LA PC POR DENTRO

ARQUITECTURA Y FUNCIONAMIENTO DE COMPUTADORAS

MARIO CARLOS GINZBURG

INGENIERO ELECTRONICO (UBA)

A Piyi, Jerónimo y Rafael

INTRODUCCION GENERAL A LA INFORMATICA:
1 LA PC POR DENTRO, ARQUITECTURA Y FUNCIONAMIENTO DE COMPUTADORAS
Cuarta Edición

M.C.Ginzburg

ISBN-10: 987-05-0916-9
ISBN-13: 978-987-05-0916-5

Ginzburg, Mario Carlos
Introducción general a la informática : la pc por dentro : arquitectura y funcionamiento de ordenadores / Mario Carlos Ginzburg ; ilustrado por Rafael Ginzburg - 4a ed. - Buenos Aires : el autor, 2006.
232 p. ; 20x28 cm.

ISBN 987-05-0916-9

1. Informática. I. Ginzburg, Rafael, ilus. II. Título
CDD 004.1

Fecha de catalogación: 03/05/2006

PROLOCO

Esta obra ha sido pensada para un lector sin experiencia previa alguna en el terreno de la computación o informática.

Al comenzar cada tema nuevo se buscan analogías con la vida real. Así, como paso previo al esquema general de un computador, se considera una fábrica, que de forma similar produce productos según las instrucciones que dejan los clientes. El contenido de cada posición de memoria se visualiza como 8 llaves de encendido de luz; y para entender qué significa que una memoria sea "random" se recurre al selector de canales de un televisor. Al tratar cómo se le ordena al computador qué debe hacer, se pone de relieve la semejanza con la forma de operar con una calculadora portátil común. Del mismo modo, el "pipe line" de un procesador se comprende a través de un lavadero automático de autos; el concepto de "buffer" mediante buzones y del papel que juega la pileta en la descarga de un lavarropas, etc.

Esta primera unidad abarca los temas fundamentales del funcionamiento de un computador.
Con ella se busca brindarle al lector *en forma clara, concisa, y práctica,* los conocimientos básicos para entender cómo funciona el interior de una PC moderna, y en general cualquier computador. Se eligió, por ser más didáctico el método de preguntas-respuestas acerca de los temas esenciales que se deben conocer.

Las primeras secciones constituyen el tronco o basamento medular de la presente unidad y de la obra en general. Las siguientes son desarrollos más detallados de las primeras.

De esta forma, **el lector progresa en el estudio del libro y de cada tema conforme a sus necesidades de profundización o aplicación práctica.**

Completan la Unidad 1 apéndices sobre el sistema binario, software (en especial sistemas operativos y virus), una Historia del desarrollo del hardware y software hasta 2006, un modelo circuital didáctico de funcionamiento de un procesador, con elementos RISC, y un Complemento para enteros y punto flotante.

La segunda unidad de la presente obra, trata en detalle el funcionamiento y uso práctico de los periféricos, algunos de los cuales se comienzan a ver en esta unidad.

Temas como la programación en Assembler, y la confección de pequeños programas típicos están contenidos en la Unidad 3. Se incluyen llamados a subrutinas, vectores interrupción y manejo de pilas.

Los circuitos lógicos de un computador: compuertas, UAL, flip flops, registros, secuenciadores, descriptos ailadamente se integran en un modelo didáctico sencillo pro RISC muy sencillo, y se tratan en la obra "De la Compuerta al Computador" próxima a aparecer. En la presente edición se ha conservado un Modelo Circuital didáctico de una UCP capaz de ejecutar instrucciones, que no requiere ningún conocimiento de circuitos lógicos.

Por qué emplear un microcomputador como modelo de funcionamiento de computadoras:

Cuando Intel lanzó el 80386 en 1985, que podía ejecutar 6 millones de instrucciones por segundo (mips), tuvo lugar un salto notable en la performance de los microprocesadores. Ya el 80286 ejecutaba 2,5 mips, y permitía operar en multitarea (*multitasking*), con un sistema operativo apropiado.

Luego, en 1991 y 1993 Intel produjo el 80486 y el Pentium I. Los siguientes procesadores de Intel contienen un núcleo RISC, siendo que los procesadores RISC (Power PC, Alpha, MIPS, i860, etc.), se lanzaron en la década de 1980.

La tendencia actual es usar cada vez en mayor medida la conexión en red de microcomputadoras (sean personales o estaciones de trabajo) para nuevos emprendimientos o en reemplazo de las grandes *mainframes.* Estas quedan restringidas para áreas reservadas o muy especiales..

Un 486 o un Pentium, al igual que los microprocesadores RISC tienen una elevada complejidad interna. Procesadores RISC operando en paralelo forman parte de supercomputadoras.

Los microprocesadores han ido incorporando paulatinamente todas las innovaciones que antes eran privativas de las grandes computadoras y supercomputadoras. Así, el "pipe line" (ya presente en el 8086), el modo protegido para "multitasking", la memoria "caché" interna, el coprocesador matemático y la concepción superescalar pasaron a formar parte del cualquier microprocesador actual.

Al aparecer en el mercado en 1986 los primeros RISC, ocurrió que estos microprocesadores presentaban innovaciones que los procesadores de las grandes y minicomputadoras (del tipo CISC) no poseían. Los buses rápidos como el PCI express mejoraron notablemente la performance de los computadores personales.

Por lo tanto, hoy día conocer el funcionamiento de un microprocesador moderno implica estar actualizado con las principales innovaciones tecnológicas ocurridas en el desarrollo de las computadoras

Los principios de funcionamiento de los dos niveles de caché en una microcomputadora son los mismos que para computadoras más grandes. Lo mismo respecto al *pipe line*, modo protegido, coprocesador, etc. Igualmente *son comunes a todos los procesadores* las funciones de la Unidad de Control la UAL, los registros, la memoria principal, las interfaces, los periféricos, los buses etc.

Resulta así que conocer estos temas en un microcomputador permite abordarlas en cualquier tipo de computador corriente, siendo que todos están basados en el modelos de Von Neumann.

Pero además de servir como modelos para explicar cómo funciona un computador, los microcomputadores tiene la ventaja de ser accesibles a más personas por su bajo costo. Ahora, quienes estudian estos temas pueden visualizar en la pantalla de una PC aspectos de su funcionamiento que antes se veían en forma abstracta en los libros, por la imposibilidad de experimentar en minis o grandes computadoras

Esta cuarta edición conserva el carácter teórico-práctico de esta obra con el uso del Debug para visualizar en "cámara lenta" los resultados de la ejecución de cada instrucción de un programa, y agrega ejercicios integradores para números enteros, flags y reales, así como las bases para la comprensión de éstos en el Complemento final.

Asimismo, la presente edición, en función de la experiencia docente permanente del autor, se han puesto de relieve conceptos importantes que se daban por supuestos, o que pueden pasar inadvertidos en una lectura rápida.

Agradecimientos:

A mis hijos Rafael que también en esta edición interpretó y realizó con claridad dibujos complementarios, y Jerónimo (Analista de Sistemas U. de Sistemas FCEyN) por su lectura crítica de las novedades escritas.

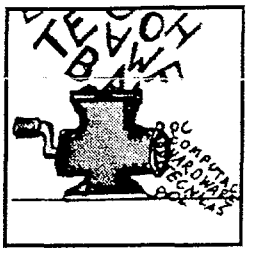
A mi ayudante Pablo Salaberri que siempre me acerca información sobre distintos temas, y que seguramente pronto colaborará en proximas ediciones de unidades de esta obra.

A mis alumnos, cuyas preguntas, dudas e inquietudes me permiten aprender cómo mejorar distintos temas.

El autor

INDICE DE TEMAS

1.1 SIMBOLOS, DATOS, PROCESOS DE DATOS E INFORMACION



¿Qué significa que un computador realiza automáticamente procesos de datos que constan de entrada, memorización, procesamiento y salida?

Se debe tener siempre presente que en esencia un computador lleva a cabo *procesos de datos*, con la particularidad que puede operar velozmente gran cantidad de datos en forma automática, sin intervención humana.

Como primer paso para ubicarse en el funcionamiento de un computador es importante tener en claro los cuatro subprocesos principales que realiza (*Entrada, Memorización, Procesamiento y Salida*), y ver que nos son familiares, por estar presentes en cualquier proceso de datos –mental, manual, o con calculadora– que hacemos sin computador.

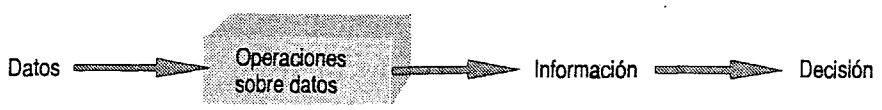
En los párrafos siguientes se pasa revista a procesos semejantes a los que ocurren dentro de un sistema de computación, a la par que se tratan conceptos tales como símbolos, datos, e información asociada a decisión.

Para no sucumbir, los seres vivientes debemos permanentemente llevar a cabo anticipaciones previsoras que aseguren una buena adaptación al mundo exterior. Por ejemplo, cuando sin darnos cuenta exponemos alguna parte de nuestro cuerpo a una fuente de calor excesivo, la retiramos rápidamente, sin pensar.
En "cámara lenta" podemos descomponer el proceso ocurrido en cuatro subprocesos:

1. Las células de nuestra piel de la zona expuesta han ***convertido*** la acción calorífica intensa proveniente del exterior en señales eléctricas que van hacia la médula, sin pasar por el cerebro. Así se ha *internalizado* dicha acción, o sea que se dio *entrada* a nuestro sistema de algo exterior representado por señales internas que podemos operar.
2. Células de la médula ***memorizan*** *informaciones anteriores* vitales –vía código genético– acerca de señales eléctricas que son "normales", por no superar su intensidad cierto umbral ancestral de "peligrosidad". Las señales que de modo continuo llegan del exterior también son *retenidas* muy brevemente.
3. Este tiempo es el necesario para que mediante otras funciones de la médula se pueda *determinar, cotejar, comparar* –como quiera llamarse– si no se superó dicho umbral. En esencia se han *procesado* señales eléctricas, con un cierto *resultado*.
4. Si del cotejo llevado a cabo resulta que la señal recibida supera ese umbral, códigos genéticos almacenados desatan reacciones, a partir de señales eléctricas enviadas a músculos del cuerpo para alejar la zona en peligro de la fuente calorífica externa. Esta vez se han ***convertido*** señales internas en acciones (movimientos) hacia el mundo exterior. O sea que nuestro organismo dio ***salida***, exteriorizó, el *resultado* del cotejo efectuado.

En un nivel de anticipación más abstracto, cerebral, no es necesario por ejemplo, que una fuente calorífica peligrosa esté cerca nuestro para alejarnos de ella. Basta que alguien nos comunique de la existencia de un incendio próximo para que, de ser necesario, tomemos la *decisión* de alejarnos del peligro. El mensaje que recibimos consiste en señales audibles, que **simbolizan**, representan, el incendio, sin necesidad que el mismo esté presente frente nuestro. Estas señales audibles, merced al órgano del oído o la vista, se convierten en señales eléctricas que permiten almacenar ese dato simbólicamente en el cerebro.
De igual forma, cuando pensamos en relación con cualquier tema, mentalmente realizamos *operaciones* con imágenes y palabras que son **símbolos** que *representan* sucesos, objetos, personas,

etc., **que *no* necesitan estar presentes necesariamente.**
Más concretamente, realizamos operaciones sobre representaciones simbólicas de *propiedades* cualidades conocidas de entes o sucesos.
Estas *representaciones simbólicas* son **datos** que *seleccionamos, reunimos,* y sobre los cuales *operamos*.
La operatoria realizada con *símbolos* que seleccionamos y reunimos, da por *resultado* **información**, que también son *símbolos* de propiedades de entes y sucesos, que antes desconocíamos. Si bien datos e información pueden ser sinónimos en el lenguaje común, en Informática son "input"-"output" de un proceso



La **información** sirve para *tomar* **decisiones**, con vistas a un accionar concreto (presente o futuro), y se obtiene realizando operaciones sobre datos. Su elaboración permite tomar conocimiento de algún aspecto de la realidad desconocido, lo cual disminuye la incertidumbre existente antes de tomar una decisión.

Por ejemplo, supongamos que una persona tiene que comprar un mismo artículo en tres comercios diferentes. Antes de concretar su compra, en su mente podrá tener las siguientes alternativas:

Si en el comercio A el precio es más barato, *entonces* compro en A
Si en el comercio B el precio es más barato, *entonces* compro en B
Si en el comercio C el precio es más barato, *entonces* compro en C

Dicha persona por los medios apropiados obtendrá el valor de venta de dicho artículo en cada comercio (datos), como ser en el A: $20; en el B: $19, y en el C: $ 21. Luego realizará operaciones de comparación entre los datos a fin de determinar cuál es el precio más bajo. Entonces habrá elaborado la información que le interesaba, supuestamente constituida por la representación simbólica "en B venden más barato", la cual le permitirá tomar la decisión de comprar en el comercio B entre las tres alternativas posibles. Obviamente, de no haber elaborado información, y si compra sin más en A ó en C, la decisión hubiese sido mal tomada, si el objetivo es el menor costo.

Los pasos necesarios para obtener información (resultado) a partir de datos (materia prima) constituyen un **proceso de datos**, o tratamiento de datos.

La información producida en un proceso de datos puede servir como dato en otro proceso.

A continuación determinaremos cuatro subprocesos en que puede dividirse un **proceso de datos mental** entre los innumerables que realizamos con el fin de obtener información, para poder conducirnos, para determinar por medio de representaciones simbólicas qué hacer en cada situación cotidiana.
Supongamos que alguien debe ir al cine a cierta hora, y debe *decidir*: si alcanza a bañarse o no; cómo ha de vestirse acorde al clima reinante, y a qué hora debe salir para llegar a horario. En este momento podría tener expectativas tales como "si tengo tiempo me gustaría bañarme", "si el tiempo está lindo me gustaría usar tal vestimenta", etc.
El procesamiento de datos que llevará a cabo puede descomponerse en una serie de acciones que ocurrirán en su interior, que están en relación con los bloques referenciados de a) hasta e) en la siguiente figura 1.1
La persona en cuestión podría proceder así:

a) *Entrada*: primero incorporaría selectivamente los datos pertinentes necesarios, usando típicamente la vista y el oído para sensarlos del exterior.

b) *Memorización*: los registraría en su mente, reunidos:
   - Ir al cine
   - Hora de inicio de la función: 20 hs (leída en un periódico)
   - Hora actual: 19 hs (leída en un reloj)
   - Estado y pronóstico del tiempo (de la radio o televisión)
   - Ropa de vestir para ese tiempo (dato en su memoria)
   - Tiempo que tarda en bañarse: 15' (dato en su memoria)
   - Tiempo de viaje normal: 15' (dato en su memoria)

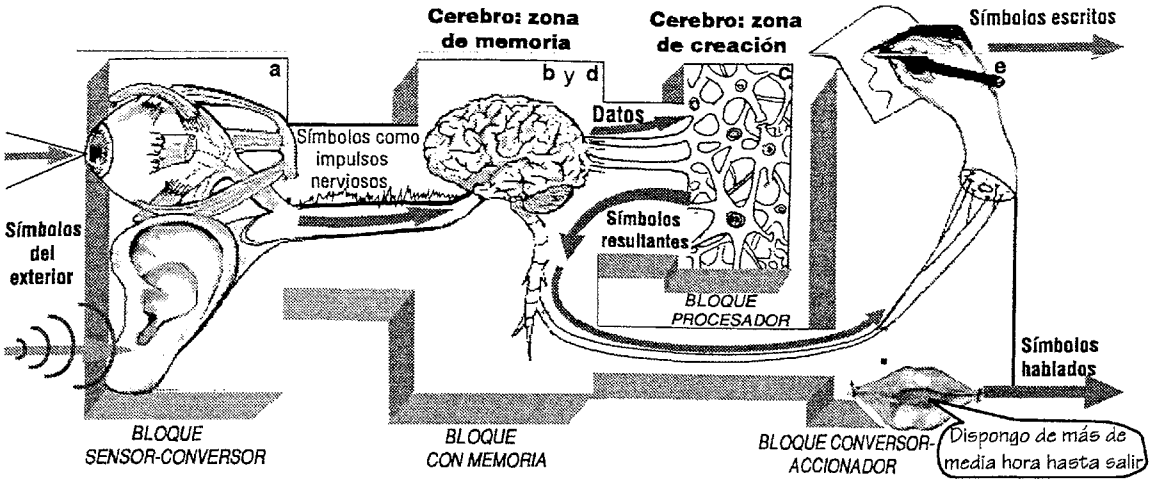


Figura 1.1

Los valores así hallados a partir de los datos primarios conocidos serán nuevos datos elaborados (*información*) Al realizar los cálculos anteriores también se establecieron *relaciones de orden*, al suponer qué se hace primero y qué después. A continuación relacionaría el pronóstico del tiempo con la ropa mas adecuada y lista para usar en esas condiciones climáticas, y establecería *relaciones de equivalencia*, entre vestimentas que tienen propiedades de abrigo similares. Eso también supone que debió realizar una clasificación. Luego podría usar su memoria, para extraer el dato estimativo de cuánto tiempo le lleva vestirse de una manera o de otra, etc.

d) Conforme al resultado del procesamiento de datos efectuado en el paso anterior, poseerá la siguiente **información**, que en forma escrita podría expresarse así: "Dispongo de más de media hora hasta salir, tiempo suficiente para bañarme rápido y vestirme con tal o tales prendas que están listas para usar. Como máximo debo salir a las 19.45 hs". *Ha resultado así un nuevo conjunto de representaciones simbólicas significativas, obtenidas a partir de aquellas correspondientes a los datos primarios.* Se han obtenido **símbolos** a partir de otros **símbolos**.

e) El resultado alcanzado (información "interna") puede ser *exteriorizado* como información externa, ya sea en forma verbal o escrita, si el cerebro ordena a los músculos relacionados con el habla que actúen, o a los músculos de la mano que escriban, respectivamente. Así exteriorizada, esta información por ejemplo podría comunicarse a otra persona, para que lo ayude a decidir correctamente. Si queda en la memoria de la persona que la elaboró, ella tomará luego las decisiones pertinentes.

Como punto de partida para comprender el funcionamiento de un computador, describiremos un proceso de datos manual auxiliado por calculadora. Este proceso, también puede desglosarse en cuatro subprocesos: entrada, memorización, procesamiento y salida, esquematizados en la figura 1.2

Antes de la aparición de computadoras, una supuesta oficina de cálculos de ingeniería funcionaba de la siguiente forma. Una persona idónea en el manejo de una calculadora común, se dedicaba a realizar operaciones con ésta. Calculistas programaban la serie de operaciones a realizar por el "idóneo" con la calculadora, y otra persona "auxiliar" las escribía ordenadamente en una planilla con renglones numerados, y también en ella los datos a procesar. La última orden de la secuencia puede indicar escribir en otra planilla una copia del resultado obtenido. Por ejemplo, una secuencia sencilla de operaciones ordenadas, que denominaremos "**instrucciones**" (simbolizadas I_1, I_2, ...) escritas en renglones sucesivos, podría ser la siguiente:

I_1: Registrar en el visor de la calculadora el número que está en el renglón 5000 (o sea el 1020)
I_2: Sumarle al número del visor el número que está en el renglón 5000 (nuevamente el 1020)
I_3: Restarle al número del visor el número que está en el renglón 5006 (o sea el 2040)
I_4: Escribir el resultado que totaliza el visor en el renglón 5010 (o sea el 0000 irá al renglón 5010)
I_5: Copiar en la planilla auxiliar el valor que indica el renglón 5010.

√ Es importante seguir en detalle esta secuencia de instrucciones, pues más adelante se repiten como instrucciones de máquina que se ejecutarán en una PC

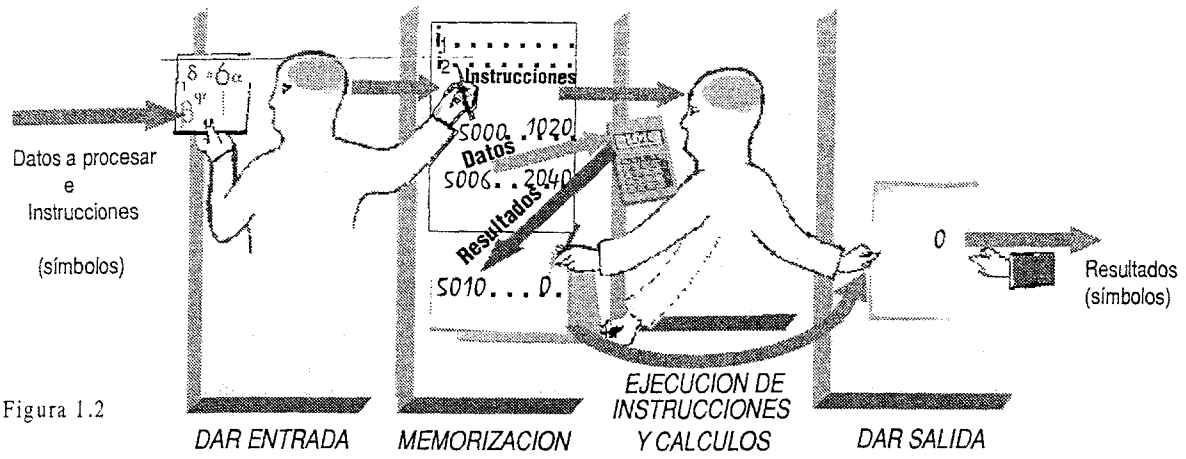


Figura 1.2

El primer bloque del esquema de la figura 1.2 da cuenta de la entrada de datos e instrucciones a la planilla, merced a la acción de la persona que programó las operaciones a efectuar. El "idóneo" **leerá en el orden dado** cada instrucción, y digitará en la calculadora la tecla correspondiente a la operación que se ordena. Luego localizará en la planilla y leerá el número que interviene en dicha operación, el cual será introducido a la calculadora por medio de su teclado.
La operación ordenada se llevará a cabo al pulsar la tecla igual (=). Un resultado parcial o total que está en el visor de la calculadora podrá registrarse (escribirse) en un renglón indicado de la planilla, cada vez que una instrucción (como $\mathbf{I_4}$) así lo ordene.
Con el cuarto bloque conversor se representa la acción del "idóneo" de dar salida hacia otra planilla auxiliar algún resultado o dato existente en la planilla principal, cuando así lo prescribe una instrucción (como $\mathbf{I_5}$)

Para los valores numéricos supuestos en los renglones, la operación realizada fue 1020 + 1020 – 2040 = 0 habiéndose asignado el resultado 0 al renglón 5010. Si llamamos **R** al resultado la secuencia de instrucciones anterior permite en general hallar el valor de la variable **R** en la expresión **P + P – Q = R**
Obsérvese que una calculadora común tiene memoria interna, utilizable con la tecla **M+** (que ordena sumar al número memorizado el que está en el visor, y el resultado memorizarlo en reemplazo de dicho primer número). Asimismo, el valor memorizado puede verse en el visor, pulsando **MR**.
Pero se trata de una memoria para un solo número. Con ella sólo se podrían llevar a cabo las instrucciones $\mathbf{I_1}$ e $\mathbf{I_2}$ de la siguiente forma. Suponiendo que en el renglón 5000 se haya escrito (memorizado) el número 1020, para hacer lo mismo en la calculadora, primero habría que tipear dicho número (1020) de forma que aparezca en el visor. Luego para memorizarlo se pulsaría **MC** (puesta a cero de la memoria) seguido de **M+**. Cuando se quiera ejecutar $\mathbf{I_1}$ se pulsaría **MR**, con lo cual necesariamente 1020 pasará de la memoria al visor (si por algún motivo 1020 fue reemplazado por otro número). Para ejecutar $\mathbf{I_2}$ se debe pulsar la tecla + seguida de **MR**, con lo cual al 1020, se le vuelve a sumar 1020 como en esencia ordena $\mathbf{I_2}$.

¿Cuáles son las operaciones primarias en los procesos de datos ?

Existen 8 acciones primarias ("**primitivas**") que pueden encontrarse en distintos procesos de datos:

- **Entrar** datos al sistema encargado de procesarlos
- **Asignar** un valor como perteneciente a un determinado nombre de datos o variables
- **Comparar** dos valores de datos para conocer la relación (< = >) existente entre ellos
- **Archivar**: almacenar datos durante un tiempo en algún medio
- **Recuperar**: leer, datos archivados en algún medio
- **Calcular**: generar, un nuevo valor aplicando una función matemática o textual
- **Borrar** datos archivados
- **Dar salida**: exteriorizar, obtener del sistema datos resultantes.

1.2 BASES PREVIAS PARA EL ESTUDIO DEL INTERIOR DE UN COMPUTADOR

Qué semejanzas tiene un computador con una fábrica que produce a pedido ?

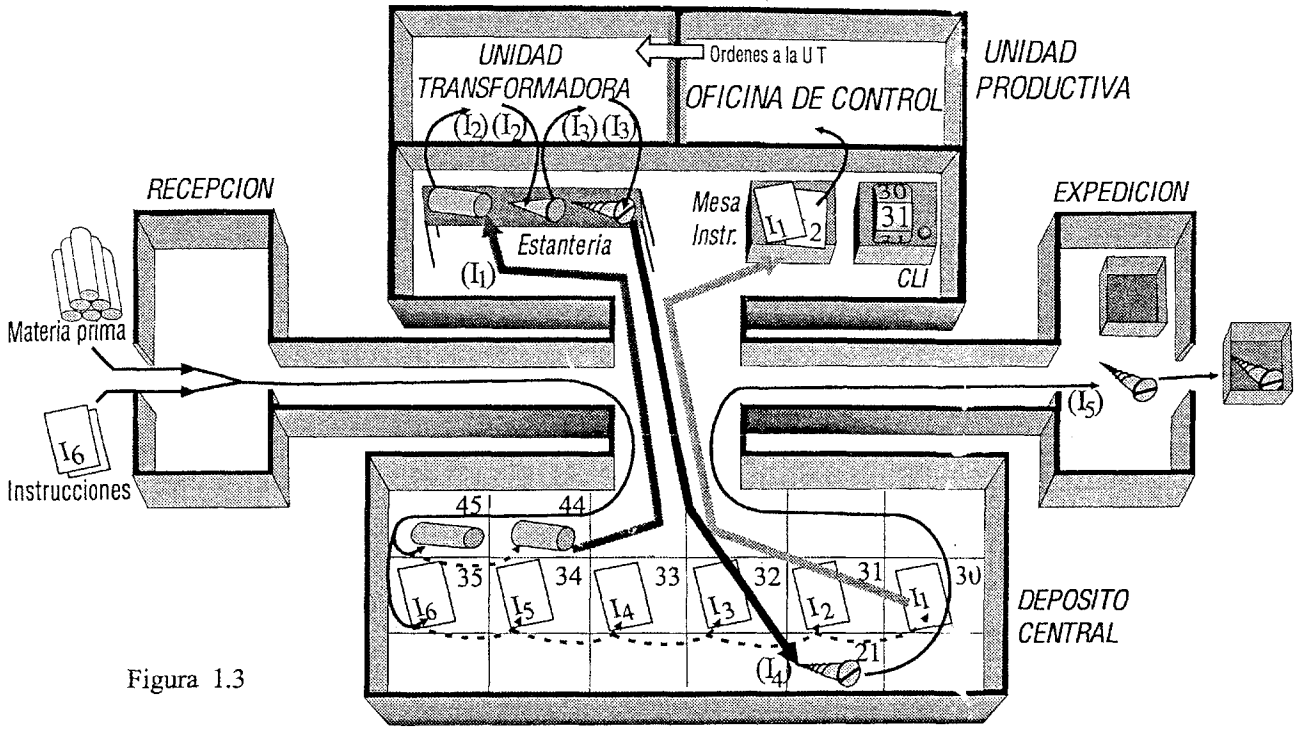


Figura 1.3

La Unidad Productiva (**UP**) de la fábrica de la fig. 1.3 produce piezas de metal a pedido (en este caso tornillos). Para ello se deben proveer (a través de Recepción) la *materia prima* y las *instrucciones* para producir cada pieza, que irán a **bŏxes** numerados del Depósito (**D**). Cada instrucción será pedida al depósito y luego cumplimentada por la Oficina de Control (**OC**), para lo cual debe llegar desde su box a la Mesa de Instrucciones (**MI**).

La materia prima (cilindro en este caso) irá desde el box donde se halla hacia una estantería **E**, y de ésta al torno automático de la Unidad Transformadora (**UT**) comandada por la **OC** según lo que ordena cada instrucción.

La OC no puede ver qué hay en cada box. Las instrucciones *siempre se ubican en boxes numerados consecutivamente* desde el 30. Para localizar cada instrucción la **OC** tiene en una mesa un contador mecánico para localizar instrucciones (**CLI**), que siempre arranca de 30 con un pulsador para avanzarlo.

Funcionamiento: una vez que instrucciones y materia prima están en **D** suena el timbre, con lo cual la **OC** pide la instrucción (I_1) que está en el box cuyo número (30) es el que aparece en el **CLI**, siendo que la misma irá hacia la **MI**. Luego desde la **OC** se pulsa el botón del **CLI** para que su número suba uno (31), a fin de que permita localizar I_2 cuando suene otra vez el timbre. Cuando I_1 llega a la **MI** en la **OC** se lee que ordena "Pasar la pieza que está en el box 44 (cilindro en este caso) hacia **E**", por lo que la **OC** impartirá directivas para que se cumpla dicha orden. Cuando ello ocurra, el cilindro que está en 44 habrá llegado a **E**, con lo cual sonará el timbre. Entonces la **OC** pedirá la instrucción (I_2) que está en el box 31, indicado por el **CLI**, y pulsará su botón para que indique 32.

Cuando I_2 llega a **MI** para ser leída por la **OC** ordena "Pasar a la **UT** la pieza que está en **E** (el cilindro), darle forma de cono y dejarlo en **E**". La **OC** ordenará los movimientos y a la **UT** la operación a realizar para ejecutar la orden, luego de lo cual sonará el timbre. Entonces la **OC** pedirá la instrucción (I_3) que está en el box 32, indicado por el **CLI**, y pulsará su botón para que indique 33.

I_3 ordena "Pasar a la **UT** la pieza **que está en E** (en este caso el cono), darle forma de tornillo y dejarlo en **E**". La **OC** llevará a cabo la orden, sonará el timbre, pedirá I_4 y el **CLI** pasará a 34. I_4 ordena "Pasar lo que está en **E** al box 21".

La **OC** cumplimentará I_4, sonará el timbre, pedirá I_5 y el **CLI** pasará a 35. I_5 ordena "Pasar lo que **está en** el box 21 a Expedición". En Expedición el tornillo será puesto en una caja para ser enviado al exterior.

Con vistas al funcionamiento de un computador el proceso descripto servirá para sistematizar y *definir las funciones* de los subsistemas tratados, existiendo las siguientes correspondencias en relación con las figuras 1.6 y 1.7:

UP = **UCP** (Unidad Central de Proceso o Procesador) **OC** = **UC** (Unidad de Control)
UT = **UAL** (Unidad Aritmético-Lógica) **MI** = **RI** (Registro de Instrucciones)
CLI = **IP** (Puntero de Instrucciones) **E** = **AX** (Registro Acumulador AX) Depósito (**D**) = Memoria

El Depósito (**Memoria**) sirve para almacenar instrucciones y materia prima (datos) y piezas terminadas (resultados). Está dividido en boxes (celdas) con números (direcciones) para localizarlos.
La **UP** (**UCP**) opera conforme a una secuencia de instrucciones que la **OC** (**UC**) obtiene, una por una, del Depósito (Memoria) para luego ejecutar cada una bajo su control. Una instrucción ordena un movimiento, o una operación y movimientos relacionados con ella. La operación la realiza la **UT** (**UAL**) que recibe materia prima (datos numéricos) y produce productos (resultados numéricos) que antes no existían, según la operación que ordena la **OC** (**UC**).

¿Cuáles son las "reglas de juego" para que el proceso siga correctamente ?
Dado que la **OC** (**UC**) no sabe dónde está en el Depósito (**Memoria**) cada instrucción, como tampoco dónde está la materia prima (datos) a operar, **se deben cumplir las siguientes reglas** para que el proceso siga correctamente:
- Antes de iniciar el procesamiento al Depósito (Memoria) deben entrar las instrucciones y la materia prima (datos) a procesar. Esto como se vio no afecta el proceso a realizar, pues con el CLI se localiza cada instrucción, y ésta indica dónde está lo que operará.
- Inicializar **CLI** (**IP**) con el número de box (dirección) que puede ser cualquiera, donde se halla la primera instrucción.
- Ubicar las instrucciones de una secuencia en boxes (celdas) **consecutivos**, siendo que cada una se localiza por su número de box dado por el **CLI** (**IP**), y luego pasa a **MI** (**RI**) donde es leída (decodificada) por la **OC** (**UC**). Si por ej. **CLI** pasa de 33 a 34 se asume que en 34 está la instrucción que sigue. De no ser así, se pierde el hilo del proceso.
- Construir cada instrucción de modo que **provea** el número (dirección) del box (celda) **donde encontrar** la materia prima (dato numérico) que se ordena operar, y que indique implícita y explícitamente el lugar a dónde irán resultados.

Cada instrucción: **1**) ordena una operación; **2**) permite localizar aquello que se va a operar; **3**) indica dónde va el resultado; **4**) a partir de su localización permite que se encuentre la siguiente instrucción a ejecutar, mediante el CLI (IP).

En la **UP** (**UCP**) existe una tercer zona para almacenamiento temporario donde están **E** (**AX**), **MI** (**RI**) y **CLI** (**IP**).
La **UP** (**UCP**) y el Depósito (Memoria) -de una sola abertura cada una- se comunican por un pasillo (líneas de datos de un bus) por el que pasa cada instrucción seguida (o no) de materia prima (datos) o resultados. Este pasillo (bus) a su vez se conecta con pasillos para la comunicación entre el Depósito y la Recepción o la Expedición (**periféricos**).

Cuando en la fábrica la materia prima pasa de un box hacia **E**, deja de estar en el box, pero con los datos (símbolos) es distinto. En el acto de leer, una copia de lo que vemos pasa a nuestra retina sin que desaparezcan del papel los símbolos leídos. En los procesos de datos, *incluidos los que hace un computador*, siempre que se lee información almacenada, **una copia de la misma** pasa al lugar de destino, quedando sin modificar la información leída. En un computador dicha copia destruye la información existente en el destino.

En definitiva, continuamente se repite el siguiente ciclo: con el **CLI** (**IP**) se localiza la instrucción a ejecutar que va a **MI** (**RI**), y luego **CLI** (**IP**) toma el valor para localizar la instrucción siguiente una vez que se ejecute la que llegó a **MI** (**RI**). Esta a su vez permite localizar la materia prima (dato) a operar según la operación ordenada, cuyo resultado irá al destino indicado en dicha instrucción. Luego otra vez con el **CLI** (**IP**) se localiza la instrucción a ejecutar que va a **MI** (**RI**), etc.

¿Qué es lo básico que se necesita conocer acerca de bits, bytes y de la equivalencia entre binario y hexa para operar con el programa debug ?

El **Apéndice 1** trata en detalle qué es un sistema numérico, el sistema binario, el hexadecimal, la suma y resta binarias y la codificación ASCII de caracteres tipeados. En lo que sigue se dan conceptos básicos para entender el sistema binario y poder operar en hexa para experimentar con el programa Debug (Sección 1.6).

100 10 1

En el sistema decimal **3 0 9** simboliza que en un conjunto con ese número de elementos se formaron 3 grupos de 100 (3x100), y que con los 309 - 300 = 9 elementos restantes no se pudo formar ningún (0) grupo de 10 (0x10), pero sí 9 grupos de 1 (9x1). O sea: 3x100 + 0x10 + 9x1 = 309.
Los grupos son de 1, 10, 100, 1000, . . . elementos. Cada uno es 10 veces el anterior, siendo 10 la cantidad de símbolos usados (0 al 9) para simbolizar cualquier número. Pueden formarse hasta 9 grupos de cada tipo

En general, en cada sistema numérico posicional, partiendo de grupos de un elemento, cada tipo de grupo es tantas veces mayor que el anterior como la cantidad de símbolos empleada en un sistema (10 veces en decimal, dos veces en binario) *Cada sistema numérico es una manera distinta de dividir en grupos de ese sistema al conjunto de elementos cuyo número se quiere simbolizar.* Se pueden formar hasta *k* grupos de cada tipo, siendo *k* el símbolo mayor del sistema: hasta 9 en decimal; sólo hasta un grupo de cada tipo, o ninguno, en binario, y hasta 15 = F en hexadeciamal.

