4. Arquitectura del procesador y de la memoria

Este capítulo constituye una primera aproximación a la arquitectura y funcionamiento de los procesadores. En los capítulos siguientes se profundiza en su operación y forma de uso.

En las secciones que siguen, se describirá la arquitectura del procesador y de la memoria. Es decir, la descripción de sus elementos desde un punto de vista funcional; sin importar cómo se construyen físicamente, sino cómo funcionan. Posteriormente, y a manera de ejemplo, se presenta el caso particular de la IA32.

Empecemos por aclarar un poco de terminología. Un computador está compuesto de diversas partes, dispone, por ejemplo, de unidades para comunicarse con el mundo exterior: teclado, pantalla, unidades de disco etc. todas estos son conocidos bajo el nombre de *periféricos* o *unidades de entrada-salida*.

La parte más importante del procesador es la encargada de ejecutar las instrucciones, la que entiende el lenguaje de la máquina: *el procesador*. Es él quien se encarga de controlar los demás componentes.

No es lo mismo un procesador que un computador, el computador está compuesto por un procesador y muchos más elementos; es decir, el computador es el sistema completo. En nuestro caso, podemos decir que un Pentium es un procesador, mientras que un PC es un computador.

Vamos a empezar con el procesador y la memoria, los cuales constituyen la esencia de un computador. El primero, el procesador, es la parte "inteligente"; es él quien interpreta y ejecuta las instrucciones. La memoria, por su lado, sirve de "papel y lápiz" para el procesador: guarda las instrucciones y se las pasa al procesador cuando este las pide, comunica valores almacenados o recibe valores para almacenar, según lo que le ordene el procesador.

4.1 ARQUITECTURA DE LA MEMORIA

Una forma de visualizar la memoria de un computador es como una secuencia de "casillas" numeradas: la dirección cero corresponde a la primera casilla, la 1 a la segunda y así sucesivamente. Cada casilla puede almacenar un valor en binario, así que cada una tiene dos números asociados: su dirección —que es fija— y su contenido —que puede cambiar—.

Las posiciones de memoria se pueden comparar con las variables de los lenguajes de programación. La dirección se puede ver como el "nombre" de la posición. Las posiciones tienen un tamaño fijo, el cual depende del computador; puede ser 8, 16 ó más bits.

Para acceder a la memoria, los procesadores tienen instrucciones que les permiten bien sea leer, bien sea escribir, una posición determinada de la memoria. Las instrucciones de lectura especifican una dirección de donde se leerán los datos; las instrucciones de escritura especifican la dirección donde se escribirá, así como el dato que se escribirá.

El hecho de que las posiciones de memoria se identifiquen con un número introduce un efecto interesante: puesto que una dirección es un número, se puede almacenar en otra variable como cualquier dato. De aquí parte el concepto de *referencia*: el contenido de una variable se refiere a otra, es decir, permite tener acceso a la otra variable. Decimos que la primera variable es de tipo *apuntador*, y que *apunta* a la segunda.

Direcciones y datos

Los procesadores trabajan con varios tipos de datos, cada uno de los cuales puede tener un tamaño diferente —un byte para los caracteres, 2 ó 4 bytes para los enteros, etc. —. En consecuencia, los procesadores necesitan tener la capacidad de leer o escribir en la memoria entidades de diversos tamaños. Esto se puede lograr de tres formas:

- Por hardware: el procesador puede leer entidades de diferentes tamaños de la memoria. Esto implica que la memoria esté diseñada para soportar los diversos tipos de acceso, y que el procesador tenga las instrucciones para especificarlos. Así, habrá una instrucción para leer un byte de memoria, otra para leer entidades de dos bytes, etc.
- Por software: la memoria solo permite acceder a entidades de un cierto tamaño fijo.[1] El manejo de accesos a otros tamaños, debe realizarse por software. Por ejemplo, si la máquina solo permite el acceso a datos de 32 bits, y se desea leer un byte, el programa deberá leer los 32 bits donde se

[1] El tamaño es fijo, pero depende del tipo de procesador: en algunos podría ser 32 bits, en otros, 64 bits, etc. Esto está relacionado con el llamado “tamaño de palabra” del procesador.

encuentra el dato para después, por medio de operaciones aritméticas y lógicas, extraer el dato que le interesa.[2]

- Mixto: como en el caso anterior, la memoria solo permite acceder a entidades de un cierto tamaño fijo. Sin embargo, el procesador tiene instrucciones especiales para extraer el dato deseado de manera automática —sin tener que hacer operaciones aritméticas o lógicas—.

Note que, en el primer caso, el procesador debe tener varias instrucciones para leer, o escribir, en la memoria: por lo menos una por cada tipo de dato que se desee leer.

Surge una pregunta: si el problema se puede resolver por hardware, ¿por qué no hacerlo así? ¿No es más fácil para el programador? En efecto lo es, sin embargo, para lograrlo, el acceso a la memoria se vuelve más complejo y demorado. Como veremos más adelante, la memoria es un cuello de botella en los computadores actuales; por eso, algunos procesadores ofrecen solo accesos de tamaño fijo; otros incurren en el costo del acceso porque es usual trabajar con entidades de diferentes tamaños.

Note que la situación anterior se puede resolver de diversas maneras: por hardware, por software o una combinación de los dos. Estas soluciones de compromiso son bastante usuales en los computadores; los límites entre el hardware y el software no están precisamente definidos y no son inamovibles: diferentes procesadores pueden tener diferentes concepciones sobre qué se hace por hardware y qué, por software.

En los procesadores que permiten direccionar entidades de diferente tamaño, la memoria se suele dividir en bytes, y las direcciones se asignan a estos. Por esto se suele hablar de dos modelos de memoria: direccionable a palabra y direccionable a byte (fig. 4.1):

- En el primero, la palabra está compuesta por varios bytes, pero estos no son direccionables independientemente: se debe leer toda la palabra; en otros términos: las direcciones se asignan a las palabras.

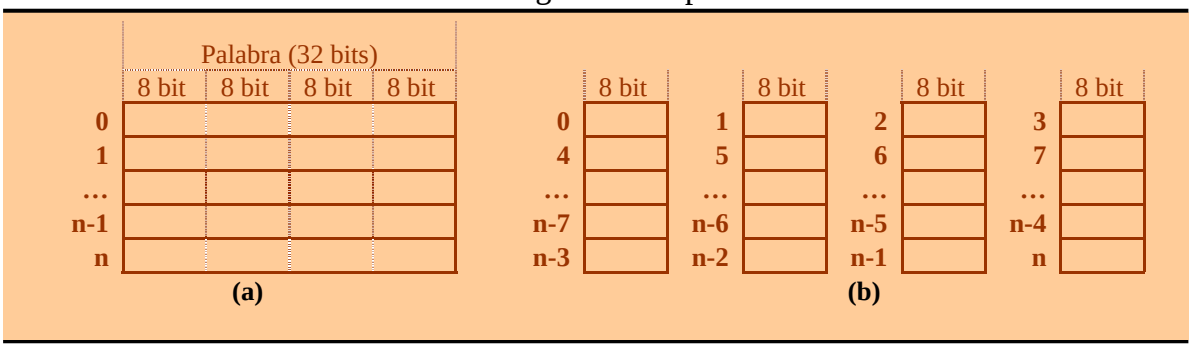

Fig. 4.1. (a) Memoria direccionable a palabra. (b) Memoria direccionable a byte.

[2] Es un problema similar al siguiente: suponga que le dan un número decimal y le piden extraer el tercer dígito menos significativo. ¿Cómo proceder? Una posibilidad es sacar el módulo 1000 —con lo cual obtenemos los tres dígitos menos significativos—, y dividir el resultado por 100 —con lo cual desaparecemos los dos dígitos menos significativos—.

- En el segundo, las direcciones se asignan a los bytes; así que es posible direccionar un solo byte. También se pueden direccionar palabras; en la figura, la primera palabra de memoria está compuesta por los bytes 0, 1, 2 y 3. Por supuesto, el procesador deberá tener dos instrucciones diferentes para indicar si se desea leer un byte o una palabra.

Esto nos conduce a otra pregunta: ¿se puede direccionar una palabra a partir de cualquier byte? Por ejemplo, en la figura, ¿se puede direccionar la palabra compuesta por los bytes 2, 3, 4 y 5? Depende del procesador, en algunos sí, en otros no.[3] Esto se conoce como *restricción de alineamiento*:

- En los procesadores que no tienen restricciones de alineamiento, como los de Intel, se puede direccionar un dato en cualquier dirección de memoria, sin importar su tamaño.
- Los que tienen restricciones imponen normas, que se podrían resumir de la siguiente manera: *un dato solo se puede localizar en direcciones múltiplos de su tamaño*. Por ejemplo, un byte es de tamaño uno, así que se puede encontrar en cualquier dirección; un entero de 4 bytes, solo puede estar en direcciones múltiplos de 4, etc.

Las restricciones de alineamiento se imponen por la siguiente razón: si observa en la figura 4.1, los bytes 2 y 3 se encuentran en una línea de memoria, y los 4 y 5, en otra. En un acceso a memoria, solo se puede leer o escribir una línea; por ende, para acceder a la palabra compuesta por los bytes 2, 3, 4 y 5, es necesario efectuar dos accesos a la memoria. Es posible hacerlo, pero al precio de complicar el sistema de acceso a la memoria; y, como se dijo anteriormente, en algunos procesadores se prefiere no hacerlo por razones de eficiencia.

Tamaño de palabra

Con frecuencia se oye decir que un procesador es de 8 bits, o de 16, o de 64. Esto está relacionado con el concepto de *tamaño de palabra*; de hecho, la expresión correcta es “el procesador tiene palabra de 32 bits”.

El concepto de “palabra” hace referencia al número máximo de bits que se pueden procesar en una sola operación; es decir, cuántos bits, máximo, se pueden obtener en una sola lectura de memoria, o cuántos se pueden sumar en una sola operación de suma, etc.

Sin embargo, no es un concepto bien definido. Por ejemplo, se dice que el Intel 8086 tiene palabra de 16 bits; en efecto, puede acceder a la memoria en grupos de 16 bits, y puede operar sobre datos de 16 bits (sumar, restar, etc.). Ahora bien, el Intel 8088 es un procesador cuya arquitectura y conjunto de instrucciones son idénticos a los del 8086;[4] sin embargo, internamente, procesa la información en grupos de 8 bits: si se le ordena leer 16 bits de memoria, efectúa dos lecturas de 8

---

[3] Por ejemplo, los procesadores Intel lo permiten; otros, como el MIPS, no lo permiten.

[4] Al margen: fue el Intel 8088, no el 8086, el procesador que IBM utilizó en el PC original.

bits y las ensambla en un solo dato; si se le ordena sumar dos números de 16 bits, efectúa dos sumas de 8 bits y ensambla el resultado.[5]

En estas circunstancias, ¿es el Intel 8088 un procesador de 8 ó de 16 bits? La respuesta depende de si se mira el hardware o el software: las instrucciones especifican operaciones de 16 bits, pero el hardware procesa en grupos de 8 bits.

También ocurre lo contrario: la interfaz con memoria del Pentium original es de 64 bits, sin embargo, las instrucciones especifican operaciones enteras de 32 bits y procesan los datos en grupos de 32 bits.

Ahora, por otro lado, en el caso particular del PC, se acostumbra entender "palabra" como 16 bits, doble palabra como 32 bits y cuádruple palabra como 64 bits. Esto es una herencia que dejo el Intel 8086 (cuyo tamaño de palabra era de 16 bits).

"Endianess"

Es usual almacenar números enteros en palabras. Aquí se debe tomar una decisión: ¿se almacenan de "izquierda a derecha" o de "derecha a izquierda"? (ver fig. 4.2) La respuesta más natural es de "izquierda a derecha" puesto que así escribimos los números. Sin embargo, en la memoria no hay tal cosa como "izquierda" y "derecha" —¡por eso lo escribimos entre comillas!—. La respuesta viene condicionada por el hecho de que, en el dibujo, numeramos la memoria de izquierda a derecha. Pero si se numera en la dirección contraria, la respuesta es diferente.

La forma correcta de plantear la pregunta es: en un computador con palabras de *n* bytes, al almacenar un número en las direcciones *i* a *i*+*n* ¿Los dígitos más significativos van en la dirección *i* o en la dirección *i*+*n*? La respuesta parece ser: ¿qué importa? Efectivamente, desde un punto de vista formal, la elección no tiene importancia: en el mercado se consiguen procesadores de uno u otro tipo. En realidad, sí hay un criterio para elegir, sólo que es completamente subjetivo: para los occidentales, es más fácil imaginar los dígitos numerados de "izquierda a derecha".[6]

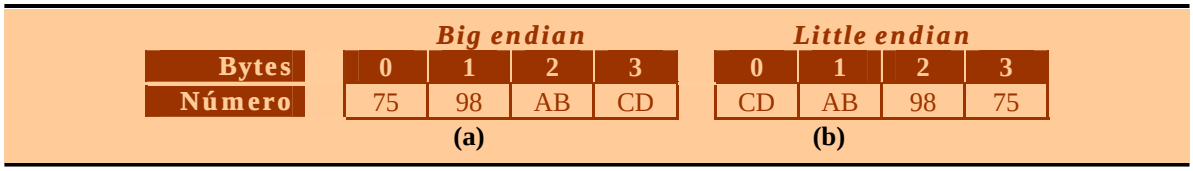

Fig. 4.2. El número 7598ABCDH representado (a) de "izquierda a derecha" (*big endian*). (b) de "derecha a izquierda" (*little endian*)

[5] Lo mismo pasa con otros procesadores: el Intel 80386 (32 bits) y el 80386 SX (16 bits en el hardware). El Motorola 68000 tenía instrucciones para operar sobre datos de 32 bits, pero internamente procesaba grupos de 16 bits.

