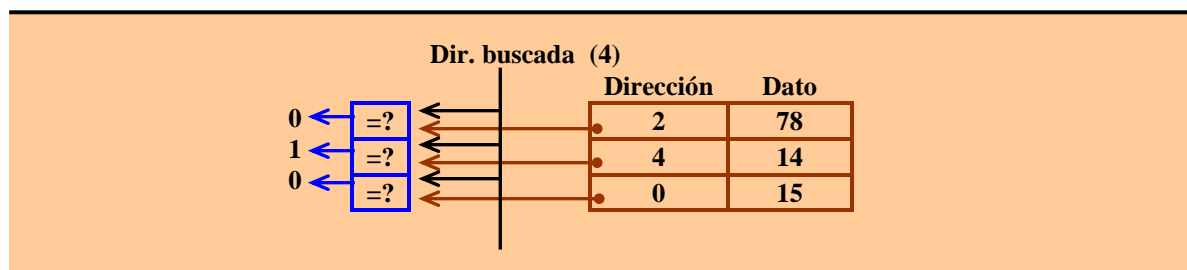


Fig. 8.11. Cache asociativo

pueden numerar usando k bits. En consecuencia se pueden usar los k bits menos significativos de la dirección para identificar la posición en el cache; los $n-k$ restantes se almacenan en el campo “dirección” del cache.⁶

De esta manera, cuando llega una dirección al cache, sabe dónde buscarla: o está en esa posición o no está en lo absoluto. Basta con leer la posición del cache y comparar la dirección almacenada con los $n-k$ bits más significativos de la dirección recibida; si son iguales, el dato está almacenado en esa posición; si son diferentes, no lo está (ver fig. 8.12).

Por ejemplo, supongamos que se tienen 16 bits de direcciones, y que el cache tiene 256 posiciones, por lo tanto, se necesitan 8 bits para direccionarlo. Si se tiene la dirección 0001H, se le asignará la posición 01 del cache, y en el campo de dirección se guardará 00; si se tiene 0002H, se le asignará la posición 02, y se guardará 00H en el campo de dirección. Ahora, si tenemos 0402H, también se le asignará la posición 02 (o sea que no puede estar al mismo tiempo con la anterior) y se guardará 04 en el campo de dirección.

Cuando a dos direcciones de memoria les corresponde una misma entrada en el cache, se dice que hay una *colisión*, y solo una de ellas se puede guardar; la otra queda por fuera del cache.

Los caches suelen estar basados en este mecanismo, aunque se acostumbra usar la variante que se describe a continuación.

Cache asociativo por conjuntos

Esta variante es, en realidad, una combinación de las dos anteriores. Consiste en tener dos caches de proyección directa en paralelo; se leen los dos al tiempo y se comparan simultáneamente los dos campos de dirección con la dirección recibida (ver fig. 8.13).

Una cierta dirección, solo se puede encontrar en una posición de los caches, pero puede estar en cualquiera de ellos; esto disminuye las colisiones. En el ejemplo de sección anterior, tanto 0002H como 0402H podrían estar en el cache (ver fig. 8.13).

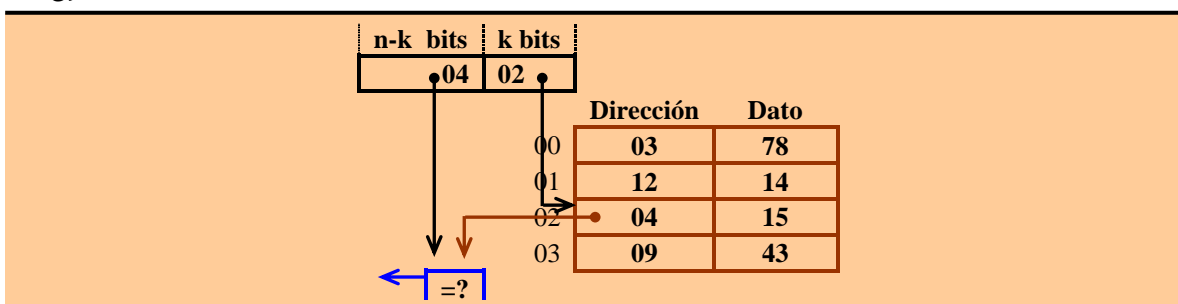


Fig. 8.12. Cache de proyección directa

⁶ No es necesario guardar los n bits, puesto que la posición del cache donde se encuentra la entrada identifica implícitamente los k bits menos significativos. Esto constituye un ahorro con respecto a los caches asociativos que sí tienen que guardar los n bits de la dirección.