	8 4 2 1	
0	0 0 0 0	0 = 0x8 + 0x4 + 0x2 + 0x1
1	0 0 0 1	1 = 0x8 + 0x4 + 0x2 + 1x1
2	0 0 1 0	2 = 0x8 + 0x4 + 1x2 + 0x1
3	0 0 1 1	3 = 0x8 + 0x4 + 1x2 + 1x1
4	0 1 0 0	4 = 0x8 + 1x4 + 0x2 + 0x1
5	0 1 0 1	5 = 0x8 + 1x4 + 0x2 + 1x1
6	0 1 1 0	6 = 0x8 + 1x4 + 1x2 + 0x1
7	0 1 1 1	7 = 0x8 + 1x4 + 1x2 + 1x1
8	1 0 0 0	8 = 1x8 + 0x4 + 0x2 + 0x1
9	1 0 0 1	9 = 1x8 + 0x4 + 0x2 + 1x1
A	1 0 1 0	10 = 1x8 + 0x4 + 1x2 + 0x1
B	1 0 1 1	11 = 1x8 + 0x4 + 1x2 + 1x1
C	1 1 0 0	12 = 1x8 + 1x4 + 0x2 + 0x1
D	1 1 0 1	13 = 1x8 + 1x4 + 0x2 + 1x1
E	1 1 1 0	14 = 1x8 + 1x4 + 1x2 + 0x1
F	1 1 1 1	15 = 1x8 + 1x4 + 1x2 + 1x1

Figura 1.4

Así, en el **sistema binario** que usa dos símbolos **0** y **1** para repre-sentar cualquier número (con el mismo significado de dichos símbolos en decimal), cada grupo será el doble que el anterior. Simbolizados en decimal serían: 1, 2, 4, 8, 16, 32, 64, 128, . . . Como el símbolo mayor es **1**, sólo se puede formar hasta un grupo de cada tipo. Un conjunto que en decimal se simboliza 13 (formado por un grupo de 10

8 4 2 1

y 3 grupos de 1) en binario será: **1 1 0 1.**

Ello significa que se pudo formar un grupo de 8 (1x8), y que con los 13 - 8 = 5 elementos restantes se pudo formar un grupo de 4 (1x4), y que con los 5 - 4 = 1 restantes no se pudo formar ningún (0) grupo de 2, pero sí un grupo de 1 (1x1). De este modo 1x8 + 1x4 + 0x2 + 1x1 = 13

Por lo tanto ahora el mismo conjunto que en decimal se había dividido en un grupo de 10 y 3 grupos de 1, en binario se ha dividido en un grupo de 8, un grupo de 4 y un grupo de 1. En la fig, 1.4 aparecen formados como cuartetos los números binarios que en decimal se corresponden con los números del 0 al 15.

128 64 32 16 8 4 2 1

El 130 decimal en binario sería: 1 0 0 0 0 0 1 0

Cada uno de los símbolos que compone un número binario es un *dígito binario*, en inglés **bin**ary digi**t**, abreviado **bit.**

Vale decir que un bit puede valer 0 ó 1. El 1101 tiene 4 bits, y el 10000010 tiene **8** bits. Cualquier conjunto de 8 bits se denomina **byte** (octeto en castellano).

El sistema numérico hexadecimal ("hexa") usa 16 símbolos, del **0** a **F** (con su correspondencia en decimal y binario indicada en la fig. 1.4), con los cuales se puede formar cualquier número. Los símbolos del 0 al 9 tienen el mismo significado que los análogos decimales.

Puesto que en binario se requiere para representar un mismo número algo más del triple de dígitos que en decimal, y dado que la información en el interior de un computador es de 8, 16, 32, 64 y hasta 128 bits, resulta engorrosa de ver en pantalla, libros y otros medios, y también para escribirlos. La tabla de la fig 1.4 permite pasar rápidamente, por simple reemplazo, cualquier número binario a hexa y viceversa.

Para ello se separa visualmente el número binario, suponiendo que sea 11000000 en cuartetos: 1100 0000.

En la figura 1.4 vemos que 1100 en hexa es C y que 0000 es 0; por lo tanto 11000000 = C0.

Si usando el Debug, en la pantalla leemos que un registro contiene A5B4, hallando el cuarteto que le corresponde a cada símbolo y reemplazando, resulta: A 5 B 4 = 1010 0101 1011 0100. O sea si leemos A5B4, en el interior del computador existen esos 16 bits.

Vale la pena recalcar que en el interior de un computador no puede existir "hexa". Sólo hay binario.

¿Cuál es la ventaja de operar en el interior de un computador con dos estados eléctricos correspondientes al 0 y 1 binarios ?

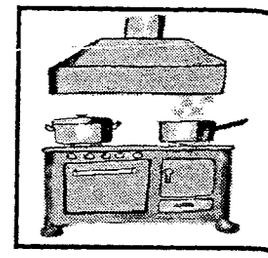
La breve explicación conceptual que sigue hace hincapié en la complejidad tecnológica y la menor confiabilidad que implicaría establecer diez estados eléctricos necesarios para representar dígitos decimales

Los millones de transistores que componen hoy día los circuitos de un computador funcionan como las llaves de dos estados, "si-no" usadas para la electricidad hogareña; pero por ser los transistores dispositivos electrónicos, pueden cambiar de un estado al otro millones de veces por segundo. Esto es, cada transistor opera en uno de dos estados perfectamente definidos: deja pasar la corriente eléctrica (**1**) o no (**0**).

Operar tecnológicamente con dos estados es mucho más simple, y también más confiable, que hacer trabajar a los transistores con diez valores de corrientes o tensiones eléctricas distintos, con el fin de generar diez estados diferentes, para poder representar los dígitos 0 al 9 del sistema decimal.

Por otra parte, dichos valores deberían permanecer fijos con la temperatura, con la complicación que además, debido a las dispersiones propias del proceso de fabricación de circuitos integrados ("chips"), los valores de las corrientes generadas variarían naturalmente en más o en menos –dentro de un cierto rango respecto a valores nominales promedio– con cada circuito que sale de fábrica.

1.3 HARDWARE DEL COMPUTADOR



¿Qué es el hardware?

Hardware son los medios físicos (equipamiento material) que permiten llevar a cabo un proceso de datos, conforme lo ordenan las instrucciones de un cierto programa, previamente memorizado en un computador.

En inglés "duro" es *"hard"*, y *"hardware"* significa "ferretería".

El hardware de un computador es la totalidad física, conformada por todos los componentes de su equipamiento: circuitos electrónicos (hoy microcircuitos contenidos en "chips" cuadrados con patas para conexión), plaquetas que los soportan, cables o caminos conductores (buses) que los interconectan, mecanismos, discos, motores, gabinetes, tornillos, pantallas, teclas, etc.
Con estos elementos se construyen los distintos bloques funcionales del hardware: procesador, memoria, periféricos, interfaces, etc.

En la figura 1.5 se indican elementos constituyentes del hardware correspondientes a la plaqueta principal de una PC, conocida como "**motherboard**" cuyos bloques principales se irán tratando.

✔ Salvo detalles de la "motherboard" y otros menores, **los conceptos y definiciones que siguen son válidos para el funcionamiento de *cualquier* computador.** Como se indica en el prefacio, **todos los computadores con un solo procesador funcionan básicamente de la misma forma que una PC actual.** Cualquier procesador actual (Pentium, Motorola, Risc, etc) o anterior, funciona sobre la base del llamado "**modelo de Von Neumann**") planteado en 1946, cuyos enunciados se dan en la sección 1.14

¿Cuáles son los bloques constituyentes básicos del hardware de un computador y qué funciones cumplen ?

En lo que sigue, si bien se parte de la fig. 1.2 es esencial tener presente lo dicho en relación con la fig, 1.3

La comprensión de las funciones que cumple cada una de las partes que conforman el proceso de datos de la figura 1.2, *permite entender cómo funciona en esencia cualquier computador, dado que cada trabajo que realiza un computador siempre es un proceso de datos, que tiene la particularidad de ser automático.*
En un proceso automático (figura 1.6) también están presentes los cuatro subprocesos constituyentes: **Entrada – Memorización – Procesamiento – Salida** que llevan a cabo los bloques a definir (periférico de entrada – memoria principal–unidad de procesamiento – periférico de salida), siendo que en un computador existen diversas posibilidades para la entrada o salida de datos. Las flechas indican movimientos semejantes en ambas figuras.
Los bloques se comunican eléctricamente entre sí a través de caminos formados por un conjunto de cables o líneas conductoras que constituyen un "**bus**". **>>>Un mapa de buses actual está en la fig 1.80.**
A los fines didácticos de mantener los cuatro bloques citados en el orden dibujado en las figuras 1.2 y 1.3, aparecen repetidos dispositivos que pueden actuar tanto para la entrada como para la salida de datos (las unidades de disquete y disco, y el módem). El bloque **I** ser refiere a una **Interfaz** intermediaria, necesaria para conectar su periférico, la cual contiene los registros **ports** (**>>Ver sección 1.10**)
En líneas muy generales, en las figuras 1.6 y 1.7 se supone lo que sigue. Un disco de la unidad de disco rígido provee un *programa*, cuyas instrucciones pasarán a través de buses hacia la memoria. Los *datos* llegarán –también a través de buses– a la memoria, provenientes del teclado.
Luego, dichas instrucciones son ejecutadas, *una por vez*. A tal fin primero cada una por un bus llega a

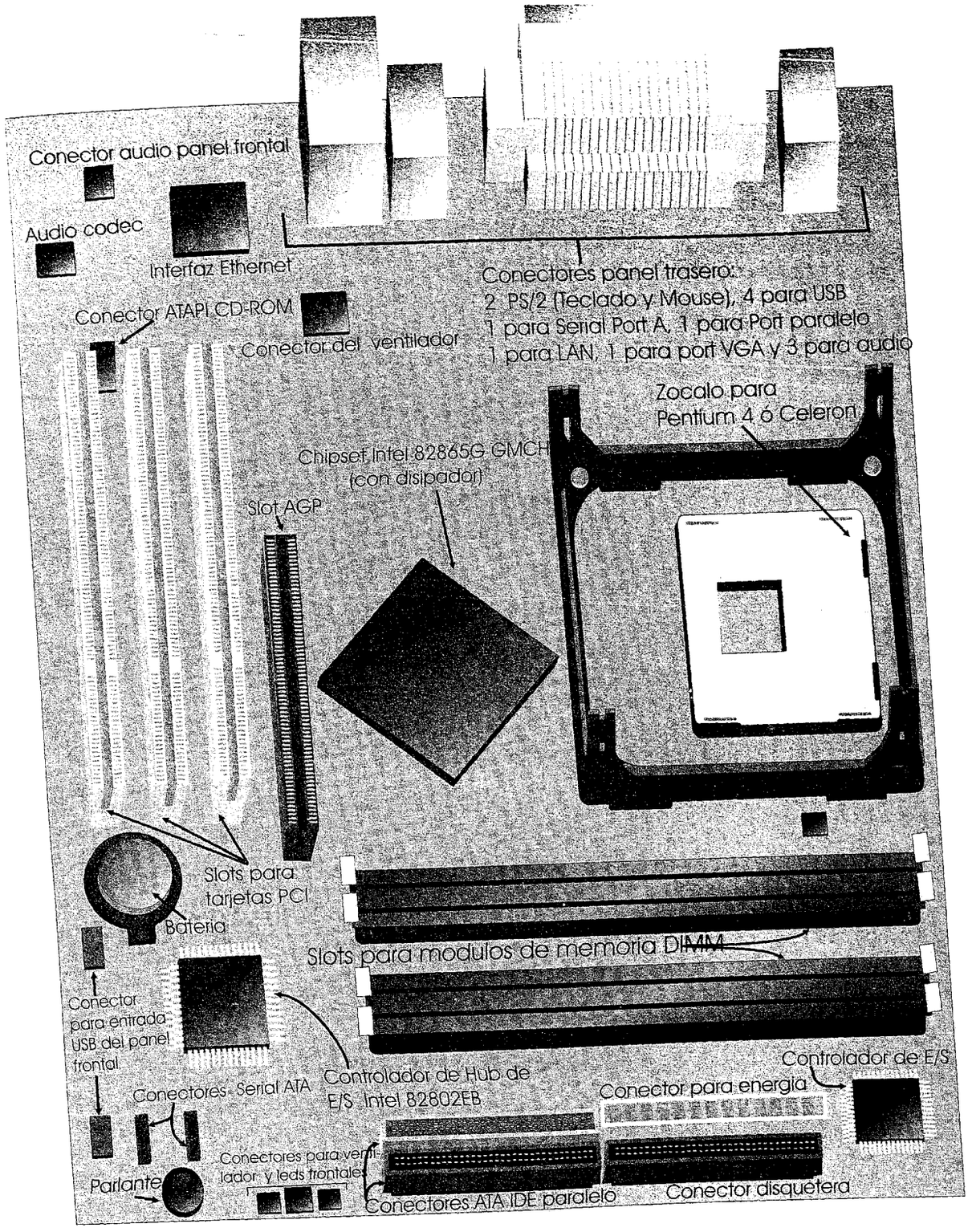


Figura 1.5

Este dibujo es un esquema con los dibujos aproximados de los conectores, chips y dispositivos más importantes para el usuario, que aparecen en la foto completa de la "motherboard" de Intel® D865GRH desarrollado por Intel Corporation para el Pentium® 4 y Celeron®.
En Internet http://www.intel.com/design/motherbd se encuentran detalles importantes de esta "mother", en particular el chipset, ROM, conexión de los módulos de memoria y otros temas importantes.

un **registro de instrucción** (RI) de la Unidad Central de Procesamiento (**Procesador**), donde permanece mientras se ejecuta, para que la Unidad de Control interprete qué operación ordena ella.
A continuación, a través del mismo bus, el dato a operar por dicha instrucción llega desde memoria a un **registro acumulador AX** [1] del procesador, antes de ser operado (conforme a la operación ordenada) en la Unidad Aritmética, a fin de obtener un resultado. Este puede sustituir en el registro AX al dato ya operado[2], y luego pasar a memoria –nuevamente a través del bus citado– si una instrucción así lo ordena.
Si por ejemplo se quiere enviar dicho resultado al exterior para ser visto en pantalla, o para ser guardado en el disco rígido o en un disquete, ello se consigue mediante la ejecución de instrucciones que así lo ordenen.

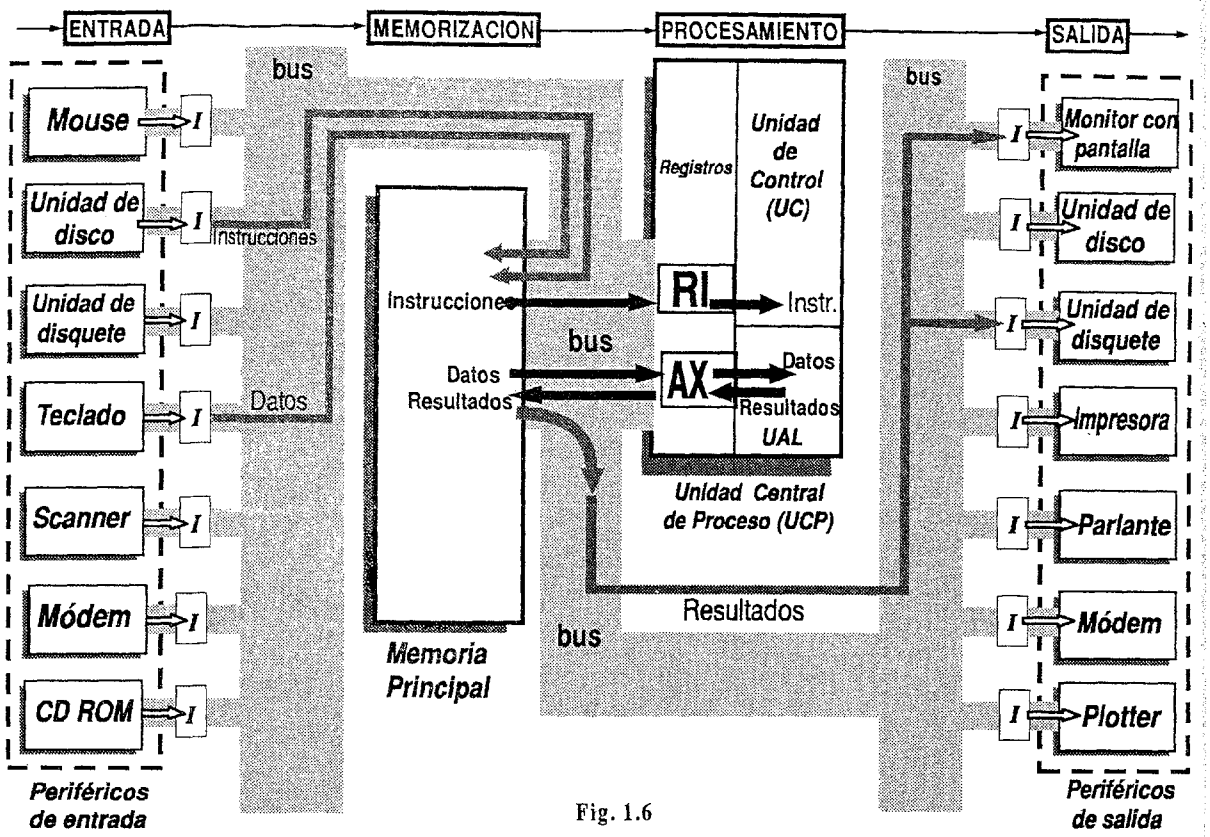


Fig. 1.6

En la figura 1.6 se ha reemplazado al "idóneo" de la figura 1.2 –*encargado de obtener de la memoria en el orden establecido cada instrucción y ejecutarla*– por un circuito denominado "**Unidad de Control**" (**UC**), capaz de realizar sus mismas acciones básicas, pero millones de veces más veloz.
Dichas acciones formaban parte de una secuencia siempre *repetitiva*:

- obtener de la memoria la próxima instrucción que corresponde ejecutar,
- localizar los datos a operar (en la memoria principal, o en un registro como AX u otro, según se ordene)
- ordenarle al circuito de la Unidad Aritmética que realice con dichos datos la operación indicada,
- guardar el resultado en un registro acumulador o en memoria principal

Por lo tanto:

> La **UC** tiene a su cargo el *secuenciamiento* de las acciones necesarias que deben realizar los circuitos involucrados en la ejecución de cada instrucción, según el código de la misma; y también tiene a su cuidado el orden de ejecución de las instrucciones de un programa, conforme como éste fue establecido

La calculadora de la figura 1.2 se ha dividido en dos porciones: una que contiene los circuitos de *cálculo*, que denominaremos **Unidad Aritmética Lógica** (**UAL** o ALU en inglés), y otra constituida

[1] Del mismo modo que al registro visualizable en una calculadora llega un número, o como en la figura 1.3 la materia prima queda transitoriamente en una estantería, antes de ser tratada por el torno.
[2] Como en el registro visualizable de una calculadora el resultado reemplaza a un número operado.

por el **registro**[1] **acumulador** designado AX, para datos y resultados (que en la calculadora es su registro visualizable). La UC ordenará mediante señales eléctricas transmitidas por cables, las operaciones (suma, resta, multiplicación, etc.) que debe realizar la UAL. Esta forma de control electrónico directo reemplaza a las lentas teclas de una calculadora de la figura 1.2. Asimismo, el hecho de tener un registro separado, permite a la UC entrar al mismo datos a operar en forma electrónica directa, mediante cables que llegan a él, sin que tampoco se requiere teclas para ello.

> La **UAL** sirve para realizar las operaciones aritméticas o lógicas que le ordena la UC, siendo auxiliada por registros acumuladores para guardar transitoriamente resultados datos y resultados.

Mientras que la UC es la encargada de ordenar operaciones de lectura-escritura de registros y de memoria, así como de las operaciones que debe realizar la UAL, ésta es pasiva: no puede emitir orden alguna.
Por lo tanto, la UAL no ejecuta instrucciones. O sea, no puede ordenar las operaciones correspondientes a los pasos que requiere la ejecución una instrucción. Sólo realiza uno de ellos: la operación aritmética o lógica que la instrucción ordena, cuando así lo requiere la UC. Mediante una operación de la UAL, a partir de uno o dos números (materia prima) se puede obtener un número (resultado) que antes no existía.
Conforme al proceso de la figura 1.2, *no debe perderse nunca de vista que el conjunto* UAL-Acumulador *forman de hecho una calculadora con funciones semejantes a una de bolsillo, siendo la UC la encargada de manejarla, según la operación ordenada.*
La palabra "Lógica" de las siglas UAL puede llevar a equívocos, en el sentido de suponer inteligencia. Como se ejemplifica en 1.8, las operaciones "lógicas" de la UAL son las operaciones AND, OR, negación, etc

> Se denomina **Unidad Central de Proceso** (**UCP** o **CPU** en inglés) al *conjunto* formado por:
> - la Unidad de Control
> - la Unidad Aritmético-Lógica
> - los registros (como el AX, RI y otros) usados durante la ejecución de cada instrucción.

La UCP es el bloque donde se lleva a cabo la ejecución de las instrucciones. Hacia ella se dirigen las *instrucciones* que serán ejecutadas (una vez que la UC decodifique su código), y los *datos* para ser operados por la UAL; a la par que de la misma salen resultados generados por la UAL.
Como se verá, además del registro acumulador definido, en la UCP existen otros registros necesarios para la ejecución de las instrucciones, que se irán definiendo.

La UCP de una microcomputadora –como lo es una PC– está contenida en el chip **microprocesador central**[2] (por ejemplo, el 80286/386/486, Pentium, 68000, Alfa, Power PC, etc.[3]).
Los términos UCP, microprocesador y procesador suelen ser sinónimos.

Al igual que el depósito fabril de la figura 1.3, la memoria *principal* o *central* o *interna* almacena materia prima (datos), instrucciones, y los resultados del proceso realizado en circuitos electrónicos[4]. O sea:

> La **memoria principal** (**MP**) almacena instrucciones de programas, que próximamente serán ejecutadas en la UCP, y los datos que ellas ordenan procesar (operar); así como resultados intermedios y finales de operaciones sobre datos recientemente llevadas a cabo en la UCP.

---

1 La palabra "**registro**" en general indica un circuito que puede almacenar (o sea registrar) temporariamente datos o instrucciones. Si bien en un computador hay registros en distintos subsistemas, (y también podemos llamar registro a cada celda de 8 bits de memoria principal y a cada "port", ubicado como ser en una plaqueta insertable), registro "a secas", sin aditamentos, implica un registro de la UCP.

> Los registros de la UCP sirven para guardar transitoriamente información relacionada con la instrucción en curso de ejecución, y con las próximas instrucciones a ejecutar. Por ejemplo, el 80386 y 80486 tienen 8 registros de uso general de la UCP.

Esta información se refiere a códigos de instrucción, datos (a operar o resultados), y direcciones de memoria.

2 Este puede contener otros circuitos, como el coprocesador matemático, la memoria caché, el reloj de xx-MHz (megahertz) y otros.

3 En el chip de un microprocesador *no está contenida la MP*. En este chip hoy encontramos el coprocesador matemático, a tratar.

4 La exigencia de una memoria interna *electrónica* para instrucciones y datos a procesar, tiene que ver con la necesidad de acceder a ella rápidamente en el interior del computador. Esto es, puesto que la UCP opera a velocidades electrónicas, debe poder leer o escribir dicha memoria a una velocidad compatible con la suya. Si la UCP tuviera como memoria un disco (memoria exterior) estaría atada a los tiempos de acceso a éste, que implican la realización de movimientos mecánicos, con lo cual se desaprovecharía la rapidez de procesamiento electrónico de la UCP. Por lo tanto, si una de las virtudes de un computador es su rapidez, la memoria también debe estar constituida por circuitos electrónicos para el almacenamiento y lectura de información.

Vale decir, *los datos que se procesan y el programa que se ejecuta para ese proceso deben estar en MP. Cada programa comparte la MP con sus datos, pero las instrucciones están en una zona y los datos en otra.*
Esta información queda almacenada *temporariamente* mientras se opera con ella[1]; pudiendo ser luego reemplazada por otras instrucciones a ejecutar, y datos que éstas procesan. También existen programas que residen en MP en forma permanente, como los del sistema operativo, que facilitan el uso de un computador, cuya ejecución se alterna con la de programas de los usuarios.

Las instrucciones y datos a procesar que pasan a la UCP llegan a la MP desde el exterior del computador (figura 1.6); y los resultados que llegan a MP provenientes de la UCP deben luego pasar al exterior:

En una **operación de entrada**, la MP es el *destino* de instrucciones y datos provenientes del exterior (que ingresan a través de unidades de discos o disquetes, teclado, mouse, módem, u otros).
Asimismo, en una **operación de salida**, la MP es el *origen* de resultados que deben salir al exterior (a través del monitor, impresora, unidades de discos o disquetes, módem, u otros).

Típicamente los programas llegan a MP –para ser ejecutados– *provenientes de archivos en discos o disquetes.*
Los datos a procesar pueden llegar a MP provenientes del exterior desde cualquier periférico.

> Los dispositivos que se encargan de entrar desde el exterior datos o instrucciones hacia el computador, o dar salida de resultados del computador al exterior, se denominan **periféricos** o **unidades de entrada/salida**.

Para tal fin su función principal es **convertir** datos externos en internos en las operaciones de entrada, o a la inversa en las operaciones de salida.
Un periférico oficia de *"frontera"* entre el exterior y el interior de un computador para la *conversión* de señales. Del mismo modo (figura 1.1), nuestros ojos, oídos, piel, etc., sensan señales externas y las transforman en señales eléctricas que van hacia nuestro cerebro o médula. Nuestras cuerdas vocales y cavidades asociadas permiten un proceso opuesto para la comunicación con el exterior.

Hay que diferenciar el periférico de lo que es su exterior. Así, para los periféricos unidades de discos (o disquetes), *el exterior está constituido por el disco*; para la impresora el exterior es el papel, para el módem es la línea telefónica, etc.
Usualmente, en un sistema de computación personal existe un conjunto de periféricos (teclado, mouse, y otros) construidos sólo para entrar del exterior datos o instrucciones, que se escriben en MP. Periféricos como el monitor con pantalla y la impresora sólo pueden dar salida a datos o resultados. En cambio, las unidades de discos o disquetes y el módem son periféricos que operan ya sea para entrada o salida, pudiendo llevar a cabo *una de estas operaciones por vez.* Por tal motivo, en el esquema de la figura 1.6 aparecen repetidos en la entrada y salida, *aunque en realidad se trata siempre de la misma unidad* actuando de una forma u otra, como ya se expresó.
La denominación "periféricos" proviene de su posición, vinculada al mundo exterior en relación con la **porción central o interna**, constituida por el microprocesador (UCP) y la memoria principal.

Por las razones que se exponen en la sección 1.10, *un periférico no se conecta directamente a la porción central*, sino por intermedio de una **interfaz** circuital (indicada con la letra **I** en la figura 1.6), que en una PC en general está contenida en una plaqueta que se inserta en un zócalo apropiado (figuras 1.60 a 1.65).

> Debe consignarse que la UC **no** gobierna directamente a los periféricos mediante líneas que llegan a ellos, sino que la UCP ejecuta un subprograma preparado para cada periférico, merced al cual desde la UCP llega a la interfaz del periférico cada comando que ordena a la electrónica de éste qué debe hacer.

Los periféricos como el teclado, el monitor, la impresora, el graficador ("plotter") y cualquier otro que permita la comunicación directa mediante símbolos usados por los hombres, se denominan **terminales**.
Las unidades de disco (magnético u óptico), de disquete, de cinta son periféricos conocidos como **unidades de almacenamiento masivo**, también denominadas **memorias auxiliares** o **externas** o **secundarias**.

Distintos circuitos de un computador se comunican entre sí mediante un conjunto de conductores (cables o

---

[1] En general los resultados son guardados como archivos en un disco (rígido o disquete) o son impresos en papel. De no hacerse así, los resultados se perderían al apagarse el equipo, pues la MP no los puede conservar si falta energía. Un programa que fue ejecutado es reemplazado por otro a ejecutar, proveniente del disco, siendo que la mayoría de los programas que están en MP son copia de los que están en el disco. En MP también existen programas y datos en forma *permanente*, como los usados cada vez que se enciende un equipo.

líneas) que interconectan eléctricamente las patas de los chips que contienen dichos circuitos.
Así, las líneas conductoras de electricidad que salen de las patas del chip de un microprocesador para transmitir direcciones, instrucciones, datos y resultados, constituyen el denominado "**bus local**".

> Un **bus** de un computador es una estructura de interconexión para la comunicación selectiva entre dos o más módulos de un computador, a fin de poder transmitir información entre dos módulos por vez.

En general, en un bus encontramos líneas para direcciones, datos, y señales de control (a veces denominadas bus de direcciones, bus de datos, y bus de control, respectivamente - Ver figura 1.8).
Las **líneas de dirección**, conducen de UCP a MP cada combinación de unos y ceros que indica dónde localizar instrucciones o datos en MP. Es *unidireccional.*
Las **líneas de datos**, en cada lectura de MP conducen de ésta hacia la UCP tanto *datos a operar* como *instrucciones;* y en una escritura conducen desde la UCP hacia MP datos resultantes. Son pues, líneas *bidireccionales.*
Las **líneas de control**, son unidireccionales individuales para que la UCP dé órdenes -cómo leer o escribir MP- y para que ella reciba señales –como la que origina la MP para indicar lectura efectivizada–

Un bus presente en las PC es el bus **PCI** (Peripheral Component Interconnect) creado por Intel, al cual –a través de zócalos (figura 1.5)– se conectan plaquetas para conexión de periféricos, y de otros buses. Este bus también emplea las tres clases de líneas citadas. No está vinculado directamente al procesador central.
En la sección 1.13 se detalla este bus, el **USB** y el **SCSI**, y en la figura 1.80 se da un mapa de cómo se comunican entre sí (como sugieren las figs 1.6 y 1.7) a través de chips que hacen de "puente" entre distintos buses.
Las señales eléctricas digitales se transmiten por las líneas de un bus como se indica en la fig 1.48

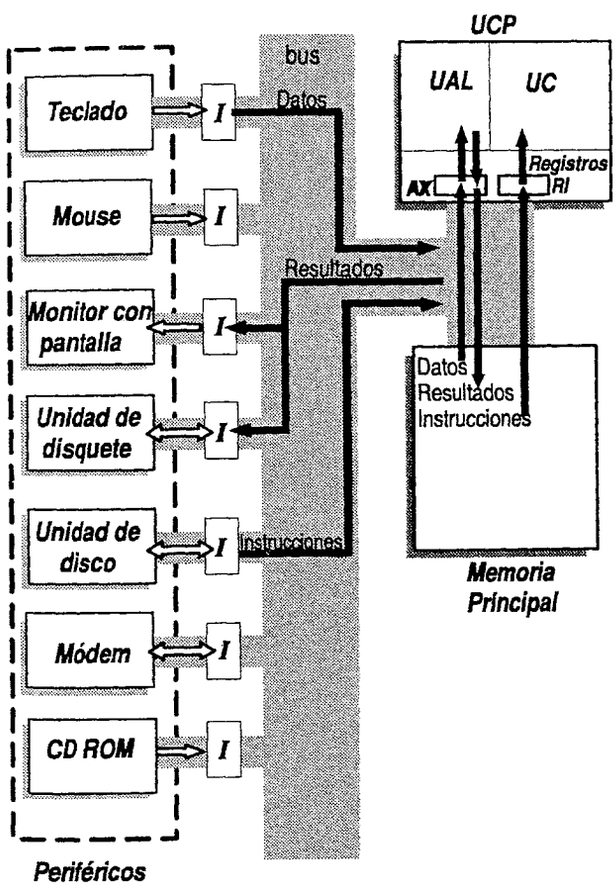


Figura 1.7

Las figuras 1.7 y 1.6 son en esencia similares, salvo la ubicación relativa de la UCP (más acorde a la figura 1.3), y que los periféricos de entrada o salida (según indica el sentido de las flechas) aparecen espacialmente en un mismo nivel de proceso, aunque como se aclaró, por ejemplo una unidad de disco cuando actúa como periférico de entrada no puede simultáneamente operar para salida, y viceversa.
En ambas figuras se han respetado los orígenes y destinos de los movimientos.
Otra diferencia menor, pero importante, es que por la forma en que se comunican los buses de la figura 1.7, es factible que datos o instrucciones que entraron por un periférico, en lugar de pasar directamente a memoria principal[1], primero pasen a un registro de la UCP, y de éste a memoria, resultando una "triangulación".
Esto es lo que sucede en una PC cuando se entran datos desde el teclado, el mouse, o el disco rígido. Asimismo, en una operación de salida, un resultado en memoria puede pasar a un registro de la UCP, y desde éste ir a un periférico. Así se realiza una impresión o un almacenamiento en el rígido en una PC.
En cambio, la unidad de disquete envía o recibe información directamente a memoria, sin "triangulación"
Estas alternativas de triangulación para entradas y salidas no se han dibujado, para no complicar la figura.

En la "mother" de una PC existen "chips" que integran distintas funciones, constituyendo el "chipset"[2]

---

[1] Conocido como "*Acceso Directo a Memoria*" (**ADM** o sea DMA en inglés). Es la forma teóricamente más rápida de dar entrada o salida a la información, que emplean masivamente los computadores medianos y grandes. Por las razones que se exponen más adelante (sección 1.10), en una PC con un procesador rápido como el 486 o el Pentium, resulta más rápida la "triangulación" que el ADM.

[2] Para el Pentium están los "chipsets" de Intel: 430FX (Triton), 430 HX (Triton II) y 430VX, y el 450GX para el Pentium Pro.

¿Cómo puede resumirse el funcionamiento básico de un computador ?

0. Los *datos* y las *instrucciones* del programa que los procesará, deben llegar a MP desde periféric
Cada instrucción está codificada mediante una combinación de unos y ceros, que constituye su códig
1. La UC localiza en MP la instrucción que debe ser ejecutada (como se explica en la siguie
pregunta), para que su código llegue a la UCP, donde la UC determinará qué ordena ese código
2. Dicho código permite: *localizar los datos* que operará la UAL, *la operación* que debe realizar
UAL, *dónde* guardar el resultado[1], y *dónde* localizar la próxima instrucción en MP.
Para realizar esto, la UC ordenará una secuencia de operaciones de lectura y escritura sobre MP o regist
de la UCP, así como la operación a realizar por la UAL. A la UAL van los datos a operar, y desde ella
generan resultados que van a MP, para luego encaminarse hacia el exterior a través de un periférico.
3. Se vuelve al paso 1.

Si una instrucción ordena una transferencia de un dato desde la UCP hasta la plaqueta donde está conect
do un periférico, tendrá lugar una operación de salida, encargándose el periférico de llevar los datos al e
terior (representado por la pantalla del monitor, un disco, etc.). Igualmente existen instrucciones para lleva
un dato que entró desde un cierto periférico hasta la UCP, mientras se desarrolla una operación de entrada

Se debe tener siempre presente que las instrucciones son interpretadas ("decodificadas") por la UC; lueg
de lo cual la UC ordena encaminar los datos hacia la UAL, y la operación que ésta debe realizar con los datos.

¿Qué registros de la UCP falta definir para realizar las primeras prácticas con e programa Debug, a fin de operar en el interior de un computador ?

En la figura 1.2 el "idóneo" recuerda, en una zona de su cerebro, cuál instrucción ejecutó, para lueg
leer la instrucción siguiente en la planilla. Esta sirve para escribir qué ordena realizar cada instrucción
Si por razones de velocidad el "idóneo" sólo pudiera leer una sola vez cada instrucción de la planill
–como ocurre con la UC– debería registrar en su mente o en algún papel dicha información.
Para tal fin, como antes se trató, el código de cada instrucción leído en memoria principal (MP)
queda registrado en el registro de instrucción (RI), para que la UC determine qué ordena el mismo
y llevar una cuenta para poder ubicar en MP la próxima instrucción a ejecutar.

Para localizar en MP la siguiente instrucción que sigue en orden de ejecución, en la UCP existe otr
registro muy necesario para la UC, conocido como "**contador de programa**" (CP) o "**registro de
próxima instrucción**", que en el modelo de PC que operaremos con el Debug será el denominado
registro puntero de instrucción (IP - Instruction Pointer). El IP indica un número, el cual permite
localizar *en una zona* de memoria principal dónde está la próxima instrucción a ejecutar. [2]

Además de los registros AX, RI e IP, definiremos un cuarto registro denominado "**registro de
estado**" (**RE**), que contiene un conjunto de bits llamados **indicadores** ("**flags**"), los cuales pueden
cambiar de valor luego de cada operación que hace la UAL. Los "flags" se usan en instrucciones de
salto, esenciales para el proceso automático de datos, pues permiten repetir muchas veces un
mismo subprograma, o saltar de un subprograma a otro que ordena acciones diferentes.
También una calculadora común muestra, junto con el resultado de una operación, otras indicacio-
nes de interés, que pueden asimilarse a dichos "flags". Así, un signo menos que aparezca en el
visor adelante del resultado de una resta, nos indica que el minuendo es menor que el sustraendo.
La no aparición de este indicador de signo, implica que el minuendo es mayor. Asimismo, si el resultado de
una resta es cero, significa que ambos números son iguales. Otro indicador es el de resultado excedido.
Igualmente, la UAL *luego de cada operación aritmética o lógica que realiza,* genera *"flags"* de estado del
resultado: si fue positivo o negativo, si fue cero o no, si como número entero sobrepasó o no el
máximo representable Cada uno de estos "si" o "no" es un bit que se guarda en el registro de estado citado,
actualizado por la UAL

---

1 Cuando el "idóneo" de la figura 1.2 lee las instrucciones I_1, I_2, I_3, . . . se supone que obtiene una información semejante.

2 En realidad este número se forma sumando el valor de IP con el valor de un registro del 80x86 denominado "de segmento de código" (CS) multiplicado por 16, pues la memoria de una PC está dividida en segmentos. Si a los fines didácticos y de simplicidad, se supone que el valor de CS es cero, ignorando su valor real, equivale a trabajar con el primer segmento de los que componen la memoria, siendo que el IP indica la posición del mismo que se quiere direccionar.

1.4 LA MEMORIA PRINCIPAL O CENTRAL

¿ Que son las direcciones y los contenidos de la memoria principal ?

La memoria principal almacena bits (unos y ceros) en *celdas* independientes, aisladas entre sí, que contienen un byte (8 bits) de información (figura 1.8, página 1.16).
Cada celda se localiza en el conjunto mediante un número binario identificatorio, que constituye su "**dirección**", o indicación de su "**posición**" en ese conjunto.
Este número no se puede alterar, pues está establecido circuitalmente.

Por lo tanto, en relación con cada celda se tiene *dos* números binarios:
- un número fijo, la **dirección** (de más de 20 bits), que presentado en los circuitos de la memoria permite acceder a una celda; y
- un número de 8 bits, que es el **contenido** informativo de esa celda, o sea la combinación de unos y ceros almacenada en ella. Este número puede cambiarse si la memoria es alterable.