[6] Se tiende a diseñar procesadores *big endian*; de alguna manera, lo seres humanos lo encontramos más "natural". Los procesadores *little endian*, en general, corresponden a "accidentes históricos": no se pensó que fuera mejor hacerlos así, sino, por giros del destino, acabaron siéndolo. Aunque hay procesadores donde este aspecto sí fue considerado explícitamente y hay razones para que sean *little endian* —fue el caso del Transputer, de la compañía Inmos—.

Los computadores que numeran de "izquierda a derecha" son llamados *Big endian a nivel byte*, puesto que empiezan a numerar a partir del dígito más significativo (el más "grande"); este es el caso del Motorola 68000. Los que numeran al contrario son llamados *little endian a nivel byte;* que es el caso del Intel 8086. Otros procesadores, como el MIPS, pueden elegir en cuál modo van a funcionar (conocidos como *bi-endian*).

Interacción procesador-memoria

La memoria interactúa con el procesador de dos maneras:

- En lectura la memoria recibe una dirección, enviada por el procesador, y retorna el contenido de dicha dirección.
- En escritura, la memoria recibe un dato y la dirección donde debe almacenarlo; el dato recibido será el nuevo contenido de la dirección indicada.

Además, el procesador envía señales de control para indicar a la memoria la operación que se desea realizar —leer (RD: *ReaD*) o escribir (WR: *WRite*)—. En la figura 4.3, puede verse el esquema de comunicación entre memoria y procesador.

En la figura 4.3, sólo hay una línea por entidad pero en realidad son varias. Los datos, por ejemplo, tienen tantas líneas como bits hay en cada posición de la memoria; cada línea transmite un bit, y estos conforman la palabra que se va a leer o a escribir. Por ende, las líneas de datos determinan la capacidad de almacenamiento en cada posición de la memoria.

La dirección también está compuesta de varias líneas, las cuales forman un número binario que se toma como la dirección donde se desea acceder. Si un procesador tiene n líneas de direcciones, puede direccionar 2^n posiciones; cada dirección es un número binario de n bits (0 a 2^n -1). En consecuencia, las líneas de direcciones determinan la capacidad de memoria en términos de cuántas posiciones puede tener.

La capacidad total de la memoria viene determinada por los dos factores anteriores: cuántas posiciones hay y cuántos bits se pueden almacenar en cada una de ellos. En consecuencia, si hay n líneas de direcciones, y cada posición tiene m bits, el total de memoria, en bits, será: $m*2^n$. En bytes, será esa cantidad dividida por 8.

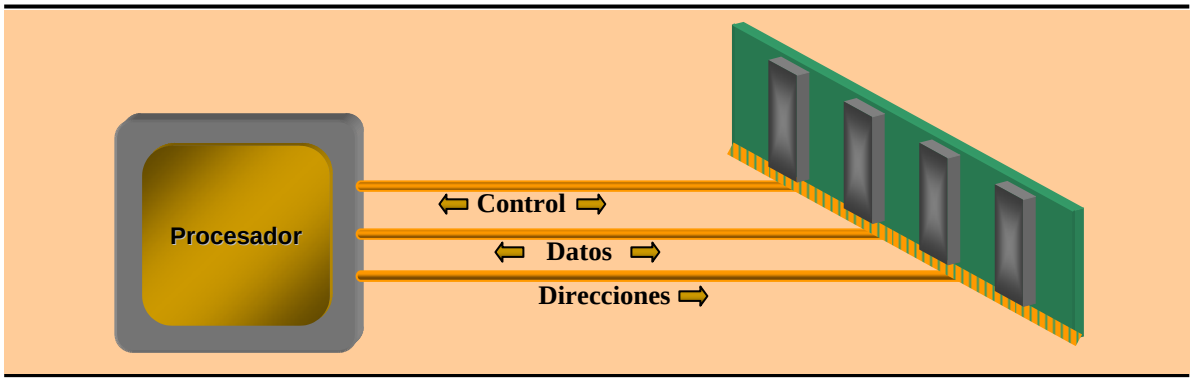

Fig. 4.3. Procesador-memoria

Nada determina cuántas líneas de direcciones debe haber. El diseñador del procesador elige el número de líneas según la cantidad de memoria que desee proveer. Pero un computador no está obligado a tener toda la memoria que puede direccionar; es decir, puede haber "huecos" en la memoria, zonas que el procesador puede direccionar pero que no existen físicamente.

De las líneas de control no nos vamos a ocupar por el momento. Baste con saber que, en el caso de los accesos a la memoria, sirven para indicar la acción que se desea realizar (lectura o escritura). En el caso general, permiten intercambiar información de control entre el procesador y la memoria.

Tipos de memoria

Las memorias se pueden clasificar según varios conceptos; uno de ellos es su grado de permanencia, o inmutabilidad:

- RAM (*Random Acces Memory*): en las memorias de este tipo se puede leer y escribir. Es posible, por ejemplo, escribir un dato, leerlo y después escribir un nuevo dato borrando el primero. En general, estas memorias son volátiles, es decir, al desconectar la corriente pierden toda la información almacenada.[7]
- ROM (*Read Only Memory*): es una memoria de sólo lectura, no es posible almacenar datos nuevos. Esta memoria se graba una sola vez, en la fábrica, después no puede ser modificada. Son memorias no volátiles.
- PROM (*Programmable* ROM): la diferencia con la memoria ROM reside en la forma de fabricarla. El usuario graba la PROM con aparatos especiales. Una vez grabada, se comporta como una ROM: no se puede modificar, y guarda la información permanentemente.
- EPROM (*Erasable* PROM): es parecida a la memoria PROM, pero es posible reprogramarla. En efecto, un "baño" de luz ultravioleta la hace perder su contenido y puede ser reprogramada a continuación. Tiene un número limitado de reescrituras.
- EEPROM (*Electrically Erasable* PROM): este es un tipo de memoria ROM que puede ser borrada eléctricamente, sin tener que recurrir a elementos externos —como luz ultravioleta—. Se puede borrar por pedazos; no es obligatorio borrarla toda de un golpe. También se puede escribir por pedazos. La memoria *flash* es un ejemplo de EEPROM. Tiene un número limitado de reescrituras, pero bastante más grande que las EPROM —cientos de miles—.

La utilidad de la memoria RAM es comprensible a primera vista. La utilidad de la ROM puede serlo un poco menos. Un comienzo de respuesta es que un computador siempre tiene que estar ejecutando un programa; ahora bien, cuando se activa el

[7] Pero no necesariamente. Por ejemplo, antiguamente, se usaban memorias magnéticas —de núcleos de ferrita—; estas memorias conservaban la información incluso sin corriente. Se han desarrollado memorias magnéticas integradas, pero todavía no son competitivas con las memorias actuales.

computador, ¿cuál programa va a ejecutar? Aquí entra en juego la memoria ROM: se almacena un programa en ROM, de tal manera que, cuando el computador empiece a funcionar, tenga un programa para ejecutar. Además, algunas rutinas básicas del sistema siempre están en memoria, así que se pueden guardar en ROM para evitar cargarlas del disco cada vez.

Medida del tamaño de la memoria

Hay un conjunto de diferentes unidades para describir el tamaño de la memoria. En primer lugar, la unidad básica es el bit; después, el byte, que, como ya se mencionó, consta de 8 bits.[8]

Las unidades superiores son conjuntos de bytes de diversos tamaños. Para notarlos, se usa una letra mayúscula (ver la siguiente tabla), que indica el tamaño, seguida de la letra B (de Byte).[9] Aunque la idea es simple, lamentablemente, se han generado algunas confusiones alrededor de ella. Procederemos a explicarlas.

Sigla	Nombre	Potencia de 10	Potencia de 2	Cantidad (2^n=)
K	kilo	10^3	2^{10}	1024
M	mega	10^6	2^{20}	1 048 576
G	giga	10^9	2^{30}	1 073 741 824
T	tera	10^{12}	2^{40}	1 099 511 627 776
P	peta	10^{15}	2^{50}	1 125 899 906 842 624
E	exa	10^{18}	2^{60}	1 152 921 504 606 846 976
Z	zetta	10^{21}	2^{70}	---
Y	yotta	10^{24}	2^{80}	---

Estas siglas pueden interpretarse de dos maneras dependiendo del contexto: en algunos contextos, se interpretan como potencias de diez, y, entonces, la secuencia: K, M, G, T corresponde a mil, millón, millardo y billón,[10] como puede verse en la tabla anterior. En otros contextos, se interpretan como potencias de 2, en cuyo caso son cercanos a los valores anteriores, sin ser iguales. Por ejemplo, con la interpretación base 2, 1 K equivale a 1024, que es cercano a mil.

En ciencias, la interpretación es decimal. En computación, suele ser binaria. Esto se debe a que el tamaño de la memoria es, en general, una potencia de dos, así que es más fácil expresarlo con la interpretación base 2.

---

[8] O, en la práctica, se suele interpretar así. Formalmente un byte es una cierta cantidad de bits, no necesariamente 8. Si se quiere se estricto, los bytes de 8 bits de deben llamar "octetos", cuyo símbolo es o.

[9] Tradicionalmente se usa B para indicar Byte y b para indicar bit. Desafortunadamente, en ocasiones no se respeta esta convención, con lo cual 1 Kb puede ser 1K bytes o 1K bits, y solo se puede determinar cuál es por el contexto.

[10] Otra desafortunada confusión: "billón", en español, es un millón de millones; en inglés, es mil millones. Mil millones, en español, se llama "millardo".

Pero no siempre es así: si bien la memoria RAM se suele expresar usando la interpretación binaria, el tamaño de los discos se expresa en ocasiones en decimal y en ocasiones en binario. Las velocidades de transmisión frecuentemente son decimales, así como las velocidades de reloj. Incluso, en ocasiones, se mezclan los dos sistemas; es el caso de los disquetes llamados "de 1.44 MB": en realidad se trata de 1.44*1000*1024.

La situación anterior ha generado mucha confusión, con los consecuentes errores y mal interpretaciones sobre las magnitudes, lo cual ha motivado la propuesta de darle un nombre diferente a las unidades binarias: la idea es tomar las dos primera letras del nombre decimal y adjuntarle el sufijo "bi" —de *bi*nario—, con lo cual las unidades binarias pasarían a ser: Kibi, Mebi, Gibi, Tebi, Pebi, Exbi, Zebi y Yobi. Su sigla es la misma decimal con una "i" al final: Ki, Mi, Gi, Pi, Ei, Zi y Yi. Aunque ya se ha formalizado esta propuesta,[11] todavía no está muy difundida.

Anteriormente era usual describir la capacidad de memoria en kilobytes (KB; o, mejor, según se explicó anteriormente, se trata de kibibytes, KiB); hoy en día, con el aumento de las capacidades de memoria, se acostumbra hacerlos en megabytes (MB; mismo comentario: en realidad son mebibytes, MiB). Esto en lo relacionado con la memoria principal; la memoria secundaria suele expresarse en gigabytes (GB) y terabytes (TB). La capacidad de los chips de memoria individuales suele expresarse en kilobits (Kb) o megabits (Mb).

Tamaños de los tipos de datos

Dependiendo de su tipo de dato, las variables ocupan más o menos bytes. Sobre esto no hay acuerdos y depende mucho de la máquina y del lenguaje de programación utilizados. Sin embargo, podemos mencionar algunas generalidades y algunos valores típicos:

Tipo de datos	Notación en C	Tamaño (bytes)
carácter	char	1
entero	int	4
entero corto	short int	2
entero largo	long int	4
entero largo largo	long long	8
punto flotante	float	4
flotante doble	double	8
apuntador	T *	4

Estos son valores típicos, sin embargo, se debe tener en cuenta que el estándar de C no especifica tamaños obligatorios, así que pueden variar entre implementaciones.

Los apuntadores se constituyen en un caso particular. En primer lugar, no necesariamente tienen el tamaño de un entero; en efecto, aunque es deseable, nada obliga a que una máquina maneje direcciones y datos del mismo tamaño.[12] Con esta

[11] Estándar IEC 60027-2. También contemplado en el estándar IEEE 1541.