El ejemplo mostrado es un cache asociativo por conjuntos de dos vías, porque tiene dos caches de proyección directa en paralelo; por supuesto, se pueden poner más: 4, 8, en cuyo caso serán de 4 vías y 8 vías respectivamente.

Otros aspectos

Bit de validez

En las secciones anteriores hemos mencionado que los caches tienen dos columnas: dirección y dato; en la práctica se necesita otra columna para indicar que la entrada es válida. En efecto, note que una entrada vacía del cache por casualidad podría tener un valor que coincidiera con una dirección de memoria; en tal caso, el cache reportaría erróneamente tener el dato. Para evitar esto la columna “valido” —de un bit— valdrá 1 cuando efectivamente haya un dato guardado, y cero cuando no.

Cuando se consulta el cache, se verifica que la dirección sea igual a la almacenada y que el bit de validez esté activo. Cuando se carga un dato en el cache, se pone en 1 el bit de validez.

Funcionamiento en escritura

Hasta el momento solo hemos tratado el caso de lectura del cache —que es el más importante por ser el más común—. La escritura presenta algunos aspectos particulares que trataremos a continuación.

En primer lugar, debemos resolver cómo proceder cuando se hace una escritura en el cache. El problema radica en que si se efectúa la escritura en el cache, el dato original en memoria queda desactualizado.⁷ Hay dos posibilidades:

- Escritura a través del cache. En este caso, se hace la escritura en el cache pero también en la memoria. Esto implica que en el caso de la escritura el cache no aporta mucho porque de todas maneras es necesario ir a la memoria.⁸
- Recopiado en memoria. Se escribe solo en el cache, pero se le agrega otra columna al cache que indica si el dato almacenado ha sido modificado o no.

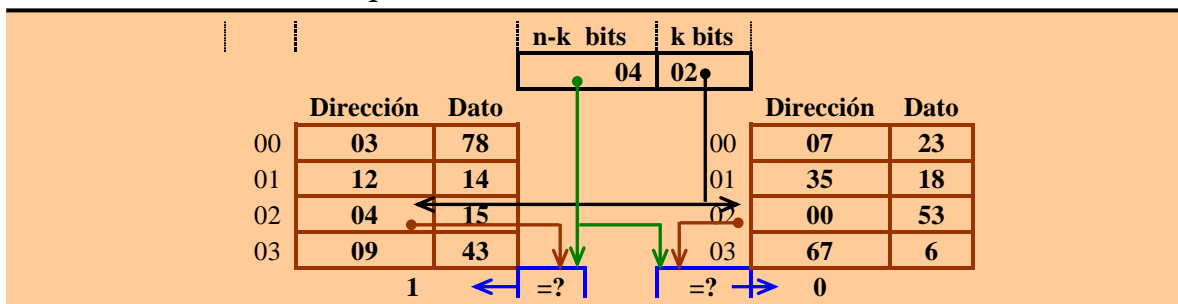


Fig. 8.13. Cache asociativo por conjuntos

⁷ Esto es importante en los sistemas multiprocesador donde puede ocurrir que dos procesadores tengan acceso a una misma variable en memoria.

⁸ Esta es una verdad a medias, puesto que el procesador puede seguir funcionando mientras se hace la escritura. Note que, cuando se hace una lectura, es necesario esperar el dato para poder proseguir; no así con la escritura.

Cuando es necesario desocupar una posición del cache, se revisa si está marcada como modificada, en cuyo caso se copiará de vuelta en memoria. Este método es más eficiente que el anterior, pero genera inconsistencias temporales con la memoria que pueden complicar el diseño de sistemas multiprocesador.

Lo anterior es aplicable cuando el dato está en el cache, si el dato no está en el cache es necesario decidir si se escribe directamente en memoria —y no en el cache— o si se le separa un lugar en el cache.