Es costumbre representar las celdas de una memoria, o una porción de ella, mediante un conjunto de casilleros verticales formando una "escalera", siendo sus direcciones números binarios consecutivos
Estos números binarios se escriben en papel o en pantalla al lado de cada celda, en su equivalente hexadecimal, como muestra la figura 1.8, a fin de no tener que visualizar largas cadenas de unos y ceros.
En la figura 1.8 (pag 1-16) se supone que en las líneas del bus de direcciones (ampliado con la "lupa") se envía la dirección 0000 0010 0000 0111 = 0207H, en la cual está almacenado el byte 01100001 = 61 H

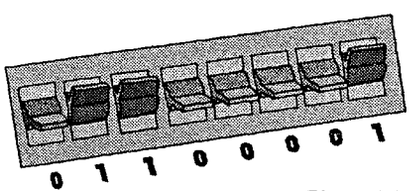


Figura 1.9

Puede ayudar a entender mejor el concepto de byte almacenado, si se piensa que en cada casillero existen 8 llaves del tipo "si-no" (figura 1.9), como las comunes de pared para encender la luz, cada una para retener un uno ("si") o un cero ("no")[1]. Entonces, para una celda dada, como la que contiene 01100001, la combinación de unos y ceros que están formando las 8 llaves es la información contenida en dicha celda. La información que almacena cada grupo de 8 llaves puede referirse a instrucciones o datos.
El modelo de las llaves también es útil para tener presente que en cada posición de MP siempre existe una cierta combinación de unos y ceros, o sea no es posible que ella no contenga "nada", pues las 8 llaves siempre están presentes, cada una en "si" o en "no".

En cada dirección de memoria (celda) sólo pueden leerse o escribirse **8 bits por vez**, sin posibilidad de operar menor cantidad de bits, o un bit aislado.

Puesto que la palabra "registro" en un significado general indica algún lugar donde datos se pueden registrar, guardar, podría designarse "registro" a cada celda de memoria, y decir que *la MP está formada por un conjunto de registros independientes de 8 bits cada uno.* Para evitar tener que aclarar "registro de la UCP" y "registro de MP", se entiende que "registro" a secas significa que es de la UCP, y las celdas de MP se denominan *"posiciones"*.
Cuando los datos o instrucciones ocupen más de un byte, se almacenan *fragmentados* en varios bytes, los cuales deben estar contenidos en celdas *consecutivas* de memoria, o sea en direcciones sucesivas
En una operación de lectura o escritura de MP *se puede acceder a varias celdas consecutivas.*
Un cierto número de éstas (2, 4, u 8 bytes) suele denominarse **palabra** ("**word**") de memoria

Tecnológicamente la MP reside en microcircuitos electrónicos, que pueden guardar un cierto número de bits, construidos sobre una fina capa de silicio (*semiconductor*), conformando un *"chip"* (figura 8.12) Este se protege con un encapsulado de plástico o cerámica en forma de pastilla con "patas" metálicas para conexionado. Se requieren varias pastillas para lograr el total de memoria necesaria (figura 1.8, a la izquierda)

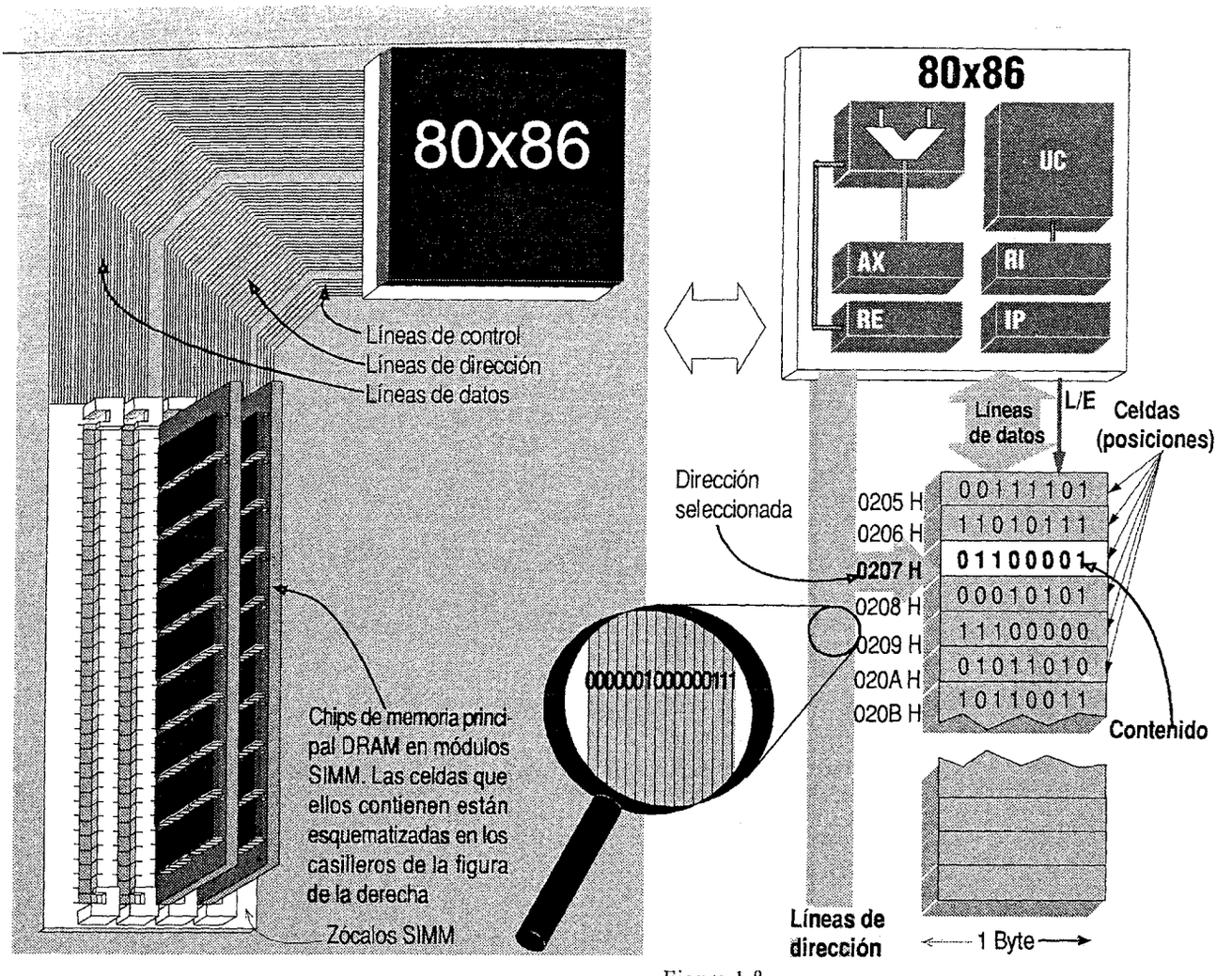


Figura 1.8

A un subconjunto de "patas" de esas pastillas se le envía cada dirección, en otro subconjunto aparecen los contenidos, etc. Como aparece en la figura 1.8, en una PC varios chips de memoria están insertos en una pequeña plaqueta que constituye un módulo o banco de memoria. Este se inserta en un zócalo ("slot"). Esta tecnología de conexionado puede ser del tipo SIMM (Single in line Memory Module) o DIMM (Dual in line Memory Module, siendo que zócalos DIMM aparecen en la motherboard de la figura 1.5).

¿Cómo se direcciona, se lee y se escribe la memoria principal ?

En relación con la MP sólo son posibles dos operaciones que puede ordenar la UCP: la *lectura* o la *escritura*, pero antes de realizar cualquiera de ellas, la UCP debe direccionar la MP.

La acción de **direccionar** (**direccionamiento**) consiste (fig 1.8) en colocar en las líneas de direcciones del bus que llegan a MP, la dirección de la celda a la que se quiere acceder, para leerla o escribirla

A continuación (figuras 1.10 y 1.11) se repetirá el esquema de la derecha de la figura 1.8 para ilustrar cómo se lee y escribe la memoria principal. Los números binarios contenidos en las celdas de memoria de esas figuras han sido convertidos a hexadecimal, y se supone que el procesador puede leer o escribir en memoria un word de 2 bytes, o sean dos posiciones consecutivas, con sólo dar la dirección de la primera.

Operación de lectura de una palabra en un acceso a memoria (figura 1.10):
La UCP "no sabe" qué combinación de unos y ceros existe en cada celda. Para conocerla debe indicar su dirección, y la MP le proporcionará la combinación que guarda la celda direccionada por la UCP. Si pide leer un word, la MP le enviará dicha combinación junto con la contenida en la posición siguiente.

Más en detalle (figura 1.10), una operación de lectura de un word comprende los siguientes pasos básicos:

1. La UCP ordena lectura mediante la línea de Lectura/Escritura (L/E = 1)[1], que va de la UCP a MP
2. En las líneas de dirección, la UCP coloca la dirección de la primer celda que se quiere leer (0207H)
3. Luego de un tiempo, una copia del contenido de la posición direccionada (61H) y del contenido de la siguiente (15H), aparecen juntas (6115 H) en las líneas de datos del bus, a disposición de la UCP

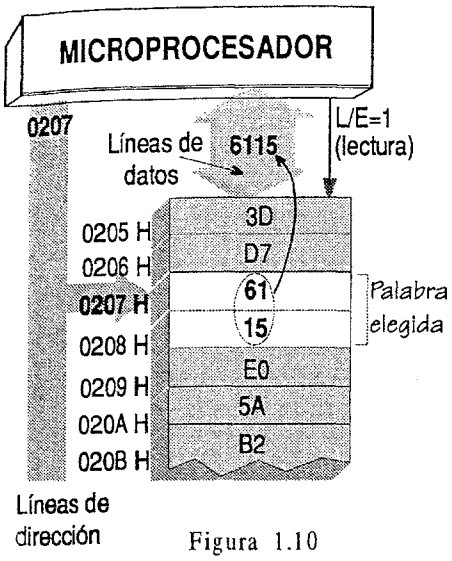


Figura 1.10

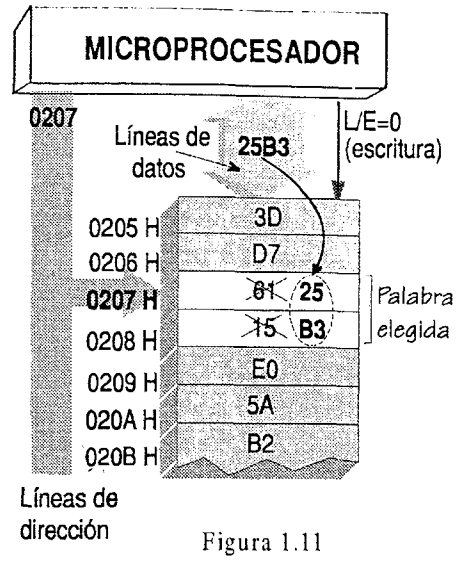


Figura 1.11

Con el modelo de las 8 llaves por celda propuesto, una lectura consiste en determinar en qué estado ("si" o "no") está cada llave, y luego transmitirlo al bus de datos.

Operación de escritura de una palabra de dos bytes en un acceso a memoria (fig. 1.11): Consiste en cambiar la combinación de unos y ceros contenida en las celdas que conforman la palabra de memoria direccionada, para lo cual:

1. En las líneas de dirección del bus la UCP coloca la dirección de la primer celda que se quiere escribir
2. La combinación binaria a almacenar en las celdas (por ejemplo 25B3 en hexa) es colocada por la UCP en las líneas de datos del bus.
3. La UCP ordena escritura mediante la línea de control L/E la cual queda brevemente en cero.
4. Luego de un tiempo, una copia de la combinación enviada a MP queda almacenada en la celda direccionada y en la siguiente (en cada una cambia la combinación de unos y ceros que forman las 8 llaves)

Se ha supuesto que las mismas celdas que antes fueron leídas, ahora son escritas con otro contenido distinto. Con la idea de las 8 llaves por celda, algunas o todas cambian su estado (de "si" a "no" o viceversa) para almacenar el nuevo contenido de unos y ceros. De esto se deduce que *una escritura es destructiva*, en el sentido que se pierde, desaparece, el contenido anterior, pues las llaves que cambian de estado no pueden volver al que tenían antes de la escritura. En el ejemplo, los contenidos 61 y 15 fueron reemplazados por 25 y B3

¿Qué es tiempo de acceso a memoria y su medida en nanosegundos ?

Tiempo de acceso: es el que transcurre entre que se direcciona una memoria, hasta que aparece en sus salidas (conectadas a las líneas de datos del bus) el contenido de la celda direccionada.

Figura 1.12

Este tiempo suele indicarse en un chip de memoria al final de su código (figura 1.12), como ser el número 70. Esto significa que es tan corto como 70 nanosegundos.

El **nanosegundo** es una unidad de tiempo que significa una *mil millonésima de segundo* (0,000000001 seg. = 10^{-9} segundo), o sea mil veces menor que una millonésima de segundo.

En el presente los procesadores operan internamente en el orden del nanosegundo.

Qué significa que el acceso a la memoria principal es al azar (random) ?

En los televisores que tenían un dial rotatorio, para pasar, por ejemplo, del canal 2 al 7 el dial debía recorrer la secuencia de números que están entre 2 y 7. Del mismo modo, en una memoria de cinta magnética

[1] Esto significa que dicha línea de control está en 5 volts. Si L/E = 0, esa línea está en 0 volts. Los procesadores 80x86 pueden leer también un byte de memoria, lo cual implica que debe existir otra línea de control (no dibujada) para indicar leer un word o un byte

–como la de un casete de audio o TV– si se quiere acceder a una determinada porción de la cinta, h
–que buscarla pasando por todas las porciones que la anteceden, a partir de la que se encuentra frente
cabezal. Decimos que en ambos casos se trata de un *acceso secuencial* al lugar donde está la informaci
buscada, que si bien requiere dispositivos sencillos, los tiempos de búsqueda a veces son inaceptables.

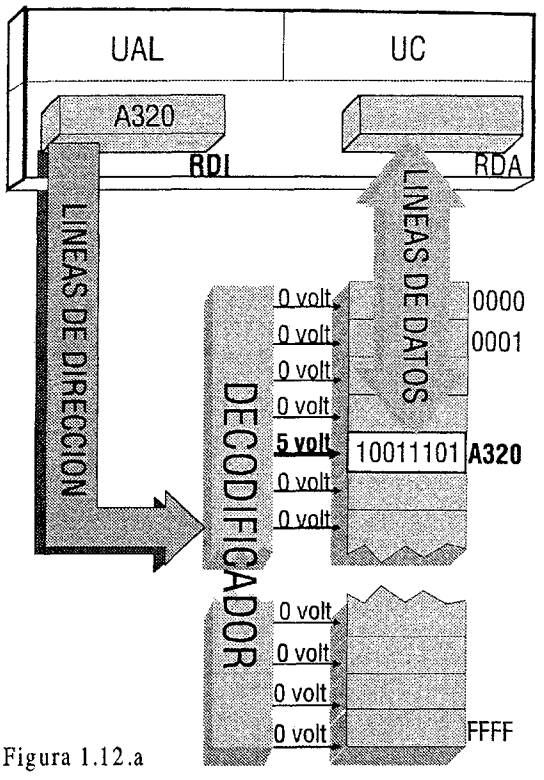


Figura 1.12.a

En cambio en una TV con control remoto, mediante la botonera forma el número de canal al que se quiere acceder, y el can aparece directamente en la pantalla, sin necesidad de ver pas otros números de canales.
Esta forma directa de acceder, seleccionar o ubicar algo, denomina "**random**" (*al azar* en el sentido de que con botonera de un control remoto puede formarse cualqui número de canal al azar, y éste aparecerá en pantalla *igual tiempo* que cualquier otro número de can seleccionado, sin importar el número del mismo y si búsqueda alguna. Un sistema de este tipo requiere un mayor complejidad, tanto en el selector como en el televiso Igualmente, acceso directo o "**random access**" en una memor implica que *cualquier* posición *puede encontrarse en igu tiempo* (para ser leída o escrita), *sin búsqueda alguna.*
Vale decir, *que el tiempo de acceso es el mismo pa cualquier dirección sin importar su número.*

A continuación veremos que en la UCP existen funciones que se quiere son análogas a las del control remoto de TV citad Para tal fin, se requiere añadir en el modelo propuesto de UC dos registros: RDI y RDA que ahora se definen, y que aparece en el esquema de la figura 1.12.a.
En el **registro de direcciones (RDI)** de la UCP *se forma cad dirección que será enviada a MP por las líneas de direcció* por lo cual está en contacto con éstas. Si bien en la UCP no existe ninguna botonera para forma números correspondientes a direcciones, la UC se encarga de indicar cómo se generará una direcció en RDI, sea por que ella ya existe en otro registro de la UCP, o por que debe ser resultado de u cálculo a realizar. Lo que importa es que *antes de que una dirección llegue a MP, la misma se forma en RD* registro del cual salen las líneas de dirección del bus local.
Del mismo modo, el número del canal de TV seleccionado se forma primero en una memoria del contr remoto, y luego automáticamente se transmite (mediante ondas infrarrojas) a los circuitos del televisor.
En un computador, la UCP y MP no se comunican en esa forma inalámbrica, sino por señale eléctricas que viajan por las líneas de dirección del bus que las interconecta.
Una vez que circuitos de MP decodifican la dirección enviada desde RDI, se accede a la celd direccionada Análogamente, cada vez que circuitos de un televisor reciben el número del can seleccionado en el control remoto, se encargan de que sea el que aparece en pantalla.

Así como por RDI debe pasar cada dirección que se envía por las líneas de dirección el bus, existe u registro en contacto con las líneas de datos del bus (en la fig 1.12.a está en la UCP, pero puede estar en MP que denominaremos **registro de datos (RDA)**. Se usa para guardar en forma transitoria la información qu la UCP envía a MP por estas líneas, o que debe recibir desde MP a través de ellas.

¿Cómo es más en detalle el acceso random a una celda de memoria ?

Los **n** bits de la dirección formada en RDI (supuesta 1010001100100000 = A320) viajan por las líneas de bus de dirección hasta las entradas del circuito decodificador, que forma parte de la memoria (fig. 1.12.a) Cada una de las 2^n líneas de salida del decodificador va a cada uno de las 2^n celdas de la memoria de líneas de dirección.. Esto es, si hay un millón de celdas, a cada una llega una salida del decodificador.
Para la combinación binaria supuesta (A320) *una sola* de esas 2^n líneas estará a 5 volts, y las restantes a volts. La celda de memoria conectada a la línea que está a 5 volts será la accedida.

Dada que una vez que llegó una dirección al decodificador cualquiera de sus 2^n salidas tarda igual en pasar a 5 volts, resulta que no importa cuál el número que recibió el decodificador, la celda correspondiente se accederá en el mismo tiempo (*esto permite el acceso random y sin búsqueda*).
En un esquema de memoria al lado de cada celda se acostumbra a escribir su dirección en hexadecimal (que en este caso van de 0000 a FFFF), siendo A320 la dirección de la celda supuestamente direccionada.
Se comprende que las celdas no tienen físicamente ningún número grabado para identificarlas, sino que es el decodificar el que se encarga de establecer la correspondencia entre el número binario que es la dirección de la celda a acceder y la celda asociada con dicho número, mediante el conductor conectado a dicha celda. Esto último implica que para cada dirección que llega al decodificador, siempre se accederá a la misma celda, puesto que el conexionado viene fijado en la fabricación de cada chip de memoria, y es idéntico para todos los chips del mismo tipo y fabricante.
Esta técnica puede asimilarse a un supuesto sistema de localización de viviendas, en el cual en cada esquina existe una botonera en la que se entra el número de la casa a localizar, y luego se enciende una luz en la entrada de dicha casa, para ubicarla sin búsquedas. Ello exige que de la botonera debe ir un cable a cada casa, de modo que con cada dirección se energice el cable que enciende la luz de la vivienda correspondiente.

Cuando se accede a una celda determinada, todo sucede *como si los 8 bits de ella se conectaran a 8 líneas de datos del bus*, para que esos 8 bits sean sensados (leídos) o modificados (escritos) según se ordene.

En la fig. 1.12.a se supuso un solo decodificador conceptual, con una salida por celda. La figura 1.12.b muestra que cada celda se localiza en la intersección de dos líneas, por ejemplo la horizontal 11 y la vertical 01 para la dirección 1101. Por lo tanto en realidad se usan dos decodificadores: a cada uno va la mitad de los bits de cada dirección. En nuestro caso A3 va al decodificador horizontal, y 20 al vertical.

¿ Cómo se accede a celdas consecutivas en un solo acceso a memoria ?

El decodificador de una memoria de 2^n celdas (figura 1.12.a) *sólo permite acceder a una celda por vez* para leerla o escribirla. Si se quiere acceder también a la celda siguiente, ésta también debe ser luego direccionada, para que luego el decodificador la identifique. Operando de esta forma, para acceder a dos celda consecutivas harían falta dos tiempos de acceso; y cuatro para cuatro celdas consecutivas, etc.
A fin de poder acceder en un solo tiempo de acceso a 2^k celdas consecutivas se debe disponer en principio de 2^k módulos (bancos) de memoria independientes. Para las DRAM existe la variante "interleaving".
Acceder por ejemplo a 2 celdas consecutivas (figura 1.10) supone 2 módulos (figura 1.12.a.2) cada uno con su decodificador y 16 líneas para datos en el bus que va a la UCP. La idea básica es la siguiente:

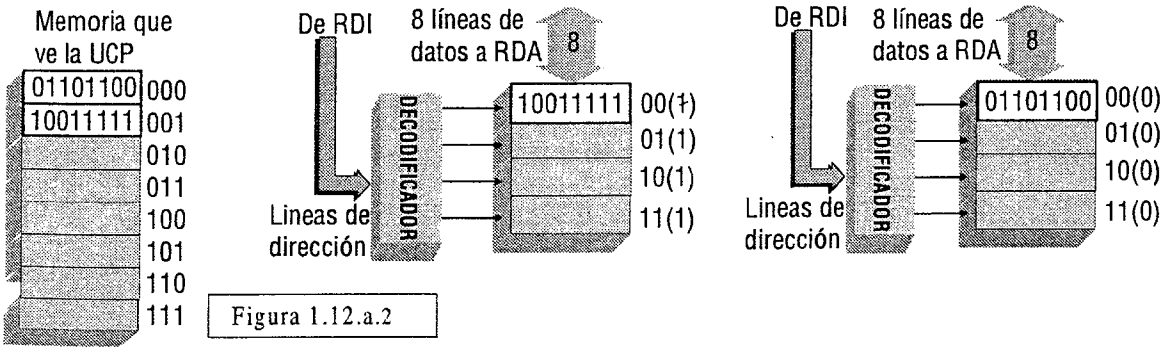


Figura 1.12.a.2

Si bien la UCP direcciona una memoria de 8 celdas (000 a 111), ésta en realidad consta de dos módulos separados de 4 celdas (00 a 11). Si la UCP necesita direccionar 2 bytes, como ser los de las celdas 000 y 001, enviará la dirección (par) de la primera (000) por el bus de direcciones. A los dos decodificadores de los dos módulos no les llega el bit extremo derecho de la dirección emitida por la UCP, sino que a ambos les llega la dirección 00, y cada módulo en una lectura enviará el contenido de 8 bits correspondiente a esa dirección al bus de datos, como se indica. Así la UCP recibirá juntos los dos bytes de las direcciones 000 y 001, en el tiempo que dura un acceso, siendo que se realizaron dos accesos de igual duración simultáneamente. El módulo izquierdo aportará las direcciones pares (000, 010, 100 y 110) y el derecho las impares restantes, o sea que el valor 0 ó 1 del último bit derecho de

cada dirección selecciona el módulo donde está su contenido. La dirección del primero de los dos bytes a acceder debe ser siempre par. Caso contrario se requieren dos accesos.
En caso de que se necesite leer el byte par o el impar en forma individual, la UCP seleccionará el que sea entre los dos bytes que proveyó la memoria. La línea que ordena lectura llega a los dos módulos, siendo que ambos son leídos juntos, y la UCP toma los dos bytes o uno de ellos, según necesite.
Para poder escribir uno u otro byte de los dos módulos en forma separada, hace falta que a cada módulo le llegue una línea de escritura propia. En definitiva, en este ejemplo la UCP debe enviar dos líneas para controlar la memoria. Por ejemplo en éstas 11 puede significar leer (uno o dos bytes), mientras que 01 y 10 ordenan escribir el módulo izquierdo o el derecho, respectivamente.
El mecanismo anterior está pensado para la situación más probable de que una memoria es accedida preferentemente en direcciones consecutivas, principio en que se basa el funcionamiento de un caché (sección 1.12). Un mínimo porcentaje de accesos no siguen esta ley, por lo que para ellos no puede aprovecharse esta concepción. El modelo de la figura 1.12.b puede generalizarse para una memoria cualquiera de 2^n celdas divididas en 4 módulos iguales de 2^k celdas.

¿Qué es el Controlador de Memoria ?

En un chip *programable* que está en la mother se encuentra el **"memory control"** junto al puente PCI (fig 1.80) vinculado a la memoria principal DRAM, el cual cumple entre otras las siguientes funciones:

1. Conforme al funcionamiento de una DRAM, divide la dirección emitida por la UCP en dos mitades (para direccionar una fila y una columna de la DRAM, respectivamente) enviándolas en forma sucesiva y en su correcta temporización a la DRAM, junto con las señales de control que la controlan (como RAS y CAS). Asimismo, en relación con la UCP, **avisa cuando la información direccionada está en el bus**, y acepta la que se va escribir en ella.
2. Realiza el manejo de los "bancos" de memoria en que se divide una DRAM, a fin de poder operar en "page mode" y en "interleaving", a los efectos de minimizar el tiempo de acceso cuando se direccionan posiciones sucesivas de la DRAM.
3. Lleva a cabo las tareas de "refresco" por bloques de los microscópicos capacitores que conforman las celdas de los chips DRAM, a fin de reponerles periódicamente su carga eléctrica, que constituye el uno o cero que guarda cada capacitor.
4. Maneja el acceso a la porción ROM (ROMBIOS) de la memoria principal, inclusive la opción de copiar los bytes de la ROM en la DRAM ("shadow") a fin de poder acceder a ellos más rápidamente.

Naturalmente que esta intermediación a cargo de este chip produce retardos en el acceso a la DRAM, en lo referente al contenido de la primera posición direccionada, pero esto resulta compensado con creces si se accede a un conjunto de posiciones sucesivas en la DRAM.
Las acciones de este controlador en gran medida están programadas: durante el booteo del computador una subrutina de la ROMBIOS inicializa registros del mismo, que determinan sus funciones, el cual también depende de lo establecido en el "set-up" de la máquina. (como la opción "shadow").

¿Qué tienen de común y diferente las zonas RAM y ROM de memoria ?

A los fines de evitar esperas, la UCP debe tener rápido acceso a celdas *de cualquier porción de MP*, o sea que no tiene sentido tener celdas de acceso secuencial.
Las primeras computadoras fabricadas comercialmente ya tenían MP con acceso random, o sea eran **Random Acces Memory (RAM)**, en las que cualquier celda puede ser escrita o leída cuantas veces se desee
Memoria **RAM** *es sinónimo de memoria de lectura y escritura* y además de *memoria "volátil"*

Las memorias **volátiles** por sus características físicas pierden la información almacenada cuando se corta el suministro de energía eléctrica al computador. Por lo tanto, la denominación RAM que expresa memoria de acceso "random", se usa también para indicar memoria volátil de lectura y escritura.

En el presente, los chips de memorias RAM que constituyen la MP son del tipo **DRAM** ("Dinamic RAM"). La palabra "**dinámica**" identifica una tecnología que en cada celda de memoria almacena un byte en 8 microscópicos capacitores, que necesitan ser periódicamente recargados eléctricamente, como diminutas baterías, lo cual implica una circulación constante de corrientes eléctricas por el chip.

La cuestión de las denominaciones se complica, desde que, para una porción de la MP, a partir de los años 70 se emplean chips de memoria **ROM**, siglas correspondientes a Read Only Memory (memoria de sólo lectura). Estos chips contienen un programa para el arranque inicial de los computadores (sección 1.15), y los programas del BIOS (ver más abajo)

La ROM es una memoria electrónica de acceso **random**, cuya escritura demanda muchísimo más tiempo que su lectura, pero que tiene la ventaja de ser *"no volátil"*, o sea que almacena la información en forma *permanente*. No necesita energía eléctrica para mantener guardados los datos. Sí para leerlos.

Las memorias ROM también son de "random acces", como se exige que sea una MP, aunque su tiempo de acceso puede ser varias veces más largo que el de las DRAM.

Típicamente la porción ROM de MP de una PC está en uno o varios chips ROM,

Por lo tanto: **MEMORIA PRINCIPAL** es **RAM + ROM**[1]

Las porciones RAM y ROM tienen en común, que al contenido de cualquier posición de las mismas se puede acceder *"al random"*, o sea en igual tiempo, sin búsquedas, indicando un número que la identifica, que es su dirección,, siendo dicho tiempo menor en la RAM. Difieren en que cada posición RAM puede ser leída o escrita cuantas veces sea, siendo su contenido volátil, mientras que se accede al contenido no volátil de una posición ROM sólo para leerla.

¿Qué contiene la porción ROM de memoria principal (ROM BIOS) ?

En una PC la porción de memoria principal que es ROM se denomina ROM BIOS ("Basic Input Output System"). Contiene por un lado programas que se ejecutan al encender un computador y sirven para:

- Verificar el correcto funcionamiento del hardware y su configuración
- Traer del disco a memoria principal (o sea escribir en ésta) una copia de programas del sistema operativo del computador (acción conocida como *"bootear"* o "arrancar" el sistema)

Por otro lado, almacena programas que se usan permanentemente para la transferencia de datos entre periféricos y memoria, sea en operaciones de entrada o salida de datos.

También la ROM BIOS contiene tablas, por ejemplo relativas a características de discos.

¿Qué tipos de memorias "random acces" de semiconductores se fabrican ?

En el siguiente cuadro se clasifican las memorias a que hace mención la pregunta en tres grupos, en relación con la facilidad y rapidez con que puede ser re-escrito cada byte de las mismas.

Memorias de semiconductores con "random acces"			
Clase	**Tipos**	**Características de re-escritura**	**Volatilidad**
Memorias de lectura y escritura	**DRAM** **SRAM** **VRAM**	Cada byte direccionado puede ser re-escrito en igual tiempo que se tarda su lectura, cuantas veces se necesite	Son volátiles
Memorias para ser mayormente leída "Read Mostly Memory" (RMM)	**EPROM** **EEPROM** **Flash ROM**	Una vez que la memoria fue escrita, se puede re-escribir ("reprogramar") un número ilimitado o muy grande de veces, pero la escritura es lenta comparada con el tiempo de lectura de un byte..	No son volátiles
Memorias para ser sólo leídas	**PROM** **ROM**	Una vez que la memoria fue escrita, no se puede re-escribir	No son volátiles

[1] Más específicamente, desde un punto de vista electrónico, la memoria principal es por lo general: **DRAM + EPROM** o también hoy día: **SDRAM + FLASH ROM** (más adelante se tratan las subclases de RAM y ROM)

Las memorias **DRAM**[1] tienen en cada celda un transistor y un capacitor microscópico. El capacitor presen dos estados: cargado guarda un uno, y descargado un cero. Al capacitor, al igual que una pequeña bater hay que reponerle la carga eléctrica que pierde, cada 10-20 milisegundos (acción de "refresco"). Es requiere una constante circulación de corrientes eléctricas hacia las celdas de una DRAM. Por ello se llam memoria *"dinámica"*.

En cambio, cada celda de las memorias **SRAM** (*"Static RAM"*) consta de 4 ó 6 transistores, qu forman un circuito con memoria, conocido como "flip-flop". Este permanece *"estáticamente"* en u estado eléctrico (0) ó en otro estado (1) mientras no se apague el computador (o se ordene pasar d un estado al otro en una escritura de la celda). Al no guardar la SRAM los bits en capacitores, n requiere circulación periódica de corrientes en su interior, como sucede en la DRAM.

Dado que un flip-flop tarda varias veces menos en cambiar de estado que un capacitor, *una SRAM e más rápida que una DRAM*. Pero debido a que es varias veces más cara que una DRAM (por su meno capacidad por chip) no se utilizan chips SRAM en memoria principal, sino en la memoria caché (a tratar Por ejemplo, si una DRAM tiene 60 nanoseg de acceso, una SRAM tiene 20 nanoseg. o menos.

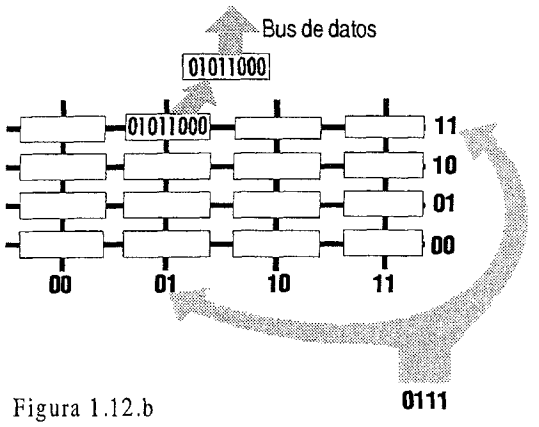


Figura 1.12.b

El esquema de la figura 1.12.b con sólo 16 celdas de 8 bits con 16 direcciones que van de 0000 a 1111, servirá par entender más en detalle una DRAM. Cada celda se localiz en la intersección de una línea horizontal (LH) con otr vertical (LV), y cada línea se selecciona con la *mitad* de lo bits de cada dirección. Si por el bus de direcciones se envía una dirección como 1101, ésta se parte en dos mitades: 11 y 01. Primero los bits 11 seleccionan la LH 11, y luego los bit 01 la LV 01. Así queda seleccionada la celda que suponemo contiene 01011000. Si es una lectura, una copia de 0101100 pasa al registro de salida, y de éste al bus de datos Igualmente, como ser, direcciones de 20 bits se parten en do grupos de 10 bits.

Las nuevas generaciones de DRAM se diseñaron para trabaja rápidamente con transferencias de bloques de bytes.

Dos tipos de DRAM se comenzaron a utilizar buses de 66 Mhz: la **FPM RAM** y la **EDO RAM** (o FPM DRAM y EDO DRAM)

Cada vez que en una DRAM corriente se accede a una celda (con LH y LV), se accede más rápido a la celd siguiente aumentando en uno sólo LV (por ejemplo con LH = 11, LV pasa de 01 a 10). Esto se denomin "Page Mode". En una FPM RAM (Fast Page Mode) luego de direccionar una celda, LV se incrementa en uno automáticamente. Esto es, en FPM se "supone" que luego de acceder a una celda se va a acceder a l siguiente. Para la primer celda, como deben llegar LH y luego LV, se tarda como ser 60 nseg, pero luego en cada acceso siguiente, como sólo debe llegar LV se tarda < 25 nseg en obtener el contenido. Esta forma d acceso, a un bloque de celdas consecutivas, se conoce como ***modo "burst"*** (ráfaga), siendo que en el presen basta dar la primer dirección y la longitud de la ráfaga, no siendo necesario perder tiempo generand direcciones consecutivas.

Una **EDO RAM** opera parecida a la FPM RAM, pero una vez que al registro de salida llega el dato, l mantiene (de ahí su nombre: "Extended Data Output") para que no se pierda, dado que entonce también a la par comienza el acceso a la celda que está en la siguiente LV, como en las FPM RAM. Est paralelismo interno que se debe a la existencia de registros intermedios, en general se conoce en hardware como *"pipe line"* o *"segmentación"*.

De concepción semejante es la pipe line Burst EDO RAM (**BEDO RAM**).

Las **EDRAM** (Enhanced DRAM, o sea DRAM mejorada) o **CDRAM** (Caché DRAM), integran en el chip un cache SRAM que almacena los contenidos completos de la última línea leída. El refresco se realiza a la par que se lee dicho caché, con lo cual éste no afecta tanto al acceso a un chip para ser leído o escrito

Las **RDRAM** de Rambus® son una innovación basada en un bus corto y rápido (500 Mbps contra 33 Mbps de una DRAM corriente) al que pueden conectarse más de 300 chips. En cada acceso sólo se indica

[1] Chips de memoria DRAM conforman los denominados módulos de memoria **SIMM** (Single In-line Memory Module), y en módulos **DIMM** (Dual In Line Memory Module), que son pequeñas plaquetas con chips DRAM, que se insertan a la "motherboard" a través de unos zócalos (figuras 1.8 y 1.5).

la primer dirección, el número de bytes, y si es lectura o escritura.
La **SDRAM** (Synchronous DRAM) intercambia datos con el caché o la UCP al ritmo de una señal de reloj externa relacionada con los Mhz de la UCP, operando así con sus entradas y salidas en sincronismo con ella. Así se evitan retardos y esperas. Si en un determinado ciclo del reloj es direccionada, luego de una cantidad fija de ciclos se obtiene el dato.
También existen las **ESDRAM** (Enhanced DRAM) con un caché SRAM incorporado
Las **DDR SDRAM** (Double Data Rate SDRAM) realizan transacciones en ambos flancos del reloj.
En los chips SDRAM figura un número dado en nseg. (por ej. 10 nseg.), que **no** se refiere al tiempo de acceso, sino a la duración del periodo del reloj que se puede aplicar (o sea 100 Mhz para 10 nseg)
Con un bus de 100 Mhz (100 millones ciclos/seg) que tenga 64 líneas, se pueden transferir 8 bytes en cada ciclo, o sea 64 Bytes/ciclo x 100 millones ciclos/seg = 64 Mbytes/seg.