[12] Sin embargo, este es el caso en la IA32: apuntadores y enteros del mismo tamaño.

aproximación surge el problema de tener dos conjuntos de registros: unos para manejar datos y otros para direcciones, y los registros de un conjunto tienen tamaño diferente a los del otro; la misma situación se presenta en memoria: el tamaño de los apuntadores es diferente al de los enteros. De hecho puede suceder que el tamaño de un apuntador dependa del tipo de la entidad apuntada; esto parece extraño pero puede ocurrir en máquinas con direccionamiento a palabra: si la entidad apuntada cabe en una palabra, o en un conjunto de palabras, se puede usar un apuntador simple; pero si la entidad es de un tamaño inferior —por ejemplo, un carácter—, el apuntador debe constar de dos partes: un apuntador a la palabra donde se encuentra la entidad y un desplazamiento precisando su posición dentro de la palabra. En cualquier caso, el estándar de C exige que un apuntador de tipo `void *` tenga el tamaño suficiente para contener cualquier tipo de apuntador.[13]

El caso de Java es distinto, puesto que fue diseñado para ejecutar sobre una máquina virtual —la JVM, *Java Virtual Machine*—, el tamaño de sus tipos de datos está estrictamente definido (hace parte de la definición de la JVM). En general, son como se describió para C, excepto por: los `long int` son de 8 bytes, los `char` son caracteres Unicode representados en 2 bytes —UTF-16— y existe un tipo adicional, `byte`, que ocupa un byte. También tiene un tipo `boolean`, pero el estándar no especifica su tamaño.

En términos generales, los tipos de datos numéricos enteros pueden presentarse en diversos tamaños, pero otros tipos de datos suelen estar sometidos a más restricciones. Por ejemplo, los números de punto flotante frecuentemente respetan el estándar IEEE 754, el cual especifica dos tamaños de números: 4 y 8 bytes; se acostumbra identificar estos dos tamaños con `float` y `double`. Por otro lado, los caracteres se ciñen a algún estándar —ASCII, Unicode u otros— el cual determina su tamaño —1 byte, en el caso de ASCII; 1, 2 ó 4, en el de Unicode, dependiendo de si se usa UTF-8, UTF-16 ó UTF-32—.

En cuanto a los tipos de datos compuestos —arreglos, estructuras, objetos—, la situación es un poco más compleja.

Vectores

En memoria, los vectores se almacenan poniendo sus elementos uno a continuación del otro a partir de una cierta dirección de memoria. En consecuencia, un vector se caracteriza por su dirección inicial (d) el número de elementos (n) y el tamaño en bytes de cada uno de estos (k).

El tamaño de un vector es el número de elementos multiplicado por el tamaño en bytes de un elemento ($n*k$).[14]

---

[13] De hecho, para eso existe; si no, no tendría ningún sentido un “apuntador a nada”. El `void *` es una suerte de apuntador genérico, de apuntador a cualquier cosa.

[14] sin embargo, debido a las restricciones de alineamiento, puede ser un poco mayor; ver la descripción de las estructuras.

Con respecto a los elementos, podemos caracterizar su posición de tres maneras distintas:

- Subíndice en el vector: este es un concepto lógico; los elementos se numeran a partir del primero, uno a continuación de otro, empezando desde cero. Es el mismo subíndice que usan los lenguajes de alto nivel.
- Desplazamiento: es el número de bytes que hay desde el comienzo del vector hasta un cierto elemento con subíndice *i*. Es igual a *i***k*. Esto es una forma de *direccionamiento relativo*: se puede ver como la dirección del elemento relativa al comienzo del vector.
- Dirección: corresponde a la dirección absoluta; es igual a la dirección inicial del vector más el desplazamiento del elemento en cuestión (*d* + *i***k*).

En cuanto a las matrices, se pueden representar de dos maneras (ver fig. 4.4):

- Como un vector de apuntadores a vectores. Que corresponde con la declaración en C: `int * v[n];`
- Como un vector cada uno de cuyos elementos es un vector. Que corresponde con la declaración en C: `int v[n][m];`

En este último caso, note que el elemento `v[i][j]` es el j-ésimo elemento del i-ésimo vector, puesto que `v` es un vector de `n` elementos, cada uno de los cuales es un vector de `m` enteros. El tamaño de cada uno de los `v[i]` es `m*4`, luego el tamaño de `v` es `n*m*4`. En consecuencia, `v[i]` tiene un desplazamiento de `i*m*4`, y `v[i][j]` tiene un desplazamiento de `i*m*4 + j *4`. Lo anterior se puede generalizar a más de 2 dimensiones.

Note que, dada la anterior representación, una matriz de *n* x *m*, se puede ver como un vector de tamaño *n***m*; es decir, las dos declaraciones siguientes pueden ser consideradas equivalentes en términos de espacio:

```
int v[n][m];
int v[n*m];
```

Es más, en ocasiones puede ser más práctico manejar una matriz como un vector.

El concepto de "vector de vectores" puede parecer extraño en una primera aproximación, pero en realidad es bastante natural. ¿Qué es un vector? Una secuencia ordenada y numerada de entidades. Estas entidades pueden ser tipos primitivos —enteros, caracteres, apuntadores, etc.—, o pueden ser tipos complejos: estructuras, objetos o, ¿por que no?, vectores.

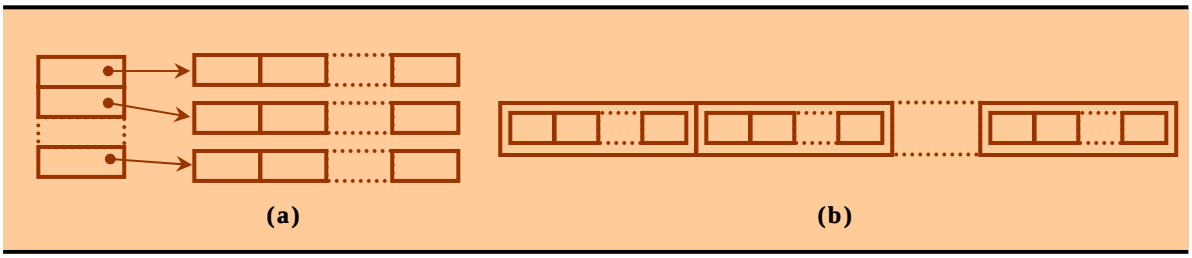

Fig. 4.4. Matriz representada como (a) vector de apuntadores (b) vector de vectores

Estructuras

Similar a los vectores, las estructuras se representan almacenando sus campos uno a continuación del otro, en consecuencia, su tamaño es la suma de los tamaños de sus campos. El desplazamiento de un campo es la suma de los tamaños de los campos anteriores.

Ahora, debido a las restricciones de alineamiento, puede ser necesario dejar "huecos" entre un campo y otro, con lo cual su tamaño se hace mayor. Por ejemplo, si el primer campo es un entero, el segundo un carácter y el tercero otro entero; en principio, esta estructura tendría 9 bytes. Sin embargo, si hay restricciones de alineamiento, es imposible que los dos enteros queden en direcciones múltiplos de 4; por lo tanto, es necesario dejar 3 bytes, sin nada, después del carácter. En este caso, cada estructura ocuparía 12 bytes (4+1+3+4) en lugar de 9 (4+1+4).

Objetos

En principio, este caso es como el de las estructuras —en memoria, los objetos se representan como estructuras—. Sin embargo, los objetos pueden tener "campos ocultos" —que el programador no ve— que aumentan su tamaño.[15]

Por otro lado, en el caso de Java, si un atributo es un objeto, este no se almacena con los otros atributos; en vez de esto, se almacena un apuntador al objeto, que se encuentra en otro sitio de la memoria. En consecuencia, el tamaño del sub-objeto no afecta el del objeto; solo ocupa los 4 bytes que requiere el apuntador.

En cuanto a C++, este permite elegir si un sub-objeto se incorpora al objeto que lo contiene, o si solo se guarda un apuntador al sub-objeto.

¿Se puede decir que un objeto "no es más que una estructura con otro nombre"? No. El concepto de objeto es una abstracción que implica cosas más allá de tener un conjunto de atributos; esta abstracción es implementada por los lenguajes de alto nivel, e implica muchas cosas, entre las cuales se cuenta el hecho de que en memoria los atributos se guardan en estructuras.

4.2 ARQUITECTURA DEL PROCESADOR

Empecemos por describir la interacción entre la memoria y el procesador desde el punto de vista de este último. Las instrucciones que componen un programa se almacenan en la memoria de manera secuencial; el procesador va leyendo las instrucciones una tras otra, ejecutando la acción indicada por cada instrucción. Dicha acción puede implicar nuevos accesos a la memoria para modificar los datos allí almacenados.

---

[15] Esto ocurre porque los objetos necesitan almacenar más información, como, por ejemplo, dónde se encuentran sus métodos.

Las instrucciones son del estilo "tome el primer operando, súmele el segundo operando y asigne el resultado al tercer operando" o "asigne el primer operando al segundo". Estos operandos pueden ser, por ejemplo, posiciones de memoria; tendríamos entonces instrucciones del tipo: "tome el contenido de la posición 7, súmele el contenido de la posición 10 y guarde el resultado en la posición 23".

Antes de entrar en los temas arquitectónicos propiamente dichos, veremos los componentes del procesador y su funcionamiento interno.

Componentes del procesador

Veamos el mecanismo necesario para realizar las acciones descritas en el párrafo anterior. Las partes de un procesador se muestran en la figura 4.5.

Registros (*register*): En una primera aproximación, se podría pensar en almacenar en la memoria todos los operandos de las instrucciones. Este esquema presenta dos inconvenientes: en primer lugar, las instrucciones son demasiado largas, puesto que tienen hasta tres direcciones de memoria para indicar los operandos.[16] En segundo lugar, los accesos a la memoria son demorados con respecto a la velocidad del procesador. Si hay un operando muy utilizado, se perdería tiempo trayéndolo cada vez desde la memoria.

Para resolver estos inconvenientes, se le da al procesador un poco de memoria local de alta velocidad. Esta memoria local, al igual que la externa, está compuesta por un grupo de "casillas" llamadas *registros*. De esta manera, las instrucciones serían del estilo: "sume el contenido de la posición 4 de memoria al quinto registro". Sólo es necesario guardar una dirección en la instrucción; además, si un operando es muy utilizado, puede guardarse en un registro y siempre estará a mano para las operaciones que se ofrezcan.

La cantidad de registros depende del procesador, puede haber sólo uno —lo cual ocurre solo en procesadores muy simples— o varios.[17] Las máquinas RISC suelen

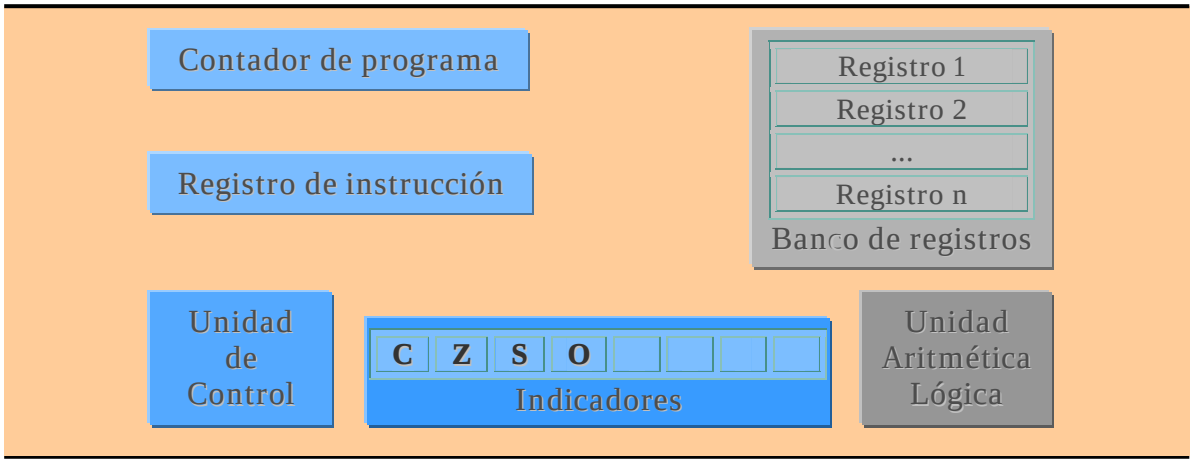

Fig. 4.5. Partes del procesador.

[16] Dos operandos fuente, que se leen, y un operando destino, donde se escribe.

tener 32, y hay algunas con más de 100 —como SPARC y EPIC—.

Vale la pena anotar que algunos registros están dedicados a tareas especiales —son de *propósito específico*—; es decir, hay algunas tareas que solo se pueden realizar en, o con, ellos. Este es el caso del registro llamado *apuntador de pila*.

La Unidad Aritmética-Lógica o UAL (ALU, *Arithmetic-Logic Unit*): se encarga de realizar las operaciones en sí. Tiene los circuitos de suma, resta, AND, etc. Recibe los operandos de la instrucción y efectúa la operación indicada sobre ellos.