En general, esta decisión está relacionada con las políticas de escritura antes mencionadas: si se hace escritura a través del cache, se prefiere escribir directamente en la memoria —puesto que de todas maneras es necesario ir a la memoria, no es útil separar un lugar en el cache—. Si se hace recopiado en memoria, se prefiere separar un lugar en el cache.

Cache unificado y escindido

El cache almacena tanto instrucciones como datos. Es posible tener un solo cache para almacenar los dos tipos de entidades, pero también es posible escindirlo en dos caches especializados: uno para datos y otro para instrucciones.

La ventaja del cache escindido radica en que el comportamiento de los datos y de las instrucciones no es el mismo; con los caches separados, es posible sintonizar cada uno por aparte según las características propias de las entidades que maneja. Por ejemplo, los caches de instrucciones suelen ser más pequeños porque la localidad de las instrucciones es mayor que la de los datos.

Líneas de cache

Hasta el momento hemos considerado caches que mueven la información por palabras. Para aumentar la eficiencia, frecuentemente los caches leen varias palabras vecinas —un bloque— en lugar de leer palabras sueltas. En efecto, el principio de localidad dice que si se presenta un acceso en una posición, es probable que las palabras vecinas sean referenciadas; puesto que acceder a la memoria principal tiene un costo, si es necesario ir por un dato, es mejor traer varios de una vez. Por esto con frecuencia en cada posición del cache se almacenan varias palabras seguidas.

Desempeño del cache

Se acostumbra medir la efectividad del cache con la *tasa de acierto* (*hit rate*), la cual se define como:

$$h = \frac{\text{número_de_accesos_en_el_cache}}{\text{número_total_de_accesos_a_memoria}}$$

Así, $h = 0.9$ quiere decir que el 90% de los requerimientos a memoria se encontraban en el cache; h es la probabilidad de que una solicitud a la memoria se encuentre en el cache.

El mecanismo del cache es bastante efectivo; con caches relativamente pequeños es posible lograr tasas de acierto superiores al 90%. Sin embargo, su rendimiento

también se satura rápidamente, como se puede ver en la figura 8.14; con un cache relativamente pequeño se logra una tasa de acierto alta, pero después aumenta lentamente. Esto implica que, siempre que se aumente el cache, aumentará la tasa de éxito, pero cada vez menos, por lo cual la relación beneficio/costo irá disminuyendo; en algún momento el crecimiento del cache aportará muy poco aumento en el rendimiento a un costo relativamente alto.

Puesto que en ocasiones los datos se encuentran en el cache y en ocasiones no, el procesador percibirá un tiempo de acceso a la memoria variable. Llamando t_c el tiempo de respuesta del cache, y t_m el tiempo de respuesta de la memoria, el tiempo medio de acceso, t_a , percibido por el procesador será:

$$t_a = t_c + (1-h) t_m$$

8.7 MEMORIA VIRTUAL

Las técnicas presentadas en las secciones anteriores pretenden lograr que el sistema de memoria presente una mayor velocidad promedio; la memoria virtual pretende que la memoria parezca más grande, así se comporte un tanto más lento.

La idea de la memoria virtual consiste en mantener parte de los datos y del código en memoria principal y parte en algún dispositivo de almacenamiento secundario—como los discos duros—. El sistema de memoria virtual se encarga de llevar los datos y el código del dispositivo de almacenamiento a la memoria principal en la medida en que el programa lo requiera. Esto se hace de manera transparente para el programador; los datos migran sin que él tenga que hacer nada, en consecuencia, percibe una memoria más grande de lo que efectivamente es (ver fig. 8.15).

Traducción de direcciones

La introducción de la memoria virtual exige que la UC cambie su forma de operar. En efecto, antes, cada vez que la UC necesitaba leer una instrucción o un dato de memoria, se limitaba a enviar la dirección a la memoria y efectuar una lectura. Ahora es necesario que, antes de enviar la dirección, revise si esta se encuentra en memoria real o en el disco.

Si el dato o instrucción se encuentra en el disco, la UC debe informar este hecho al sistema operativo para que él se encargue de traer la información a memoria real.