Entre las **memorias estáticas** (**SRAM**) muy usadas en los cachés externos (4 nseg < tacc < 20 nseg) encontramos:
La SRAM asincrónica (**Async SRAM**): en el momento que pone un dato en las líneas de datos, activa una línea que va a la UCP. Es la más lenta de las SRAM para caché externo.
La SRAM sincrónica rápida (**Sync burst SRAM**), que funciona en sincronismo con los pulsos reloj de la UCP. Así, cada dato leído llega a la UCP en sincronismo con un pulso reloj.
La "pipe lined burst" SRAM (**PB SRAM**): merced a sus registros de entrada y salida, una vez que ellos están cargados, permiten que mientras está presente en las salidas el contenido de una celda direccionada, se realicen accesos a celdas de direcciones siguientes ("pipe line"). Para buses de hasta 133 Mhz

Los chips de memoria de vídeo **VRAM** (Video RAM) forman parte de la memoria principal, pero están en la plaqueta de vídeo (fig 1.65). Guardan la información que sale por pantalla. Estos chips tienen casi el doble de patas que los de cualquier RAM. Esto se debe a que una VRAM tiene un bus de direcciones y datos para ser escritos por la UCP, y otro bus para direcciones y datos para ser leídos por los circuitos de la plaqueta de vídeo, que manejan el monitor. Por eso se dice que la VRAM es una memoria de "dos puertas", dado que *simultáneamente* que es escrita por la UCP, puede ser leída por la plaqueta.
También existen **EDOVRAM**, semejantes a las subclases DRAM antes citadas.
Otras RAM para las plaquetas de vídeo son: la **WRAM** (Window RAM) que acelera la generación de gráficos, dado que ha sido pensada para anticipar las operaciones típicas que se realizan en vídeo; y la **3D RAM**, especialmente creadas para manejar gráficos en tres dimensiones. Contienen varias UAL en su interior.

Los primeros chips ROM (hoy denominados **ROM** *"fabricados a medida"* se encargaban a los fabricantes de chips, indicándole el contenido que debía tener cada celda. Entonces, cuando se fabricaba cada chip, se grababan los contenidos que siempre tendría. La adquisición de estos chips hoy sólo se justifica económicamente si se encargan decenas de miles de chips iguales.

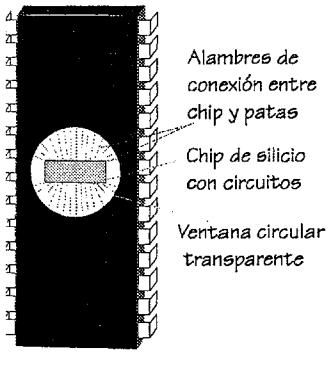


Figura 1.13

A diferencia, los chips **PROM** ("Programmable ROM") se fabrican en serie, pero el interior de cada chip está preparado para que –en una segunda etapa– quién utilice uno o miles de estos chips pueda escribir una sola vez los contenidos que tendrán sus celdas. Esto se realiza en un dispositivo electrónico que se vende para tal fin. Una vez así grabada ("programada"), una PROM no puede ser re-escrita.
Las ROM de la clase **RMM**, para ser mayormente leídas, son reprogramables, o sea se pueden volver a escribir, siendo que *la escritura es poco frecuente frente a las operaciones corrientes de lectura.*
Un tipo de RMM aún usada para ROM BIOS en las PC es la **EPROM** ("Erasable-Programmable ROM"). Los pastillas EPROM se carac-terizan por presentar una ventanilla transparente en su cara superior (fig. 1.13), que normalmente está cubierta por una etiqueta plateada autoadhesiva
Se puede cambiar el contenido de todas las celdas, re-escribiendo el nuevo contenido que tendrá cada una en un dispositivo semejante al usado para las PROM. Previo a ello se debe "borrar" (*"erase"*) el contenido de *todas* las celdas[1]. Esto se logra haciendo pasar luz U.V por la ventana durante unos 15 minutos.
Una **EEPROM**, o **E²PROM** ("Electrically Erasable ROM")[2] no requiere el borrado de todas las celdas con luz U.V, dado que –como en una RAM– se puede seleccionar cada dirección que se quiere re-escribir., sin sacar la pastilla del circuito donde opera. Esto puede hacerse unas 10.000 veces.

---

1 Obsérvese al respecto, que este proceso de borrado no es necesario en una RAM, pues una escritura borra un contenido anterior.

2 A veces identificada con las denominadas **EAROM** ("Electrically Alterable ROM")

Las "flash" **ROM** son un tipo de EEPROM mejorado. En una fracción de segundo se bor eléctricamente por bloques, y luego se re-escriben las celdas consecutivas de un bloque. Este tipo ROM en el presente se usa en reemplazo de disqueteras en "notebooks", y como parte ROM de memoria principal. Así puede actualizarse el BIOS con un programa, sin sacar el chip de la mother. Una **FRAM** (Ferroelectric RAM) es una memoria no volátil que incorpora hierro magnetizable en su chip.

¿Qué es capacidad de memoria, y qué son las unidades KB, MB, GB ?

Capacidad (de almacenamiento) de una memoria es la cantidad total de bytes que puede guarda En el presente, lo común es que en cada celda se guarde un byte de información. Entonces una memor con **N** celdas tendrá una capacidad de **N** bytes. Dicho número es siempre una potencia de dos: $N = 2^k$ El número de celdas (bytes) de una memoria se puede expresar en kilobytes (KB), Megabytes (MB Gigabytes (GB), Terabytes (TB) . . . según sea.

Un grupo de 2^{10} = 1024 celdas sucesivas, o sea 1024 Bytes se denomina **1 KByte (KB)** por ser 102 un valor cercano a 1000 ("kilo"). Por lo tanto **1024 B = 1 KB**

Dado por ej. una memoria de 2^{16} = 65536 salidas, el cociente 65536/1024 = 64 indica que en 6553 entran 64 grupos de 1024, o sea 64 grupos de 1 KB, por lo que se dice es una memoria de 64 KByte

El múltiplo siguiente a 1 KB es 1024 veces mayor: **1 MB** = 1024 x 1 KB = 1024 x 1024 bytes 1.048.576 bytes = 2^{10} x 2^{10} = 2^{20}, siendo 2^{20} la potencia de dos más cercana a 1.000.000 (1 "mega"). Así, para una memoria de 2^{22} = 4.194.394 celdas (bytes), el cociente 4.194.394/1.048.572 = $2^{22}/2^{20}$ = 2^2 = 4 indica que en 4.194.394 entran 4 grupos de 1.048.576; se dice que es una memoria de 4 MB.

Así evitamos manejar números como 4.194.394, redondeando en forma práctica hacia abajo.
Siguen los múltiplos **1 GB** = 1024 x 1MB = 1024x1.048.576 bytes (supera a mil millones = "giga")
1 TB = 1024 x 1GB (supera el millón de millones = "tera").

Conforme a estas unidades, la siguiente tabla de potencias de dos es útil para fijar y generaliza conceptos, siendo que la columna con las unidades en bytes, KB y MB es la que se acostumbra a usar

2^0	=	1	byte	=	1	byte	
2^1	=	2	bytes	=	2	bytes	
2^2	=	4	bytes	=	4	bytes	
2^3	=	8	bytes	=	8	bytes	
2^4	=	16	bytes	=	16	bytes	
2^5	=	32	bytes	=	32	bytes	
2^6	=	64	bytes	=	64	bytes	
2^7	=	128	bytes	=	128	bytes	
2^8	=	256	bytes	=	256	bytes	
2^9	=	512	bytes	=	512	bytes	
2^{10}	=	1024	bytes	=	1	KB	= 1024/1024
2^{11}	=	2048	bytes	=	2	KB	= 2048/1024
2^{12}	=	4096	bytes	=	4	KB	= 4096/1024
2^{13}	=	8192	bytes	=	8	KB	= 8192/1024
2^{14}	=	16384	bytes	=	16	KB	= 16384/1024
2^{15}	=	32768	bytes	=	32	KB	= 32768/1024
2^{16}	=	65536	bytes	=	64	KB	= 65536/1024
2^{17}	=	131072	bytes	=	128	KB	= 131072/1024
2^{18}	=	262144	bytes	=	256	KB	= 262144/1024
2^{19}	=	524288	bytes	=	512	KB	= 524288/1024
2^{20}	=	1048576	bytes	=	1024	KB = 1 MB	= 1048576/1048
2^{21}	=	2097152	bytes	=	2	MB	= 2097152/1048576
2^{22}	=	4194394	bytes	=	4	MB	= 4194394/1048576

. . . . . .

Figura 1.14

Las unidades elegidas permiten compara rápidamente el tamaño de una memori respecto a otra, y hacen más simple l expresión de la capacidad.
Así, en vez de decir "la memoria es d 1048576 bytes, y voy a pasar a una d 4194394 bytes, con lo cual la capacida aumentará 4194394/1048576 = 4 veces", es más práctico decir "la memoria e de 1 MB y voy a pasar a otra de 4 MB con lo cual la capacidad aumentará 4 veces"
Se debe tener presente que para pasa de bytes a KB se debe dividir por 102 Si un programa ocupe 8000 posicione de memoria de 1 byte. No son 8 KB. Exactamente son: 8000/1024 = 7,8 KB En una PC cuando se habla de un memoria de por ejemplo 8 MB d capacidad, se asume que se trata de l porción DRAM de la misma, que está en los módulos de tecnología SIMM o DIMM

¿Qué relación existe entre la capacidad de una memoria, la cantidad de bits que tienen sus direcciones y el número de líneas de dirección ?

En una memoria que guarda un byte por posición, debe haber tantas posiciones como bytes tenga la memoria. Esto es, una memoria de 2 MB implica que se deben identificar 2097152 posiciones distintas, asignándole un número binario distinto para cada una, que es su dirección. Se acostumbra a asignar el cero a la primer posición, por lo cual la última sería en decimal el número 2097151.
Una memoria tiene un número N de celdas que siempre es una potencia de dos, por lo cual las mismas se localizan mediante direcciones que son números binarios que van de 000....000 hasta 111....111, siendo que a cada celda le corresponde uno de esos números como dirección.

Las potencias de dos de la figura 1.14 sirven para determinar cuántos bits deben tener los números bina rios que son las direcciones de una memoria, de forma de adjudicar un número distinto a cada posición. De la figura 1.4 resulta que con 4 bits se pueden formar 16 = 2^4 combinaciones binarios (o sea números binarios) distintas. Inversamente, si se quiere formar 16 números binarios distintos hacen falta 4 bits.
Del mismo modo, por ejemplo, dado que 2 MB = 2^{21}, resulta que *el exponente 21 indica la cantidad de bits que debe tener cada dirección,* para formar 2.097.152 de direcciones distintas, como se necesita.
Por ejemplo, una dirección de 21 bits sería 0 1001 0111 1001 1010 1110 = 0979AE en hexa, según lo visto.
La primer dirección de memoria sería un número binario con 21 ceros = 0 0000 0000 0000 0000 0000 = 00000 en hexa; y la última otro de 21 unos = 1 1111 1111 1111 1111 1111 = 1FFFFF en hexa.

De la tabla resulta que al subir uno el exponente del número dos, se duplica la cantidad de memoria. O sea, si una memoria tiene el doble de capacidad que otra, los números que forman las direcciones de la primera deben tener un bit más que los números que forman las direcciones de la segunda.

Si se quiere operar mentalmente, se debe tener presente que **2^{10} = 1 KB,**
y que **$2^{20} = 2^{10} \times 2^{10}$ = 1 KB x 1KB = 1024 x 1024 = 1 MB.**
A fin de calcular cuántos bits tienen las direcciones de una memoria de 8 MB, partiendo de que 1 MB necesita 20, que 2 MB necesitan 21, que 4 MB necesitan 22, resulta que 8 MB necesitan 23 bits.
Puesto que por cada línea de direcciones de un bus transmite un bit, en principio deben utilizarse al menos tantas líneas de dirección como cantidad de bits tenga cada dirección.

¿Qué es el bit de paridad en memoria principal, y para qué se emplea ?

Supongamos que el contenido de una posición de memoria leída es 01000001, pero que por un ruido (interferencia electromagnética) durante la operación de lectura, la UCP recibe 01000011, o sea que el bit marcado se recibe invertido. Entonces la combinación recibida será otra, sin que se pueda notar el error.
Como se verificará, *ampliando la capacidad de una memoria de forma que cada posición contenga un bit extra* (o sea 9 bits), *puede detectarse si se ha producido un solo error por inversión,* como el ejemplificado.
El bit denominado de **"paridad"**, *que se agrega al byte a almacenar,* debe ser siempre de valor tal que el conjunto de los **9** bits almacenados tenga un número par de unos (paridad par de unos).
En el momento en que se escribe cualquier posición de memoria un circuito calcula el valor (**0** ó **1**) que debe tener dicho bit; y cuando se lee cualquier posición, el mismo circuito verifica que los 9 bits leídos presenten paridad par de unos. Caso contrario se interrumpe el programa en curso para avisar que hay un problema de error de paridad en memoria.
En una memoria con paridad, cuando por ejemplo se escribe la combinación 01000001 antes citada, dicho circuito le agregará un bit de valor 0 (en negrita): 010000010[1]. Así el nuevo conjunto de 9 bit. almacenado presentará un número par de unos.
De esta forma, si durante la lectura en lugar de leerse 010000010, se lee 010000110 por que el bit subrayado se invirtió, se detectará un error, dado que estos 9 bits tendrán un número impar de unos, en lugar de ser par Por lo tanto, esta convención –de igual paridad en la escritura y lectura– sirve para detectar si *uno* solo de los bits recibidos cambió de valor, que es la mayor probabilidad de errores en una lectura de memoria.
Si los bits errados son dos, la paridad par seguirá, y no hay forma de detectar una combinación mal recibida.

---

[1] Para todas las combinaciones de 8 bits a escribir que tengan un número par de unos, el bit de paridad será **0**, mientras que las que tengan un número impar de unos, dicho bit valdrá **1**. Por ejemplo 01101101 se almacenará en memoria como 011011011.
ECC (Error Correction Code) usado en servers, usando paridad no sólo permite detectar bits errados, sino también corregirlos.

¿Qué es un microprocesador de 8, 16 ó 32 bits y qué relación tiene ello co los registros, y la memoria principal ?

Anteriormente se distinguía entre procesadores de 8, 16 y 32 bits -según sea- para indicar el número máxi de bits que puede tener un dato a operar por la UAL de cada uno. Desde el 80386 en adelante y hasta Pentium 4 pueden operar en su UAL dos números de 32 bits, siendo que en correspondencia sus registros uso general pueden almacenar 32 bits. Lo mismo ocurre con la mayoría de los microprocesadores de distint fabricantes, por lo que el número 8, 16 ó 32 que caracterizaba la potencia y por ende la velocidad procesamiento de un procesador, ha dejado de usarse como distintivo de potencia.

Una UAL de 32 bits puede operar con número del doble de magnitud que otra de 16 bits. Asimismo, si l registros para datos y resultados son de 32 bits, a los efectos de una mayor velocidad de transferencia entre UCP y memoria, deben existir por lo menos 32 líneas de datos en el bus. De esta forma, de una sola vez transfieren 32 bits entre memoria y UCP o viceversa. Esto también implica que la memoria debe est organizada para que *en un solo acceso* puedan leerse o escribirse 32 bits (4 bytes consecutivos), dando sólo dirección del primero.

En consonancia con ello, *los registros de la UCP para datos o resultados deben tener una longitu igual al tamaño de los datos que opera la UAL.*

Por ejemplo, el Pentium 1[1] es un procesador de 32 bits[2], con una longitud de palabra de 64 bits. O sea qu en cada acceso a MP puede leer/escribir 8 bytes consecutivos, para lo cual se debe indicar sólo la dire ción del primero de ellos. A tal efecto, este microprocesador tiene 64 patas destinadas a conectarse a 6 líneas de datos que forman parte del bus ("local bus") que lo conecta con el caché externo y con la MP.

Por lo tanto, si bien la UAL del Pentium opera con 32 bits (y con 64 bits su coprocesado matemático incorporado), pueden llegar al mismo datos o instrucciones de a 64 bits por vez[3], l cual acelera el procesamiento de los datos.

También son procesadores de 32 bits el 386 y el 486, pero operan la MP con palabras de 32 bits[4].

Se comprende que un procesador de mayor número de bits sea más rápido que otro que opere con menos pues en una sola instrucción puede manipular más bits. Por ejemplo, si un 286 (procesador de 16 bits) quier sumar dos números de 32 bits cada uno, primero mediante varias instrucciones debe sumar los primeros 16 bits d ambos números, y luego mediante otras tantas instrucciones sumar los 16 bits restantes.[5]

¿Puede decirse que los registros de la UCP conforman una pequeña RAM ?

Con relación a una pregunta anterior, se vio que se podía considerar una RAM como constituida po registros de un byte. Así, una memoria de 1 MB puede considerarse formada por 1.048.576 registros de un byte, aunque en general no se acostumbra a llamar registros a las posiciones de memoria.

Un registro de una UCP, por ejemplo de 32 bits, al igual que una posición de memoria, se puede considerar constituido por 32 llaves "si-no", que operan como las 8 llaves de la figura 1.9).

Asimismo, los registros de cualquier UCP (cuyo número en general es 8, 16, 32) se identifican por un número interno que oficia de dirección dentro de la UCP (a partir de cero, como una pequeña memoria). Dicho número es el que debe figurar en el código de una instrucción que hace mención a un determinado registro de la UCP.

Vale decir, que por ejemplo, los registros de un 80x86 designados simbólicamente AX, BX, CX,..., como circuitos de la UCP tienen un número binario que permite localizarlos, de la misma forma que se localiza una posición de memoria por su dirección, siendo su acceso del orden del nanoseg.

Entonces, salvo aspectos cuantitativos referidos a la cantidad de registros de una UCP, y al número de bytes que guardan los mismos (4 bytes en un 386, 486 o Pentium), *los registros de la UCP constituyen una pequeña RAM*, siendo que cada registro se localiza al random mediante un número, para ser leído o escrito.

---

1 Al igual que el P6 y la mayoría de los microprocesadores tipo RISC

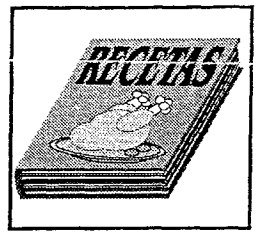
2 Intel define para todos sus procesadores un word como 16 bits, y como "doble word" un grupo de 32 bits (palabra doble).

3 La memoria interna para instrucciones (caché), puede ser leída de a 32 bytes = 256 bits por vez (unas diez instrucciones juntas)

4 El modelo 386SX lee la memoria sólo de 16 bits por vez, como el 286. La UAL del 286 también opera con 16 bits.

5 Asimismo, es factible que un procesador necesite operar en su UAL un dato cuyo tamaño sea *menor* que los bits que procesa (por ejemplo de 8 ó 16 bits, si procesa 32). En consonancia debe poder ordenar leer o escribir un número acorde de posiciones de memoria. Por ejemplo, un Pentium o un 3/486 pueden sumar dos números de 8 (ó 16) bits cada uno, y ordenar acceder a una (o dos para 16) posición(es) de memoria (8 es la mínima cantidad de bits que se pueden leer o escribir en memoria). Esto está pensado para que los nuevos modelos de procesadores sean compatibles con los anteriores, de modo de poder ejecutar los programas ya desarrollados para estos últimos.

1.5 EL SOFTWARE, LOS DATOS, Y SU CODIFICACION



¿Qué es el software o "logical" ?

Además de la velocidad y confiabilidad, no cabe duda que *la versatilidad* es otra de las características relevantes de las computadoras. A partir de su uso inicial como calculadoras automáticas, poco a poco han ido invadiendo distintas áreas de aplicación: científica, industrial, administración pública, apoyo profesional, entretenimientos, modelación, enseñanza, etc. (En el Apéndice se hace una reseña de estas aplicaciones).
Esta versatilidad es también observable en muchas herramientas que usamos, incluidas las primeras: nuestras manos. Pensemos también en los múltiples usos que puede tener un simple cuchillo, según la forma de tomarlo, moverlo, porción de filo usada, etc. En el conjunto cuerpo humano-cuchillo, el cambio de plan de uso se *realiza fácil y rápidamente* en nuestro cerebro, sin modificación física visible.
Otra actividad o proceso cotidiano, útil para comprender los conceptos de hardware y software, es la elaboración de comida. En ella distinguimos:

1. **Una parte fija, estable y tangible**, constituida por una persona, la cocina y sus utensilios, que serán los medios físicos *("hardware")* con que se cuenta para llevar a cabo la elaboración.

2. **La receta o procedimiento**, que depende del plato deseado. Ella instruye ordenadamente cómo usar los medios físicos en cada paso de la elaboración de los ingredientes. Es el plan lógico *("software")*.

Nos encontramos con un tipo de proceso versátil, con múltiples resultados (platos distintos), llevado a cabo mediante un mismo equipamiento físico, y un sinnúmero de planes lógicos de elaboración posibles. Éstos son fáciles de cambiar en nuestra mente. Están pensados en función de los medios disponibles, y están limitados a los mismos. Por ejemplo las recetas de una cocina a gas no sirven para un asador al aire libre.
Asimismo, *con el equipamiento físico solo no se puede hacer nada, si falta el plan lógico correspondiente.*

Un computador no sabe cómo procesar datos. Se le debe indicar mediante instrucciones: *dónde están los datos a procesar, qué procesamiento realizar con ellos, y hacia que medio irán los resultados obtenidos*

Hacia 1946 en la Universidad de Pennsylvania funcionaba el prototipo de computador ENIAC descripto en el apéndice A3 de esta unidad. El mismo, una vez programado realizaba velozmente una serie de cálculos que se necesitaba efectuar en forma repetida, en cada oportunidad con datos distintos. Si hacía falta realizar otros cálculos distintos había que *reprogramarla cambiando el cableado de un panel*, lo cual podía insumir horas. El matemático de esa universidad, John von Neumann, para solucionar esto planteó la necesidad de almacenar en una sola memoria interna electrónica de rápido acceso (la memoria principal), por un lado el *programa* a ejecutar, y por otro los *datos* a procesar. Este es uno de los postulados del llamado "**modelo de Von Neumanan**" (Ver apéndice A3 citado)
De esta forma, de poderse cambiar rápidamente en dicha memoria el programa a ejecutar, se podrían realizar nuevos cálculos, sin necesidad de modificar el conexionado de circuitos de la máquina, con la pérdida de tiempo que ello implica.
Un programa (software) almacenado en la memoria principal de un computador, si bien está registrado en ella, no forma parte material de los circuitos que la constituyen (hardware)
Estar registrado significa que esos circuitos quedarán en determinados estados eléctricos en su interior, que representarán al programa citado, que en la figura 1.9 hemos asimilado a posiciones de llaves "si-no"
Estos estados pueden hacerse equivaler a una determinada configuración espacial de "unos" y "ceros" en la memoria, que se mantendrá mientras no se ordene memorizar otro programa en reemplazo del existente.
De ser así resultará otra configuración interna de estados eléctricos, que corresponderán a los "unos" y "ceros" del nuevo programa.

Entonces, **cuando se cambia el software no hay cambio material de componente alguno d computador: el hardware permanece invariable. Sólo se modifica el estado eléctrico d material semiconductor que compone los circuitos de la memoria**

Han ocurrido cambios energéticos localizados; la materia sigue constante. En el caso de las llav "si-no" citadas, en cada una se habrá estirado o encogido un resorte por la acción de nuestra man pero no hay cambios materiales.

Por esta manera fácil y veloz de cambiar de programa en el interior de un computador, és resulta una herramienta tan versátil para efectuar cualquier proceso de datos. Cambiando programa se modificará el comportamiento de la máquina.

Una computadora para aplicaciones comerciales puede convertirse en una de tipo científic utilizando un programa adecuado. Se modifica el procedimiento sin cambiar de equipo.
En todos los casos descriptos nos encontramos con el aprovechamiento *múltiple* de un *mism* equipamiento físico (**hardware**), como si cada procedimiento programado que se ordena realiza (**software**) generara un dispositivo físico *a la medida* del proceso que ordena ese plan.

Conforme con lo anterior, resulta que *el hardware provee funciones procesadoras, que son activad por cada una de las instrucciones de un programa ejecutado según ellas.*

Cada marca y modelo de computador *tiene un repertorio definido de códigos de instrucciones qu puede ejecutar,* acorde a las posibilidades funcionales de su hardware.
No acepta instrucciones que no pertenezcan al mismo. Una misma instrucción, como ser sumar, tien distinto código en dos procesadores distintos.

Software de un computador, es cualquier programa[1] que puede ser almacenado total o parcial mente en su memoria principal, para ser ejecutado por el procesador de dicho computador.

Su nombre hace referencia al hecho de que los programas son materia dúctil, "blanda" (*"soft"* e inglés). Esto es, *los programas son fáciles de modificar, y de cambiar unos por otros en la memoria principa* de un computador, para que éste, siendo hardware fijo, sea una herramienta de múltiples usos.
Dicha facilidad se debe a que *los programas no forman, físicamente, parte del hardware, sino que és les sirve de soporte material.* Únicamente se modifica el estado eléctrico de los circuitos de la memori principal, mediante señales eléctricas "transparentes" al operador o al proceso.
El software también puede registrarse en cintas, discos magnéticos u otros medios.

La *esencia* del software *son los algoritmos*[2] *que él expresa, representa.* Ellos son *expresados* mediant programas usando determinados lenguajes. Cada programa que puede ejecutarse en un computado –al ser almacenado en memoria, y luego ejecutado– permite llevar a cabo un cierto proceso de datos.
Debido a esta relación del software con algoritmos, es que a veces se lo denomina "**logical**".

Un computador con todos sus circuitos electrónicos energizados, pero sin ningún programa e memoria principal, no puede procesar datos. *No sabe qué hacer.* Es sólo "puro hardware"
Cada instrucción que es ejecutada por la UC da lugar a una secuencia predeterminada de movimiento de datos, y operaciones en la UAL, que se desarrollan con los pulsos reloj. O sea *que cada código de ins trucción determina implícitamente qué partes del hardware se activarán y en que orden,* según se verá.

Por lo tanto, puede afirmarse que *"el software controla al hardware"*. En ese sentido cuando de una se cuencia de instrucciones se salta a otra distinta, correspondiente a una subrutina o a otro programa, se habl de *"transferencia de control"*, vale decir que el control se ha transferido a la nueva secuencia en ejecución.

En el apéndice A.2 se encontrará una clasificación del software, donde se incluyen la función de un siste ma operativo y de programas utilitarios, así como el software constituido por los denominados "virus".

---

1 Sean del sistema operativo, del usuario (procesadores de texto, base de dato, traductores de lenguajes, de juegos), de diagnóstico, etc.

2 Un *algoritmo* es un procedimiento que asegura, mediante un número finito de pasos, una salida requerida, a partir de una entrad dada, independientemente del tiempo en que se realiza. Las conocidas reglas aprendidas en la escuela primaria para realizar manualmente las cuatro operaciones aritméticas, son algoritmos.

¿Qué es el firmware ?

Un computador encendido sin ningún programa en memoria principal (MP) no puede hacer nada.
Los programas que residen en la porción RAM de MP *desaparecen cuando se apaga un equipo.*
Por ello cada vez que éste se enciende, hay que traer del disco a memoria una copia del sistema operativo (SO). Esta acción se conoce como *"arranque"* o *"boot"* o **"buteo"** (ver detalle en la sección 1.15)
Todavía en la década del 70, luego del encendido había que escribir en MP un pequeño programa en forma manual, mediante llaves "si-no" dispuestas en el panel frontal del computador. Al ser ejecutado este programa se traía a MP, del disco o cinta, programas constituyentes del SO.
En el presente, al encender un computador el arranque es automático, sin intervención manual, merced a que está almacenado en la porción ROM de MP un primer programa, que permite traer a MP los programas del SO archivados en un disco.
También están en esta ROM programas de diagnóstico (que verifican el correcto funcionamiento y configuración del hardware antes de traer el SO), y programas que son invocados cada vez que se necesita realizar una entrada/salida, constituyentes del BIOS (Basic Input Output System).
Se trata, pues, de software que está *permanentemente fijo* en el hardware, o sea que una vez que un programa o varios se han escrito en la porción ROM de MP, permanecen *siempre* almacenados en MP. Así, estos programas se conservan inalterados aunque se corte la energía del equipo, por ser la ROM "no volátil", listos para ser usados toda vez que sea necesario si el equipo está encendido.
No necesitan ser traídos de un disco para ser re-escritos en MP. Recordemos que una ROM es también una memoria "random" como una RAM, con tiempo de acceso 3 a 5 veces mayor que ésta.
Además de programas, una ROM se usa para conservar en forma permanente tablas de datos y constantes.

> En forma extensiva, se denomina ***firmware*** al software (programas) almacenado permanentemente en el hardware constituido por una memoria ROM *soportada por circuitos electrónicos.*

Decimos en forma extensiva, por que la palabra "firmware" se acuñó en relación con la ROM que existe en la UC de los procesadores "CISC"[1], como los 80x86 de Intel. Según se verá (sección 1.7), en esta ROM están guardadas las combinaciones binarias ("micro-códigos") que le indican a la UC la secuencia de acciones necesarias para ejecutar cada instrucción. Ellas constituyen la "inteligencia" de un procesador.
Un programa contenido en un disco CD ROM no es firmware, pues no es una ROM circuital.

¿Qué es un microprocesador dedicado ?

El microprocesador (80x86, Pentium, Power PC, 88000, etc.) que caracteriza un computador, es su procesador "central". La memoria de la cual lee las instrucciones a procesar, es *predominantemente RAM*, si bien tiene una pequeña porción ROM para el arranque y el BIOS. Esto es así por que en un computador para uso general debe poderse cambiar el software residente en memoria principal, según las necesidades del usuario (programas para procesar texto, para planillas de cálculo, para base de datos, para juegos, etc.).

Un sistema de computación también presenta "**procesadores dedicados**" y **microcontroladores** (éstos en el chip también contienen memoria principal y registros "ports". *Son como una computadora en un solo chip*).
Dada la complejidad de los periféricos actuales, su electrónica requiere de un microprocesador para controlar las operaciones que realiza, siendo que su costo en general es mucho menor que el del procesador central citado
Encontramos procesadores dedicados en el teclado, en una impresora láser, en la plaqueta de un módem, en la electrónica de una unidad de disco rígido, entre otros.
Puesto que *un procesador dedicado se caracteriza por ejecutar siempre los mismos programas*[2], éstos se encuentran residentes en algún tipo de memoria ROM *(firmware)* que oficiará de memoria principal del procesador dedicado. No debe confundirse con la memoria principal del procesador central. En general, para este último, los procesadores dedicados y los programas de los mismos le son "transparentes".
Se comprende que un procesador dedicado necesita que sólo una pequeña porción de su memoria principal sea RAM. Esta puede no necesitarse si los datos que procesa y los resultados que obtiene entran en los registros del procesador dedicado, en cuyo caso la memoria principal del mismo sería totalmente ROM.

---

1 Siglas de Complex Instruction Set Computer, o sea computador con instrucciones complejas en su repertorio de instrucciones.
2 Por lo cual, dada su sencillez, en general no requiere de un sistema operativo.

¿Cómo se prepara el proceso de datos en el computador antes definido, cómo se le ordena a éste qué debe hacer ?

En cualquier proceso de datos con computador, es indispensable escribir en memoria los datos las instrucciones del programa, antes de comenzar a ejecutar éstas. El programa Debug perm realizar fácilmente esto por medio del teclado[1], como se describe en la próxima pregunta.

Es **esencial** releer paso a paso la explicación relacionada con las figuras 1.2 y 1.3, pues sirve de base pa comprender cómo funciona un computador, y para experimentarlo en una PC, así como para los temas qu siguen. Al final del Apéndice A1 se da un ejercicio integrador basado en el ejemplo aquí desarrollado.

El proceso de datos a codificar, que será luego llevado a cabo en una PC es semejante al proceso manual que ilust la figura 1.2.. Dicho proceso estaba descrito por la secuencia de instrucciones $\mathbf{I_1}$ a $\mathbf{I_5}$. Por simplicidad didáctic codificaremos y ejecutaremos las 4 primeras, o sea que efectuaremos $\mathbf{R = P + P - Q}$
Supondremos que los datos son **P** = 1020H y **Q** = 2040H[2].

1. ***Codificación de los datos:***
Primero escribiremos en memoria estos datos de dos bytes que en binario serían: son 0001 0000 0010 0000 = 1020H y 0010 0000 0100 0000 = 2040H, conforme a las equivalencias dadas por la tabla de la figura 1.4, detallad en el Apéndice 1. Dado que cada celda de memoria guarda un byte, **P** y **Q** deben ocupar dos celdas sucesivas cada uno[3]
Se han asignado *arbitrariamente* las direcciones *5000*H y *5001*H[4] a **P**, y *5006*H y *5007*H[5] a **Q** (en la fig 1.15 se supon escritas **P** y **Q**). Nótese que primero (en la dirección más baja (*5000*H para **P** y *5006*H para **Q**) se escribe en la UCP de Intel™ la mitad derecha de cada número (20 y 40 respectivamente). O sea **XXYY** se escribe **YYXX.**
Al resultado **R** de hacer $\mathbf{P + P - Q}$ le asignaremos direcciones *5010* y *5011*, asumiendo que ocupará 2 bytes.
Las direcciones también están escritas en hexa, aunque *dentro de un computador sólo pueden existir unos y ceros.*

La dirección de un dato puede ser elegida arbitrariamente, siempre que dicha dirección sea la que forme parte de la instrucción que va a operar dicho dato, puesto que una instrucción debe indicar cómo localizar el dato a operar.

2. ***Codificación de las instrucciones en código de máquina:***
Habiendo establecido que el dato 1020H ocupará las direcciones *5000*H y *5001*H, y que el dato 2040H estará en binario en *5006*H y *5007*H, ahora se pueden codificar y escribir las instrucciones, puesto que como veremos, *para codificarlas se requiere primero conocer las direcciones* ***elegidas*** *donde se han guardado los datos.*

La combinación binaria que codifica una instrucción constituye su **"código de instrucción"** o **"código de máquina"**[6], en el sentido que es el código que "entiende" la máquina, o sea los circuitos de la UC. **Cada procesador tiene sus propios códigos de máquina que puede ejecutar.**

El código de máquina comprende necesariamente el **código de la operación** (cod-op) ordenada (transferir, sumar, restar, comparar, dividir, saltar a otra instrucción ...). Si el dato a operar está en memoria, **al cod-op le sigue una dirección**[7] que indica dónde éste se encuentra (o dónde memorizar un resultado que está en un registro de UCP, si se ordena una escritura). **No confundir el valor de un dato con el valor de su dirección.**

---

[1] En la práctica esto sólo ocurre cuando se desarrollan o modifican programas, dado que el sistema operativo se encarga de manejar la escritura rápida en memoria principal de las instrucciones de los programas a ejecutar, que trae del disco. Así, cuando desde Windows llamamos a un procesador de texto, este programa en pocos instantes está en memoria, esperando qué módulo del mismo se va a ejecutar, de acuerdo con nuestros requerimientos. Los datos (letras, palabras) van entrando a memoria a través del teclado, a una velocidad que depende del usuario, siendo también ubicados en la zona de memoria que se les ha asignado mediante la ejecución de subrutinas.

[2] Sumar 1020 + 1020 resulta 2040 tanto en decimal como en hexa (H), pero como se indicó anteriormente, la cantidad de unidades que representan los sumandos y el resultado son distintas en ambos sistemas de numeración. Se han elegido ex profeso, por razones didácticas, los datos indicados, para evitar resultados que recién pueden interpretarse en conocimiento el Apéndice 1. Se ha previsto que el resultado sea cero.

[3] **Cuando en un computador un dato o una instrucción no entra en una posición de memoria, se fragmenta en posiciones sucesivas, y se localiza por la dirección de la primera.**

[4] Como se verá, el Debug permite escribir direcciones desde la 0100 H = 0000 0001 0000 0000 binario hasta la FFFF H = 1111 1111 1111 1111 binario = 65535d, por lo que se dispone cerca de ese número de celdas (bytes) libres para almacenamiento.

[5] Las letras **P** y **Q** que en alto nivel se denominan ***variables***, **a nivel de máquina se corresponden con las direcciones de memoria** (5000H y 5006H) elegidas para las mismas.. Los valores particulares 1020H y 2040H que en esta oportunidad tienense han asignado a dichas variables se corresponden con los contenidos a escribir en esas direcciones de memoria. (o sea los mismos 1020H y 2040H)

[6] Los archivos que guardan instrucciones en código de máquina se denominan "**archivos de código**" o "**ejecutables**"

[7] O un número relacionado con ella. Si el dato está en un registro de la UCP el código de operación determina de qué registro se trata. Si el dato a operar es siempre el mismo (constante) en lugar de dicha dirección puede ir el valor del dato, pero con un cod-op particular.