Es una unidad pasiva; se limita a hacer lo que la CU le diga, sin saber para qué, o por qué.

El Contador de Programa o CP (PC, *Program Counter*, o IP, *Instruction Pointer*): Es un registro cuyo contenido indica cuál es la instrucción que se debe ejecutar. Puesto que un programa es una secuencia de instrucciones, es necesario que el procesador "recuerde" en cuál instrucción va. Ese es el papel del PC: sirve como un apuntador a la instrucción que se va a ejecutar.

Inicialmente este contador tiene la dirección en memoria de la primera instrucción del programa. Después de que una instrucción ha sido ejecutada, la unidad de control incrementa el contador para que quede apuntando a la instrucción siguiente.

Registro de instrucción (IR, *Instruction Register*): Es un registro interno; es decir, lo utiliza la unidad de control pero no el programador. Este registro pertenece a la estructura interna y no es necesario tenerlo en cuenta cuando se programa. Una vez mencionado, podemos desentendernos de él —atención: ¡eso no es lo mismo que olvidarlo! —.

La unidad de control lo utiliza para almacenar la instrucción que se está ejecutando; de ahí obtiene información como: de qué instrucción se trata, cuáles son las direcciones de los operandos, dónde se deja el resultado, etc.

La Unidad de Control o UC (CU, *Control Unit*): Es el "director de orquesta"; quien ordena las acciones que cada elemento debe ejecutar. Su funcionamiento sigue los siguientes pasos:

- Lectura de instrucción (*instruction fetch*): la CU envía el PC a la memoria y activa una línea de control —RD: *ReaD*— para indicar que se trata de una lectura. Después de un cierto tiempo, la memoria responde enviando el contenido de esa posición de memoria por las líneas de datos.

  La CU retira la información, que debe ser una instrucción, de las líneas de datos, y la almacena en el Registro de Instrucción.

- Decodificación de instrucción (*instruction decode*): la CU examina el IR para identificar de qué instrucción se trata —"sumar", por ejemplo—. También identifica el tipo y número de los operandos —dos para una suma; por

---

[17] Incluso puede no haber; algunas arquitecturas trabajan solo sobre la memoria, como las llamadas *arquitecturas de pila*. Hoy en día no son usuales; una excepción notoria es la JVM.

ejemplo: “posición 4 de memoria”, “registro 5”—, los obtiene, sea de la memoria, sea de los registros.

- Ejecución: la CU envía los operandos y el código de la operación ALU, la cual, después de un tiempo, pondrá el resultado en las líneas de salida.

  Al mismo tiempo, puede actualizar el PC; es decir, incrementarlo en el tamaño de la instrucción para que quede apuntando a la siguiente.

- Escritura del resultado: la CU toma el resultado de las líneas de salida de la ALU y lo transfiere al sitio donde debe quedar almacenado.

El proceso anterior se conoce como el *ciclo de ejecución*.

Como puede notar, la unidad de control también se encarga de enviar las señales de control al mundo externo; en particular, a la memoria.

Los indicadores (*flags*): Este es un registro, solo que no se utiliza para guardar números. En su lugar, cada bit indica algo sobre el estado del computador. Por ejemplo, uno de estos bits indica si el último resultado producido por la unidad aritmética fue cero o no (*bit de cero*); este bit se pone en 1 cuando el resultado es cero y en 0 cuando no.[18]

El número y función de los indicadores dependen del procesador.[19] A manera de ejemplo podemos mencionar: bit de signo, para indicar si el último resultado fue negativo o positivo; bit de desbordamiento (*overflow*), indica que el último resultado sobrepasó la capacidad de cálculo del procesador y es erróneo; el bit de cero antes mencionado, etc. Anteriormente habíamos dicho que el procesador recuerda el acarreo (*carry*): es en este registro de indicadores donde lo hace.

Cada uno de los operadores de la ALU afecta los bits indicadores según como lo determine el diseñador del procesador. De esta manera, cada vez que se efectúa una operación, los bits indicadores "recuerdan" las características del resultado. Solo recuerdan el último resultado, puesto que cada vez que se efectúa una operación, el nuevo resultado se sobrepone al anterior.

Los indicadores permiten definir instrucciones condicionales del tipo: "si el resultado fue cero entonces... " o "si el resultado fue negativo...". A partir de estas condiciones primitivas, se pueden desarrollar otras comparaciones; por ejemplo, si se quiere saber si dos números son iguales, se restan y se pregunta si la operación dio cero (indicador de cero vale 1). Algunas condiciones más complejas pueden requerir la verificación de varios indicadores.

Toda la estructura anterior es conocida como *Unidad Central de Proceso* o UCP (CPU, *Central Processing Unit*).

---

[18] Puede verlo como un valor lógico: 1 quiere decir “es cierto que dio cero”; 0 quiere decir “es falso que dio cero”.

[19] Incluso hay algunos procesadores que no tienen.

Para comprender mejor el funcionamiento, vamos a ver un ejemplo: supongamos que en la posición 203 de memoria, tenemos codificada la instrucción "sumar la posición 7 al registro 5". Los registros tienen los siguientes valores:

PC = 203
Registro 5 = -10

El contenido de la posición 7 de memoria es 4.

La secuencia de acciones es:

- La unidad de control —CU— busca la instrucción que debe ejecutar. Para esto envía el PC por las líneas de dirección, y activa la señal de control que indica lectura —RD—.
- La memoria recibe la dirección, 203 en el ejemplo, y, por la señal RD, sabe que se trata de una lectura. Pone el contenido de la posición 203 en las líneas de datos.
- La CU recibe lo enviado por las líneas de datos. Como la CU sabe que se trata de una instrucción, la almacena en el registro de instrucción.
- La CU decodifica la instrucción. Se da cuenta de que es una suma (suponemos que ese es el contenido de la posición 203). Extrae la dirección del operando en memoria, 7, lo envía por las líneas de dirección y activa la señal RD.
- La memoria recibe el 7 enviado, extrae el contenido de esta posición (4, según nuestras suposiciones) y lo envía por las líneas de datos.
- La CU le ordena a la ALU que reciba lo que está en las líneas de datos como primer operando.
- La CU mira, en el registro de instrucción, cuál es el segundo operando. Se da cuenta de que es el registro 5. Extrae el valor del registro (es decir -10) y le ordena a la ALU que lo tome como segundo operando.
- La CU le ordena a la ALU que efectúe una suma sobre sus dos operandos.
- La ALU realiza la suma, la cual da -3 (operandos -7 y 4). Pone el bit de cero en 0 —el resultado es diferente de cero—, y el bit de signo en 1 —es negativo—.
- La CU transfiere el resultado al registro número 5.
- La CU incrementa el PC, para que apunte a la siguiente instrucción.

La CU continúa efectuando el mismo ciclo para siempre —enviar PC por las líneas de dirección, etc.—.

Elementos arquitectónicos

Podemos definir la arquitectura de un procesador como su interfaz binaria; es decir la interfaz que ofrece para su programación en binario. En consecuencia, si dos máquinas tienen la misma arquitectura, las dos están en capacidad de ejecutar los mismos programas binarios.

En contraste, la estructura se ocupa de cómo implementar la arquitectura; note que dos máquinas pueden tener estructuras diferentes, mientras la arquitectura sea la misma, continúan siendo compatibles.

En lo que sigue, veremos los elementos principales que definen una arquitectura.

Tipos de datos

Parte de la definición de una arquitectura consiste en establecer los tipos de datos que soportará. No solo se trata de establecer los tipos, sino, también, sus características: representaciones y tamaños.

Arquitectura de la memoria

Son los factores que tratamos en la sección anterior: tamaño de palabra, granularidad —mínima entidad direccionable—, tamaños de entidades soportados, restricciones de alineamiento, capacidad y *endianess*.

Lenguaje de máquina

Las instrucciones que procesa la máquina están almacenadas en la memoria, lo cual implica que, como los datos, deben estar codificadas en binario. Las instrucciones, así como los tipos de datos, tienen unas convenciones para su representación; en el caso de las instrucciones, estas convenciones se llaman *lenguaje de máquina*.

Para definir estas convenciones, primero se debe decidir qué componentes debe tener una instrucción. Una instrucción no es más que una operación (suma, resta, etc.) que se efectúa sobre unos datos; lo cual nos define los elementos básicos que se necesita codificar. En consecuencia, la codificación de una instrucción se divide en dos campos importantes: el código de operación y el, o los, operando(s).

Los códigos de operación no son más que unos códigos arbitrarios que designan las diferentes operaciones. El número de bits asignados para el código de operación depende del número de instrucciones de que se quiera dotar la máquina. Supongamos, a manera de ejemplo, que se asignan 6 bits; dicha máquina no podría tener más de 64 instrucciones.[20]

En cuanto a los operandos, la situación es más compleja. En primer lugar, es necesario determinar cuántos operandos va a tener la instrucción:

- Tres operandos. Estas son instrucciones del estilo:

  ADD O_d , O_{f1} , O_{f2}

  Donde O_d es el *operando destino*, y los O_f son los dos *operandos fuente*. La operación efectuada es: $O_d \leftarrow O_{f1} + O_{f2}$. Este tipo es usual en los procesadores RISC.

- Dos operandos. Estas son instrucciones del estilo:

  ADD $O_{d\text{-}f1}$, O_{f2}

[20] Y podría tener menos; previendo que a futuro se desee adicionar otras instrucciones no previstas.

Donde $O_{d\text{-}f1}$ es al mismo tiempo el operando destino y uno de los operandos fuente, y O_{f2} es el otro operando fuente. La operación es: $O_{d\text{-}f1} \leftarrow O_{d\text{-}f1} + O_{f2}$. La IA32 es de este tipo.

- Un operando. En realidad son de dos operandos, pero uno de ellos es implícito; se trata de un registro especial llamado *Acumulador* —que notaremos *Acc*—. Son del estilo:

  ```
  ADD   O
  ```

  La operación es: $Acc \leftarrow Acc + O$. Solo se usa en procesadores muy sencillos.

- Cero operandos. En realidad son operaciones de cero o un operando. Corresponde a las arquitecturas de pila; más adelante volveremos a trata el tema, por el momento nos limitaremos a decir que estas máquinas tienen una pila —*stack*— de ejecución y las operaciones son del estilo:

  ```
  ADD
  ```

  Que saca dos valores del tope de la pila de ejecución, los suma y guarda el resultado de nuevo en la pila. Como se mencionó anteriormente, son poco usuales hoy en día, sin embargo, la JVM es de este tipo.[21]

Note que cuantos más operandos haya, más grande debe ser la instrucción, puesto que es necesario codificar cada uno de ellos.

Cuando hablamos del número de operandos, estamos hablando en términos generales; por supuesto, en una máquina concreta el número de operandos puede variar según la operación, por ejemplo, no es lo mismo una operación binaria —como la suma— que una operación unaria —como la negación—.

En segundo lugar, es necesario decidir qué tamaños de operandos se va a soportar: ¿solo palabras?, ¿o también bytes o medias palabras? Esto incide en la representación, porque, por ejemplo, si un operando está en la memoria, no basta con codificar la dirección donde se encuentra, sino que también es necesario codificar, de alguna manera, de qué tamaño es el acceso.

En tercer lugar, es necesario decidir la localización de los operandos: donde pueden estar. Esto lo trataremos en la parte de modos de direccionamiento; por el momento digamos que pueden estar básicamente en tres sitios: en la memoria, en los registro o en la instrucción misma.

Todo lo anterior reunido determina el tamaño total de las instrucciones. Algunas arquitecturas, como la IA32, tienen instrucciones de tamaño variable, es decir, no hay un tamaño estándar; en otras, como las RISC, todas las instrucciones tienen el mismo tamaño.

Para cerrar este tema, mencionemos que se suele hablar de varios tipos de instrucciones:

---

[21] Otro ejemplo típico es la forma de utilizar las calculadoras Hewlett-Packard.

- De movimiento. Son instrucciones que solo mueven los datos: de un registro a otro, de memoria a los registros o viceversa; son el equivalente de la asignación de los lenguajes de alto nivel.
- Aritméticas-lógicas. Efectúan alguna operación —suma, multiplicación Y-lógico, etc.— sobre los operandos.
- De control. Los programas, normalmente, se ejecutan en secuencia: una instrucción tras otra. Estas instrucciones permiten cambiar el orden de ejecución: saltarse una o más instrucciones, hacia adelante o hacia atrás, de manera incondicional o condicional —i.e. que efectúan el salto obligatoriamente o que pueden efectuarlo o no dependiendo de alguna condición—. Sirven para implementar las construcciones de control de los lenguajes de alto nivel —if, while, for, etc.—.
- Del sistema. Son instrucciones especiales que solo utiliza el sistema operativo, le permiten controlar el funcionamiento de la máquina.

Esto introduce una consideración de diseño: es necesario elegir las operaciones que se consideren convenientes o necesarias para el conjunto de instrucciones.

Arquitectura de registros

Con esto hace referencia a la manera como están organizados los registros. En principio, todo lo que se debe decidir es cuántos registros hay y cuál es su tamaño. Sin embargo, en la práctica se presentan otros aspectos que es necesario considerar.