Si la información se encuentra en memoria real, la UC debe identificar dónde en memoria real se encuentra localizado el dato; esto porque, como se aprecia en la

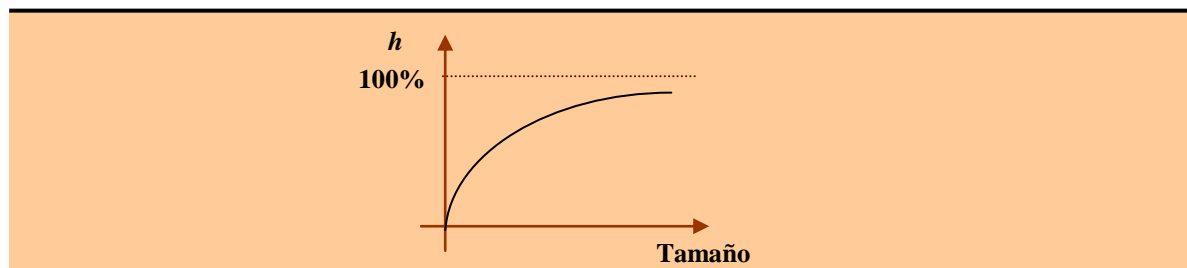


Fig. 8.14. Tasa de acierto contra tamaño.

figura 8.15, los datos no necesariamente ocupan la misma posición en memoria real y en memoria virtual. Esta operación se conoce como *traducción de direcciones*.

Para implementar la memoria virtual, tanto el espacio virtual como el real se dividen en bloques llamados *páginas*; el tamaño de estas páginas depende del procesador utilizado y del sistema operativo, pero típicamente son unos cuantos KiBytes.

Las páginas virtuales pueden estar en uno de dos estados: asignadas a una página real o almacenadas en el disco.⁹ Para poder ubicar las páginas, en primer lugar, se procede a numerarlas secuencialmente —tanto las páginas virtuales como las reales; ver fig. 8.15—. En segundo lugar, se dispone de una estructura de datos llamada *tabla de descriptores de página*; se trata de un vector indexado por el número de página virtual y cuyo contenido indica si la página está en el disco o en memoria real; en este último caso también indica en cuál página real se encuentra. Por ejemplo, para el caso de la fig. 8.15, la tabla sería:

	Presencia	Página real
0	1	0
1	1	3
2	0	—
3	1	1
4	1	2
5	0	—

El campo “presencia” indica si la página se encuentra o no en memoria real; el campo “página real” indica cuál es la página correspondiente en memoria real —si la página virtual no está en memoria real, este campo no se usa—.

Para agilizar la traducción, las direcciones, tanto virtuales como reales, se dividen en dos partes: los primeros m bits indican en cuál página se encuentra, y los últimos n bits indican la posición dentro la página. Por ejemplo, en una máquina con direcciones de 32 bits, se podrían partir las direcciones en los 20 bits más significativos y los 12 menos significativos, con lo cual se tendrían 2^{20} páginas cada una de 2^{12} bytes; esto implica que la tabla de descriptores puede llegar a tener 2^{20} entradas.

Cuando la UC necesita traducir una dirección virtual, toma los 20 bits más

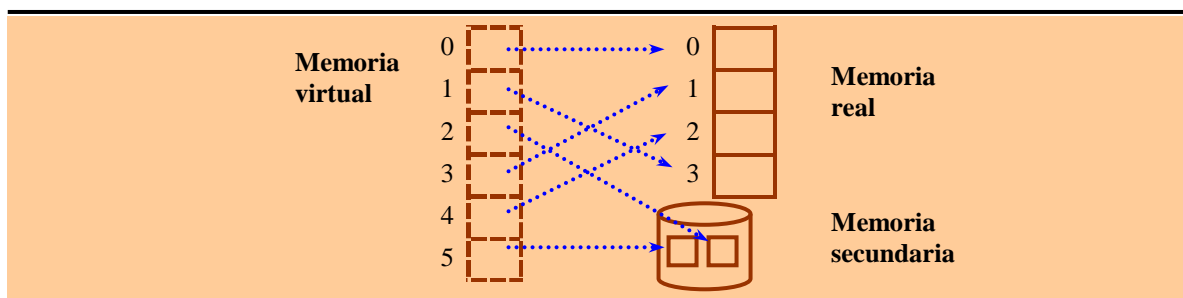


Fig. 8.15. Memoria virtual.