En general una instrucción ordena una operación (mediante su cod-op), y en relación con ésta **permite localizar uno o dos datos a operar** (en este caso dando su dirección en memoria), e indica **dónde guardar el resultado.**

El formato de las instrucciones usadas es:

CODIGO DE OPERACIÓN	DIRECCIÓN DEL DATO

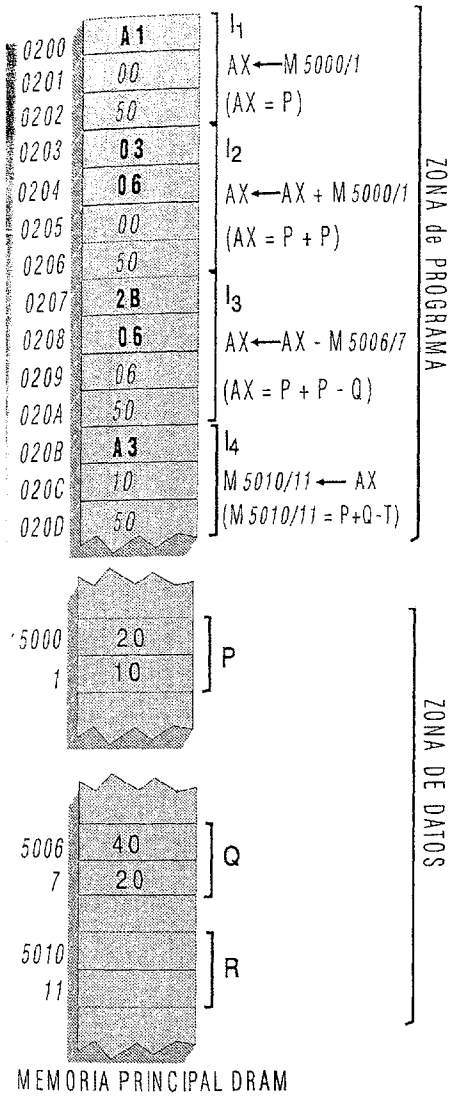


Figura 1.15

Teniendo presente el proceso de datos con calculadora de la figura 1.2, la instrucción I_1 del mismo, adaptada para una PC, ordenaría la operación de "transferir (mover) hacia el registro **AX** (de **2 bytes**) una copia de un número de 2 bytes localizable en la dirección de memoria que indican los dos últimos bytes de la instrucción" (en este caso es *5000*H). O sea escribir en **AX** una copia del número 1020H, que es el que se encuentra en dicha dirección[1] en este ejemplo, lo cual implica que se destruirá el número que antes estaba en **AX**.

Esto puede simbolizarse así: **AX** ← **M***5000 y 5001*

Para un procesador de Intel o de AMD el cód-op que ordena la operación escrita entre comillas es **A1**H = 10100001 [2].

El *código de máquina* de I_1 comprende su cod-op **A1**, seguido **de la dirección del dato** a operar (en este caso fue elegida 5000H[3]). Por lo tanto, el código de máquina completo que se escribirá en memoria (figura 1.15) es **A1***0050*H = **1010 0001** *0000 0000 0101 0000* en binario, o sea que I_1 consta de 24 bits = 3 bytes, por lo que ocupará tres posiciones consecutivas de memoria. Se ha elegido arbitrariamente para I_1 (y también para la secuencia de instrucciones a ejecutar) que la primer dirección a partir de la cual se escriben los tres bytes que la forman sea *0200*H. O sea que I_1 ocupa las direcciones *0200*H a *0202*H, con su código de máquina fraccionado en A1 , 00 y 50 en esas direcciones.

Es importante notar que *00 50* o sea *5000* **es la dirección** del dato, que forma parte del código de la instrucción, sin la cual cuando se ejecuta I_1 no habría manera de encontrar el dato que es 20 10, o sea 1020.

La instrucción I_2 ordena "sumar al número contenido en el registro **AX**, una copia del número de 2 bytes localizable en la dirección de memoria que indican los dos últimos bytes de la instrucción" (que en este caso vuelve a ser *5000*H, por haberse sumado **P** + **P**).

En símbolos: **AX** ← **AX** + **M***5000 y 5001*

O sea sumarle al número que existía en **AX** una copia del número 1020H (que es el que está en dicha dirección), y el resultado escribirlo en lugar del número que existía en **AX**. La operación escrita entre comillas la ordena el cod-op 0306H, al cual debe seguirle la dirección del dato (*5000*H), por lo que el código de máquina de I_2 será 0306 *0050*H (*0050* corresponde a *5000* con las mitades traspuestas) que en binario es: **0000 0011 0000 0110** *0000 0000 0101 0000*. Estos 32 bits = 4 bytes *obligatoriamente* deben ser escritos a continuación del código de I_1, o sea que ocupará las posiciones *0203*H a *0206*H (figura 1.15).

I_3 ordena "restar al número contenido en **AX** una copia del número de 2 bytes localizable en la dirección de memoria que indican los 2 últimos bytes de la instrucción"[4] (*5006*H en este caso, donde está el dato 2040 H).

---

[1] En gran medida es equivalente a pulsar **MR** en una calculadora de bolsillo, si previamente se ha memorizado 1020, con lo cual este número aparecerá en el visor, sin importar lo que el visor tenga antes. Dado que, como dijimos, los circuitos de cálculo y el registro acumulador de la UCP se comportan en gran medida como una calculadora, **es importante ir notando que las instrucciones** (códigos) **de máquina más simples** –que son también las mas usadas– **ordenan operaciones del tipo de las que realiza una calculadora**.

[2] Los códigos que usaremos para las instrucciones son los reales que "entiende" cualquier microprocesador de Intel™, del 80286 al Pentium, o sus "clones" de AMD. Pueden hallarse usando el Debug, como se indicará, al tratar Assembler en la Unidad 3.
En relación a los códigos, **si en lugar del registro AX se hubiera usado el BX** para la misma orden, el código hubiera sido 8B1E H

[3] Si bien el dato a operar está en las direcciones 5000H y 5001H, basta con dar la primera, dado que el código de operación (A1H) cuando se ejecute la instrucción ordenará a la UCP pedir lo que está en 5000H y 50001H.
El Debug simula un 286, por lo que en este ejemplo estamos operando con números del tamaño de un word. Cambiando el código de la instrucción puede ordenarse leer sólo la dirección 5000.

[4] Como si en una calculadora que hemos memorizado el número 2040 pulsáramos sucesivamente las teclas – **MR** =

No confundir el dato (2040) con su dirección (5006) indicada en la instrucción en estas instrucciones.
Su código de máquina será: 2B06 *0650* H, que también ocupará 4 bytes (direcciones *0207* H a *020A* H[1]).

En símbolos la operación que ordena I_3 puede escribirse: **AX** ← **AX** − **M***5006 y 5007*

Por último, I_4 ordena "transferir hacia las 2 posiciones de memoria cuya primer dirección indican los dos últimos bytes de la instrucción (en este caso *5010*) una copia del número que está en el registro **AX**". El código de operación A3 ordena la operación entre comillas, seguido de la dirección que en este caso es 5010, o sea que el código de máquina será A31050.

En símbolos la operación que ordena I_3 puede escribirse: **M***5010 y 5011* ← **AX**

En la figura 1.15 en las celdas de R no aparece ningún valor, puesto que el resultado (0000) queda en memoria luego de ejecutar las cuatro instrucciones, como se ejemplifica usando el Debug en la sección que empieza en la página siguiente. También en ella se verá el procedimiento por el cual se escriben en memoria en las direcciones elegidas, los valores de los datos a procesar, y las instrucciones (*en la práctica programas del sistema operativo se encargan de ello*).

La dirección (0200) de I_1 es *arbitraria*, dado que si cuando I_1 se va a ejecutar el registro **IP** tiene esa dirección será localizada por la **UC** encargada de ejecutarla. A partir de esa dirección, el cod-op de esa instrucción y de las siguientes indica lo que debe sumarse al **IP** para localizar la siguiente instrucción a ejecutar. El número a sumar al **IP** es la cantidad de celdas que ocupa cada instrucción. Es el sistema operativo, quién decide cual será la dirección de la primera instrucción de un programa, la cual llegará al **IP** merced al programa cargador.

¿Qué sería "alto nivel" y "bajo nivel" en la codificación realizada ?

Cuando escribimos la expresión **R = P + P − Q** estamos indicando una serie de operaciones aritméticas, cuyo resultado –que depende de los valores que les asignemos a **P** y **Q** les asignemos– debe asignarse a **R**.
Se trata de una forma bastante abstracta y compacta de codificar *varios* pasos a realizar, que el lenguaje matemático permite así formular. Dicha expresión que resume un proceso a realizar, en una PC hemos visto que requiere cuatro instrucciones (procesos a realizar) que hemos codificado en hexa A10050 03060050 2B060650 y A31050, códigos que en memoria principal se almacenan en binario..
Las dos formas de codificar qué debe hacerse (**R = P + P − Q** y los códigos citados), ordenan realizar el *mismo* proceso. La expresión matemática lo hace en una forma más compacta y fácil de expresar para nosotros, mientras que los códigos de las instrucciones a ejecutar amen de resultar poco manejables para el hombre, están codificados en un lenguaje que sólo pueden entender los procesadores de Intel™ o AMD. Asimismo, el proceso a realizar ha sido descompuesto en 3 subprocesos (uno por instrucción) acorde a lo que es capaz de realizar y "entender" (decodificar) la máquina.
La forma "matemática", abstracta, de codificar más familiar a nosotros, se dice de "**alto nivel**", y es la que se emplea para programar procesos en los lenguajes de alto nivel (Fortran, Pascal, Cobol, Basic, C, Java, etc.).
En cambio, la codificación binaria al nivel de códigos de instrucción que "entiende" una máquina determinada se denomina de "**bajo nivel**" o "**nivel de máquina**" o "**código de máquina**" o "lenguaje absoluto". de máquina"
Cuando antes codificamos la expresión matemática en cuatro instrucciones, de hecho hemos pasado de alto a bajo nivel. Esta *traducción* en las primeras computadoras era realizada por personas que codificaban en binario las instrucciones para el computador, mediante llaves existentes en un panel frontal.
Posteriormente este proceso fue realizado por programas denominados "**compiladores**" encargados de traducir las órdenes codificadas en un lenguaje de alto nivel que estaban en memoria principal, en códigos de instrucciones, que también quedaban en memoria. Para Java se usan traductores tipo "**intérpretes**".

Un programa en alto nivel que fue traducido por ejemplo para un 80x86, no puede ser ejecutado por otro procesador y viceversa. Para cada procesador existe un compilador creado sólo para el mismo.

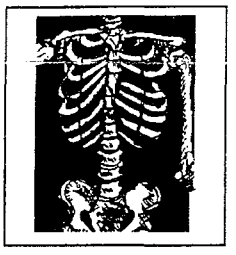
Como se verá, cuando una instrucción se ejecuta, para que se lleve a cabo el subproceso que ella ordena, la UC lo lleva a cabo mediante una serie de movimientos o subprocesos menores, más simples, que los circuitos de la UCP y memoria pueden realizar.

---

[1] En hexa al 0209 le sigue el 020A, y no el 0210 como sería en decimal,. Esto es acorde con que en la tabla de la figura 1.4, al 9 le sigue A. Esto se trata en detalle en el Apéndice A1.

[2] Esta instrucción en una calculadora es equivalente a pulsar **M+** (suponiendo que MC fue pulsada previamente). Así en ésta para efectuar **A + B** y memorizar, primero hay que llevar **A** al visor y luego sumarle **B**, y tercero memorizar con **M+** el resultado (pudiendo ser necesario pulsar antes **CM**).

1.6 USO DEL PROGRAMA DEBUG DEL DOS PARA VISUALIZAR EL INTERIOR DEL COMPUTADOR



¿Cómo se usa el Debug para escribir datos e inst

A continuación se indican los pasos a dar para llamar al program
está en Windows 98 y NT), y luego poder leer y escribir los datos
figura 1.15. *El procedimiento sirve en general para leer o escribir*
En negrita aparece aquello que escribe el DOS (y que interesa
mayúscula lo que debe escribir el usuario (con la tipografía que sea
Los números que se tipean y los que aparecen en pantalla son h
en el interior de un computador solo puede haber unos y ceros.

C:\> *DEBUG* ↵
–

Un guión que aparece titilante debajo de la C, indica que el Debug está listo para que a continuación del guión se escriba un comando. Para leer o escribir la memoria se tipea la letra *E* (examinar) seguida de la primer dirección que se quiere leer o escribir. Por ejemplo, si se quiere conocer el contenido de posiciones de memoria a partir de la dirección 5000 H, primero se tipea el comando *E 5000* seguido de un Enter (↵), con lo cual en la pantalla se verá, por ejemplo, lo siguiente:

– E 5000 ↵		(tipeado por el usuario para examinar memoria)
309D:5000	**1F.**	(mostrado por el Debug)

Esto nos dice que la dirección 5000H (0101 0000 0000 0000) de memoria[1] tiene por contenido la combinación 1F H = 00011111. **Esta combinación es la que estaba almacenada en la dirección 5000H en la PC que hemos utilizado, pero que otro día, o en otra PC será distinta**[2]. Lo mismo vale para 309D
Si luego de leer la 5000H se quiere leer el contenido de la posición *siguiente* 5001H, no hace falta repetir el comando E. Basta con pulsar la barra de espaciado luego del último valor leído –en este caso 1F – para que en pantalla aparezca el número que guarda la posición 5001 (06 en este caso como aparece a continuación).

–E 5000 ↵ (Examinar memoria)

309D:5000	**1F.**	**06.**	**50.**	**8D.**	**46.**	**B0.**	**A0.**	**6A.**
309D:5008	**09.**	**6A.**	**FF.**	**9A.**	**66.**	**0E.**	**DF.**	**00.**
309D:5010	**FD.**	**11.** ↵						
–		5000	5001 . . .				5006	5007

Del mismo modo si se vuelve a pulsar la barra se conocerá el contenido de la posición 5002H, y así de seguido para determinar contenidos de posiciones *consecutivas* de memoria. Por razones de claridad, aparecen hasta 8 lecturas de posiciones por renglón, y para no llenar la pantalla con números, *sólo se muestra*

---

1 Aunque esto por el momento no es relevante, conviene aclarar que realidad se trata de una **zona** (*segmento*) de memoria con más de 64000 posiciones (bytes), numerados con direcciones que van de 0000H hasta FFFFH. La dirección 0000H es la primer posición de este segmento de memoria, y no el cero de la memoria. La dirección de memoria donde comienza este segmento *está en relación* con el valor 309D que aparece a la izquierda de 5000. Dicho 0000 corresponde a la dirección de memoria 309D0 H (309D con el agregado de un cero, conforme a la convención para los 80x86 de Intel). Si a 309D0 le sumamos 5000 H se tendría 359D0 H = 0011 0101 1001 1101 0000 en binario (20 bits), que sería una dirección de memoria que está 5000 H posiciones después de 309D. Puesto que las direcciones de memoria ejemplificadas son de 20 bits, significa que se trata de una memoria de 1 megabyte, como se deduce de la figura 1.14 (2^{20} = 1 MB)
Este es el tamaño de memoria que "ve" el DOS, desde la dirección 00000000000000000000 = 00000 H hasta la 11111111111111111111 = = FFFFF H. Este tema se verá en detalle al tratar este sistema operativo en la Unidad 3.

2 *Conforme al modelo de 8 llaves "si-no" por celda de memoria, las mismas siempre están formando una combinación de unos y ceros determinada. No es factible que no haya "nada" en una posición.* Cuando se enciende un computador, en cada posición se forma un número al azar, que no tiene por qué ser el 1F ejemplificado en esta obra.

1.6 USO DEL PROGRAMA DEBUG DEL DOS PARA VISUALIZAR EL INTERIOR DEL COMPUTADOR

¿Cómo se usa el Debug para escribir datos e instrucciones en memoria ?

A continuación se indican los pasos a dar para llamar al programa **Debug** desde el DOS (que también está en Windows 98 y NT), y luego poder leer y escribir los datos e instrucciones como se plantea en la figura 1.15. *El procedimiento sirve en general para leer o escribir la memoria principal de una* PC.
En negrita aparece aquello que escribe el DOS (y que interesa a los fines didácticos), y en cursiva mayúscula lo que debe escribir el usuario (con la tipografía que sea). El símbolo ↵ indica la tecla "Enter". *Los números que se tipean y los que aparecen en pantalla son hexadecimales, aunque como sabemos en el interior de un computador solo puede haber unos y ceros.*

C:\> *DEBUG* ↵
–
Un guión que aparece titilante debajo de la C, indica que el Debug está listo para que a continuación del guión se escriba un comando. Para leer o escribir la memoria se tipea la letra *E* (examinar) seguida de la primer dirección que se quiere leer o escribir. Por ejemplo, si se quiere conocer el contenido de posiciones de memoria a partir de la dirección 5000 H, primero se tipea el comando *E 5000* seguido de un Enter (↵), con lo cual en la pantalla se verá, por ejemplo, lo siguiente:

– E 5000 ↵		(tipeado por el usuario para examinar memoria)
309D:5000	**1F.**	(mostrado por el Debug)

Esto nos dice que la dirección 5000H (0101 0000 0000 0000) de memoria[1] tiene por contenido la combinación 1F H = 00011111. **Esta combinación es la que estaba almacenada en la dirección 5000H en la PC que hemos utilizado, pero que otro día, o en otra PC será distinta**[2]. Lo mismo vale para 309D
Si luego de leer la 5000H se quiere leer el contenido de la posición *siguiente* 5001H, no hace falta repetir el comando **E**. Basta con pulsar la barra de espaciado luego del último valor leído –en este caso 1F – para que en pantalla aparezca el número que guarda la posición 5001 (06 en este caso como aparece a continuación).

–E 5000 ↵	(Examinar memoria)							
309D:5000	**1F.**	**06.**	**50.**	**8D.**	**46.**	**B0.**	**A0.**	**6A.**
309D:5008	**09.**	**6A.**	**FF.**	**9A.**	**66.**	**0E.**	**DF.**	**00.**
309D:5010	**FD.**	**11.** ↵						
–		5000	5001 . . .				5006	5007

Del mismo modo si se vuelve a pulsar la barra se conocerá el contenido de la posición 5002H, y así de seguido para determinar contenidos de posiciones *consecutivas* de memoria. Por razones de claridad, aparecen hasta 8 lecturas de posiciones por renglón, y para no llenar la pantalla con números, *sólo se muestra*

---

1 Aunque esto por el momento no es relevante, conviene aclarar que realidad se trata de una **zona** (*segmento*) de memoria con más de 64000 posiciones (bytes), numerados con direcciones que van de 0000H hasta FFFFH. La dirección 0000H es la primer posición de este segmento de memoria, y no el cero de la memoria. La dirección de memoria donde comienza este segmento *está en relación* con el valor 309D que aparece a la izquierda de 5000. Dicho 0000 corresponde a la dirección de memoria 309D0 H (309D con el agregado de un cero, conforme a la convención para los 80x86 de Intel). Si a 309D0 le sumamos 5000 H se tendría 359D0 H = 0011 0101 1001 1101 0000 en binario (20 bits), que sería una dirección de memoria que está 5000 H posiciones después de 309D. Puesto que las direcciones de memoria ejemplificadas son de 20 bits, significa que se trata de una memoria de 1 megabyte, como se deduce de la figura 1.14 (2^{20} = 1 MB)
Este es el tamaño de memoria que "ve" el DOS, desde la dirección 00000000000000000000 = 00000 H hasta la 11111111111111111111 = = FFFFF H. Este tema se verá en detalle al tratar este sistema operativo en la Unidad 3.

2 *Conforme al modelo de 8 llaves "si-no" por celda de memoria, las mismas siempre están formando una combinación de unos y ceros determinada. No es factible que no haya "nada" en una posición.* Cuando se enciende un computador, en cada posición se forma un número al azar, que no tiene por qué ser el 1F ejemplificado en esta obra.

una dirección a la izquierda de cada renglón. Así, luego de la 5000 no aparecerán en pantalla 5001 a 5007 (que deben determinarse visualmente, como señalan las flechas dibujadas con los valores), y luego aparece 5008. Entonces, si a partir de la indicación anterior pulsamos repetidamente la barra de espaciado, se obtendría un vuelco de memoria del tipo siguiente (*distinto para cada oportunidad y computador*):
Cuando se quiere dejar de leer se pulsa Enter (↵), como se indica en el último renglón, con lo cual vuelve a aparecer el guión titilante del Debug, esperando un nuevo comando. En general se debe pulsar Enter luego de escribir un comando o cuando se quiere que aparezca el guión titilante, para indicarle un nuevo comando al Debug. Los valores que aparecen varían de un computador a otro, y en un mismo computador. Con el Debug es factible leer *cualquier* zona de memoria, sea RAM o ROM.

Escritura de los datos en memoria mediante el Debug:
El ejemplo anterior mostró la lectura de posiciones consecutivas de memoria. También es posible luego de leer una posición *cambiar su contenido*, o sea escribirle un número de dos dígitos en hexa (equivalente a un byte binario). Este número se debe escribir a continuación del punto que acompaña a cada valor leído, en el espacio que para tal fin se ha reservado. Luego de escribir el nuevo contenido que tendrá una posición, se puede leer la siguiente pulsando la barra de la forma vista. Si también se desea cambiar el contenido de esta posición, debe escribirse el nuevo valor después del punto del contenido anterior, y así de seguido.
Volviendo al proceso de datos de la figura 1.15 escribiremos en 5000 y 5001 los contenidos 20 y 10 (que corresponden al dato **P** = 1020 H, y en 5006 y 5007, los contenidos 40 y 20, correspondientes a **Q** = 2040 H

```
-E 5000 ↵   (Examinar memoria y escribir en ella)
309D:5000  1F.20  06.10  50.  8D.  46.  B0.  A0.40  6A.20 ↵
-
```

Figura 1.16[1]

Si se quiere corroborar que los valores recién escritos son los nuevos contenidos de las posiciones modificadas, nuevamente se ordena examinar, debiendo resultar:

```
-E 5000 ↵   (Examinar memoria)
309D:5000  20.  10.  50.  8D.  46.  B0.  40.  20. ↵
-
```

De esta manera se han escrito en las posiciones 5000, 5001, 5006 y 5007 los datos que en la figura 1.15 aparecen escritos en la "escalera" (que representa las posiciones de memoria). En las posiciones 5010 y 5011 deberá ir el resultado de la operación a realizar, en reemplazo de los contenidos que existan (en este caso FD y 11, como surge de la primer lectura realizada), por lo que *no interesa* estos valores (FD y 11)

Con el mismo procedimiento se escriben en memoria los códigos de máquina de las instrucciones, a partir de la dirección elegida 0200H

```
-E 200 ↵   (Examinar memoria y escribir en ella)
309D:0200  25.A1  F3.00  AA.50  DD.03  09.06  56.00  00.50  AB.2B
309D:0208  49.06  FF.06  00.50  12.A3  FF.10  FA.50 ↵
-
```

Verificando luego si la escritura anterior es correcta, resulta:

```
-E 200 ↵   (Examinar memoria)
309D:0200  A1.  00.  50.  03.  06.  00.  50.  2B.
309D:0208  06.  06.  50.  A3   10   50 ↵
-
```

De esta forma hemos realizado la escritura en memoria de las instrucciones.

Suponiendo que se quiera salir del Debug, se debe tipear Q (Quit) para volver al DOS:

```
-Q ↵   (para salir del Debug)
C:\>   (el DOS espera comando)
```

[1] De acá en adelante se designa con la palabra "figura" a porciones de la pantalla que se verían usando el programa Debug.

¿Cómo encuentra la UC en memoria la primer instrucción y las siguientes de un programa a ejecutar, mediante el registro IP ?

Anteriormente se estableció que el **registro puntero de instrucción (IP)** tiene la función de indicar la dirección de memoria donde se encuentra la próxima instrucción a ejecutar.
En el programa codificado (figura 1.15) la primer instrucción comienza en la dirección 0200H, y las dos siguientes en 0203H y en 0207H. Por lo tanto, en este caso IP deberá contener sucesivamente los valores binarios que en hexa son: 0200H, 0203H y 0207H, como se verificará cuando se ejecuten esas instrucciones.
Suponiendo que IP empiece con el valor 0200H –ya veremos cómo– puesto que el código de I_1 ocupa 3 bytes, (esto lo "conocerá" la UC al ejecutar I_1), si la UC hace la suma (con su calculador que es la UAL) 0200 + 3 = 0203, podrá determinar el nuevo valor que debe tener IP, que es la dirección donde está I_2
Del mismo modo, cuando ejecuta I_2–que ocupa 4 bytes–, podrá calcular 0203 + 4 = 0207 para ubicar I_3.
Por último si a 0207 le sumamos 4 (cantidad de bytes de I_3), obtenemos 020B, dirección de I_4 (en hexa después de 7 siguen 8, 9, A, B) Así sucesivamente la UC va cambiando el valor de IP [1] para ir localizando en memoria las sucesivas instrucciones que debe ejecutar.
Esta es la forma pensada para que la UC localice y ejecute, en el orden establecido, cada una de las instrucciones que forman una secuencia. Por lo tanto, para respetar estas "reglas de juego" *las instrucciones a ejecutar deben estar escritas en posiciones consecutivas de memoria.*
Una "**instrucción de salto**" –a tratar– permite no seguir la ejecución de un programa con la instrucción que está escrita a continuación de ella en memoria, sino continuar con otra instrucción, cuya dirección debe permitir formar instrucción de salto. El valor de esta dirección debe pasar al IP. O sea que igualmente esta instrucción permite encontrar la que le sigue.

Por la tanto, *cada instrucción permite determinar donde está la siguiente a ejecutar, y por consiguiente, establecer el valor que tendrá el registro IP*

¿Quién se encarga de proporcionar la dirección de la primer instrucción de cada programa a ejecutar ?

En las prácticas que realizaremos con el Debug, seremos nosotros quienes nos encargaremos de escribir en el IP la dirección de la primer instrucción de una secuencia que queremos ejecutar. Puesto que un computador pasa automáticamente de la ejecución de un programa a otro, se supone que tiene que existir un mecanismo para hacer esto. A continuación daremos un esbozo acerca de cómo esto último tiene lugar, o sea quién se encarga de dar al IP la dirección donde comienza cada nuevo programa a ejecutar, una vez que el mismo y los datos se han escrito en memoria.
Cuando se enciende un computador, el primer programa que se ejecuta es siempre el mismo. Los códigos de sus instrucciones están permanentemente en la porción ROM de memoria principal, siendo que el hardware (circuitería) provee la dirección de la primer instrucción a ejecutar. O se que el hardware inicializa el valor del IP al encender el equipo, así como el valor de otros registros.[2]
Las primeras generaciones de computadoras que se construyeron –hoy visibles en fotos o museos– presentaban un gran panel frontal lleno de llaves "si-no". El operador tenía que cargar mediante dichas llaves, un registro equivalente al IP con la dirección donde estaba la primera instrucción del primer programa a ejecutar. Algo semejante haremos muy pronto mediante el teclado y el programa Debug.
Como resultado de la ejecución de dicho programa almacenado en ROM, se escribe en memoria principal una copia de programas que están en el disco. Estos al ser ejecutados traen a memoria los programas del "sistema operativo", que también están en el disco como archivos. Luego de lo cual en pantalla aparece C> o alguna imagen con opciones, para indicar al sistema operativo mediante un comando, qué programa se desea traer a memoria.
Esto es, un programa del sistema operativo permite traer del disco a memoria, una copia del programa que necesita el usuario, *y se encarga de que en el IP aparezca la dirección de la primer instrucción del programa a ejecutar.*
Esto se consigue si la última instrucción del programa del SO que se encarga de esta tarea, es una instrucción de salto, que indique la dirección donde está la primer instrucción del programa del usuario a ejecutar.

---

1 La UC necesita "anotar" en IP el valor de la dirección que calculó mediante la UAL. De no memorizarlo en IP, se perdería dicho valor

2 Como el registro CS (code segment) de la UCP que junto con IP constituyen un **contador de programa (CP)**

Como se describió, esta dirección se instala en el IP, con lo cual en forma automática, *sin intervención humana*, la próxima instrucción que la UC ejecuta – luego que ejecutó la última instrucción de salto citada será la primera del programa de usuario. A partir de ésta se hallan las siguientes de la forma vista.
Debe consignarse que el sistema operativo registra la dirección dónde dejó en memoria la primer instrucción del programa que trajo del disco, la cual es proporcionada al IP por la instrucción de salto citada. Cuando termina de ejecutarse este programa, la última instrucción del mismo llamará al sistema operativo. Será una instrucción que en esencia ordena saltar a la dirección de memoria donde está la primer instrucción de un programa del SO. Este decidirá cuál es el próximo programa que debe ejecutar la UC.
De esta forma se pasa automáticamente de la ejecución de un programa a otro, sin intervención humana.
Las instrucciones de salto, son pues las que permiten cambiar automáticamente el valor del IP, y pasar a ejecutar una instrucción que está en cualquier lugar de la memoria, que será la primera de una secuencia escrita en posiciones sucesivas de memoria. Esta secuencia puede ser del propio programa en ejecución, o pertenecer a otro programa que debe ejecutarse a continuación.

¿Cómo se cambia la dirección de instrucción que indica el iP ?

Conforme a lo anticipado mas arriba, mediante el Debug cargaremos en el IP la dirección de la primer instrucción a ejecutar, que como hemos establecido es 0200H. Para realizar esto, una vez que mediante el Debug hemos escrito los datos y las instrucciones, le damos la orden:

```
-R IP ↵      (comando al Debug para examinar el valor del Registro IP y cambiarlo si se desea)
IP 0100      (el Debug informa que actualmente el IP contiene 0100)
: 0200 ↵     (al lado de los dos puntos que deja el Debug escribimos 0200, nuevo valor que debe tener IP)
-
```

Figura 1.17

Teniendo en el IP la dirección de I_1 ya se puede ejecutar la secuencia de instrucciones escritas, como sigue.

¿Cómo puede visualizarse mediante el Debug la forma en que se van procesando los datos, al ejecutarse las instrucciones en una PC ?

Así como el Debug permite examinar y modificar la memoria, también puede mostrar en pantalla una "radiografía" del contenido de registros del microprocesador (UCP). Observando los datos a procesar en memoria, y como a partir de ellos se obtienen resultados, puede seguirse la evolución de un proceso de datos. Además con el Debug esto puede hacerse paso a paso, dado que permite ejecutar en "cámara lenta" una a una las instrucciones de un programa o secuencia, de modo de poder ver como van cambiando de valor los registros de la UCP y posiciones de memoria. Esto es lo que haremos a continuación, con las instrucciones escritas anteriormente, experiencia que puede hacerse con cualquier PC. Antes de ejecutar dichas instrucciones, y por método, primero conviene examinar los registros de la UCP y otros datos que muestra el Debug[1], de los cuales sólo nos interesan en esta etapa los que indicaremos en negrita.
Hemos usado el comando *R IP* ↵ para conocer el valor de un registro particular[2] y poder luego cambiarlo. Si se usa el comando R a secas, se visualiza el valor de registros de la UCP sin poder modificarlos:

```
-R ↵          (Examinar registros)
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=309D  ES=309D  SS=309D  CS=309D  IP=0200  NV UP EI PL NZ NA PE NC
309D:0200  A10050          MOV AX,[5000] [3]          DS:5000=1020
-
```

Figura 1.18

---

[1] El *Debug simula un 80286*, pero como éste es compatible con el 386, 486 y el Pentium (que tienen registros de 32 bits) puede usarse con cualquier PC. Hay programas mas completos que el Debug, pero lo hemos elegido por que está a mano en cualquier PC.

[2] Este comando, seguido del nombre de cualquier registro de la UCP, permite leer su valor y poderlo cambiar. Así ***R AX*** permite leer y cambiar AX. La lista de todos los comandos del Debug se obtiene mediante la tecla ? seguida de Enter.

[3] En cada tercer renglón, a la derecha del código de máquina de la próxima instrucción que se ejecutará (en este caso A10050), siempre aparece su codificación equivalente en lenguaje **Assembler** (en este caso MOV AX, [5000], a tratar en la Unidad 3 de la presente obra

Los dos primeros renglones indican el estado de registros de la UCP. De éstos en primer lugar por el momento únicamente importan AX e IP[1]. Como primer medida verificamos que IP está con el valor 0200 que le habíamos asignado mediante R IP. Si bien AX contiene 0000H (16 ceros en binario), este valor particular no interesa, pues I_1 ordena escribir 1020H, destruyendo el valor anterior (0000H).
El tercer renglón se refiere a la memoria principal. El valor 0200 (igual al de IP) corrobora la dirección donde empieza la próxima instrucción a ejecutar, y al lado del mismo el código A10050, que es el que habíamos escrito en memoria para I_1, como puede verificarse (que será el valor que tendrá el registro RI)
O sea que es la instrucción que queremos ejecutar en primer lugar. En la parte derecha indica que en la dirección 5000 (involucrada en la instrucción) se tiene el valor 1020 (dato antes escrito)
Puesto que IP=0200, el dato 1020 y el código A10050 de la instrucción son correctos, podemos ejecutar ésta. Para ello simplemente se da el comando T, y luego el Debug visualiza la misma información que la figura 1.18, pero con los cambios habidos luego de la ejecución de la instrucción:

```
-T ↵    (Ejecución de la instrucción I1)
AX=1020   BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=309D   ES=309D  SS=309D  CS=309D  IP=0203  NV UP EI PL NZ NA PO NC

309D:0203 03060050          ADD AX,[5000] [2]          DS:5000=1020
-
```

Figura 1.19

Constatamos que I_1 se ha ejecutado correctamente, pues se ha cumplido la orden que portaba su código: escribir en AX una copia del contenido de la posición 5000, que era 1020. Si observamos el tercer renglón vemos que en 5000 sigue estando 1020. También ha cambiado IP a 0203, para apuntar la dirección de I_2, como habíamos previsto al hablar de IP. Asimismo vemos que el código 03060050 de I_2 es el correcto, por lo que podemos ejecutar I_2 [2]:

```
-T ↵    (Ejecución de la instrucción I2)
AX=2040   BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=309D   ES=309D  SS=309D  CS=309D  IP=0207  NV UP EI PL NZ NA PO NC

309D:0207 2B060650          SUB AX,[5006] [2]          DS:5006=2040
-
```

Figura 1.20

Se ha realizado lo que ordenaba I_2: sumar al valor 1020 de AX el contenido de 5000 (que es 1020), y el resultado (2040) escribirlo en lugar de 1020. También se verifica que el dato es 2040, y que 2B060650 es el código de I_3, instrucción de resta que pasaremos a ejecutar:

```
-T ↵    (Ejecución de la instrucción I3)
AX=0000   BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=309D   ES=309D  SS=309D  CS=309D  IP=020B  NV UP EI PL ZR NA PE NC

309D:020B A31050            MOV [5010],AX              DS:5010=FD11
-
```

Figura 1.21

I_3 ordenaba restar a AX el contenido de 5006, que es 2040H, o sea que se ha efectuado 2040H – 2040H = = 0000H, como aparece en AX.

```
-T ↵    (Ejecución de la instrucción I4)
AX=0000   BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=309D   ES=309D  SS=309D  CS=309D  IP=020E  NV UP EI PL ZR NA PE NC

309D:020B BB1026            MOV BX,2610
-
```

Figura 1.22

---

[1] En la Unidad 3 al ejemplificar secuencias mas complejas, usaremos casi todos ellos, al igual que muy pronto los "*flags*" o indicadores de estado (NV UP EI, etc.). Los registros DS, CS, SS, ES, todos con el valor de referencia 309D mencionado en un pie de página anterior. De tener valores diferentes cada uno, permiten definir para Datos, Código, Stack y Extensión, cuatro zonas independientes de memoria de 64000 direcciones (posiciones de un byte). Como todos tienen igual valor, las cuatro zonas (*"segmentos"*) están superpuestas en una sola.

[2] Por razones didácticas se ha elegido el mismo dato para I_1 e I_2, pudiendo estar en otra dirección el dato que se opera en I_2

El registro IP pasó a 020E H. En esa posición y en las dos siguientes quedaron al azar los valores BB y 26 (como resulta de leer el tercer renglón) que el Debug interpreta como correspondientes a o instrucción. Como ya hemos ejecutado I_1, I_2, I_3, e I_4 no seguimos ejecutando ninguna instrucción má

Un programa termina con una instrucción (que no hemos usado) cuyo código ordena que se pase a ejecu un programa del sistema operativo, para que éste decida cuál es el próximo programa que se debe ejecuta

Si queremos verificar que en las direcciones 5010 y 5011 se escribió el resultado que está en **AX**. Hacemos

```
-E 5010 ↵   (Examinar memoria)
309D:5010  00.    00. ↵
```

con lo cual constatamos que efectivamente se cumplió lo que ordena I_4

¿Cómo ordenar que los códigos de máquina del proceso anterior sea ejecutados una tras otro automáticamente, conforme sucede realmente

Por razones didácticas se ha ejecutado instrucción por instrucción, para ver luego de cada ejecución com cambiaba el interior del computador. Fue necesario antes de ejecutar cada instrucción la intervención human para pulsar *T* ↵ entre otras cosas. Esto se parece al manejo de una calculadora cada vez que apretamos tecla = . Si las computadoras funcionaran de esa manera, no tendría mucho sentido su existencia, puesto qu justamente su velocidad de procesamiento de datos es quizás el mayor de sus atributos.
Con el Debug también es factible hacer que se ejecuten, una tras otra, a la velocidad del computador, la instrucciones escritas en memoria, y así obtener casi instantáneamente los resultados requeridos.

> Así puede corroborarse que una vez que datos e instrucciones están en memoria, la UCP realiza u proceso de datos en forma *automática*, *sin intervención humana*, merced a la acción de circuito electrónicos que responden a las órdenes que portan las instrucciones.