A grandes líneas, podemos hablar de dos tipos de registros:

- De propósito general. Son registros que se pueden utilizar en cualquier operación.
- De propósito específico. Son registros asignados para tareas especiales, que solo en ellos se realizan. En algunas máquinas este el caso de la multiplicación: solo se puede hacer en registros especialmente designados para esta operación. También hay arquitecturas donde se diferencia entre los registros que contienen datos y los que contienen direcciones.

Un conjunto de registros relacionados se suele llamar un *banco de registros*. Hay arquitecturas que tienen varios bancos de registros; esto porque se pueden tener registros para varias funciones distintas. Por ejemplo, la IA32 tiene un banco de registros generales, otro para las operaciones de punto flotante y otros dos para las operaciones sobre datos multimedia.

Vale la pena plantearse la pregunta de si son necesarios los registros. Como explicamos anteriormente, la respuesta es no —por ejemplo, todos los operandos pueden estar en memoria— pero es conveniente por razones de eficiencia.

Hay otro caso que ya hemos mencionado: las arquitecturas de pila. En estas arquitecturas no hay registros sino una pila de ejecución; los operandos se guardan en ella, y las operaciones los toman de allí, y allí dejan sus resultados. En estas arquitecturas hay dos instrucciones especiales:

- `PUSH` *O*

  Toma el operando *O* de memoria y lo guarda en el tope de la pila.

- `POP` *O*

  Toma el tope de la pila y lo guarda en el operando *O* —en memoria—.

Las demás instrucciones no tienen operandos; hacen todo sobre la pila.

Modos de direccionamiento

Como mencionamos anteriormente, los operandos pueden estar en diferentes sitios: los registros, la memoria y la instrucción misma. En el caso de la memoria, además, hay diferentes maneras de accederla: no es lo mismo una variable estática que un vector o un acceso por apuntador. Los modos de direccionamiento son las diferentes formas que tenemos para especificar dónde está un operando.

Los modos de direccionamiento cambian bastante de un procesador a otro, así que sólo vamos a ver los más generales. Posteriormente veremos cuáles modos posee la IA32 en particular; pero intentaremos que los ejemplos sean lo más cercanos posibles a la IA32, en particular, supondremos instrucciones de dos operandos.

Direccionamiento de registro:

En este modo, todos los operandos son registros. Se supone que los operandos que intervienen en la operación ya han sido llevados desde la memoria a unos registros del procesador. Ejemplos de este modo son:

- `ADD reg1, reg0`

  Sumar el registro cero al registro 1.

- `MOV reg4, reg5`

  Mueve (*move*) el contenido del registro 5 al 4.

En este caso, en la instrucción se debe codificar un número que identifica al registro.

Direccionamiento inmediato:

En lugar de tener el operando en alguna posición de memoria, lo podemos tener codificado en la instrucción misma. Es útil para usar valores constantes, ¿Para qué desperdiciar una posición de memoria si nunca se va a modificar? Además, como el operando está incluido en la instrucción, se evita hacer un acceso a la memoria.

- `SUB reg4, 3`

  Resta el número 3 al registro 4.

Observe que se está restando el valor 3, no el contenido de la posición 3 de memoria ni el registro 3.

Este modo de direccionamiento sirve para manejar las expresiones que incluyen constantes, como, por ejemplo, en C, C++ o Java:

```
x = y - 3;
```

En este caso, en la instrucción se debe codificar el valor numérico de la constante.

Direccionamiento directo:

Este modo se refiere directamente a una posición de memoria; es decir, "traiga el contenido de la posición *n* de memoria".

- `MOV reg1, [3]`

  Asigna el contenido de la posición 3 de memoria al registro 1.

Note los corchetes: denotan que estamos hablando del contenido de la posición de memoria 3 y no de la constante 3. Este modo de direccionamiento sirve para manejar las expresiones que operan sobre variables estáticas.

Los anteriores son los tres modos básicos de los que hemos hablado; pero, en el caso de la memoria, hay otras variantes que agregan flexibilidad:

En este caso, en la instrucción se debe codificar la dirección de la variable.

Direccionamiento indirecto por registro:

Note que, en el direccionamiento directo, se escribe explícitamente la dirección de los datos, por ende la dirección no puede cambiar; se accede siempre a la misma variable. En ocasiones, es necesario cambiar la dirección dinámicamente; tal es el caso cuando se manejan vectores o apuntadores en el interior de un ciclo:

```
for (i = 0; i < n; i++) x += v[i];
```

En el ciclo anterior, en cada iteración, `v[i]` es una variable diferente, porque la variable `i` cambia; pero la variable `x` siempre es la misma.

Puesto que las direcciones son números, una forma de lograrlo es mantener la dirección en un registro, de tal manera que se pueda cambiar dinámicamente. Es lo que se llama el *direccionamiento indirecto por registro*; por ejemplo:

- `ADD reg3, [reg1]`

  Suma al registro tres el contenido de la posición de memoria *apuntada* por el registro 1.

Note que no se le asigna el contenido de `reg1` a `reg3`, sino que se interpreta el contenido de `reg1` como una dirección, se trae el contenido de dicha dirección y se le asigna a `reg3`; `reg1` es un apuntador a la dirección en memoria que nos interesa. En definitiva, ¿qué posición de memoria le estamos sumando a `reg3`?, depende de qué hay en `reg1`; por ejemplo, si antes de la instrucción anterior escribimos:

- `MOV reg1, 20`

Le estaríamos sumando a `reg3` el contenido de la posición 20 de memoria.

Este modo sirve para manejar vectores y apuntadores en los lenguajes de alto nivel; por ejemplo:

```
x = *p;
```

En este caso, aunque en la instrucción se codifica un número que identifica al registro, se debe tener presente que el operando efectivo es la posición apuntada.

Direccionamiento indexado:

Este modo es una generalización del anterior. Es un modo muy flexible que sirve para diversas acciones: acceder vectores en la memoria, campos de una estructura y, como veremos más tarde, variables locales y parámetros en la pila.

Antes de exponer este modo, es necesario estudiar el concepto de *direccionamiento relativo*. Las direcciones que hemos venido usando son absolutas, es decir, se expresan con respecto a la posición cero de memoria. También es posible expresar las direcciones con respecto a otras direcciones, por ejemplo, "5 posiciones adelante de la posición 1000". En este ejemplo, la dirección 1000 es la *dirección base* y el 5 es el *desplazamiento*. Esta forma de direccionar se conoce como *direccionamiento relativo*. Un direccionamiento relativo tendrá, entonces, dos componentes: la *dirección base*, que es una dirección absoluta, y el *desplazamiento*, que es un número con signo; la dirección absoluta correspondiente viene dada por: *dirección base* + *desplazamiento*.

En el direccionamiento indexado, el operando está compuesto por dos partes: una constante y un registro. Uno cualquiera de ellos es una dirección base, y el otro, un desplazamiento. La dirección absoluta es la suma del registro y la constante.

Tenemos entonces dos casos posibles:

- El registro es un apuntador a la memoria y la constante un desplazamiento con respecto a él —ver fig. 4.6 (a)—.
- La constante es una dirección de memoria y el registro un desplazamiento con respecto a ella —ver fig. 4.6(b)—.

El primer caso sirve para direccionar los campos de una estructura[22], o, en el caso de la programación orientada por objetos, los atributos de un objeto[23]. El registro

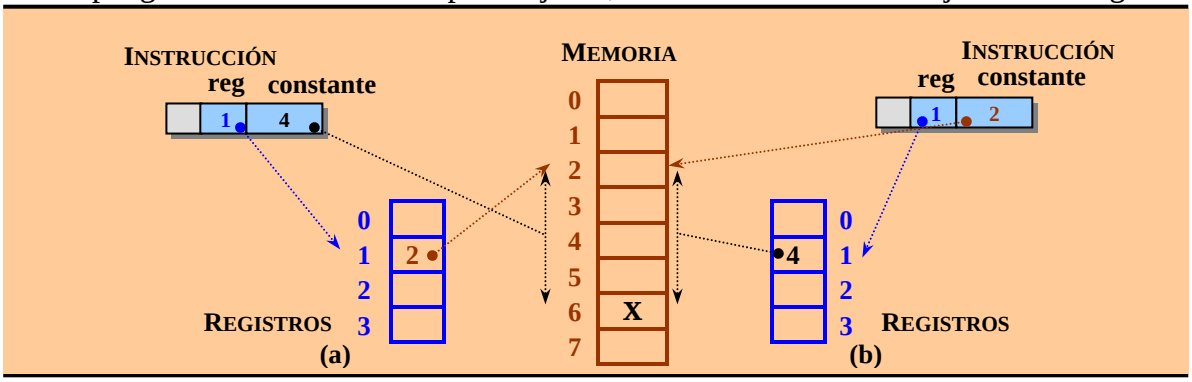

Fig. 4.6. Direccionamiento indexado. (a) el registro es la base. (b) la constante es la base

[22] Las estructuras se representan almacenando sus campos uno detrás de otro, seguidos, en la memoria.

apunta al comienzo de la entidad, y la constante es el desplazamiento necesario para alcanzar el campo respectivo. En el caso de C, serviría para evaluar expresiones del estilo:

```
x = p->a;
```

O, lo que es lo mismo:

```
x = (*p).a
```

El segundo caso sirve para direccionar vectores: la constante indica la dirección dónde empieza el vector en la memoria. El registro juega el papel de subíndice: contiene el desplazamiento del elemento del vector que nos interesa. Sirve, entonces, para evaluar expresiones del estilo:

```
x = v[i];
```

Note que, en el caso del vector, la posición inicial es fija —¡el vector no se anda desplazando por la memoria!—, en tanto que el subíndice cambia dinámicamente —es una variable—; por esto es importante que el desplazamiento esté en un registro.

En cualquiera de los dos casos, el procesamiento es el mismo: la CPU suma el contenido del registro con la dirección —no con el contenido de la dirección—, interpreta el resultado como una dirección y obtiene el contenido de esta última.

En una primera aproximación puede parecer extraño que se trate igual un direccionamiento de vectores y uno de apuntadores, pero sí: son sustancialmente lo mismo. Esto explica por qué en C se puede usar indistintamente `v[i]` o `*(v+i)`.[24]

La notación para este modo es:

- `MOV reg1, [const + reg4]`

Donde *const* es una constante cualquiera.

Cuando se usa para manejar vectores, es usual utilizar la notación `20[Reg2]` en lugar de `[20+Reg2]`; así se parece más a la notación de los lenguajes de alto nivel (`v[i]`).

Es fácil verificar que, si la constante es igual a cero, el direccionamiento indexado es equivalente al indirecto por registro.

En este caso, en la instrucción se deben codificar tanto el registro como la constante.

Para terminar esta sección, dos comentarios:

---

[23] Desde el punto de vista de la máquina, no hay mayor diferencia entre un objeto y una estructura; los atributos se representan como los campos de una estructura.

[24] Y, por extraño que parezca, incluso `i[v]`, puesto que `i[v]` es igual a `*(i+v)`, que es igual a `*(v+i)`, lo cual equivale a `v[i]`.

Con frecuencia las arquitecturas ponen restricciones sobre los modos de direccionamiento de los operandos en una misma instrucción. Por ejemplo, las arquitecturas *load-store*, típicas de los RISC, exigen que en las instrucciones aritméticas todos los operandos sean registros. En cambio, la IA32 permite que sus dos operandos tengan cualquier modo de direccionamiento, pero no permite que los dos estén en memoria.

Se debe diferenciar entre dos conceptos: el operando en la instrucción y el *operando efectivo*. El operando en la instrucción es lo que en ella se codifica, pero el operando efectivo es el valor que efectivamente se usa. Por ejemplo, en el direccionamiento directo, en la instrucción se codifica la dirección, pero lo que se usa es el valor almacenado en esa posición.

La figura 4.7 ilustra las diferencias entre los diversos modos de direccionamiento.

4.3 ARQUITECTURA INTEL DE 32 BITS (IA32)

En esta sección vamos a aplicar los conceptos vistos en las secciones anteriores al caso concreto de la IA32. En primer lugar, veremos cómo ha sido la evolución de la arquitectura de los procesadores de Intel.

El Intel 8086 fue el primer procesador de 16 bits diseñado por la compañía Intel. Hace parte de una familia de procesadores compatibles "hacia arriba", es decir, los nuevos procesadores son compatibles con los antiguos pero tienen más capacidades. Algunos miembros de la familia son: 8080, de 8 bits; 8086, de 16 bits; 8088, internamente es igual al 8086 pero externamente trabaja con 8 bits; 80286, 16 bits; 80386, 32 bits. Con el 80386, Intel definió las grandes bases de la IA32; después vienen el 80486 y las diversas variantes de Pentium; todos son, en lo fundamental, IA32, aunque con extensiones en su conjunto de instrucciones —en algunos casos, substanciales—, e implementados con diferentes técnicas y tecnologías que los hacen más veloces.