⁹ Estrictamente hablando, puede haber un tercer estado: la página no está asignada ni a una página real ni está en el disco; es decir, la página no tiene existencia física.

significativos e indexa con ellos la tabla de descriptores; del descriptor correspondiente toma el número de la página real y con este valor reemplaza los 20 bits más significativos de la dirección virtual;¹⁰ la dirección así obtenida corresponde a una dirección real, y, por tanto, puede ser enviada a la memoria.

Funcionamiento

Los programas y los datos se instalan en la memoria virtual, lo cual quiere decir que todas las direcciones generadas por un programa, de instrucciones y datos, son con respecto al espacio virtual de direcciones.

Adicionalmente, como vimos anteriormente, la UC sigue permanentemente un ciclo de ejecución: lectura de instrucción (lo cual implica un acceso a la memoria), decodificación de la instrucción (que también puede implicar una o más lecturas de la memoria para obtener los operandos), ejecución y escritura del resultado (que puede implicar otro acceso a la memoria). Cada uno de los accesos antes mencionados parte de una dirección virtual, y, por ende, esta debe ser traducida, siguiendo el proceso descrito en la sección anterior, antes de hacer el acceso efectivo a la memoria.

Ahora bien, durante el proceso de traducción, puede ocurrir una de dos cosas:

- El bit de presencia está en 1. En este caso todo transcurre como se describió en la sección anterior.
- El bit de presencia está en 0. En este caso, la página virtual se encuentra en el dispositivo de almacenamiento, y, en consecuencia, no puede ser direccionada.

En el segundo caso, la UC no puede traducir la dirección, así que se limita a informar al sistema operativo sobre la ocurrencia de un *fallo de página*. Note que, en este punto, el tratamiento pasa al software; el sistema operativo toma el control del asunto, y se suspende el programa que estaba ejecutando (el que causó el fallo de página).

El sistema operativo ubica la página en el dispositivo de almacenamiento, busca una página libre en memoria real y carga en ella la página almacenada en el dispositivo. Después de esto actualiza la tabla de descriptores (bit de presencia en 1, y número de la página real donde cargó la información). Una vez hecho lo cual, retorna el control al programa para que vuelva a intentar el acceso, el cual será exitoso esta vez puesto que la página ya ha sido cargada en la memoria real.

Al intentar cargar la página, el sistema operativo se puede encontrar con que no hay páginas disponibles en la memoria real. En tal caso, el sistema operativo debe descargar otra página al dispositivo de almacenamiento, y utilizar el espacio que se acaba de liberar para cargar la página que presentó el fallo.

¹⁰ Dado que las páginas virtuales y las reales tienen el mismo tamaño, un desplazamiento es igual en una página real o en una virtual y no necesita ser traducido.

utilidad: protección (del sistema, entre usuarios, al interior de un mismo usuario).
administración (reduce fragmentación, supervisar uso de zonas, manejo del crecimiento de zonas (pila)).

Ejercicio de conexión de líneas a chips (el 1) si hay 4 chips y si (a) el procesador es direccionable a palabra (b) es direccionable a byte. (las líneas se pueden conectar a cualquier chip)

(otra sección: velocidad, NUMA?. ECC (en ejercicios?)).

EJERCICIOS

- 1- Se tiene un chip de memoria con 16 pines de direcciones (A'0 a A'15) y 8 de datos (D'0 a D'7). Este chip se va a conectar a un computador con el mismo número de líneas (A0 a A15 y D0 a D7).
 - a- ¿Importa el orden en que se conecten las líneas de direcciones, es decir, A'i debe estar conectada a Ai?
 - b- ¿Importa el orden en las líneas de datos? ¿D'i debe conectarse con Di?
- 2- Se tiene un procesador de 32 bits (datos y direcciones), con direccionamiento al byte. El total de memoria es de 128 MiB, dividida en cuatro bancos. ¿Cuántos Mibits debe tener cada banco? ¿Cuántas líneas de direcciones se necesitan para direccionar cada banco?
- 3- Se tiene un procesador de 64 bits (datos y direcciones), con direccionamiento al byte. El total de memoria es de 256 MiB, y se construye con chips de 64 Mibits que tienen 8 líneas de datos.
 - a- ¿Cuánta memoria tiene en cada banco?
 - b- ¿Cuántos bancos tiene?
 - c- ¿Cuántas líneas de direcciones deben ir del procesador a la memoria? ¿Cuántas se usan para direccionar cada banco? ¿Cuántas para seleccionar el banco?
- 4- Se tiene una memoria, direccionable a palabra, con 4Mi posiciones cada una de 4 bytes. Se sabe que fue implementada con 4 chips idénticos.
 - a- ¿De cuántas maneras podría estar armada esa memoria? Es decir, ¿de cuántas maneras se pueden organizar 4 chips para configurar el total de memoria?
 - b- Para cada una de las configuraciones que haya encontrado, describa cómo serían los chips (cuántas líneas de datos y de direcciones tendría cada chip).
- 5- Se tiene un procesador con m bits de direcciones.
 - a- Si se tienen 2^k bancos, ¿cuántos bits de direcciones se deben usar para seleccionar el banco?