Luego de haber ejecutado las instrucciones una por una, puesto que los datos e instrucciones permanecen a memoria sin cambios, y que I_1 no requiere que AX esté en cero, si se ejecutan nuevamente dich instrucciones se obtendrán los mismos resultados. Esto es lo que haremos, pero en vez de ejecutarlas una po una, daremos la orden de ejecutar una tras otra, I_1, I_2, I_3 e I_4 Antes de dar el comando debemos hacer que IP –que quedó en 020B– pase a tener el valor 0200 H, para apuntar a la dirección donde comienza I_1, para l cual debemos ordenar R IP de la forma vista. Después, como paso previo a la ejecución, para constatar qu todo está en orden podemos hacer como antes:

```
-R ↵(Examinar registros)
AX=0000   BX=0000   CX=0000   DX=0000   SP=FFEE   BP=0000   SI=0000 DI=0000
DS=309D   ES=309D   SS=309D   CS=309D   IP=0200 NV UP EI PL NZ NA PO NC

309D:0200  A10050          MOV  AX,[5000]                  DS:5000=1020
```

El comando para ejecutar las instrucciones que van de la dirección 200 a la 20F [1] es

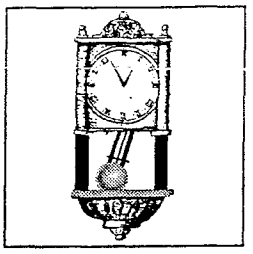
```
-G =0200 020E ↵
AX=0000   BX=0000   CX=0000   DX=0000   SP=FFEE   BP=0000   SI=0000 DI=0000
DS=309D   ES=309D   SS=309D   CS=309D   IP=020E NV UP EI PL ZR NA PE NC

309D:020B  BB1026          MOV BX,2610
```

Puede verificarse que se ha obtenido el mismo resultado que en la figura 1.22, correspondiente al mismo proceso realizado instrucción por instrucción con intervención del operador.

Es importante aclarar que *G =0200 020E ↵ es una* orden para el Debug, para que se ejecute el program que comienza en la dirección indicada, y no una orden para la UC, que ***permanentemente*** *está ejecutand programas*. Sin ir más lejos se ejecuta un programa cada vez que pulsamos o soltamos cada tecla de comando *G = 0200 020E*, así como se ejecuta otro cada 0,018 seg. para actualizar la hora y el calendari del computador, entre otros. Como se estableció, es mediante instrucciones de salto, antes citadas, que s cambia el valor del IP para que la UC en forma *automática* pase de la ejecución de un programa a otro.

[1] Dirección donde comienza una instrucción que sigue a I_5 que no queremos ejecutar.

1.7 PAPEL DE LA UC Y DE LOS MHz DEL RELOJ EN LA EJECUCION DE LAS INSTRUCCIONES



¿Cómo se ejecutan las instrucciones I_1 a I_4 mediante movimientos simples entre memoria y registros de la UCP ordenados por la UC ?

Con el fin de comprender mejor como opera la UC, veremos más en detalle la forma en que se ejecutan las instrucciones, a partir del esquema de la figura 1.8, con el agregado de los registros de direcciones (RDI) y de datos (RDA), definidos en la sección 1.4 al tratar el acceso al azar.
Como se vio en relación con la figura 1.15 cada **sentencia** (orden) de un programa en alto nivel (como R = P + P - T) es traducida por un programa compilador en una secuencia de **instrucciones** (órdenes más simples, en nuestro ejemplo $\mathbf{I_1}$, $\mathbf{I_2}$, $\mathbf{I_3}$ e $\mathbf{I_4}$) que una UC puede ejecutar.

> **A su vez**, la ejecución de cada instrucción se divide en **pasos** aún más simples (4 en nuestro caso, figs 1.23 a 1.25). Las acciones que debe ordenar/controlar la UC en cada uno de estos 4 pasos están determinadas por 4 combinaciones binarias llamadas "**microcódigos**" que van apareciendo una tras otra en las **líneas de control** con cada uno de los pulsos que constituyen los Mhz (figurass1.29 y 1.30).
> Estas líneas salen de la UC hacia la UAL, los registros y la memoria (fig. 1.31). La ejecución de una instrucción implica una secuencia de movimientos de transferencia de bytes entre memoria principal y registros de la UCP (o entre estos últimos), establecidos por la UC en un orden determinado, según el código de dicha instrucción. También puede ordenarse una operación en la UAL

Cada segundo puede ejecutarse algunos millones de instrucciones, para lo cual deben sucederse muchos *millones* de estos movimientos de pasaje de direcciones, códigos, datos y resultados, *al ritmo de millones de impulsos eléctricos por segundo* ("**megahertz**", abreviados **MHz**[1]) que le llegan a la UC, generados regularmente por un cristal piezo-eléctrico de cuarzo o "**reloj**" (*"clock"*) .
Así se habla de microprocesadores (con reloj) de 100 MHz, 1 GHz, etc. En principio, a mayor número de MHz podrán suceder más de estos movimientos por segundo, con lo cual se podrán ejecutar más instrucciones por segundo. Un Pentium actual de 1 GHz puede ejecutar más de mil millones de instrucciones por segundo (1000 MIPS), y en ciertos casos hasta 3000 MIPS.
Describiremos *en un modelo simplificado,* el orden, origen y destino de estos movimientos, que deben llevarse a cabo durante la ejecución de una instrucción, para $\mathbf{I_1}$, $\mathbf{I_2}$, $\mathbf{I_3}$ e $\mathbf{I_4}$, y los agruparemos por etapas
Dichos movimientos el Debug no los puede mostrar, como tampoco muestra los registros RI, RDI y RDA.

> Así se comprenderá que la UC tiene como función primera dar órdenes de operaciones de lectura o escritura a la memoria y registros de la UCP, y ordenar qué operación debe hacer la UAL, o sea **controlar**, *en el sentido de dar órdenes*, a esos dispositivos. De ahí su nombre de "unidad de control"

A fin de hacer más simple la explicación, supondremos que cuando la UC ordena leer la zona de instrucciones de memoria –a la cual apunta el valor de IP– *pide 4 bytes consecutivos de código de instrucción*[2]. Para ello, si por ejemplo es IP = 0200, asumiremos que primero pide leer 0200 y 0201, y luego 0202 y 0203[3]. Los contenidos de estas posiciones llegan al **registro de instrucción RI** (figura 1.23).
Para leer dos (o más) direcciones consecutivas basta dar el número correspondiente a la primera de ellas.

Durante la obtención y ejecución de una instrucción, ocurren en definitiva las siguientes acciones y

---

1 En Electricidad, si un fenómeno sucede X veces por segundo se dice que tiene una frecuencia de repetición de X Hertz (hercios), en honor a Hertz, descubridor de las ondas electromagnéticas. Un Hertz (Hz) es un ciclo por segundo; 1000 Hz son un kilohertz (Khz); 1000000 Hz son un megahertz (Mhz)

2 En los microprocesadores actuales, para ganar tiempo, mientras se están ejecutando instrucciones pedidas anteriormente, se van leyendo de MP códigos de instrucciones a ser ejecutados localizados en posiciones consecutivas. Las instrucciones pedidas con anticipación se guardan en una memoria interna del microprocesador. Esto se describe en la sección 1.14.

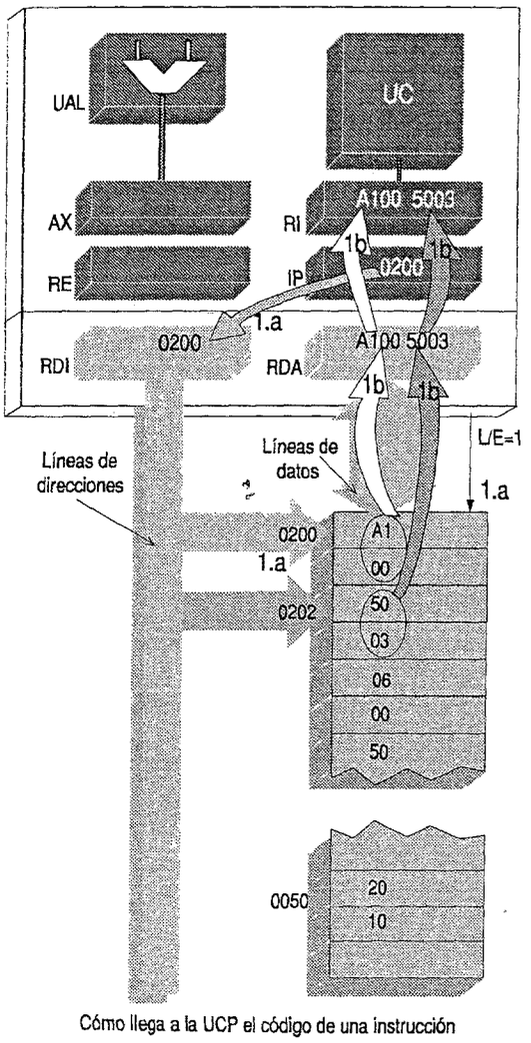
3 O sea estamos suponiendo un microprocesador como el 80286 que opera con un word de 16 bits

movimientos principales (figuras 1.23 a 1.26), con los objetivos que se indican, que como se verá son *comunes en general a todas las instrucciones*. Para las operaciones de lectura de la memoria principal (MP), debe tener presente el esquema de la figura 1.10. Comenzaremos con $\mathbf{I}_1$ (figura 1.23).

1. <u>Movimientos para direccionar y obtener el código de la instrucción en el registro RI</u> (*Igual para cualquier instrucción*)

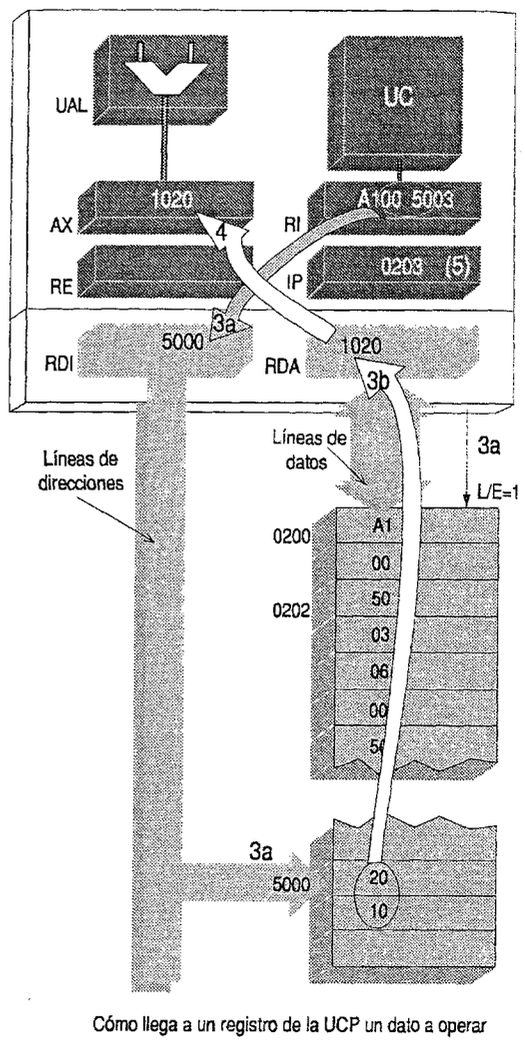
a. La UC pone en **1** la línea L/E (lectura), y ordena enviar al registro RDI una copia de la dirección 0200 H = 0000 0010 0000 0000 *que indica el IP*. De este modo dicho número, de 16 bits, llegará a M a través de 16 líneas de dirección del bus (una línea para cada bit).

b. La MP envía juntos los contenidos de la posición direccionada y de la siguiente (0200 y 0201), o sea en el caso el número A1 00, que en binario sería 10100001 00000000. Estos 16 bits van por las 16 líneas de dat del bus, hacia el registro RDA[1], y de éste al RI. Luego siguen la misma ruta los contenidos 50 y 03 de direcciones 0202 y 0203, como se planteó más arriba. En consecuencia, al cabo de estos movimientos, en RI existirá en binario la combinación que en hexa es A1005003, que corresponde al código de máquina la instrucción pedida ($\mathbf{I}_1$ en este caso).



Cómo llega a la UCP el código de una instrucción

Figura 1.23



Cómo llega a un registro de la UCP un dato a operar

Figura 1.24

[1] Es importante notar que ***en una lectura de MP, los datos de las posiciones leídas permanecen intactos, y una copia de los mismos reemplaza a los que existían en el registro de destino, los cuales se pierden.***
O sea que una lectura de MP implica una escritura en un registro de la UCP.
Esto es semejante cuando en un calculador se pulsa la tecla RM, y una copia de lo que está memorizado pasa al visor, perdiéndose el número que éste contenía anteriormente.

2. **Decodificación** (Determina los próximos movimientos a realizar por la UC para ejecutar la instrucción que está en RI y *ocurre para todas las instrucciones*)

Cuando un código de máquina (en este caso A1005003) llega al registro RI, el código de operación (en este caso **A1**) es "**decodificado**"[1] por la UC. Esto es, el código es detectado por circuitos de la UC, y *su combinación particular de unos y ceros desencadena una secuencia de acciones que ya han sido preparadas para esa combinación cuando se diseñó el procesador, a saber:*

3. **Movimientos para direccionar y leer un operando** (dato a operar), **cuyo destino es el registro RDA** (fig 1.24)
   a. La UC pone en **1** la línea L/E (lectura), y ordena enviar al registro RDI una copia de la dirección formada por los dos bytes del código de máquina que siguen al código de operación (en este caso 0050), pero traspuestos (o sea 5000), con lo cual dicho número llega a MP a través de las 16 líneas de dirección del bus.
   b. La MP envía juntos los contenidos de la posición direccionada y de la siguiente (5000 y 5001), o sea en este caso el dato 1020H. Los 16 bits del mismo llegan por las líneas de datos al registro RDA.

4. **Movimientos y acciones para cumplimentar la operación que ordena la instrucción:**
   I_1 ordena transferir desde MP hacia AX un dato. Puesto que éste ya se encuentra en RDA, sólo resta el movimiento de pasar dicho dato del RDA al registro AX (figura 1.24), donde queda almacenado.
   De esta forma se ha ejecutado lo ordenado por I_1.

5. **Movimientos y acciones para que IP contenga la dirección de la próxima instrucción a ejecutar:**
   En la UAL se debe sumar al contenido del registro IP, la cantidad de bytes que ocupa la instrucción ejecutada (en este caso 3), y reemplazar el valor anterior (0200) por el resultado de la suma (0203)[1].

La ejecución de I_2 (que ordena sumar al registro AX el dato que está en 5000H) empieza con los pasos **1a.** y **1b.** de la figura 1.23 (iguales para *todas* las instrucciones). En **1a.** será 0203 la dirección que apunta IP; y en **1b.** al registro RI llegará 0306 0050 que es el código de I_2 Además, I_2 tiene en común con I_1 el movimiento **3a.** y la dirección en este caso es también 5000 H. En el movimiento **3b.** (figura 1.25) el dato (en este ejemplo otra vez 1020H) obtenido de la lectura de la dirección 5000H llega al registro RDA.
La operación ordenada en el paso **4** ahora es sumar el operando (1020H) –que está en RDA– al dato contenido en AX (1020H). El resultado de la suma (2040H) debe guardarse en AX reemplazando al valor anterior 1020H, que se pierde (igual que una calculadora cuando se suma).
El paso **5** consistirá en cambiar el valor de IP, de modo que apunte a la dirección de I_3, para lo cual la UC debe sumar 4 (pues I_2 ocupa 4 posiciones de memoria) al valor 0203, de forma que IP indique 0207 H.

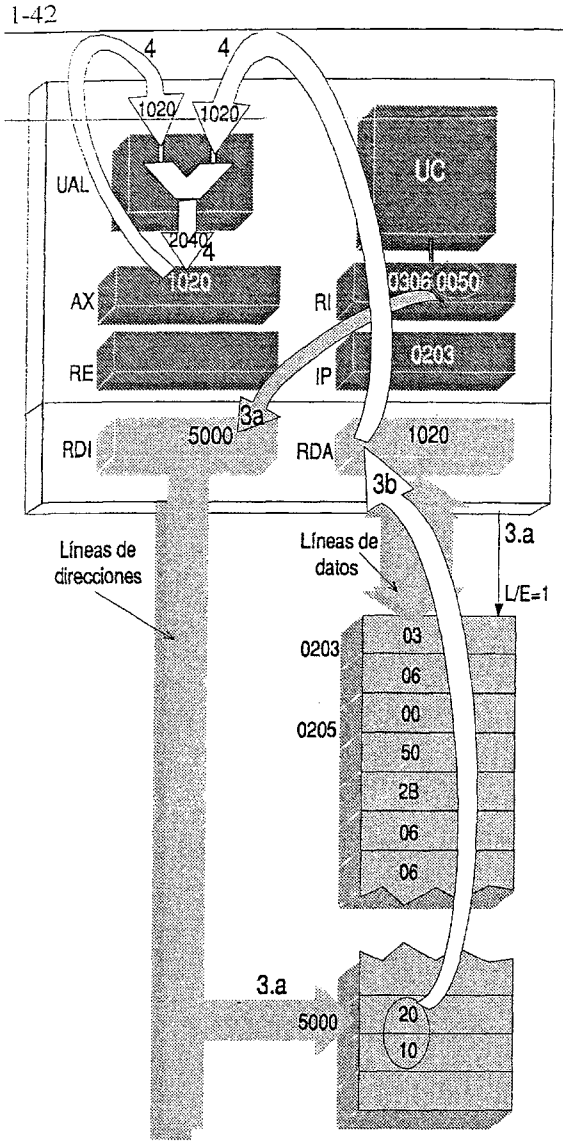
I_3 se ejecuta con los mismos movimientos que I_2 con la única diferencia que la UC ordena una resta a la UAL.[2]
Puede verificarse que mediante ellos se llega a los resultados hallados con el Debug. (figura 1.21).
Para la instrucción I_4 (que ordena guardar en 5010H el contenido de AX), luego del movimiento **1b.** (figura 1.23) se tendrá en RI su código A31050.
En el movimiento **3a.** –como en I_1– se ordena enviar al registro RDI una copia de la dirección formada por los dos bytes del código de máquina que siguen al código de operación (en este caso 1050), pero traspuestos (o sea 5010H), con lo cual dicho número llega a MP a través de las 16 líneas de dirección del bus. (fig. 1.26).
En el paso **3b.** el dato (0000 H) que se debe enviar a MP para cumplimentar la operación, pasará al RDA.
Puesto que la operación ordenada en esencia es una escritura de memoria, el paso **4.** consiste simplemente en que la UC pone en **0** la línea L/E (escritura), con lo cual enviará hacia MP, por las líneas de datos, una copia del contenido de RDA (en este caso 0000 H). Este se escribirá en las posiciones 5010 y 5011[3]. En el paso **5** el IP se actualiza en 020E (no dibujado).

---

[1] Esta acción circuital no es visualizable en la figura 1.23. Supondremos que cuando la UC lee el primer byte del código (**A1**) de una instrucción detecta cuantos bytes la componen y qué representa cada uno. Por lo tanto la UC así "sabe" que 03 no forma parte del código de la instrucción. Recordar que ésta ordenaba enviar hacia AX una copia del número contenido en la dirección 5000 (y en la 5001)

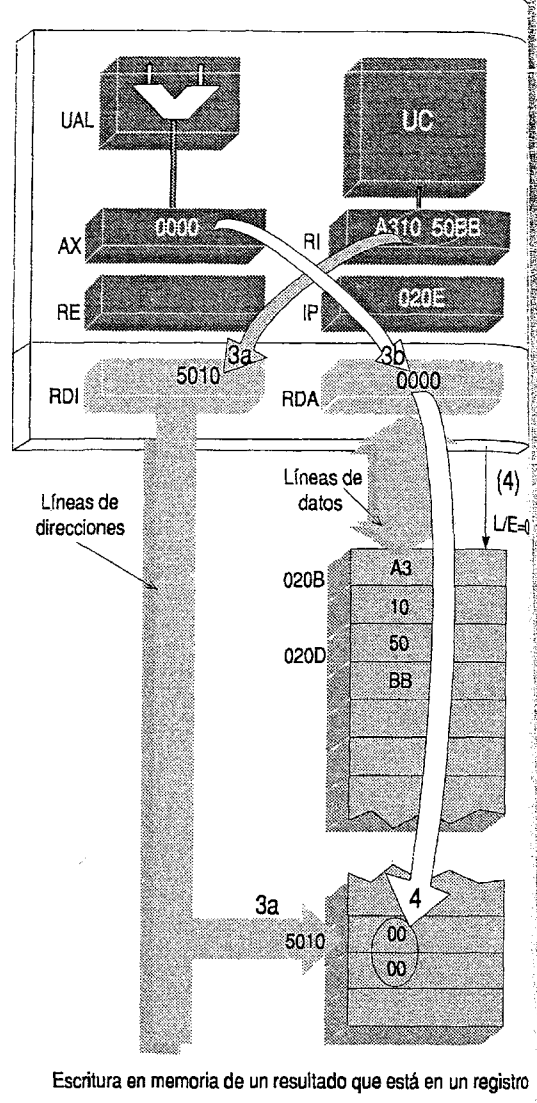
[2] En los pasos 3b de las instrucciones I_2 e I_3 tiene lugar una lectura de memoria seguida de una operación aritmética.

[3] En general, *en una operación de escritura en MP (o en cualquier registro), se destruye el contenido que tenía antes la posición escrita, la cual pasa a almacenar el nuevo valor escrito. Los datos leídos en el registro de origen, cuya copia fue escrita en MP (destino), permanecen intactos.* Asimismo, *una escritura de MP supone una lectura de un registro de la UCP.*



Cómo se suman dos números en la UAL

Figura 1.25



Escritura en memoria de un resultado que está en un registro

Figura 1.26

¿Qué secuencia de pasos ordena la UC para ejecutar cada instrucción ?

Si recapitulamos (figuras 1.23 a 1.26) cómo se ejecutaron las instrucciones en el esquema de UC supuesto, resulta que la estructura de la UCP está pensada para que repita *permanentemente* la siguient secuencia de pasos, con las instrucciones del programa a ejecutar que está en memoria principal (MP):

1. **Obtener** (direccionar) **la instrucción a ejecutar de la memoria principal**[1]**:**
   El IP indica la dirección de MP donde comienza el código de máquina[2] de l instrucción a ejecutar, el cual luego de ser leído de MP llega al registro RI
2. **Decodificar:**
   El código de operación indica: la operación a realizar, cómo encontrar un dat a operar, y la cantidad de bytes que tiene la instrucción, para que la UC lleve cabo la secuencia de movimientos preparada para ejecutar dicho código

---

1 En inglés "**fetch**".

2 Por ejemplo en la instrucción I_1 el *código de máquina* era **A10050** (3 bytes), siendo su *código de operación* **A1**, y los 2 byte restantes **0050** permiten formar la dirección 5000 donde está el dato a operar. I_2 es una instrucción cuyo código de máquina es de bytes, y su código de operación es de 2 bytes. En general los bytes que siguen al código de operación permiten determinar la direcció de MP o el registro dónde está el dato a operar, o dónde escribir un resultado.

3. Obtener un dato a operar:
   - **3a.** Si el dato está en MP, con una dirección que resulta del código de máquina de la instrucción, se direcciona la MP para obtener un dato a operar ("operando")
   - **3b.** Dicho dato llega al registro RDA (lo mismo en una escritura en MP).

4. **Realizar la operación ordenada y almacenar el resultado:**
   Según lo ordenado, puede tener lugar una operación en la UAL y almacenar el resultado en un registro, o consistir la operación en un simple movimiento de un registro a otro, donde queda guardado un dato.

5. **Cambiar el contenido del registro IP, para que tome la dirección de la próxima instrucción a ejecutar, y vuelta al paso 1.** (El cambio del contenido de IP puede hacerse junto con el paso 3.)

Las etapas o pasos citados –sintetizados en la figura 1.27– describen, al igual que las figuras 1.23 y 1.24, el **ciclo de una instrucción**, que puede dividirse temporalmente en una fase de *obtención de la instrucción* seguida de otra fase de *ejecución*.

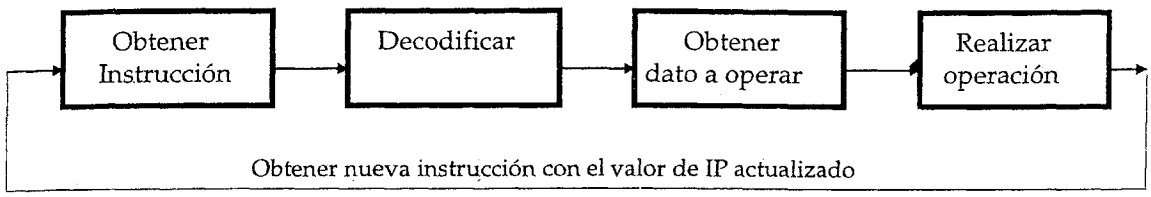


Figura 1.27

¿Cómo hace la UC para no equivocarse con tantos números contenidos en memoria que pueden ser instrucciones, datos o direcciones ?

En memoria principal existen almacenadas combinaciones de unos y ceros, números binarios que pueden representar códigos de instrucciones, datos o direcciones. El procesador "no sabe" con cuál de estos tipos de información está tratando, pero el orden, la secuencia repetitiva que realiza –descripta en la respuesta anterior– ha sido perfectamente planeada para que no existan problemas de interpretación al respecto.
Este orden empieza cuando se enciende un computador, pues lo primero que pide la UCP de la MP es un código de máquina, el que corresponde a la primera instrucción del primer programa a ejecutar[1].
La dirección de dicha instrucción está preestablecida, y pertenece a la porción ROM de MP, por lo que al encenderse el equipo el número de dicha dirección siempre debe formarse en el IP (y en el registro CS[2]). Luego se suceden en orden los 4 pasos descriptos en la respuesta de la pregunta anterior. De esta forma, lo primero que recibe la UCP de MP es el código de máquina de una instrucción, que irá al RI.

Por lo tanto, *un computador está pensado para que la UCP comience a operar leyendo de MP un número que debe ir al registro de instrucción (RI), por lo que dicho número será interpretado como un código de una instrucción.*

Conforme se estableció antes, luego de decodificar el código de una instrucción (paso 2), la UCP está pensada para que forme la dirección de MP donde está un dato a operar. Entonces (paso 3a), *lo siguiente que la UCP lee de MP* (o de un registro de la UCP si el código lo ordena) es *un número que es un dato*
Este dato si bien llega al RDA, no va al RI, como el código de operación, sino hacia un registro de la UCP (como el AX ejemplificado, o es conducido a la UAL para ser operado). También puede ocurrir que un número se escriba en esa dirección de MP (como en la ejecución de I_4).

Después de sumar al IP la cantidad de bytes que tenía la instrucción ejecutada, IP contendrá la dirección de la próxima instrucción a ser ejecutada, con lo cual vuelve a empezar otro ciclo, leyendo

---

[1] Como ya se describió, este programa está en la porción ROM de MP, dado que el primer programa a ejecutar debe estar siempre en memoria, aunque se apague el equipo, para poder traer del disco los programas del sistema operativo que se pierden en la porción RAM de memoria al apagar el computador. Al ser ejecutado comienza una secuencia de pasos que permiten traer trae del disco a MP otro programa, que cuando es ejecutado a su vez trae del disco a MP programas del sistema operativo.

[2] En un 80x86, el contenido del registro de segmento CS multiplicado por 16 siempre se suma a IP para formar cualquier dirección.

de MP un número que es el código de dicha instrucción, y como tal es interpretado dado que va al RI en reemplazo del código anterior, y así de seguido.

El orden establecido supone que *las instrucciones deben estar escritas en posiciones sucesivas de memoria*, y que los datos a operar por dichas instrucciones están en otra zona de memoria.
Acorde con estas "reglas de juego", si las instrucciones han sido escritas en posiciones sucesivas de memoria y su código es el correcto, no habrá problemas en lo concerniente a cómo la máquina "interpreta" cada combinación binaria que pide de memoria. Esto es así por que el orden establece que lo primero que llega es un código, el cual permite localizar otro número que será un dato, y luego nuevamente lo próximo que llegue será un código, etc.
En caso que la UC decodifique en el RI un código que no reconoce, está previsto que la UC pase a ejecutar una subrutina, que de ser necesario, por ejemplo haga aparecer en pantalla un aviso de error insalvable.

¿Qué analogía didáctica puede establecerse para visualizar la actividad básica de organizar movimientos y operaciones que realiza la UC ?

Si observamos los movimientos indicados en las figuras 1.23 a 1.26 podemos establecer ciertas similitudes con los que *un sistema de control automático de vías de trenes* (figura 1.28) realizaría entre un galpón con trenes estacionados (símil de la memoria), y los andenes de una estación de tren (símiles de registros de la UCP) vinculados por *una* única vía bidireccional (similar al bus de datos), para que cada tren vaya al destino que corresponda, según una cierta planificación establecida.
Desde el centro de control se comandaría, por ejemplo, que un tren que está estacionado en un lugar del galpón, se dirija hacia un andén, y que luego otro tren estacionado en otro lugar se dirija a otro andén. Estos movimientos tendrían su correlato en el movimiento **1b** y **3b** de las figuras. 1.23 y 1.24. También es factible imaginar un lugar de transformación (enganche y desenganche de vagones) para formar nuevos convoyes (símil de la UAL). Por ejemplo, un tren que estaba en un andén sería conducido a ese lugar para ser acoplado, total o parcialmente, con otro que viene del galpón formándose un nuevo tren que luego iría al andén de donde partió el primero de los trenes citados. Algo semejante ocurre, cuando durante la ejecución de un programa procesador de texto se unen los caracteres de dos párrafos para formar uno nuevo.

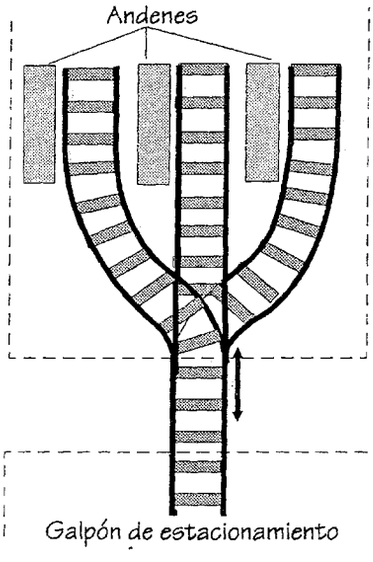


Figura 1.28

La función de la UC de encaminar datos hacia un registro de destino, puede apreciarse en este modelo "ferroviario", en el cambio de vía que debe realizarse, para que un tren que viene desde el galpón de estacionamiento –por la única vía de comunicación con la estación– vaya hacia el andén de destino.

Dentro del microprocesador la UC permanentemente –mediante llaves electrónicas (transistores) que comanda– está abriendo y cerrando caminos eléctricos (buses) internos[1], para habilitar en cada movimiento previsto el camino que permita *encaminar* datos del registro de origen al registro de destino, deshabilitando los restantes caminos. Como se describirá, el control de estos movimientos lo realiza la UC mediante líneas que salen de ella hacia los buses internos, registros y memoria (cable de lectura/escritura), al ritmo de los pulsos que genera el "clock".

Esta analogía también permite visualizar que el bus que comunica memoria con el microprocesador sólo permite un envío por vez, en un sentido u otro. También la vía principal de la figura 1.28 sólo permite que circule por ella un solo tren por vez, sea de un andén a un lugar de la playa o en sentido inverso.
Este modelo que pone de relieve la función de la UC de abrir y cerrar caminos eléctricos mediante líneas de control que salen de ella, puede también servir para aclarar ciertas asociaciones erróneas en torno a la palabra "control" que caracteriza a la UC.[2]

---

1 En inglés *"data paths"*

2 Si bien la semejanza realizada vale en cuanto a los movimientos ordenados, es importante notar una diferencia sustancial en relación con los procesos de datos. Según se vio, por ejemplo si se lee la memoria, una *copia* del dato direccionado es la que llega al registro de destino. A diferencia, un tren "desaparece" del lugar donde estaba estacionado cuando va hacia algún destino

En primer lugar, en las figuras citadas resulta que **ni datos ni instrucciones entran a la UC**, sino que van a registros, encargándose la UC de que ello ocurra habilitando en cada caso los caminos correspondientes.
La UC no se encarga de controlar si un dato llegó correctamente, sin bits errados, a la UCP, o si un resultado de la UAL es correcto[1] o no, dado que *los datos no van a la UC*, sino directamente a registros asociados a la UAL. Lo mismo pasa con la materia prima en la figura 1.3
Asimismo, cuando un código de instrucción llega al registro RI, la UC determina qué ordena ella, para poner en marcha los movimientos preparados para ejecutarla. Sólo si el código no corresponde a ninguna instrucción, se interrumpe la ejecución del programa en curso y se pasa a ejecutar una subrutina preparada para tal eventualidad. Del mismo modo, en el modelo "ferroviario", la oficina de control de vías no tiene por objetivo controlar cuántos vagones tiene cada tren que pasa, o la carga que lleva.

¿Qué relación existe entre los movimientos que ocurren durante la ejecución de una instrucción y el reloj de sincronismo del procesador ?

Anteriormente afirmamos que los movimientos que componen la ejecución de cada instrucción se realizan *en sincronismo* con impulsos eléctricos que se suceden *regularmente*, a razón de millones de ellos por segundo, generados por un cristal piezo-eléctrico de cuarzo, denominado "*clock*" ("reloj") Profundizaremos más este tema, suponiendo que se generan 50 millones de impulsos por segundo.

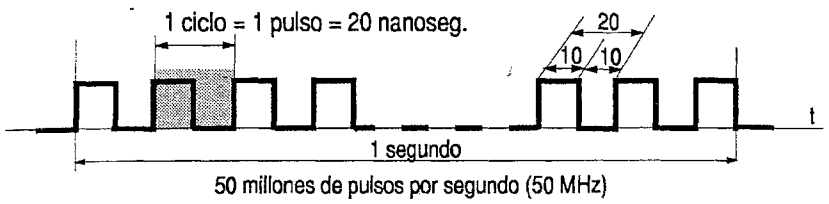


Figura 1.29

Si dichos impulsos se visualizan con un instrumento apropiado, como un osciloscopio electrónico, tienen una forma de onda como indica la figura 1.29. Se trata de una señal eléctrica que pasa cíclicamente por dos niveles denominados "*bajo*" y "*alto*", cada uno de una duración fija. Esto implica que si midiésemos la señal que sale de dicho cristal oscilador, por ejemplo, durante diez nanosegundos (1/100 de millonésima de segundo) se tendría 0 volts (nivel bajo), y en los siguientes diez nanosegundos, 5 volts (nivel alto).
Por definición, un ciclo tiene lugar cada vez que se repite un mismo fenómeno. En la figura se ha indicado un ciclo del reloj, considerado desde que comienza un nivel bajo, hasta que dicho nivel se inicia de nuevo, con un nivel alto (pulso) en el ciclo. O sea que en un ciclo tiene lugar un pulso.
En el ejemplo, un ciclo o pulso se repite 50 millones de veces por segundo, por lo cual se tiene una frecuen-cia de repetición de 50 millones de Hertz = 50 Megahertz = 50 **MHz**, siendo 1 Hertz = 1 ciclo por segundo.

Entonces, *hablar de megahertz (o megahercios) es lo mismo que hablar de millones de pulsos por segundo, y es lo mismo que decir millones de ciclos por segundo.*

Son comunes los microprocesadores con reloj de 100 Mhz a más de 1 Ghz. El reloj está incorporado al micro-procesador. En general, un procesador será más rápido si funciona a más MHz[2]

Como se planteó, *estos pulsos marcan, sincronizan, los instantes en que comienzan los movimientos que tienen lugar durante la ejecución de cada instrucción.* Vale decir, que un movimiento empieza al comienzo de un pulso y tiene tiempo de consumarse hasta que el inicio del pulso siguiente, cuando comienza otro movimiento.

Con un reloj de 50 MHz de reloj, un movimiento debe concretarse durante un ciclo reloj, siendo que éste dura **1/50 millonésima de segundo**, para el caso que ocurran 50 millones de ciclos por segundo.
En la fig 1.30 se han indicado los movimientos que empiezan en consonancia con cada pulso, para las instrucciones $\mathbf{I}_1$ e $\mathbf{I}_2$ antes ejecutadas (ver figs 1.23 y 1.24), suponiendo que con cada pulso tenga lugar un paso de la ejecución de una instrucción, y que se requiere 4 pulsos (pasos) para ejecutar cada una de dichas instrucciones.

Resulta así que, en general, *una instrucción requiere para su ejecución varios pulsos reloj.*

---

[1] Al encender un computador la ejecución de un programa de diagnóstico que está en ROM lleva a cabo una serie de verificaciones en el hardware, en relación con el correcto funcionamiento de la UAL ,y la memoria, entre otros.y la configuración del sistema entre otros.

[2] La frecuencia de los pulsos reloj no sirve para compara la performance de procesadores *distintos*.

¿De qué forma la UC pasa de un movimiento a otro abriendo y cerrando caminos?

En la figura 1.23 se tiene que para leer en memoria el código de máquina de una instrucción a ejecutar, en movimiento **1a.** una copia del contenido de IP pasaba a RDI. Esto lo ordena la UC *mediante una línea de control que sale de ella* (figura 1.31) que habilita –por estar en **1**– el camino ("bus") que une IP con RDI. Los contenidos de memoria llegan al registro RDA pasando por el bus de datos que comunica ambas.
Este movimiento de lectura es ordenado por la UC *mediante su línea de control L/E* (de lectura/ escritura *que va de la UC a memoria*, para lo cual dicha línea de control debía estar en **1** (5 volts).
Una vez que dichos contenidos leídos llegaron al RDA, tendrán como destino el registro RI, dado que se tra de un código de instrucción. Para ello, la UC habilitará *mediante otra línea de control* de valor **1** el camin que une RDA con RI. Obsérvese en el esquema propuesto, que las restantes líneas de control que salen de UC por estar en **0** (0 volts), no permiten la transferencia de datos entre otros registros de la UCP, por cortar comunicación entre los caminos (buses) que los unen.
De manera análoga (figura 1.32), en la lectura del dato a operar, las líneas de control que están en **1** permite los movimientos **3a.**, **3b** y **4**, que aparecen en la figura 1.24. Estos movimientos son ordenados por las líne de control que en la figura 1.32 están en **1** (que en la figura 1.31 estaban en **0**), las cuales habilitan l correspondientes caminos entre registros.
También salen líneas de control de la UC hacia la UAL, para ordenar sumar, restar u otra operación.
Si el código de una instrucción (como el de I_4), ordena un movimiento de escritura en memoria, cuando tenga lugar mismo, un cable de control de la UC habilitará el camino que va de un registro acumulador hacia el registro RDA, y cable L/E que va a memoria deberá estar en 0 volts, para que se escriba la posición de memoria previamen direccionada, en correspondencia con los movimientos de la figura 1.26.