Después de los Pentium, se empezó a pensar en desarrollar una arquitectura de 64 bits. Para esto existen dos caminos, y los dos se han tomado: diseñar una arquitectura de 64 bits compatible hacia arriba —así como el 80386, de 32 bits, es compatible con el 8086, de 16 bits— o diseñar una nueva arquitectura diferente de la IA32. Esta última opción ha sido seguida por Intel, que desarrolló la arquitectura EPIC —*Explicitly Parallel Instruction Computing*—; se trata de una arquitectura completamente diferente de la IA32.

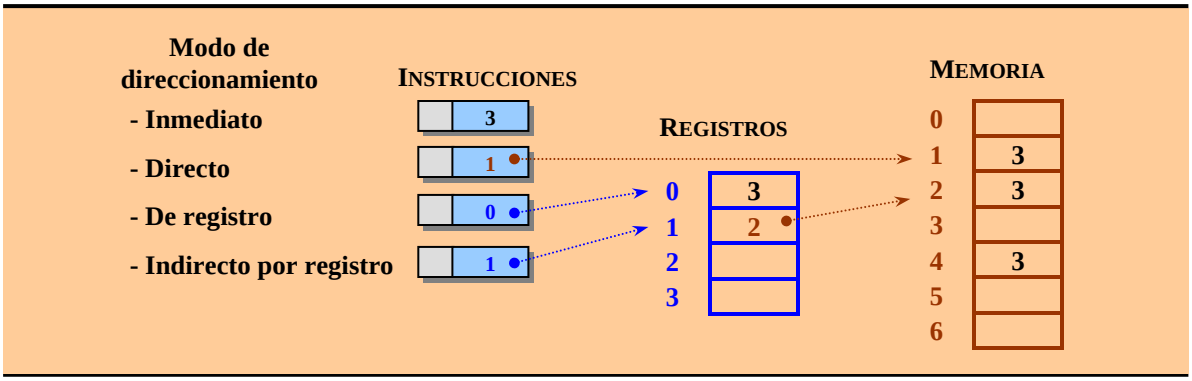

Fig.4.7. Modos de direccionamiento. En todos los casos, el operando efectivo es el número 3

Otros fabricantes, en particular AMD, fabrican procesadores compatibles con Intel: tienen la misma arquitectura pero diferente implantación, por lo cual su desempeño y otras características físicas pueden variar.

La memoria

La IA32 maneja datos y direcciones de 32 bits, lo cual le permite tener memoria de hasta 4 GB. La memoria es direccionable al byte; en una lectura o escritura, el procesador puede pedirle a la memoria que envíe 1, 2 ó 4 bytes,[25] a partir de cualquier dirección: no tiene restricciones de alineamiento.[26]

La IA32 es una arquitectura *little endian* a nivel byte. Recuerde que esto implica que los dígitos menos significativos de un número quedan en las direcciones más bajas. En general, esto no es importante para el programador, pero, en ocasiones, hay que tenerlo presente.

En terminología propia de Intel: un grupo de 16 bits (2 bytes) es una *palabra*, un grupo de 32 bits (4 bytes) es llamado *doble palabra* y un grupo de 64 bits (8 bytes) es llamado *cuádruple palabra*.

Inicialmente, en los PC de IBM y clones, la memoria estaba organizada de la siguiente manera:

Rango de direcciones	Función
00000H – 9FFFFH	memoria RAM
A0000H – BFFFFH	área de pantalla
C0000H – FFFFFH	memoria ROM

Como puede notar en la tabla anterior, el 8086 solo tenía 20 bits de direcciones. En la IA32 se dispone de 32 bits de direcciones, por lo cual la memoria se puede extender por encima de FFFFFH hasta llegar a 4 GB.

Como mencionamos anteriormente, la zona de memoria ROM contiene el programa de arranque del computador, así como unas rutinas básicas para el sistema. El área de pantalla es una región especial de la memoria que el procesador usa para comunicarse con la pantalla —ahí se "dibuja" la imagen que sale en la pantalla—.

Arquitectura del procesador

La arquitectura de los registros de la IA32 está basada en la del 8086 —ver fig. 4.8 (a)—, así que comenzaremos por ver someramente al 8086, para pasar, a continuación, a la IA32.

Registros del 8086

El 8086 tiene 8 registros de propósito general llamados: AX, BX, CX, DX, SI, DI, BP y SP. Estos nombres pueden parecer un tanto caprichosos; en realidad, tienen una explicación histórica, pero, para la IA32, esto no tiene mayor relevancia, así que no

---

[25] Y más para otros tipos de datos, como el punto flotante, pero obviaremos este caso.

[26] Sin embargo, la lectura de datos no alineados es más demorada.

entraremos en el asunto. Además, tiene el registro de indicadores y el IP —*Intruction Pointer*, nombre que Intel le da al PC—. Por último, hay 4 registros de segmento llamados: SS, CS, DS y ES, de los cuales no nos ocuparemos.

Los registros AX, BX, CX y DX presentan la particularidad de que se pueden dividir en dos subregistros —ver fig. 4.8(a)—. Note que no son registros diferentes; por decirlo de alguna manera, AX está compuesto por AH y AL. Si AH o AL son afectados de alguna manera, también lo es AX y viceversa.

Esta capacidad se ofrece para permitir trabajar con 8 ó 16 bits a voluntad. Por ejemplo, cuando se quiere una operación de 16 bits, se utiliza AX, BX, etc.; cuando se quiere operar con 8 bits, se utiliza AH, AL o cualquier otro registro de 8 bits. Cuando estudiamos la memoria, vimos que podía ser accedida en grupos de 8 ó16 bits: la subdivisión de los registros permite realizar dichos accesos.

La H y la L, significan *High* y *Low*, puesto que representan las partes alta y baja, respectivamente, del registro de 16 bits en cuestión. Los otros registros — SI, DI, BP y SP— no se subdividen.

En el 8086, los registros SP, BP, SI y DI tienen algunas funciones especiales, pero esto pierde importancia en la IA32. El único que conserva un papel especial es el SP extendido, como se explicará en lo que sigue.

Registros de la IA32

Cuando se quiso diseñar la IA32, surgió el problema de cómo manejar números de 32 bits dado que los registros eran de 16. Se decidió conservar la arquitectura original y definir unos registros extendidos que recubren a los anteriores. Estos 8 registros extendidos de propósito general se llaman: EAX, EBX, ECX, EDX, ESI, EDI, EBP y ESP —la "E" quiere decir *Extended*—, como se puede ver en la fig. 4.8 (b). Los registros extendidos tienen 32 bits; los 16 bits superiores son nuevos, los 16 inferiores corresponden a algunos de los registros originales de 16 bits. Por

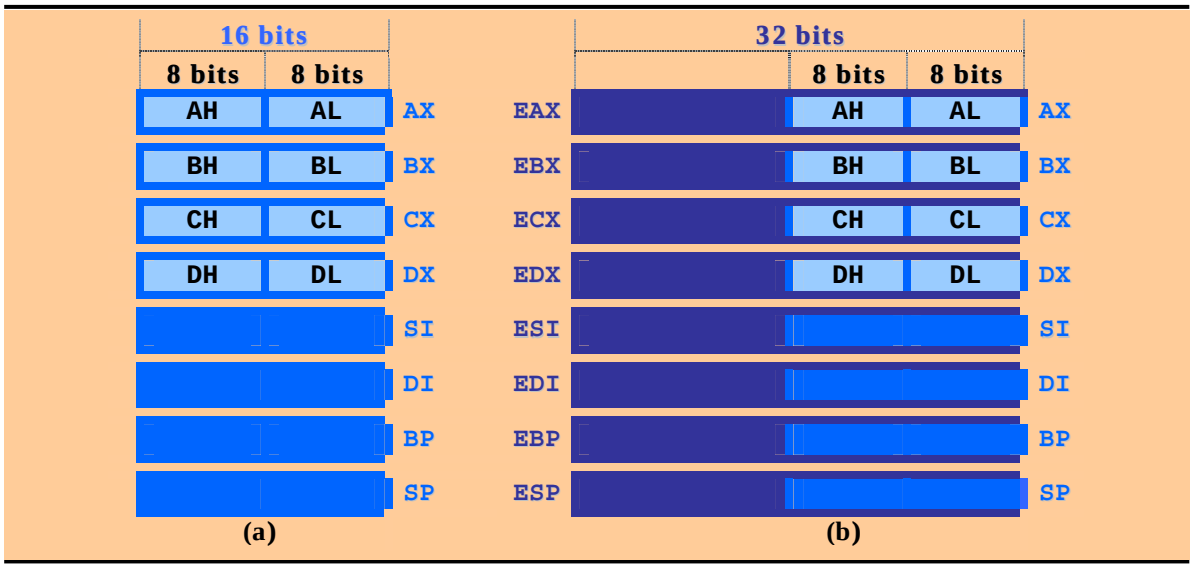

Fig. 4.8. Registros de (a) Intel 8086 (b) IA32

ejemplo, EAX tiene 32 bits; los 16 bits inferiores corresponden a AX y pueden ser manipulados independientemente de los 16 bits superiores.

El registro ESP tiene un papel especial: como se verá más adelante, la IA32 maneja una pila (*stack*) por hardware; este registro es el apuntador al tope de la pila —*Extended Stack pointer*—.

Aparte de los anteriores hay otros registros que el programador no manipula directamente pero que influyen en la ejecución de los programas: el registro de indicadores y el EIP —*Extended Intruction Pointer*—. El EIP también es de 32 bits, con lo cual un programa puede estar en cualquier parte de la memoria y extenderse todo lo que sea necesario. En cuanto a los indicadores, es un registro de 32 bits, pero no todos se usan, y aquí solo nos interesan los 12 inferiores:

Bit No	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
	O	D	I	T	S	Z	-	A	-	P	-	C

- O (*Overflow*): ya conocemos su uso. Si se efectúa una operación sobre dos cantidades, y el resultado sobrepasa el número de bits asignados para el resultado, este bit se pone en 1 para indicar un *desbordamiento*.
- C (*Carry*): otro viejo conocido. Cuando se suman dos números, aquí se almacena el acarreo de la suma de los bits más significativos —el acarreo final de la suma—. Este bit puede ser afectado de manera diferente por otras operaciones (por ejemplo, los corrimientos).
- Z (*Zero*): indica si la última operación dio cero o no. Se comporta como un valor lógico: si es cierto (vale 1), entonces la última operación dio cero.
- A (*Auxiliary carry*): lo usa el procesador para efectuar las operaciones de aritmética decimal (BCD). Es el acarreo que pasa del 4º al 5º bit.
- S (*Sign*): indica el signo del último resultado obtenido. Este bit es igual al bit más significativo del resultado —que es el bit del signo—.
- P (*Parity*): cuando el resultado de una operación tiene un número par de unos, el bit P se pone en 1; si el número de unos es impar, se pone en 0.
- D, I, T: no son de interés en este momento, puesto que no son indicadores de estado, sino que permiten controlar ciertos comportamientos de la máquina.

Como se mencionó anteriormente, las operaciones afectan los indicadores de diferentes maneras, incluso puede que algunos no se vean afectados. En el caso de la suma, la forma de modificar los indicadores es evidente para C, S, Z y O. En otros casos, por ejemplo los corrimientos, es menos obvio cuáles indicadores afectan y cómo deben hacerlo; posteriormente estudiaremos estos aspectos.

Por último, la IA32 agrega 2 registros de segmento a los del 8086, queda así con: SS, CS, DS, ES, FS y GS. Estos registros juegan un papel en el direccionamiento, pero, en la práctica, no son muy utilizados y no nos ocuparemos de ellos.

La arquitectura IA32 es bastante extensa, y esta presentación solo puede ser considerada como una introducción. En particular, quedan fuera de esta discusión

los registros de: punto flotante, MMX y XMM, así como varios registros de control que influyen en diversas tareas más relacionadas con el sistema operativo —en particular, con la memoria virtual—.

Modos de direccionamiento

La IA32 tiene instrucciones con dos operandos, y los dos pueden tener cualquier modo de direccionamiento, con la excepción de que los dos operandos no pueden estar simultáneamente en la memoria.