- b-** Si se tienen k bancos, ¿cuántos bits de direcciones se deben usar para seleccionar el banco?
- c-** Si se usan k bits de direcciones para seleccionar el banco, ¿cuánto es la capacidad de cada banco?
- 6-** Se tiene un procesador con 32 bits de direcciones (A0 a A31). La memoria se implementa usando 4 bancos.
- a-** Si para seleccionar el banco se utilizan las 2 líneas menos significativas — en lugar de las más significativas como se hizo en el texto—, ¿cómo quedan distribuidas las direcciones en los bancos?
- b-** Y si para seleccionar el banco se utilizan las A2 y A3, ¿cómo quedan distribuidas las direcciones en los bancos?
- c-** Generalizando, si se usan las líneas A j a A $j+k$, ¿cuántos bancos hay?, ¿cómo quedan distribuidas las direcciones en los bancos?
- 7-** Se piensa hacer un computador de 8 bits con 16 bits de direcciones, lo cual permite tener hasta 64Kbytes de memoria. Se quiere dividir la memoria en 4 bancos:

8K	-	Direcciones más bajas.
16K		
32K		
8K	-	Direcciones más altas.

Los chips tienen los mismos pines usados en los ejemplos. ¿Cómo deberían ser los decodificadores para este esquema de memoria?

- 8-** Ocasionalmente puede ocurrir que los chips de memoria de un computador fallen; para detectar esto se usan bits de paridad. La idea es almacenar cada palabra de n bits usando $n+1$ bits, el último de los cuales sirve para la paridad. Cuando se almacena una palabra, se calcula el bit de paridad y se almacena junto con los datos; cuando se lee una palabra se recalcula el bit de paridad y se compara con el almacenado, si no son iguales, ha ocurrido un error.
- a-** Supongamos que se tiene un procesador con palabra de 32 bits y direccionamiento a palabra. Se quiere detectar errores de un bit en una palabra, ¿cuál es el sobrecosto en memoria en el que hay que incurrir para implementar este mecanismo?
- b-** Si el procesador tiene direccionamiento al byte, y se quiere detectar errores de un bit en cada byte, ¿cuál es el sobrecosto en memoria en el que habría que incurrir?
- 9-** Con respecto al punto anterior, si además de detectar un error se desea corregirlo es necesario recurrir a códigos de corrección de errores, como los códigos de Hamming. En este caso se almacena cada palabra de n bits usando $n+k$ bits, los k últimos sirven como bits de paridad del código de Hamming. Cuando se almacena una palabra, se calcula el código de Hamming y se almacena junto con los datos; cuando se lee una palabra se recalculan los bits

de paridad y se comparan con los almacenados, basados en las diferencias, se puede corregir el bit dañado.

a- Supongamos que se tiene un procesador con palabra de 32 bits y direccionamiento a palabra. Se quiere poder corregir un error de un bit en una palabra, ¿cuál es el sobrecosto en memoria en el que hay que incurrir para implementar este mecanismo?

b- Si el procesador tiene direccionamiento al byte, y se quiere corregir errores de un bit en cada byte, ¿cuál es el sobrecosto en memoria en el que habría que incurrir?

- 10-** Se quiere construir una memoria con palabras de 2^k bits. Se están considerando dos posibilidades: usar chips de $2^n \times 1$ bits o de $2^{n-k} \times 2^k$ bits. La memoria debe tener $a \cdot 2^n$ palabras. ¿Cuántos chips se necesitan en cada una de las dos opciones?