Por lo tanto, de la UC sale un conjunto de "líneas de control" que van
1. hacia la UAL
2. hacia los caminos entre registros de la UCP
3. hacia la memoria (línea de lectura/escritura -L/E) y hacia los ports de las interfaces (figuras 1.61 y 1.62)

Según el valor (1 ó 0) de estas líneas la UC ordena la operación que hace la UAL, de qué registro a cual otr se pasará la información, y si la memoria será leída o o escrita.

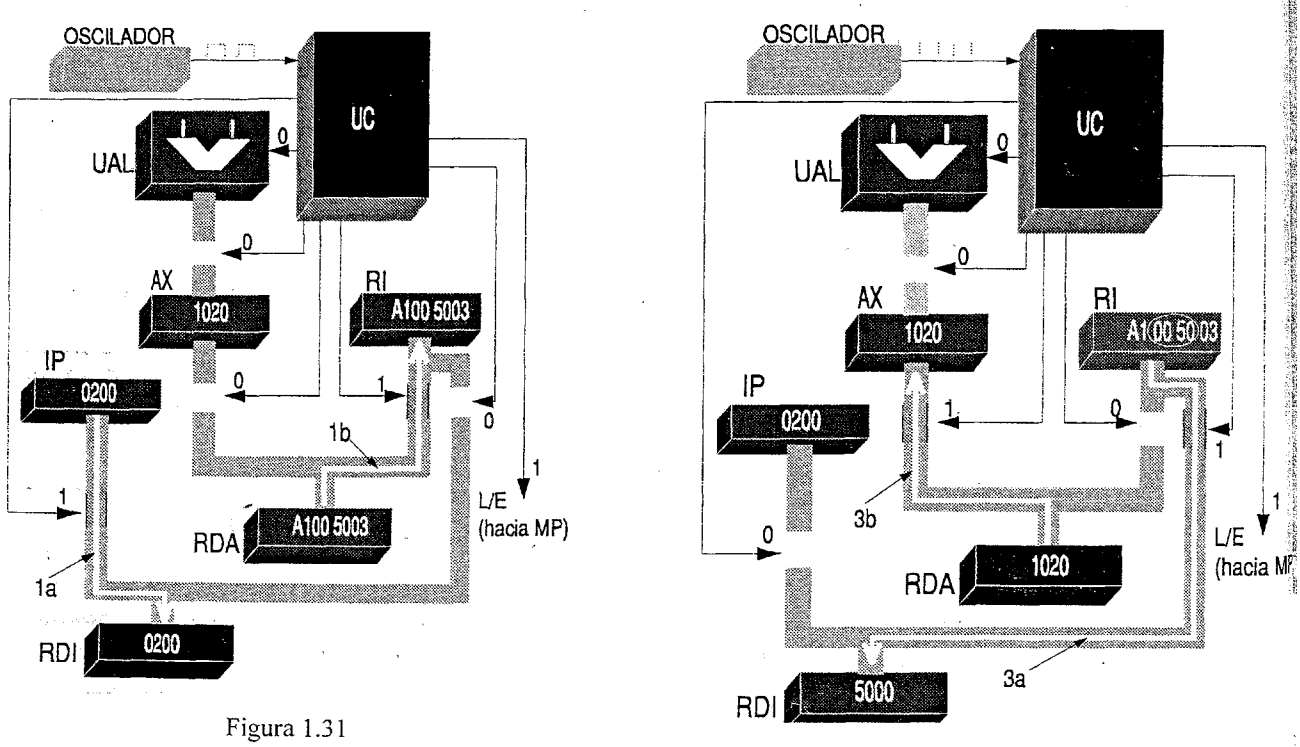


Figura 1.31

De esta forma se realiza la acción de control de la UC, mediante las líneas que salen de ella, a fin de lograr los movimientos y operaciones necesarios para ejecutar cada paso de una instrucción. Es como un "director de orquesta" que ordena en cada momento qué instrumentos deben ponerse en juego.

En la UCP dichos momentos son iniciados, sincronizados, por cada pulso reloj.
Puesto que –como se trató– cada movimiento se lleva a cabo en un pulso reloj, cada uno de las líneas que salen de la UC puede cambiar de **0** a **1** ó viceversa en cada pulso reloj.
Por ejemplo, si durante un pulso reloj la línea que en el movimiento **1b** habilita la escritura de RI desde RDA (figura 1.31), en el pulso siguiente debe inhabilitarla (cambiando de **1** a **0**), para que el dato no vaya a RI, sino a AX, como aparece en la figura 1.32.

> Por consiguiente, ***con cada pulso reloj avanza un movimiento o paso la ejecución de una instrucción, y cambia la combinación de unos y ceros presente en las líneas que salen de la UC,*** *a fin de que puedan llevarse a cabo dicho movimiento.*

Cada vez que se repite un determinado movimiento –como el **1a** ó el **1b**– se repite también en las salidas de la UC la combinación de unos y ceros que determina (controla) dichos movimientos

¿ Dónde reside la "inteligencia" de la UC, para "saber" los movimientos y operaciones en la UAL a realizar; y cómo localizar cada microcódigo ?

Esta pregunta equivale a plantear *de dónde sale cada combinación de unos y ceros que aparecen con cada pulso reloj en las líneas de salida de la UC.*

Según se describió (fig. 1.27) la ejecución de cada instrucción se divide en **pasos** aún más simples (4 en nuestro caso, pero que son más en instrucciones complejas). Las acciones que debe ordenar/controlar la UC en cada uno de estos 4 pasos están determinadas por 4 combinaciones binarias llamadas **"microcódigos" (µcod)** que van apareciendo una tras otra con cada .pulso reloj (Clock = **Ck**) en las **líneas de control (LC)**. Estas salen de la UC con destino a la UAL, los registros de la UCP, y laa memoria (fig. 1.31). También van hacia los ports.
Con cada **Ck** el valor (1 ó 0) de cada **LC** que sale de la UC determina los movimientos (como ser de **IP** a **RDI**) que deben tener lugar, y si interviene la UAL, qué operación debe hacer. El valor de cada **LC** puede cambiar con cada **Ck**. Así, para 500 Mhz (500 millones pulsos/seg) las **LC** cambian 500 millones de veces por segundo, o sea que se generan en ellas 500 millones de **µcod**/seg.)
Este funcionamiento es común a *todos* los procesadores, sean CISC o RISC (sección 1.14).

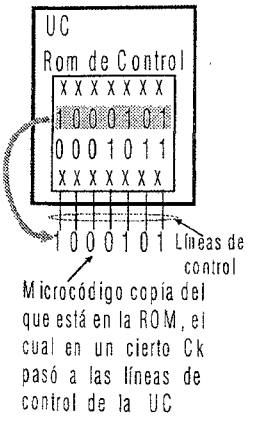


Figura 1.33

En un CISC las salidas de la UC, o sea las **LC**, son salidas de una ROM denominada ROM de Control [1] (**RC**), que contiene escritas en su interior todas las combinaciones binarias que pueden aparecer en las **LC** para determinar qué debe hacer la UC en cada paso de la ejecución de una instrucción. Ello implica que en la **RC** reside la "inteligencia" de la UC, que obviamente fue originada por quienes crearon la UCP.
En general de la UC salen *n* **LC** (como ser $n = 100$), por lo que cada **µcod** será de *n* bits (un bit por cada **LC**), y está guardado (grabado) en una sola celda de *n* bits de la **RC**. O sea que en la **RC** las celdas no son de 8 bits, sino de *n* bits (figura 1.34).
En la figura 1.31 se supone que durante un cierto **Ck** los valores de las 7 **LC** supuestas que salen de la UC son 1000101, y en la fig. 1.32 se asume que con el **Ck** siguiente dichos valores son 0001011. Ambos valores en la figura 1.33 aparecen guardados en 2 celdas sucesivas de 7 bits de una **RC**. Como en cualquier ROM, cuando se accede a una celda una copia de su contenido (**µcod**) pasa a sus líneas de salida, que son las **LC** de la UC. Esto sucede con cada **Ck**.
Con xxxx se indican otros **µcod** en la **RC**, que como todos los **µcod** constan de unos y ceros.
Cuando el **µcod** 1000101 está en las **LC** permite realizar los movimientos de la figura 1.31

El esquema siguiente generaliza la fig. 1.33 e intenta acercarse conceptualmente a cómo se localizan los µcod en una ROM de Control. Sobre la base de la fig 1.33, y suponiendo que cada una de las instrucciones de la fig 1.15 se ejecute en 4 pasos indicados en la figura 1.30 en concordancia con 4 pulsos, hacen falta 4 **µcod** (**µcod1**, **µcod2**, **µcod3** y **µcod4**) que deberán aparecer en las **LC** con cada pulso, para indicarle a la UC qué hacer en cada uno de los 4 pasos de una instrucción. Con **Ck1**, **Ck2**, **Ck3**, **Ck4**, designamos a cada uno de los 4 pulsos necesarios para que avance un paso la ejecución de cada instrucción, ya sea I_1, ó I_2 , ó I_3 ó I_4 (fig. 1.15).
En las celdas de la **RC** de una UCP CISC se guardan los **µcod** para pedir y ejecutar cada instrucción de su repertorio.

---

[1] También llamada **ROM de microcódigo** o de **microinstrucciones**, inmodificable y forma parte del chip del procesador **Esta ROM no tiene nada que ver con la ROM de memoria principal**, que contiene los programas de arranque y el BIOS (Basic Input Output System) en una PC, **ni tampoco tiene que ver con el sistema operativo elegido para un computador**. Esta concepción circuital se empleó en procesadores y microprocesadores CISC de distintos fabricantes, inclusive hasta el Pentium 1, prevaleciendo luego la concepción RISC.

Puesto que una **RC** es una memoria random -cuyas celdas guardan tantos bits como **LC** existan- *cada celda de* **RC** *se localiza por su dirección.* Cada **µcod** proveerá la dirección del siguiente, salvo la dirección del tercero (**µcod3**), que se determina en la decodificación a partir del cod-op de la instrucción que llegó al **RI**.
Como se tratará, *dado que en la* ***RC*** *existen miles de* ***µcod*** *para ejecutar todo el repertorio de instrucciones de una UCP, la localización de los sucesivos* ***µcódigos*** *para ejecutar la instrucción que llegó al RI y luego pedir la siguiente, se realiza siempre a partir de la localización del* ***µcod3****, merced al cod-op de dicha instrucción.*

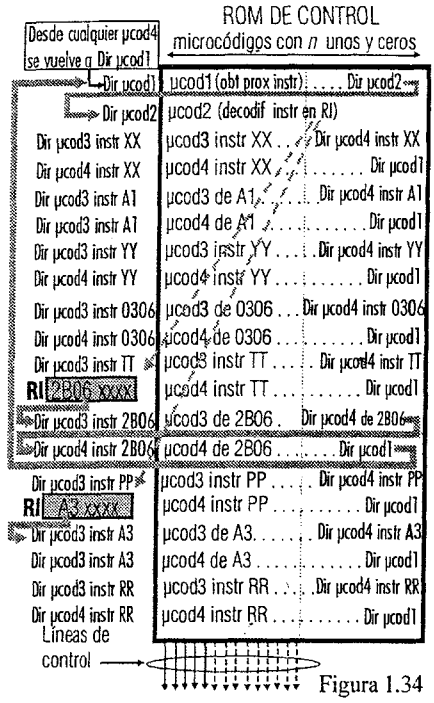


Figura 1.34

Supondremos que después de los pulsos **Ck**1 y **Ck**2, en las **LC** han aparecido **µcod**1 y luego **µcod**2, con lo cual llega a **RI** (fig. 1.34) la instrucción de cod-op 2B06; y que en la decodificación dicho cod-op 2B06 que está en **RI** permite localizar la dirección del "**µcod**3 de 2B06" en la **RC**. O sea que cuando llegue Ck **los 0s y 1s** del **µcod**3 de 2B06 al aparecer sobre las **LC** abrirán y cerrarán los caminos que corresponda mediante las llaves que los gobiernan, para que se lleve a cabo el paso 3 de 2B06: direccionar el dato y llevarlo a RDA.
Asimismo, un subconjunto de bits del **µcod**3 de 2B06 no van a las **LC** sino que proveen la dirección del **µcod**4, supuesto en la dirección que sigue a la del **µcod**3.
Cuando se genere el pulso **Ck**4, el **µcod**4 de 2B06 aparecerá en las **LC** determinando que se lleve a cabo el paso 4: realizar la resta ordenada por dicho cod-op 2B06.
A su vez, un conjunto de bits del **µcod**4 de 2B06 no van a las **LC**, sino que proveen la dirección (Dir µcod1) del **µcod**1 para pedir de memoria la siguiente instrucción (de cod-op supuesto A3). Si al pulso que sigue a **Ck**4 lo volvemos a llamar **Ck**1, al ocurrir este pulso el **µcod**1 aparecerá en las **LC** para que se lleve a cabo este paso 1 del pedido.
Del mismo modo, bits del **µcod**1 que no van a las **LC** proveerán la dirección (Dir µcod1) del **µcod**2 que para cualquier instrucción sigue al **µcod**1, supuesto en la dirección que sigue a la del **µcod**1. El **µcod**2 permitirá que el cod-op A3 de la instrucción supuesta ahora en **RI** sea decodificado, para localizar el **µcod**3 de A3, *como sucedió antes* con 2B06, dado que el **µcod**2 no indica la dirección del **µcod**3 de cada nueva instrucción a ejecutar.
De esta forma, luego de ejecutar cada instrucción (con **µcod**3 y **µcod**4), se pide la siguiente (con **µcod**1 y **µcod**2), para seguir con los **µcod**3 y **µcod**4 que determinan la ejecución de esta siguiente instrucción, y **así sucesivamente**. En cada secuencia **µcod**3 =>**µcod**4 =>**µcod**1 =>**µcod**2 la dirección de cada **µcod** la provee el **µcod** anterior, siendo que la dirección de **µcod**3 la provee el cod-op de la instrucción que llegó a RI, como lo señalan las dos simbolizaciones del registro RI en grisado en la figura 1.34.
Los dos primeros pasos (obtener instrucción en RI y decodificarla -fig 1.31) como se trató, **son comunes a todas las instrucciones**, por lo que los **µcod**1 y **µcod**2 serán compartidos por todas las instrucciones.
Tanto **µcod**3 de XX como **µcod**4 de XX hacen referencia a **µcod** de otra instrucción que puede ejecutar la UCP.

Sistematizando:

Sentencia (R = P+P-Q)

- I1: µcod1, µcod2, µcod3 de A1, µcod4 de A1
- I2: µcod1, µcod2, µcod3 de 0306, µcod4 de 03061
- I3: µcod1, µcod2, µcod3 de 2B06, µcod4 de 2B061
- I4: µcod1, µcod2, µcod3 de A3, µcod4 de A3

Cada sentencia (por ej. R = P + P - Q) de un programa de alto nivel se traduce en una secuencia de instrucciones (I_1, I_2, I_3, I_4 en este caso), y cada instrucción se ejecuta mediante una secuencia de **µcódigos** que en un CISC están en una ROM de Control, y que aparecen en las líneas de control al ritmo de los pulsos reloj. Ellos determinan las acciones que debe realizar la UC en cada paso de la ejecución de una instrucción.

Cuando el código de operación de una instrucción llega al registro RI, el mismo permite ubicar (en este ejemplo) la tercera de dichas combinaciones que debe aparecer en las líneas de la UC para que comience la ejecución de dicha instrucción

Este conjunto de combinaciones constituyen la "inteligencia" de un computador, habiendo sido *preparadas por el hombre*, para que la UC pueda ejecutar cada una de las instrucciones de máquina que forman parte de cualquier programa contenido en memoria principal, sea de usuario o del sistema operativo.
Los procesadores que tienen dichas combinaciones almacenadas en una ROM de Control se dice que son "máquinas **microprogramadas**".

Todo sucede como si la UC fuera como un robot preprogramado, para que con cada orden distinta que se le imparta (instrucción) responda mediante una serie de acciones específicas (ordenadas por las combinaciones que se generan en sus líneas de salida), programadas en su interior por sus creadores.

1.8 UAL: OPERACIONES LOGICAS, DE COMPARACION Y "FLAGS" EL COPROCESADOR MATEMATICO

¿Cuáles son las operaciones lógicas que realiza la UAL, y cómo se comparan números en un computador por medio de ella ?

La denominación "**Lógica**" de la UAL se debe a que también efectúa operaciones **lógicas**, como la **AND** ($\wedge$), **OR** ($\vee$) y **la negación** ($-$). Es como la calculadora que brindan ciertos sistemas operativos, que además de tener las teclas corrientes tienen otras con los símbolos $\wedge$ $\vee$ $-$ que indican dichas operaciones lógicas. Los valores *"verdadero"* y *"falso"* se representarían por **1** y **0**.

La operación **AND** queda definida así: **0∧0 = 0** **0∧1 = 0** **1∧0 = 0** **1∧1 = 1**
Vale decir si en el visor hay un **0**, y se pulsa la tecla $\wedge$ seguida de la tecla **1**, al pulsar = resulta un **0** en el visor, etc.
También pueden entrarse más bits: como ser primero entrar al visor 10010110, después apretar la tecla de operación $\wedge$ luego entrar la combinación 00000010. Al apretar la tecla = resultaría 0000010, como puede verificarse haciendo la operación AND columna por columna, conforme ella fue definida, disponiendo los operandos como sigue:

```
  10010110
∧ 00000010
  00000010     esto mismo es lo que hace en esencia la UAL para hacer la operación lógica AND
```

Aprovecharemos la operación realizada para mostrar cómo puede usarse ella en la práctica.
Supongamos que se realiza una encuesta con ocho preguntas que se contestan por "si" (1) o por "no" (0). Entonces la combinación 10010110 representaría las 8 respuestas de una persona. Si ahora se desea conocer entre todos los encuestados, cuántos contestaron por el "no" a la anteúltima pregunta, habría que detectar en cada grupo de 8 respuestas almacenadas (como la 10010110 anterior), si el bit anterior al de la extrema derecha es 0 ó 1. De ocurrir lo primero, el programa encargado de este procesamiento hará incrementar en uno un determinado registro que oficiará de contador.
Para detectar si el anteúltimo bit citado vale 0 ó 1 (supuesto desconocido) hacemos la operación AND con la combinación 00000010 (llamada *"máscara"*). Puesto que en este caso dicho bit vale 1, se obtuvo un resultado distinto de cero. Cualquier otra combinación que tenga ese bit en 1, dará como resultado 00000010, como puede verificarse. Si el bit en cuestión valiera 0, la combinación que representa las respuestas sería 10010100. Haciendo la misma operación AND con la máscara 00000010, esta vez se obtendrá 00000000, en lugar de 00000010. Lo mismo sucederá para cualquier combinación que tenga 0 como anteúltimo bit. Entonces, si el resultado es cero implica que dicho bit es 0 ("no"). Si el resultado es distinto de cero implica que ese bit vale 1 (respuesta "si"), con lo cual se debe incrementar en uno el contador de respuestas positivas citado.

Habiéndose ejemplificado una operación lógica, debe quedar bien en claro que **la UAL es un simple circuito calculador** que realiza automáticamente operaciones aritméticas o lógicas. *La UAL no tiene "inteligencia" de tipo deductivo como puede insinuar su denominación "lógica".*

Para **comparar** dos números A y B –a fin de saber si A es menor, igual o mayor que B– en la UAL **se resta** A–B. La indicación de la UAL de resultado negativo, cero, o positivo, permitirá conocer (suponiendo que son enteros), como es A respecto de B.

Cuando nos presentan dos números escritos, basta con observarlos para darnos cuenta en forma mecánica por los símbolos que los componen, cuál es el mayor, por la experiencia que hemos acumulado en cuanto al número de unidades que representan. La UC no "sabe" que está operando números, ni el valor de una combinación binaria.

El método usado en un computador para conocer, por ejemplo, si un número binario es el 3192, es restarle 3192. Si el resultado es cero, será dicho número, caso contrario no. La UAL tampoco determina como es A respecto de B, sólo efectúa A – B e indica si el resultado fue cero o no. Se necesita luego considerar esta indicación para tal determinación.
Por lo tanto, la **comparación es una operación aritmética de resta**, no lógica, siendo que las otras operaciones aritméticas que realiza la Unidad Aritmético Lógica son la suma, la multiplicación y división de números enteros y naturales.

¿Qué son los indicadores ("flags") de resultado generados por la UAL, contenidos en el Registro de Estado de la UCP?

Al tratar los registros de un computador se estableció que junto con el resultado de una operación la UAL genera –*mediante indicadores ("flags") que pueden valer uno o cero*– un pequeño "resumen" de las características del mismo, que puede ser o no utilizado por la instrucción siguiente a la que ordenó dicha operación.
Estos indicadores constituyen **el Registro de Estado**. *Son esenciales para estipular* (mediante una instrucción de "salto"), *en función del valor de uno o varios de ellos, la condición para que la UC pase a ejecutar otra, que no es la que sigue a continuación en memoria. La dirección de esta instrucción debe ser indicada en la instrucción de salto.*

Esta indicación es semejante a la que aparece en el visor de un calculador, cuando al efectuar una resta de números naturales, aparece el resultado con un signo negativo, indicador que el minuendo es menor que el sustraendo. Si este signo no aparece deducimos que el resultado es positivo. Del mismo modo, si al hacer una resta aparece un cero implica que los números restados son iguales; y si el resultado es distinto de cero, que son distintos. Otro indicador aparece cuando el resultado sobrepasa el valor máximo que se puede representar. Si el mismo no aparece, asumimos que el resultado está bien.
A continuación se definen tres indicadores importantes que genera la UAL, de los cuales en esta unidad daremos un ejemplo de utilización del primero en una instrucción de salto, condicionada al valor del mismo

El indicador **Z** (de "zero") vale **1** (ZR) si el resultado fue cero, y vale **0** si el resultado *no* fue cero (NZ).

El indicador **S** (de signo) para enteros vale **1** (NG) si el resultado fue negativo, y valor **0** si el resultado fue positivo (PL)

El indicador **V** (de "overflow") vale **1** (OV) si el resultado como número entero supera el máximo valor representable; caso contrario vale **0** (NV). Este indicador y los otros se tratan en detalle en la Unidad 4.

Entre paréntesis se da la forma en que el Debug representa el valor de esos indicadores, como aparecen en las figuras 1.18 a 1.22 después de la ejecución de cada instrucción, acorde con el resultado obtenido.
En esa secuencia de instrucciones antes ejecutada, el indicador **Z** luego de ejecutar la instrucción $\mathbf{I_1}$ quedó en **0** (**NZ**– no zero, en la figura 1.20), y después de ejecutar $\mathbf{I_2}$ cambió a **1** (**ZR** en la figura 1.21)

Los flags se definen en detalle en la Unidad 4 de esta obra, así como al tratar las instrucciones de salto (sección 1.19 y en el modelo circuital del Apéndice de esta unidad), donde se ve cómo *la máquina toma decisiones en función del valor de los flags*, tema también tratado en la Unidad 3 de esta obra.

¿En qué se diferencian la UAL y el coprocesador matemático que opera con números reales representados en "punto flotante" y cómo opera éste?

La UAL sólo realiza operaciones aritméticas con números naturales o enteros, siendo que las instrucciones para sumar y restar con estos números son muy rápidas de ejecutar. Para operar en la UAL números fraccionarios, el programa que ordena las operaciones aritméticas debe controlar el lugar donde está la coma, y operar los números como si fueran enteros. Esto es lo que hacemos con papel y lápiz cuando, por ejemplo, multiplicamos dos números y luego al final ubicamos la posición de la coma en el resultado, sumando la cantidad de dígitos fraccionarios que presenta cada uno de los operandos. Estas determinaciones y otras, demoran los resultados.

El **coprocesador matemático** ("copro") es un microprocesador dedicado entre otras funciones a realizar rápidamente operaciones con números enteros y fraccionarios, encargándose sus circuitos de controlar a cada instante el lugar donde debe ir la coma. También realiza operaciones trigonométricas y logarítmicas.

De esta forma no se pierde tiempo en la ejecución de instrucciones adicionales, que lleven la cuenta de la ubicación de la coma, como se requiere si se usa la UAL.
Un programa con muchas instrucciones que ordenen operaciones con el "copro", puede obtener resultados hasta cien veces más rápido que otro programa que realice los mismos cálculos simulando números con coma y usando la UAL. Esto se hacía cuando un "copro" era costoso.
Los números enteros y fraccionarios (racionales) y los irracionales (pi, raíz de dos, etc.) constituyen los números **"reales"**. Para que el coprocesador pueda operarlos deben estar codificados en **"punto flotante"** (*"floating point"* – **FP**), que en castellano sería "coma flotante o desplazable". Por tal motivo el "copro" también se denomina "*Unidad de Punto Flotante*" (**FPU** las siglas en inglés).

Esta convención –similar a la notación científica– implica que todos los números que llegan al "copro" deben representarse de una manera normalizada, que ejemplificaremos en decimal, siendo que en realidad en el interior del computador se representa sólo con números binarios, sin comas.
Cuando en el visor de una cierta calculadora "científica" aparece 3.078 E 3 se conviene que es $3{,}078 \times 10^3 = 3078$ ó sea que desplazamos la coma tres lugares hacia la derecha. De manera inversa, partiendo del 3078, este número queda representado con la notación científica anterior compuesto por dos números, de los cuales uno es el exponente de diez.
Del mismo modo $243{,}78 = 2{,}4378 \times 100 = 2{,}4378 \times 10^2$ = 2.437 E 02 es, en decimal, una notación semejante a la de punto flotante en binario. Esta se ve en detalle en la Unidad 4 de esta obra.
Siempre será factible, corriendo la coma, representar un número *"normalizado"* de la forma anterior, con un solo dígito como parte entera, seguida de otra fraccionaria y otro número independiente que indique cuántos lugares se debe correr la coma para obtener el número originario en lugar del "normalizado".

Existen instrucciones para cada tipo de número a procesar, cuyos códigos determinan, si la operación aritmética se realizará en la UAL o en el "copro". Igualmente, dado un número binario en memoria, si el mismo se quiere imprimir en decimal, **según sea el tipo de datos que está operando el programa en ejecución** (Integers, Reales, etc.) **será interpretado** dicho número.
Del mismo modo que 260850 puede interpretarse como el número 260.850 ó como la fecha 26/08/50

Por otra parte, por ejemplo el "copro" de un Pentium permite operar en doble precisión *extendida* con números de 80 bits, mientras que la UAL del mismo puede operar dos números de 32 bits.
Esto permite realizar con un "copro" cálculos que requieran una *mayor precisión* (apreciar mayor número de dígitos), que de realizarse en la UAL requerirían tiempos adicionales, puesto que números que superen el tamaño que la UAL maneja, deben operarse por partes según este tamaño.
Con el mismo fin, almacena en su interior con muchas cifras, números como PI.

El 486DX y el Pentium I ya presentan el "copro" incorporado en su chip, (el de un Pentium es 5 veces más rápido que el del 486). En el 80286 y 80386 sus "copros" 80287y 80387 estaban en chips separados
Las siglas 80X87 se refieren a "copros" de Intel. Estos tienen estructura distinta a los de otros fabricantes.
Poseen 8 registros de 80 bits que el programador los ve como formando parte de una pila, con un registro oficiando de cima. O sea que los 80X87 son máquinas para datos apilados (existen microprocesadores de este tipo). Las instrucciones para llevar datos de memoria al "copro" los van apilando de forma que el último en entrar quede siempre en el registro que es la cima de la pila (designado ST), como en una pila de platos, y sacan los resultados de la cima. Los 7 registros debajo de la cima se designan ST(i).
Los números son más largos en los registros de la pila (80 bits) que cuando están en memoria (64 bits).
Realizan las cuaatro operaciones aritméticas básicas, calculan raíz cuadrada, exponenciaciones, valor absoluto, , logaritmos, funciones trigonométricas y trascendentales en general (o sea aquellas que no son polinómicas).
Procesan datos como números en FP (reales) de 32, 64 y 80 bits, enteros de 16, 32 y 64 bits, y datos BCD de 18 dígitos decimales. En los 8 registros citados **sólo puede haber datos traducidos a FP**, existiendo instrucciones indicadas para ello (como ser cargar en la pila en FP un número que en memoria es un entero).
Las instrucciones para el Pentium y para el "copro" pueden ser ejecutadas por ambos en forma simultánea. Cada uno intercepta y ejecuta sus instrucciones. Si la instrucción empieza con 11011 (ascii de ESC) la ejecuta el "copro". En la pila de registros citada están los operandos y se guardan

resultados. Al respecto, al final de la Unidad 4 de esta obra se da un ejercicio integrador donde ejecutan instrucciones para el "copro" usando el registro que oficia de cima de la pila.
El Registro de Estado del "copro" informa entre otras, si un resultado no está fuera de rango representación, o si un operando está mal representado, y acerca del valor de los flags del "copro" Estos últimos permiten determinar si un número es > = < que el de la cima (top) de la pila.
Un Registro de Control permite seleccionar la precisión (simple, doble) y el redondeo a utiliz Existe una instrucción del "copro" que transfiere el contenido del Registro de Estado del "copro" registro AX de un Pentium. Asimismo "copro" y Pentium se pueden comunicar a través de "port direccionables reservados, que no pueden usarse para el manejo de periféricos. Así, cuando el 38 detecta una instrucción del "copro" la pasa a éste a través de un port mediante un ciclo de E/S.
El "copro" sólo puede acceder a memoria a través del 80x86 o Pentium.
Las extensiones para multimedia (MMX™) de los actuales Pentium se realizan mediante instruccione para ese tipo de datos, las cuales se ejecutan en un coprocesador dedicado a MMX, el cu comparte los 8 registros del "copro".
En un computador pueden existir otros coprocesadores, como ser para vídeo, para Entradas/Salidas, et Un coprocesador es como una extensión de procesador central, que colabora con éste trabajando e paralelo y proporcionando registros extras.

¿Qué son los MIPS y las MFLOPS ?

Actualmente un procesador puede ejecutar millones de instrucciones por segundo. Estas últimas palabras s abrevian con la sigla **MIPS**[1]. Este dato puede servir relativamente para comparar, a una misma frecuencia d operación, la performance de un mismo procesador o de distintos procesadores entre sí, ejecutando instru ciones de tipo semejante, siendo que los MIPS de un mismo procesador varían de un programa a otro. L ejecución de distintos tipos de programas es siempre la mejor medida de comparación.
Los procesadores 80x86 de Intel aumentaron sus MIPS en promedio como sigue:

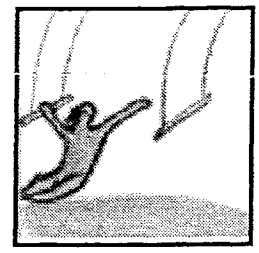
8088 y 8086: 0,33 MIPS a 5 MHz.
80286: 1,2 MIPS a 8 MHz.
80386 SX : 2,5 MIPS a 16 MHz
80386 DX: 6 MIPS a 16 MHz.
80486 SX: 20 MIPS a 25 MHz
80486 DX: 20 MIPS a 25 MHz y 40 MIPS a 50 MHz
80586 (Pentium I): 100 MIPS a 66 MHz

Un Pentium actual de 1 Ghz en promedio puede terminar de ejecutar hasta 3 instrucciones por ciclo, lo cu extrapolando implicaría unos 3000 MIPS !

Dados los requerimientos actuales de las operaciones en punto flotante, la evaluación de la performance d procesadores para las mismas, pasó a ser importante. Los **MFLOPS** –pronunciados "megaFLOPS"– (de inglés *mega floating points operations per second*) son las millones de operaciones en punto flotante po segundo que puede realizar un procesador, calculados como el cociente:
Número de operaciones en punto flotante de un programa/Tiempo de ejecución x 10^6
Conforme a esta definición –a diferencia de los MIPS– teóricamente la evaluación de los MFLOPS depend del procesador y del programa elegido. O sea que se debieran comparar los MFLOPS de distintos procesadore ejecutando una misma tarea con igual número de operaciones en punto flotante del mismo tipo, aunque un mismo programa en cada procesador en general tendrá distinto número de instrucciones.

[1] Esta unidad se acuño con los computadores IBM 370 y VAX-11-780 de DEC, que comercialmente fueron los primeros de 1 MIPS. Debe consignarse que existe la MIPS Computer Systems, Inc propietaria de las arquitecturas RISC MIPS R2000 y MIPS R3000

1.9 UTILIDAD DE LAS INSTRUCCIONES DE SALTO



¿Cómo operan las instrucciones de salto condicionado y por qué son esenciales ?

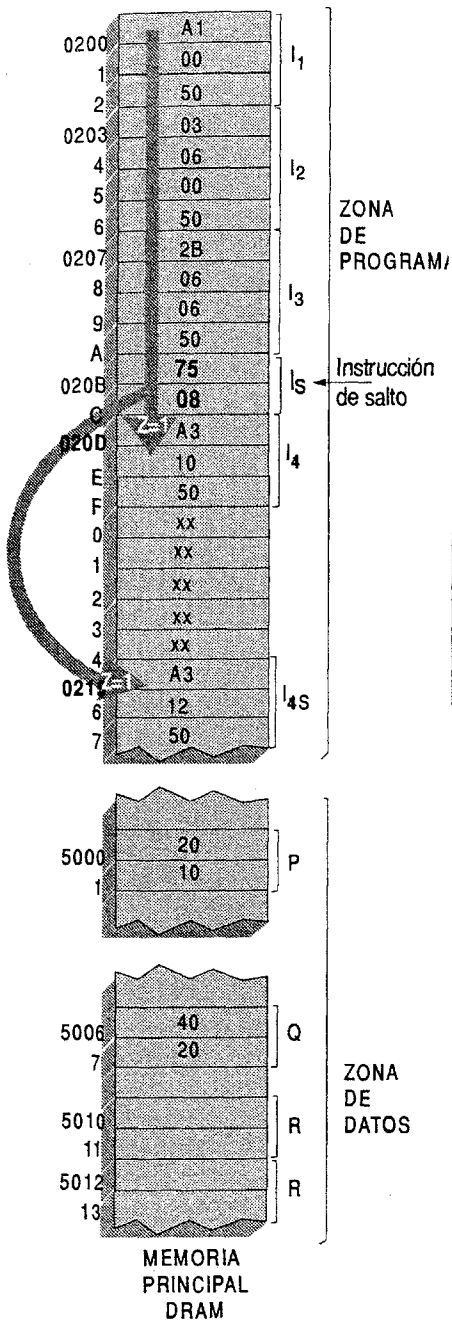


Figura 1.35

Las instrucciones de salto condicionado son *esenciales*, pues permiten pasar en forma automática de un programa (proceso) a otro, en función de ciertos resultados alcanzados en el primero, determinables mediante los indicadores.

A fin de entender cómo operan estas instrucciones, modificaremos ligeramente la secuencia I_1, I_2, I_3, I_4 que aparece codificado en la figura 1.15, (en las figuras 1.36 a 1.40 se recuerda qué ordenan las mismas).

Luego de la instrucción I_3 de resta plantearemos *dos alternativas* de ejecución: si el resultado de efectuar **P + P – Q** es cero, debe guardarse en la dirección 5010 H), como se hizo antes mediante I_4.

De no valer cero, dicho resultado se asignará a la dirección 5012 H. Mediante una instrucción semejante a I_4, que designaremos I_{4S} (por que se la ubica mediante una instrucción de salto)

La posibilidad de dos alternativas de ejecución en función de un resultado anterior (el de la resta en este caso) se realiza insertando una instrucción de salto I_S entre I_3 e I_4.

> Una **instrucción de salto**, *permite decidir cuál es la próxima instrucción que se ejecutará luego de ella, entre dos instrucciones posibles*: la que le sigue a continuación en memoria u otra cuya dirección en memoria ella indica. Este salto de una zona a otra de memoria tendrá lugar o no, según el valor de uno o más indicadores de un resultado anterior.

En lo que sigue, codificaremos las instrucciones que ejecutaremos usando el Debug. Para poder concretar las dos alternativas planteadas, primero ejecutaremos las instrucciones con los datos anteriores, para obtener un resultado igual a cero, y luego las volveremos a ejecutar, pero cambiando el valor de un dato, para que el resultado no sea cero.

Las instrucciones I_1, I_2, e I_3 serán las mismas que en la figura 1.15, por lo cual conservaremos sus códigos, escritos también a partir de la dirección 0200H , de modo de efectuar primero **P + P – Q**

Estos códigos vuelven a aparecer en la figura 1.35

Ahora vamos a codificar la instrucción I_S de salto que ordena:

"Si como resultado de la instrucción anterior (I_3) el indicador **Z** vale 1 (indicación RZ de resultado cero en el Debug) la próxima instrucción a ejecutar es la escrita a continuación en memoria (I_4).

Si **Z = 0** (NZ en el Debug) saltar a ejecutar la instrucción (I_{4S}) que está en la dirección que resulta de sumar a la dirección de la instrucción escrita a continuación (I_4) el valor indicado en esta instrucción". El código de operación que ordena lo que está escrito entre comillas es 75 en hexa, y el código de la instrucción es 75 XX H, donde XX es el valor a sumar citado

Esta información puede obtenerse usando el Debug, como se verá.

Para el caso que se cumpla la condición de salto (Z=0), hemos elegido XX = 08, por lo que se pasará a ejecutar la instrucción I_{4S} que está 8

posiciones abajo de la dirección de la instrucción I_4, escrita abajo de I_S. En las direcciones 020B 020C se han escrito los dos bytes del código 7508 de I_S (intercalada entre I_3 e I_4).
Sintetizando, el código 7508 ordena: si **Z=1** *ejecutar la instrucción que sigue por orden en memoria* si **Z=0** *saltar a ejecutar la instrucción que está 8 posiciones abajo de la que se ejecuta si* Z=0.