La IA32 tiene todos los modos descritos en la sección anterior, así que los ejemplos ahí presentados son válidos para esta —por supuesto, cambiando "`reg1`", "`reg2`", etc. por registros de la IA32: `EBX`, `DX`, `ESI`, `CL`, etc.—. Sin embargo, hay algunas precisiones que es necesario conocer para poder utilizarla con propiedad:

- Direccionamiento de registro: los dos registros involucrados deben ser del mismo tamaño; los dos de 32 bits, los dos de 16 bits o los dos de 8 bits (ejemplo: BL con CL, o AX con SI, o EBX con EBP)[27].
- Direccionamiento inmediato: el valor constante debe tener un número de bits menor o igual que el registro. Es decir, a AL se le puede asignar 9AH ó 1H pero no 0FA1H; mientras que este último sí se le puede asignar a BX.
- Direccionamiento directo: el tamaño del registro destino (8, 16 ó 32 bits) determina el número de bits que se van a traer de la memoria.
- Direccionamiento indirecto por registro y direccionamiento indexado: en el Intel 8086 había algunas restricciones para estos modos; en la IA32 se puede usar cualquiera de los registro de 32 bits. Se pueden traer datos de 8, 16 ó 32 bits, según el registro destino que vaya a recibir el dato.
- Direccionamiento basado: este modo no lo presentamos en la sección anterior, ya que su mecanismo físico de direccionamiento es parecido al del indexado. En este modo se usan dos registros y una constante; la dirección efectiva se calcula sumando los tres valores. Este modo es un tanto exótico pero tiene su utilidad para manejar vectores declarados localmente en un procedimiento o en un método. Tiene varias notaciones, pero la básica es: [*constante*+*reg1*+*reg2*], donde *reg1* y *reg2* son alguno de los registros de 32 bits —EAX, EBX, etc.—.
- Escalamiento: esto, más que un modo de direccionamiento, es una característica que se puede aplicar a los modos indexado y basado. Consiste en que el registro de indexamiento se puede multiplicar por un factor *k*, pero

[27] Hay unas pocas instrucciones que permiten mezclar registros de tamaños diferentes, pero se trata de casos especiales; por ejemplo, hay una instrucción para convertir un byte en una palabra: uno de sus operandos es de tamaño byte y el otro de tamaño palabra.

este factor solo puede ser 1, 2, 4 ó 8. Se nota: [*constante+k*reg*], done *reg* es un registro de 32 bits y *k* es el factor. En el caso del direccionamiento basado, se nota: [*constante+reg1+k*reg2*]. Se acostumbra llamar *registro índice* al que se escala (*reg2*) y *base* al otro (*reg1*). Esta característica facilita el acceso a vectores en algunos casos, como veremos posteriormente.

4.4 CONSIDERACIONES GENERALES

Como vimos en las secciones anteriores, el diseño de una arquitectura involucra muchas características: arquitectura de memoria, arquitectura de registros, formato de instrucciones, etc.

La pregunta que surge es: ¿qué orienta a un arquitecto de computadores para tomar las decisiones de diseño de una arquitectura? ¿Es este un proceso subjetivo guiado solo por conceptos como "elegancia"? ¿Hay criterios?

A lo largo del tiempo ha habido criterios, más o menos subjetivos, pero han ido evolucionando y cambiando.

En un principio se trataba de algo más bien subjetivo, guiado por consideraciones tales como "comodidad de uso" o instrucciones "poderosas".[28] Esto porque en las primeras máquinas era usual la programación en lenguaje de máquina o en lenguaje ensamblador. En consecuencia, las máquinas se pensaban en términos de programadores humanos, y se intentaba facilitarles la labor de programación.

El surgimiento de los lenguajes de programación de alto nivel cambió la situación en dos aspectos:

- El programador no se relaciona directamente con la máquina. Realmente programa una máquina abstracta, independiente del hardware.
- Hay un intermediario entre el programador y la máquina: el compilador.

Como consecuencia de todo esto, el verdadero programador de la máquina no es un ser humano, lo cual cambia el panorama de diseño de las máquinas.

En los años 70 surgió una discusión alrededor de la "diferencia de nivel" entre procesadores y lenguajes: se argüía que los conceptos en los lenguajes de alto nivel habían avanzado mucho, pero que, en contraste, las máquinas seguían siendo igual de "simples" que siempre; a esta diferencia de nivel se le denominó *brecha semántica* (*semantic gap*). Se afirmaba que esto generaba una labor más compleja para el compilador: traducir de lenguajes de cada vez más alto nivel a máquinas relativamente simples; lo cual, supuestamente, redundaba en traducciones ineficientes a lenguaje de máquina.

Como solución se propuso crear máquinas de más alto nivel. Estas máquinas trataban de recrear, o soportar, en hardware conceptos existentes en los lenguajes de alto nivel. Algunas de estas máquinas, por ejemplo, tenían los datos etiquetados en memoria, de manera que sabían a qué tipo de datos pertenecía cada uno; otras

[28] Y, quizás en algún grado, "lo que se pueda hacer".

eran capaces de acceder a vectores o matrices directamente, a partir solo de los subíndices.

Aunque pronto se cayó en cuenta de que subir demasiado el nivel de las máquinas no era una buena idea, de todas maneras, en términos generales, se tendió a crear máquinas más bien complejas; años después se acuñaría el término CISC (*Complex Instruction Set Computer*) para describir este tipo de máquinas.

Diversos estudios que se realizaron por esos años mostraron que la mayoría de los programas tendían a no hacer uso de las características complejas; la mayoría de las instrucciones efectivamente usadas por los programas eran más bien simples. Por otro lado, la eficiencia de los compiladores había aumentado notoriamente, en particular su capacidad para optimizar el código de máquina generado. Adicionalmente se descubrió que las arquitecturas simples, por el mismo hecho de ser simples, permitían o aligeraban la tarea de crear implementaciones más eficientes.

En síntesis, se encontró que las arquitecturas más simples tendían a originar máquinas más veloces. Este resultado es un tanto paradójico, porque una máquina CISC tiende a tener programas más compactos, ya que cada instrucción realiza más acciones. En cambio, en una máquina simple, los programas son más largos y, en consecuencia, debería tomar más tiempos de ejecución. En principio esto es así, pero, al mismo tiempo, las arquitecturas simples permiten hacer implementaciones más eficientes donde cada instrucción se ejecuta más rápidamente. Podríamos decir que es mejor ejecutar varias instrucciones rápidamente que una más lentamente.

Las arquitecturas que se generaron tras toda esta discusión se llamaron RISC (*Reduced Instruction Set Computer*). Se caracterizan por tener bastantes registros, instrucciones simples —hacen una sola acción—, de tres operandos, con un formato estándar (el formato de instrucción no cambia, o muy poco, entre instrucciones). En particular son arquitecturas del tipo *load-store*: las instrucciones que acceden a la memoria no efectúan operaciones, y las que efectúan operaciones, no tienen acceso a la memoria —trabajan solo en los registros—.

La discusión no termino aquí; posteriormente, los fabricantes de máquinas CISC, específicamente Intel, mostraron que, aunque con alguna complicación, sí podían utilizar las técnicas de las máquinas RISC. Estas últimas, por su parte, continuaron con su evolución, recurrieron a nuevas técnicas —divergiendo en algún grado de sus planteamientos originales— hasta el punto de que se empezó a hablar de arquitecturas post- RISC.

En realidad no es exacto decir que RISC o CISC son arquitecturas como tales; en el mejor de los casos se trata de un conjunto de recomendaciones de diseño, o de idiosincrasias, que han conducido a máquinas con arquitecturas más o menos parecidas.

Sin embargo, toda la discusión alrededor de tema dejó varias lecciones útiles. La más importante, quizás, es que no se puede disociar completamente arquitectura

de estructura; las decisiones tomadas en uno de estos niveles pueden condicionar decisiones en el otro nivel.

En la visión moderna, el compilador es el responsable de generar código, y la estructura es la responsable de ejecutarlo; la arquitectura es una interfaz entre los dos: es un vehículo, un mediador, entre el compilador y la estructura. En consecuencia, debe diseñarse para que le permita al compilador comunicar a la estructura de la manera más clara y precisa cómo debe ejecutar el código por él generado.

Hay otra problemática relacionada con el diseño de arquitecturas. En principio, un diseñador puede crear la arquitectura como a bien tenga; sin embargo, en la práctica esto no es tan cierto: hay arquitecturas que se han convertido en estándar —como la IA32—, y, en consecuencia, puede que el diseñador de un procesador tenga que respetarla; esto hace que su labor de diseño de la estructura se vea restringida por la arquitectura preexistente.

Un último aspecto de diseño muy relevante es la capacidad para soportar el paso del tiempo. Diseñar una arquitectura es un proceso largo y costoso; no es frecuente que una compañía se decida a emprenderlo. Por lo tanto, al diseñar una arquitectura, es necesario hacer algún análisis prospectivo y construirla pensando en el futuro. Un caso interesante es la comparación del Intel 8086 —elegido por IBM— y el Motorola 68000 —elegido por Apple—. Los dos son más o menos contemporáneos, pero el Intel fue diseñado pensando en las memorias de la época, por lo cual tenía 20 bits de direcciones; el Motorola fue diseñado para las memorias del futuro, y, por ende, tenía 32 bits de direcciones. Las consecuencias se vieron en la evolución de la memoria en los dos tipos de computadores: en la medida en que las necesidades de memoria aumentaron, las máquinas de Apple crecieron sin ningún problema, en tanto que los PC se vieron sometidos a todo tipo de restricciones, y la configuración y manejo de la memoria se convirtieron en una pesadilla.

EJERCICIOS

1- **a-** Estime cuántos granos de arena caben en la tierra y expréselos usando la unidad más conveniente (la más cercana al valor: kilo, mega, etc.).

b-Se tiene una memoria de 2 GiB, exprésela en MiB, en KiB y en B.

c- Exprese 2 MiB en KB.

d- Exprese 2000 KiB en MiB.

e- Se tiene un disco de 40 GB, ¿cuánto es en GiB?

f- Estime cuántos libros (texto puro, en ASCII) caben en el disco anterior.

2- Se está recibiendo información transmitida a 1024 Mbit/s, y la transmisión dura 1024 segundos. Se tiene un disco duro con capacidad de 123 GiB. ¿Cabe la información recibida en el disco duro?

3- Se tiene un disco1 de 42 GB y un disco2 de 40 GiB, ¿cuál es la relación de las capacidades de almacenamiento? (Es decir, capacidad de disco1 es igual, mayor o menos que la de disco2).

4- **a-** El Intel 8086 tiene direcciones de 20 bits ¿Cuál es el tamaño máximo de la memoria, en KiB, de un computador que use el Intel 8086 como procesador?

b- La IA32 tiene direcciones de 32 bits ¿Cuál es el tamaño máximo de la memoria de un computador que use la IA32 en el procesador? (use la unidad más conveniente para expresarlo).

c- La IA64 tiene direcciones de 64 bits ¿Cuál es el tamaño máximo de la memoria de un computador que use la IA64 en el procesador? (use la unidad más conveniente para expresarlo)

5- ¿Qué quiere decir que un computador tiene tamaño de palabra de *n* bits?

6- ¿Qué relación sería razonable esperar entre el tamaño de palabra y el tamaño de los registros (mayor, menor, igual...)?

7- Para cada uno de los siguientes tipos de datos, indique cuál es el rango de valores que pueden representar (suponiendo los tamaños indicados en el texto):

Tipo de datos	Rango
char	
int	
short int	
long int	

8- Se tiene un vector de `int` en C. En el vector se almacenó una cadena de caracteres; 4 caracteres en cada posición. Escriba un programa en C para extraer el i-ésimo carácter; use solo operaciones lógicas y aritméticas, no utilice el *cast*.

9- Se tiene un vector de `char` en C. En el vector se almacenaron enteros (`int`); 4 `char` para cada `int`. Escriba un programa en C para extraer el i-ésimo entero; use solo operaciones lógicas y aritméticas, no utilice el *cast*. ¿Depende su programa de si la máquina es *little endian* o *big endian*?

10- Se tiene un vector de `char` en C, pero se usa como un vector de bits. Los bits se numeran de menos significativo a más significativo —al interior de un byte— y del inicio hacia el final del vector —de `v[0]` a `v[n]`—

a- Encuentre una operación aritmética para determinar en qué posición del vector se encuentra el i-ésimo bit.

b- Dado que conoce la posición del vector, encuentre una operación aritmética para determinar a cuál bit del byte corresponde el i-ésimo bit del vector.

c- Escriba un programa en C para extraer el i-ésimo bit; use solo operaciones lógicas y aritméticas. ¿Depende su programa de si la máquina es *little endian* o *big endian*?

11- Se tiene un vector de `int` en C. En el vector se almacenaron enteros, pero desfasados 2 bytes; es decir, los primeros 2 bytes no se usan y luego viene la secuencia de enteros, cada uno de 4 bytes. Escriba un programa para extraer el i-ésimo entero; use solo operaciones lógicas y aritméticas, no use *cast*. ¿El programa debe tener en cuenta si la máquina es *little endian* o *big endian*?

12- Se tiene un computador con direcciones de n bits y palabras de m bytes. ¿Cambia la cantidad total de memoria si es direccionable al byte o si es direccionable a palabra? Si piensa que la capacidad cambia, diga en cuánto; si no, explique por qué no cambia.

13- **a-** Se están manejando enteros de 64 bits. Se representan en vectores de `int` de 2 posiciones; `v[0]` es la parte menos significativa. Escriba un programa para sumar dos números representados de esta manera; no use *cast*, solo operaciones lógicas o aritméticas.

b- Escriba un programa para restar números representados de esta manera.

14- En un computador con palabra de 16 bits, se almacenó en memoria el número 89 AB CD EFH a partir de la posición cero.

a- Si el computador es *big endian* ¿Qué obtiene al leer un byte de la posición 0? ¿Y si lee de 1 ó de 2?

b-¿Qué obtiene al leer dos byte a partir de la posición 0? ¿Y si los lee de la posición 1 ó de la 2?

c- Responda las preguntas anteriores para el caso de un computador *little endian*.

15- Para cada fila de la siguiente tabla, complete el dato, o datos, que falta. Si no es posible calcularlo, o es absurdo, o no es consistente con los otros datos escriba guiones (---) en la casilla respectiva.

Bits de direcciones	Número de celdas	Bits por celda	Total de memoria
20		8	8 MiB
16	65536		16*64 Kibits
	1´000.000	8	
10	1024		8704 bits
	2^{24}	32	
		16	4 MiB
16	2^5*1024		64 KiB

16- Se tiene una memoria con una capacidad de p Bytes, con n_1 líneas de direcciones y m_1 líneas de datos. Se tiene otra memoria con una capacidad de $p/2$ Bytes, n_2 líneas de direcciones y m_2 líneas de datos.

a- Si $n_1 = n_2$, ¿Cuál es la relación entre m_1 y m_2? (m_2 como función de m_1).

b- Si $m_1 = m_2$, ¿Cuál es la relación entre n_1 y n_2? (n_2 como función de n_1?

17- Se tiene una memoria con una capacidad total de X Bytes, con m líneas de direcciones y n de datos. Se desea cuadruplicar su capacidad ($4*X$ bytes); para esto se puede incrementar el ancho de palabra y/o el número de direcciones.

¿De cuántas maneras se podría hacer esto? Describa cada una de ellas, en particular, diga cuántas líneas de datos (bits) y cuántas líneas de direcciones (bits) habría que agregarle en cada uno de los casos.

18- **a-** Se tiene una memoria de 4 MByte. La palabra es de 64 bits, el procesador maneja bytes, medias palabras y palabras y la memoria es direccionable al byte. ¿Cuántas líneas de datos y cuántas de direcciones maneja el procesador?

b- Se tiene una memoria de 4 MByte. La palabra es de 64 bits, el procesador solo maneja palabras y la memoria es direccionable a palabra. ¿Cuántas líneas de datos y cuántas líneas de direcciones debe manejar el procesador?

c- Se tiene una memoria de 2 MByte con 19 líneas de direcciones, ¿de qué tamaño es la palabra?

19- **a-** Un apuntador es una variable que contiene una dirección. Si estamos en una máquina con direccionamiento al byte, tenemos `char * p` y hacemos `p++`, ¿en cuánto aumenta el valor almacenado en `p`?

b- Y si tenemos `int * p` y hacemos `p++`, ¿en cuánto aumenta?

20- Se tiene una máquina sin restricciones de alineamiento, y un programa en C que declara una estructura con 3 campos: un `char`, un `int` y un `char`.

a- ¿Cuanto espacio en memoria ocupa esta estructura?

b- ¿Cuanto espacio en memoria ocupa un vector de estas estructuras?

c- Repita a- y b- para una máquina con restricciones de alineamiento.

21- En una máquina de 16 bits, se efectúan las siguientes operaciones; diga en qué valores quedan los indicadores mostrados después de cada una de ellas:

Operación	S	C	O	Z
0F32H + 0002H				
0F32H + FFF2H				
0032H + F002H				
0032H + FFCEH				
FF32H + FFCEH				
6F32H + 6111H				
8032H + E002H				

22- Como se indicó en el texto, los indicadores sirven para comparar números; una forma es restarlos y ver qué ocurre en los indicadores. Por ejemplo, si son iguales, el indicador de cero vale 1.

a- Si se está trabajando con números en complemento a 2 ¿Cómo se puede detectar si *a* es menor que *b* usando los indicadores? (lla respuesta no es trivial. Pruebe su respuesta con los números 4000H y -4000H).

b- ¿Y si son números sin signo?

23- ¿Cuántos accesos a la memoria se efectúan durante la ejecución de cada una de las siguientes instrucciones?

Instrucción	Accesos
`mov eax, 4`	
`add al, ch`	
`mov al, [4]`	
`mov ax, [4]`	
`mov eax, [4]`	
`mov [4], eax`	
`mov [ecx], eax`	
`add [4], ax`	
`add ax, [ecx]`	
`sub ax, [4+ecx]`	

24- ¿Cuántos bytes se leen de la memoria durante la ejecución de cada una de las siguientes instrucciones?

Instrucción	Bytes
`mov al, [4]`	
`add ax, [4]`	
`or eax, [4]`	
`and dl, [ecx]`	
`mov dx, [ecx]`	
`add edx, [ecx]`	
`mov edx, [8+ecx]`	

25- En algunas arquitecturas existe un modo de direccionamiento llamado indirecto por memoria (este modo no existe en la IA32). Funciona como el indirecto por registro pero usando una posición de memoria para la indirección en lugar de un registro. Lo notamos de la siguiente manera:

```
MOV  reg1, [[4]]
```

¿Cuántos accesos a memoria se efectúan durante la ejecución de cada una de las siguientes instrucciones?

Instrucción	Accesos
`mov reg1, [[4]]`	
`add reg1, [[4]]`	
`mov [[4]], reg1`	
`add [[4]], reg1`	

26- ¿Cuánto queda valiendo `ebx` después de ejecutar el siguiente programa?

```
MOV  ebx, 3
MOV  eax, 4
MOV  [eax], 7
ADD  ebx, [eax]
```

27- Sobre la IA32, ¿cuánto espacio en memoria ocupa cada una de las siguientes declaraciones en C?

Declaración	Bytes
`char c;`	
`char * p;`	
`int * p;`	
`char v[20];`	
`char * v[20];`	
`char v[20][10];`	

28- Se tiene la siguiente matriz:

```
int   m[e1][e2][e3];
```

a-¿Cuánto espacio ocupa en memoria?

b- ¿Cómo se calcula es desplazamiento del elemento m[i][j][k]?

c- Generalice el ejercicio anterior para matrices de *n* dimensiones.

29- Sobre la IA32, dadas las declaraciones mostradas, ¿cuántos accesos a memoria efectúa cada una de las siguientes instrucciones de C?

```
int x, y;
int * p, *q;
```

Instrucción	Accesos
`x = 5;`	
`x = y;`	
`p = q;`	
`x = *q;`	
`*p = *q;`	
`*p++ = *q++;`	
`p = &x;`	

30- En la siguiente tabla, diga si los segmentos de código en cada fila son equivalentes o no.

Segmento 1	Segmento 2	¿Equivalen? (S/N)
`mov ebx, 5` `mov eax, [ebx]`	`mov eax, 5`	
`mov ebx, [5]`	`mov ecx, 5` `mov ebx, [ecx]`	
`add eax, 5` `mov ebx, [eax]`	`mov ebx, [5+eax]`	
`mov al, 7` `mov ah, 5`	`mov ax, 507H`	
`mov ax, [5]`	`mov al, [5]` `mov ah, [6]`	
`mov ax, [ebx]`	`mov ah, [ebx]` `mov al, [ebx+1]`	

31- En la siguiente tabla, diga si los segmentos de código en cada fila son equivalentes o no, dadas las declaraciones mostradas a continuación (algunos segmentos pueden incluso ser código inválido):

```
int v[10];
int * p = v;
int i;
```

Segmento 1	Segmento 2	¿Equivalen? (S/N)
`v[i] = 7;`	`(p+i) = 7;`	
`v[i] = 7;`	`(*p+i) = 7;`	
`v[i] = 7;`	`*(p+i) = 7;`	
`*(v+i) = 7;`	`p[i] = 7;`	
`v[i] = 7;`	`p += i;` `*p = 7;`	
`v[i]++;`	`(p+i)++;`	
`v[1]++;`	`(*(++p))++;`	

32- Se tiene un procesador con 53 instrucciones diferentes. No tiene registros. La memoria se direcciona al byte, y su tamaño es de 256 bytes. Las instrucciones son de dos operandos. Diseñe el formato de instrucción.

33- Se tiene un procesador con 53 instrucciones diferentes. Tiene 16 registros. El tamaño de palabra es 16 bits. La memoria se direcciona al byte, y su tamaño es de 64 KiB. Las instrucciones son de dos operandos; el primer operando siempre es un registro; el segundo tiene 4 modos de direccionamiento posibles: inmediato, de registro, directo e indirecto por registro. Para codificar el modo de direccionamiento del segundo operando se incluyen dos bits en la instrucción.

a- ¿Cuántos bits se necesitan para codificar cada una de las entidades que participan en una instrucción?

b- ¿Es posible tener un único formato de instrucción? Sí no, explique por qué; de lo contrario, indique las implicaciones.

c- Diseñe el o los formato(s) de instrucción.

34- En ocasiones se quiere trabajar con subvectores de un vector o de una matriz. Esto es muy útil en ciertos casos; por ejemplo, si se tiene una matriz almacenada por filas y se quiere trabajar con columnas o diagonales, hay que hacer algoritmos específicos para cada caso. En lugar de esto, se puede crear una representación general para subvectores.

La representación es así: se tienen dos arreglos; uno de ellos es el vector original con todos los elementos —vector de contenido—; el otro es un vector de bits —vector de control— con tantos bits como elementos tiene el vector de contenido. El valor de cada bit del vector de control indica si el elemento correspondiente del vector de contenido hace parte o no del subvector; el bit en 1 indica que sí hace parte. Por ejemplo, si se tiene el vector:

9	13	0	3	6	57	7	3	2	-1	-8	27	5	5	4	1

Y el vector de control:

0	0	0	1	0	0	1	0	0	1	0	0	0	0	1	1

El subvector representado es:

3	7	-1	4	1

Se tiene un vector de *n* enteros (de 4 bytes), y se quiere manejar subvectores de tamaño *k*:

a- Encuentre una fórmula en términos de *n* que exprese el espacio ocupado por los subvectores usando esta representación.

b- Encuentre una fórmula en términos de *n* que determine a partir de cuál *k* es más eficiente (en uso de espacio) esta representación que almacenar el subvector directamente.

35- Otra representación de subvectores se logra con la estructura siguiente:

```
Subvector struct
   nElementos     dword?
   base           dword?
   factor         dword?
   apContenido    dword?
Subvector ends
```

nElementos es el número de elementos del subvector. *apContenido* es un apuntador al vector original. *base* y *factor* sirven para obtener la posición de cada elemento del subvector de la siguiente manera:

```
Subvector [i] = (*apContenido)[Base + Factor*i]
```

Por ejemplo, si se tiene una matriz de tamaño *n* representada por filas, la segunda columna se representa por: *base* = 1 y *factor* = *n*.

a- ¿Cuáles son los valores de *base* y *factor* para seleccionar la tercera fila?

b- ¿Cuáles son los valores de *base* y *factor* para seleccionar la diagonal?

c- ¿Cuáles son los valores de *base* y *factor* para seleccionar la anti-diagonal (de la esquina superior derecha a la inferior izquierda?

d- Compare esta representación con la del punto anterior. ¿Cuáles son sus ventajas y desventajas?

36- Otra representación de subvectores consiste en lo siguiente: se tienen dos arreglos; uno de ellos es el vector original con todos los elementos —vector de contenido—; el otro es un vector de índices con tantos elementos como el subvector. El valor de cada elemento del vector de índices denota un elemento del vector de contenido. Por ejemplo, si se tiene el vector:

9	13	0	3	6	57	7	3	2	-1	-8	27	5	5	4	1

Y el vector de índices:

3	6	9	14	15

El subvector representado es:

3	7	-1	4	1

Los índices no necesariamente están en orden ascendente, e incluso pueden estar repetidos.

a- Dé una expresión para obtener el i-ésimo elemento del subvector.

b- Compare esta representación con las de los puntos anteriores. ¿Cuáles son sus ventajas y desventajas?

37- Un vector disperso es aquel que tiene muchos elementos en cero, lo cual desperdicia memoria. Para evitarlo, se desarrolla una representación especial para este tipo de vector.

La representación es la siguiente: se tiene un arreglo de bits —vector de control— con tantos elementos como el vector disperso; el valor de cada bit indica si el elemento correspondiente del vector disperso vale 0 —bit de control en cero— o no —bit de control en 1—. Por otro lado, se tiene el vector de contenido que es un arreglo de palabras con los valores de los elementos diferentes de cero del vector disperso. Por ejemplo, el vector disperso:

0	0	0	3	0	0	7	0	0	-1	0	0	0	0	4	1

Se representa por el vector de control:

0	0	0	1	0	0	1	0	0	1	0	0	0	0	1	1

Y el vector de contenido:

3	7	-1	4	1

Se tiene un vector disperso de *n* enteros (de 4 bytes), de los cuales *k* por ciento son diferentes de cero:

a- Encuentre una fórmula en términos de *n* y *k* que exprese cuánto espacio se necesita para representar el vector disperso con la representación anterior.

b- Determine a partir de cuál *k* es más ineficiente (en uso de espacio) esta representación que almacenar todo el vector directamente.