Abajo de la instrucción de salto (dirección 020D en este caso) se debe escribir en memoria la **instrucción I_4** (de código A31050, como en 1.15) que se debe ejecutar luego de I_S para el caso que no se cumpla la condición; o sea si de la instrucción anterior I_3 (de resta) resultó **Z=1** (ZR), por ser cero el resultado de la operación que ordenó I_3.
Recordemos que este código ordena enviar a la dirección 5010 H una copia del número que está en AX.

Ahora hay que escribir la instrucción I_{4S} a ejecutar en la alternativa que se cumpla la condición de salto (Z=0 o sea si aparece NZ en el Debug). Debe escribirse 8 posiciones más abajo de I_4 que comienza en 020D (dado que hemos elegido dicho valor 08). Si escribimos las 8 direcciones que siguen a 020D, se tiene: 020E, 020F, 0210, 0211, 0212, 0213, 0214, 0215. O sea, en hexa 020D + 8 = 0215 será la dirección donde escribir I_{4S} (figura 1.35)
Esta instrucción ordena lo mismo que I_4, pero asignando el resultado (que está en AX) a la dirección 5012 como se planteó. Entonces su código de operación también será A3, y su código de instrucción será A31250.

Habiendo codificado la secuencia de instrucciones a ejecutar, debemos escribirla en memoria usando el Debug, de la forma vista en la figura 1.16, a partir de la dirección 0200 H. Una vez realizada la escritura, si verificamos la misma, debe obtenerse lo siguiente

```
-E 200 ↵     (Examinar memoria)
309D:0200  A1.   00.   50.   03.   06.   00.   50.   2B.
309D:0208  06.   06.   50.   75    08    A3    10    50 ↵
-
```

También habrá que escribir en la dirección 0215 el código A3 12 50 (figura 1.35) sin importar (indicado XX) qué está escrito entre 0210 y 0214, debiéndose verificar que debe quedar:

```
-E 215 ↵     (Examinar memoria y escribir en ella)
309D:0215  A3.   12.   50.   ↵
-
```

Conforme a lo establecido, primero ejecutaremos estas instrucciones con los datos del ejemplo anterior **P** =1020 H y **Q** = 2040 H, que están en las direcciones 5000 y 5006, respectivamente, de modo que no se cumpla la condición de salto, o sea que se siga la secuencia I_1, I_2, I_3, I_S e I_4.
Entonces volvemos a escribir estos datos (pues los mismos antes escritos ya se borraron), debiendo quedar[1]:

```
-E 5000 ↵    (Examinar memoria)
309D:5000  20.   10.   50.   8D.   46.   B0.   40.   20. ↵
-
```

Con estos datos y los códigos de instrucción escritos en memoria ejecutaremos instrucción por instrucción, siguiendo el mismo procedimiento llevado a cabo a través de las figuras 1.19 a 1.22, que por tratarse de los mismos códigos y datos elegidos antes coincidirán hasta la figura 1.15

```
-R IP ↵  (Examinar IP)
IP 0100
: 0200 ↵
-
-R ↵ (Examinar registros)
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=309D  ES=309D  SS=309D  CS=309D  IP=0200  NV UP EI PL NZ[2] NA PE NC

309D:0200  A10050          MOV   AX,[5000]                  DS:5000=1020
```

1 Tanto en esta verificación como en la anterior de los códigos de instrucción, sólo interesan los valores indicados en negrita.

2 El flag Z que usaremos en este caso vale **NZ** (Z=0), pero podía haber sido en otro momento, o en otro computador: **ZR** (Z=1)

```
-T ↵   (Orden de ejecución de I1 que ordena asignar al registro AX el contenido de la dirección 5000H)
AX=1020   BX=0000   CX=0000   DX=0000   SP=FFEE   BP=0000   SI=0000 DI=0000
DS=309D   ES=309D   SS=309D   CS=309D   IP=0203 NV UP EI PL NZ NA PO NC

309D:0203   03060050          ADD AX, [5000]                DS:5000=1020
-
```

Figura 1.36

```
-T ↵   (Orden de ejecución de I2 que ordena sumar al registro AX el contenido de la dirección 5000 H)
AX=2040   BX=0000   CX=0000   DX=0000   SP=FFEE   BP=0000   SI=0000 DI=0000
DS=309D   ES=309D   SS=309D   CS=309D   IP=0207 NV UP EI PL NZ NA PO NC

309D:0207   2B060650          SUB AX, [5006]                DS:5006=2040
-
```

Figura 1.37

Hasta acá existe coincidencia total con las indicaciones de las figuras 1.19 y 1.20, como debe ser.

```
-T ↵   (Orden de ejecución de I3 que ordena restar al registro AX el contenido de la dirección 5006 H)
AX=0000   BX=0000   CX=0000   DX=0000   SP=FFEE   BP=0000   SI=0000 DI=0000
DS=309D   ES=309D   SS=309D   CS=309D   IP=020B NV UP EI PL ZR NA PE NC

309D:020B   7508          JNZ 0215 [1]
-
```

Figura 1.38

El código 7508 indica que la próxima instrucción a ejecutar es la de salto (I_S), y que de la resta efectuada resultó que el **indicador Z** cambió de NZ (**Z=0**) a ZR (**Z=1**), de resultado cero (pues al ejecutar I_3 se hizo 2040 – 2040 = 0000), por lo cual podemos anticipar que luego de ejecutar I_s la siguiente será I_4

```
-T ↵   (Orden de ejecución de Is que ordena saltar si Z=0)
AX=0000   BX=0000   CX=0000   DX=0000   SP=FFEE   BP=0000   SI=0000 DI=0000
DS=309D   ES=309D   SS=309D   CS=309D   IP=020D NV UP EI PL ZR NA PE NC

309D:020D   A31050          MOV [5010], AX                DS:5010=XXXX
-
```

Figura 1.39

Verificamos que de la ejecución de I_S ha resultado que la próxima instrucción a ejecutar es la que sigue por orden (dirección 020D) y tiene código A31050, o sea que es I_4. Vale decir que no se ha producido un salto hacia I_{4S} de dirección 0205. Esto era previsible, por que al obtenerse **Z=1** (ZR) en la instrucción anterior, **no** se cumple la condición de salto (que exige Z=0). Luego de ejecutar I_S los indicadores no cambian.
Esto es general: *luego de ejecutarse una instrucción de salto no cambian los indicadores de estado,* puesto que en ella no se realiza ninguna operación aritmética o lógica.
Para terminar esta alternativa, ejecutaremos la instrucción I_4, de código A31050:

```
-T ↵ (Orden de ejecución de I4, que ordena escribir en la dirección 5010H una copia del contenido de AX)
AX=0000   BX=0000   CX=0000   DX=0000   SP=FFEE   BP=0000   SI=0000 DI=0000
DS=309D   ES=309D   SS=309D   CS=309D   IP=0210 NV UP EI PL ZR NA PE NC

309D:0210          (no interesa lo que continúa escrito en este renglón)
-
```

Figura 1.40

Como ya hemos ejecutado la alternativa I_1, I_2, I_3, I_S, e I_4 no seguimos ejecutando ninguna instrucción más.

Si queremos verificar que en las direcciones 5010 y 5011 se escribió el resultado que está en AX. Hacemos:

---

[1] JNZ en Assembler son las siglas del Jump NZ, o sea Saltar si resultó NZ al ejecutarse la instrucción anterior.

–E 5010 ↵ (Examinar memoria)
309D:**5010** **00.** **00.** ↵

A continuación, haremos que se ejecute la alternativa I_1, I_2, I_3, I_S, e I_{4S} para lo cual cambiaremos el valor de un dato, haciendo **Q** = 2000 H, de modo que **P** + **P** – **Q** sea distinto de cero, para que se cumpla la condición de salto.
Entonces, si escribimos en 5006 y 5007 el nuevo valor asignado a **Q** debe resultar en memoria:

–E 5000 ↵ (Examinar memoria)
309D:**5000** **20.** **10.** **50.** **8D.** **46.** **B0.** **00.** **20.** ↵
–

En lo que sigue ejecutaremos los *mismos* códigos de instrucción –antes escritos en memoria y que permanecerán en ella mientras sigamos con el Debug o no los borremos– con los datos modificados. Nuevamente se dará una coincidencia de valores con pasos similares realizados.

–R IP ↵ (Asignar a IP la dirección de la próxima dirección a ejecutar)
IP 0100
: *0200* ↵
–

–R ↵ (Verificación general para control)
AX=0000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=309D ES=309D SS=309D CS=309D **IP=0200** NV UP EI PL **ZR**[1] NA PE NC

309D:**0200** **A10050** MOV AX,[5000] DS:**5000=1020**
–

–T ↵ (Orden de ejecución de I_1)
AX=1020 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=309D ES=309D SS=309D CS=309D **IP=0203** NV UP EI PL **ZR** NA PO NC

309D:**0203** **03060050** ADD AX, [5000] DS:**5000=1020**
– :

–T ↵ (Orden de ejecución de I_2)
AX=2040 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=309D ES=309D SS=309D CS=309D **IP=0207** NV UP EI PL **NZ** NA PO NC

309D:**0207** **2B060650** SUB AX, [5006] DS:**5006=2000**
–

Figura 1.41

Hasta acá, como debe ser, hay coincidencia con las indicaciones de las figuras 1.36 y 1.37, salvo que en 5006 hay 2000

De la ejecución de la instrucción de resta resulta:

–T ↵ (Orden de ejecución de I_3)
AX=0040 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=309D ES=309D SS=309D CS=309D IP=020B NV UP EI PL **NZ** NA PE NC

309D:**020B** **7508** JNZ 0215
–

Figura 1.42

El código 7508 indica que la próxima instrucción a ejecutar es la de salto (I_S), y que de la resta efectuada resultó la indicación NZ (Z=0) de resultado no cero, por lo cual es previsible que luego de ejecutar I_S la siguiente será I_{4S}

[1] Este valor ZR (Z=1) del indicador Z, podía haber sido en otra oportunidad o en otra PC, el valor NZ (Z=0).

–T ↵ (Orden de ejecución de la instrucción de salto I_S)

```
AX=0040  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=309D  ES=309D  SS=309D  CS=309D  IP=0215  NV UP EI PL NZ NA PE NC

309D:0215  A31250          MOV [5012], AX                DS:5012 = XXXX
–
```

Figura 1.43

Verificamos luego de ejecutar I_S, que la próxima instrucción a ejecutar es la que está en 0215 y tiene código A31250, o sea que es I_{4S}. Vale decir, que se ha producido un salto hacia I_{4S} de dirección 0215. Esto era pronosticable, por que al ser Z=0 (NZ) en la instrucción anterior, se cumple la condición de salto. Lo anterior equivale a verificar que el valor de **P + P** = 2040 es distinto al valor de **Q** = 2000. De ser iguales aparecería ZR, como fue antes
Para terminar esta alternativa, ejecutaremos la instrucción I_{4S}, de código A31250:

–T ↵ (Orden de ejecución de I_{4S})

```
AX=0040  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=309D  ES=309D  SS=309D  CS=309D  IP=0218  NV UP EI PL NZ NA PE NC

309D:0218          (no interesa lo que continúa escrito en este renglón, pues no se utilizará)
```

Figura 1.44

Dado que hemos ejecutado la alternativa I_1, I_2, I_3, I_S, e I_{4S}, no ejecutamos ninguna instrucción más.

Para verificar que en las direcciones 5012 y 5013 se escribió el resultado (0040) que está en AX,. hacemos:

–E 5012 ↵ (Examinar memoria)
309D:**5012** **40.** **00.** ↵ (recordar que los números se escriben con los bytes traspuestos).
–

De esta forma se ha comprobado cómo una instrucción de salto condicionado permite que se ejecute una secuencia u otra según un resultado interno alcanzado (en nuestro ejemplo el resultado cero o no cero de una resta). Esto es fundamental para poder pasar automáticamente, por programa, de un proceso a otro.

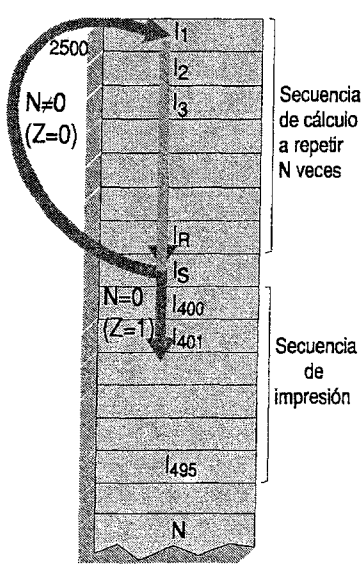
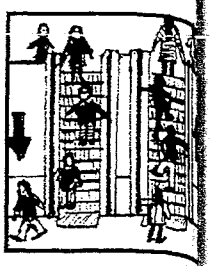


Figura 1.45

Las instrucciones de salto para llamar subrutinas operan en esencia como I_S de la figura 1.35, siendo que ordenan saltar a una instrucción que es la primera de una subrutina que está en otra zona de memoria.
La ejecución de la subrutina termina con otra instrucción de salto, que ordena retornar a la instrucción que en memoria sigue a I_S.
También I_S puede ordenar saltar hacia atrás (figura 1.45) [1] para permitir, por ejemplo, repetir N veces una secuencia de instrucciones de cálculo, y luego pasar automáticamente a otra de impresión.
El valor N se escribe primero en memoria, y se le resta uno cada vez que se ejecuta la secuencia de cálculo, mediante la instrucción I_R, de modo que ella se habrá repetido N veces cuando sea N=0.
A la instrucción que ordena restar uno (I_R), le sigue una instrucción de salto I_S (ubicada entre las dos secuencias). Esta en esencia determina luego que se le resta uno a N, si el resultado es cero o no.
Mientras no sea cero (Z=0), I_S ordena saltar a la dirección 2500 donde está I_1, para que se vuelva a ejecutar la secuencia de cálculo.
Cuando la resta resulte cero, no se cumple la condición e I_S ordena seguir con la secuencia de impresión de resultados, cuya primera instrucción (I_{400}) está escrita debajo de la instrucción de salto I_S.

[1] En ésta se ha supuesto, por simplicidad, que las instrucciones ocupan un byte

1.10 ENTRADAS Y SALIDAS: SEÑALES, PERIFERICOS, BUSES Y PORTS EN EL CAMINO QUE REALIZAN LOS DATOS



Hasta el presente habíamos mencionado brevemente la función de los periféricos en la sección 1.3 (figuras 1.6 y 1.7), y nos habíamos ocupado de la **porción central** –vale decir, del microprocesador y de la memoria principal– sin importarnos cómo datos e instrucciones llegan a memoria, ni como de ésta salen al exterior. Ahora en un primer acercamiento trataremos esta cuestión, siendo que en la Unidad 2 de esta obra referida a periféricos y E/S se la considera más en detalle.

¿Cómo viajan los bits de un lugar a otro en un computador ?

Durante un proceso de datos, los bits de los números binarios que representan ya sea datos, instrucciones, direcciones o resultados, pueden existir dentro de un computador de dos formas:

- *almacenados* (en registros o posiciones de memoria durante un tiempo que depende de su próxima utilización), o
- *transmitiéndose* por líneas conductoras a velocidades electrónicas (cercanas a la de la luz) de un lugar de almacenamiento (origen) a otro (destino), siendo que viaja hacia el lugar de destino una copia de los bits del lugar de origen.

Al tratar la memoria se planteó una analogía con llaves (figura 1.9), en la cual cada bit se guarda como uno o cero, según uno de dos estados posibles "si-no" de cada llave, en la memoria o en registros. En un computador esos dos estados son generados por medio circuitos electrónicos, por lo cual en una escritura se puede pasar muy rápidamente de un estado al otro para cambiar el valor del bit almacenado.

Con el fin de aproximarnos más al funcionamiento eléctrico de estos circuitos supondremos (figura 1.46) que a cada llave de cualquier celda de memoria o registro donde se guardan bits, por un lado está conectada a 5 volts, proporcionados por una pila que simula la fuente de alimentación del equipo, y por otro lado está conectada a un "cable de salida".

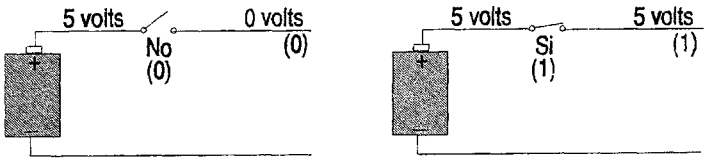


Figura 1.46

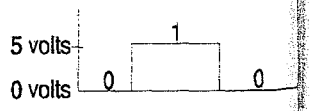


Figura 1.47

Cuando la llave está en "no" (**0** almacenado), por estar abierta no permite que los 5 volts de tensión –que le llegan por el cable que la une con la pila– pasen al "cable de salida, por lo que en éste no existirá voltaje (0 volts). Podemos decir, que *cuando en el "cable de salida" hay 0 volts implica que la llave está en "no", o sea que existe almacenado un bit de valor cero.*
Por el contrario, con la llave en "si" (**1** almacenado), por estar ella cerrada, unirá el cable que sale de la pila con el "cable de salida". Entonces, los 5 volts de la pila se manifestarán también en el "cable de salida", lo cual indica que la llave está en "si" (existe almacenado un bit que vale **1**).

Si la llave durante un tiempo está en "no", durante otro está en "si", y luego otra vez está en "no", graficando a lo largo del tiempo el valor de la tensión eléctrica que se mediría (en volts) en el extremo del cable que sale de ella, resulta la figura 1.47. Este tipo de señales eléctricas, restringidas a tomar solo *dos* valores o *niveles* (alto o bajo), se denominan **señales digitales binarias** o simplemente binarias.

Resulta así que el valor (0 volts ó 5 volts) que aparece en el "cable de salida", es una señal eléctrica indicadora en el lugar de destino (en el extremo del cable), de la existencia de un cero o un uno almacenado.

Puede decirse que *el bit de valor 0 ó 1 que sucesivamente memorizó la llave, se transmitió a través del cable que sale de la llave hasta el lugar de destino, donde su valor (0 ó 1) es detectado merced al valor de la señal eléctrica binaria sensada en el destino.*
Esta señal eléctrica es pues "portadora" y transportadora de información binaria.

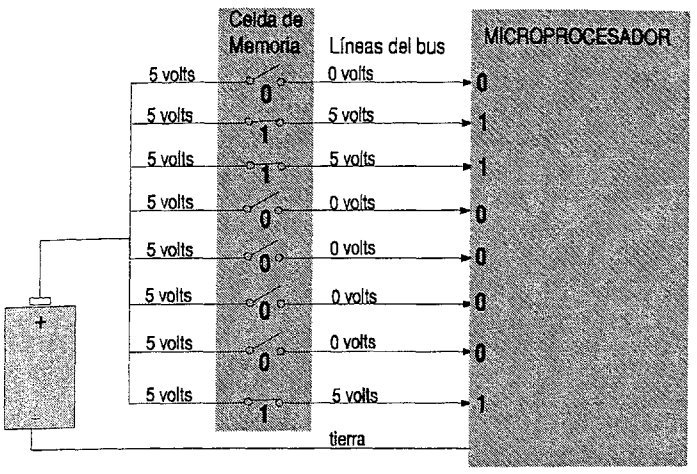


Figura 1.48

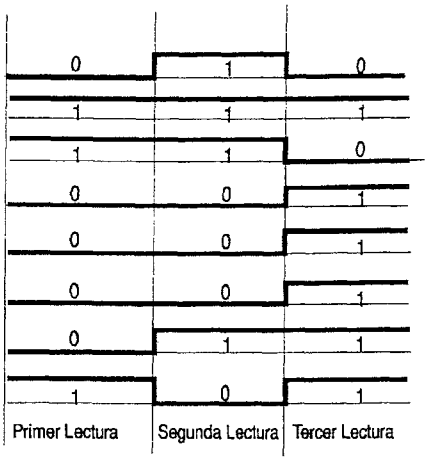


Figura 1.49

Si consideramos 8 llaves como las que imaginamos para una posición de memoria (figura 1.9), suponiendo que de cada una de ellas llega un cable hasta un microprocesador (figura 1.48), éste podrá sensar si cada llave guarda un **0** ó **1** de la forma vista. Basta una sola pila conectada a todas las llaves, para detectar si cada una transmite o no –según esté cerrada o abierta– los 5 volts de la pila. Asimismo, deberá llegar al microprocesador el cable conectado al negativo común ("masa" o "tierra") respecto del cual se mide la tensión eléctrica del cable de salida de cada llave.
Así operan los conductores del bus que comunican la memoria con el microprocesador.

En la figura 1.49 se supone que en una primer operación de lectura se leyó la combinación 01100001, en una segunda la 11100010, y después la 01011111. En correspondencia las líneas del bus a lo largo de ese tiempo tendrán los valores dibujados, resultando en cada línea una señal digital binaria.
Para un bus de 32 líneas de datos puede imaginarse un mecanismo similar de transmisión hacia la UCP, desde 32 llaves, cuando se leen 4 posiciones consecutivas de memoria.

Lo descripto para un movimiento de datos desde memoria al microprocesador vale también para otro en sentido inverso, por ejemplo, cuando en una escritura de memoria, una copia del registro de datos (RDA) debe ir hacia memoria (figura 1.26). Hay que imaginar llaves que en RDA guardan una combinación de bits, con cables que salen de ellas hacia la memoria. El esquema sería el mismo que la figura 1.48, cambiando sólo los nombres del origen y destino. Estos cables del bus de datos, son los mismos que los anteriores, pero cambia el sentido de transmisión cuando se usan para enviar bits de la UCP a memoria.

También opera conforme al esquema de la figura 1.48 el bus de direcciones, que como se trató, tiene un solo sentido de envío: del microprocesador a memoria. En la sección 1.4 se planteó que –como en un control remoto de TV– en el registro de direcciones RDI se formaba (suponiendo con llaves "si-no") el número binario que era la dirección que se enviaba a memoria para acceder directamente a la posición a leer o escribir. Los cables que salen de estas llaves van hacia memoria, constituyendo el *bus de direcciones.*

¿Qué diferencia existe entre transmisión de bits en paralelo y en serie

En la figura 1.48, la existencia de 8 cables para transmitir bits de datos, implica que en el dest (microprocesador en este caso) se puede conocer *simultáneamente* el estado de las 8 llaves, a trav del valor que indica el cable que sale de cada llave.

> Esta forma de enviar varios bits juntos[1] de un lugar a otro por varios cables, se denomi **transmisión paralelo**.

En cualquier punto de un cable, la tensión eléctrica en un instante corresponde en volts al valor **0** ó **1** del b que se está transmitiendo. **No es posible que en un mismo cable un tramo esté a 5 volts, y otro a 0 volts.**

Lo anterior implica que *por un cable se puede enviar un solo bit por vez.*

Si se quiere enviar 8 bits a la vez, hacen falta 8 cables, pudiendo estar cada uno a 0 ó 5 volts.

La combinación 01100001 transmitida por 8 cables, también puede enviarse por un solo cable.

> La **transmisión serie** o "serial" supone enviar por un solo cable, uno tras otro, los bits que se quiere transmitir.

Cuando se transmite por 8 líneas una combinación de bits en paralelo, como la 01100001 (figur 1.48), en el extremo de dichas líneas el valor **0** ó **1** del bit transmitido se determina en cada cabl individual, conforme al valor de tensión (0 ó 5 volts) existente. Asimismo, la posición relativa de la líneas –siempre fija– determina la ubicación de cada bit en el conjunto de los 8 transmitidos.

En la transmisión serie, para poder distinguir ceros o unos repetidos; se requiere que los 5 ó volts que representa cada bit, **duren un lapso de tiempo fijo estipulado.**

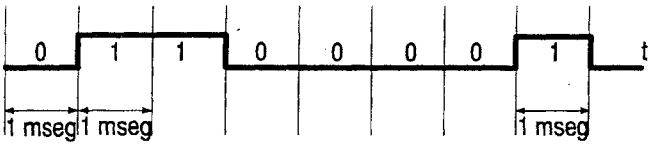


Figura 1.50

Por ejemplo, si en un cable cada bit dur un milisegundo, para transmitir en ser la combinación 01100001 (figura 1.50), en un extremo del mismo durante 1 milis gundo se deben detectar 5 volts (por e "uno" final) seguidos de 0 volts durant 4 milisegundos (por los cuatro "ceros" seguidos de 5 volts durante 2 milisegun dos (dos "unos"), y por último 0 volts durante otro milisegundo (por el "cero" de la izquierda)[2] O sea, que la transmisión en serie requiere *sincronizar* ambos extremos de la línea. Llevaría 8 milésimas d segundo enviar los 8 bits (contra una milésima si se hace en paralelo con la misma duración de cada bit).

En lugar de la figura 1.50 completa, suele escribirse la secuencia 01000011, ó también directamente la forma de onda dibujada. Siempre debe tenerse presente que se trata de una representación *en el tiempo* y no en el espacio. O sea que 01000011 ó dicha onda **no simbolizan una transmisión de bits "cruzando" un cable al mismo tiempo** (como autos que pasan sobre un puente en caravana). Como se subrayó, cuando por un cable se transmiten en serie dichos bits, sólo puede enviarse un bit por vez por el cable. En la figura 1.51 se supone que de la secuencia 01100001 ya se ha enviado 01 y se está transmitiendo un "cero". *Es como tener un puente que puede ser cruzado de a un vehículo por vez.* Igualmente, para ocho cables (figura 1.52), si se deben enviar sucesivamente las combinaciones 01100001, 11100010, y 01011111 antes citadas, sólo se podrá enviar una por vez, resultando en el tiempo en cada línea los valores indicados en la figura 1.49.

Mientras que en la transmisión en paralelo con n cables se pueden enviar y recibir n bits simultáneamente, en la transmisión serie enviar bits demanda n veces el tiempo asignado a la duración de cada bit.

---

[1] En general en un bus de datos se envían 8, 16, 32 ó 64 bits simultáneamente, según sea el word de memoria que maneja el procesador. En cambio en un bus de direcciones, la cantidad total de bits depende de la capacidad de la memoria. En las memorias DRAM una dirección se envía dividida en dos tandas.

[2] Puede pensarse que en el otro extremos de la línea una llave puede permanecer abierta o cerrada un tiempo mínimo de una milésima de segundo. Para generar los 4 ceros sucesivos debe permanecer abierta 4 milésimos de segundo, etc.

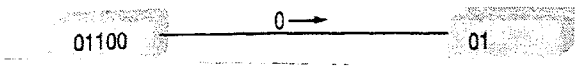


Figura 1.51

Por tal motivo, *en el interior de un computador sólo se transmiten bits en paralelo, a fin de poder enviar más millones de bytes por segundo (megabytes/seg)*

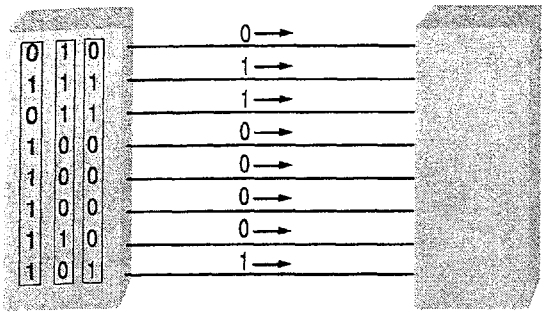


Figura 1.52

Ya al tratar el bus que une el microprocesador con la memoria (figura 1.8), se indicó que todos los bits de datos se enviaban simultáneamente, por líneas en paralelo.

Transmisión de bits en serie tienen lugar, por ejemplo, a través de uno[1] de los conductores del cable de conexionado del teclado o del mouse al computador.
El uso de un solo cable para transmitir datos es económico para enviarlos a distancia.
La transmisión paralelo sólo puede darse entre lugares próximos, como los existentes dentro del gabinete de un computador, dado que la interferencia electromagnética, existente a altas frecuencias entre cables que están próximos en un recorrido común, aumenta con la distancia común recorrida. Es por ello que una impresora se conecta en paralelo si está próxima al computador, y en serie en caso de estar a varios metros del mismo.

¿Qué es información digital, y qué significa computador "digital" ?

Cuando contamos con los dedos (*"dígitos"*) establecemos una correspondencia entre dos conjuntos de elementos individualizables, discretos (separados): el que se quiere contar y el de los dedos. Estos levantados forman una figura que representa un cierto *número* de tales elementos.

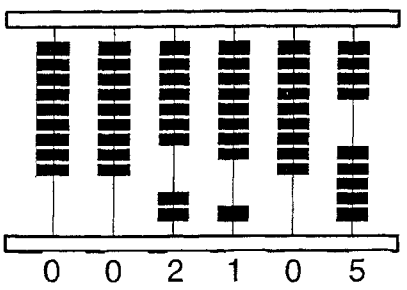


Figura 1.53

Esta correspondencia es la que también se establece mediante el ábaco (figura 1.53), pero de modo más complejo. En éste, una pieza individual ("cuenta") –deslizable en un alambre o varilla fina de madera– tiene por ejemplo valor cien en la varilla de las centenas, indicando que se han contado cien elementos mediante las dos varillas que están a su derecha. Así un ábaco puede representar un conjunto muy grande de elementos *mediante otro conjunto de elementos mucho menor*, constituido por las piezas que están en sus alambres. En cada varilla las cuentas pueden estar en uno de diez estados distintos, según la cantidad de ellas que estén arriba y abajo en la misma.

Asimismo, según *la posición* que tenga una varilla, cada cuenta de la misma vale uno, diez, cien, etc. De la combinación de los estados individuales de cada varilla resulta un estado del ábaco que representa el número de elementos contados.

El sistema numérico arábigo, posterior al ábaco, asignó un símbolo del 0 al 9 para representar cada una de los diez estados posibles de las cuentas de cada varilla.
En esencia, las piezas del ábaco (que simbolizan objetos contados) se reemplazaron por otros símbolos abstractos que cumplen la misma función, más fáciles de transportar y manipular en el papel, los cuales constituyen *números*.
Si una varilla tiene cinco cuentas en su parte inferior, y luego se apila otra sobre ellas, se pasa de un estado de las cuentas a otro distinto, *sin posibilidad de estados intermedios con significado*.
Del mismo modo, del número 2025 se "salta" al 2026, sin que exista un dígito entre el 5 y el 6.

---

[1] Dicho cable está acompañado por otros, para enviar señales de sincronismo, control y alimentación. Una vaina de plástico recubre el conjunto.

Tanto en el ábaco como los números son representaciones simbólicas de la acción de contar unidades

En uno y otro caso la información consta de símbolos separados entre sí, cada uno con un valor propio y otro que depende de su posición en el conjunto (no es lo mismo un 2 en las decenas que un 2 en unidades de mil)

Esta forma de información se denomina **digital**, siendo que se ha conservado la palabra "dígitos" para indicar símbolos que combinados representan un conjunto de elementos contados, siendo que también a cada uno de ellos según su posición relativa se les ha atribuido un valor numérico (uno, diez, cien, mil .... a las varillas del ábaco; uno a cualquier dedo)

Es característica de la información digital y de los fenómenos digitales en general, que entre estados, posiciones, o valores definidos, no existen otros intermedios.

Digital se ha convertido en sinónimo de **número**. Estados físicos como la posición de las cuentas de un ábaco, los engranajes de los medidores domiciliarios de gas, agua, etc, es posible codificarlos mediante símbolos numéricos.

Los relojes digitales indican cuantitativamente la hora mediante números, provenientes de la acción de contar eléctricamente la cantidad de pulsos (conjunto a contar) generados a intervalos regulares por un oscilador de cuarzo.

La información recién descripta es del tipo **digital decimal**. En relación con la señal eléctrica de figura 1.46, restringida a tomar sólo *dos* valores, estados o *niveles* (alto o bajo) que habíamos definido como **digital binaria**, podemos corroborar que tal denominación se ajusta al concepto establecido de información digital. Los 8 cables de la figura 1.48 son, como las varillas del ábaco, piezas separadas de información, con la diferencia que en un instante dado cada uno puede estar en un nivel alto o bajo.

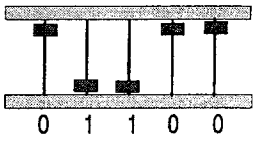


Figura 1.54

Es factible construir un ábaco binario equivalente (figura 1.54) Asignando el símbolo binario **0** al nivel bajo y **1** al alto, los niveles en que se encuentran los cables constituyen la combinación de símbolos binarios 01100001 que puede hacerse corresponder con un número del sistema binario de representación. Lo mismo en una posición de memoria, el estado "si-no" de las 8 llaves con que simulamos circuitos electrónicos con dos estados eléctricos son asimilables a un número binario.

Cuando en un cable la tensión eléctrica cambia de un nivel al otro va pasando físicamente por una serie de valores intermedios, que no son detectados por los circuitos, diseñados para reconocer sólo dos niveles. O sea que cualquier valor intermedio de tensión eléctrica *no tiene significado* para el sistema. Del mismo modo, en un ábaco no tiene significado una cuenta que se desliza de un extremo al otro de una varilla.

Los conductores del interior de las computadoras presentan uno de dos niveles o estados eléctricos posibles. Grupos de 8, 16, 32, 64 cables –según sea– en un instante dado *representan un número binario* que codifica un dato, instrucción, o resultado, mientras que otros representan números que sirven como dirección. Al mismo tiempo, cables individuales que están en **1** ó **0** son usados para controlar (dar órdenes) a los circuitos del computador acerca de las operaciones a realizar.
En el interior de un computador en esencia se guardan y se transmiten números binarios.

La denominación **computadoras digitales** indica el tipo de máquinas que operan con señales eléctricas digitales binarias, que siempre es posible representar mediante números binarios.

Computador digital es sinónimo de computador, desde los primeros prototipos de la década del 40 hasta los actuales. En laboratorios de física o matemática pueden existir "**computadores analógicos**". En éstos la tensión eléctrica no está restringida a dos valores como en los digitales.

Los niveles alto/bajo, así como los símbolos **1** y **0** , pueden adjudicarse a los valores "verdadero" y falso" de la lógica clásica, como lo hizo George Boole hacia 1850. Es por ello que también se llaman *señales lógicas* las señales digitales binarias tratadas, y en correspondencia *circuitos lógicos o binarios* los encargados de generarlas y sensarlas.

¿Qué es información analógica?

Los relojes digitales y analógicos son exponentes de dos formas diferentes de obtener información.
Un reloj solar es un dispositivo analógico, muy ilustrativo de las características de la información llamada "analógica". Se basa en el movimiento de la sombra de un objeto iluminado por el sol, que sirve como *símil* o *analogía* del transcurso continuo del tiempo. Algunas marcas circunstanciales en el recorrido o tamaño de la sombra, sirven para indicar determinados instantes de interés horario. En su recorrido, la sombra pasa teóricamente por infinitos puntos, sin solución de continuidad. Cada uno de los puntos significa información, y entre dos posiciones, siempre será posible encontrar otras intermedias. En esencia se ha convertido el movimiento del sol en una indicación en un cuadrante.
Las manecillas de un reloj analógico, que se mueven en forma continua, pasan también por infinitas posiciones para estimación horaria. La existencia de números en tales relojes puede no ser necesaria. Sólo bastan unas pocas marcas de referencia en el cuadrante, como presentan muchos modelos, para estimar la hora.
Existe un sinnúmero de dispositivos mecánicos ideados para estimar o medir magnitudes físicas, los cuales presentan variaciones *análogas* a éstas. La temperatura se mide por su efecto dilatador en una línea de mercurio de un termómetro. La altura del aceite en el motor de un auto se representa por la película oscura que deja en la varilla que está sumergida en el cárter. La velocidad de un vehículo se representa por la posición de la aguja en un cuadrante en función del número de vueltas por segundo que da una rueda.
El peso de un objeto, se representa en una balanza a resorte, por el estiramiento que sufre éste a causa de dicho peso. La longitud de una pieza de género puede medirse en relación con el número de vueltas que debe dar un cilindro donde se enrolla.

En los ejemplos citados; se tiene una magnitud física a evaluar, medir, (tiempo, altura, velocidad, peso, etc.), cuyo valor puede sufrir una sucesión de variaciones –en más o en menos– tan pequeñas como sea. Esto es, ella puede tomar un *rango continuo de valores* (teóricamente infinito, pues entre dos valores siempre hay otro intermedio).

Dicha magnitud se ha ***reemplazado*** por algún sistema que pueda *representarla* o *sustituirla*, por variar en general *proporcionalmente*, en igual medida.

Se ha construido en todos los casos un *análogo*, un sistema analógico, cuya salida brinda **información analógica**, esto es una indicación (señal) que puede variar en forma semejante a la magnitud física de rango continuo de valores a medir, evaluar o transmitir.

Esto en algunos casos se hace por razones práctico-constructivas, como el caso de la observación directa del nivel de aceite, o de combustible en un vehículo. En el caso de un cuerpo sólido a pesar, no es factible fraccionarlo en porciones de 1Kg ó submúltiplos, so pena de destruirlo. Igualmente una tela no se puede cortar en trozos de un metro para medirla. Asimismo, resulta más sencillo evaluar variaciones de temperatura por sus efectos dilatadores, que en forma absoluta por el grado de agitación molecular.

Nos interesan en particular las señales eléctricas que se hacen variar en forma análoga (*señales analógicas*) a la variaciones que presentan ciertas magnitudes, como las tonalidades de gris o color de una imagen cuando es barrida por un escáner, o la presión del aire que actúa sobre el micrófono de un teléfono cuando hablamos por él.
En un escáner (ver unidad 2) mediante una fila de fotodiodos –uno al lado de otro– muy sensibles a las variaciones luminosas, se barre una imagen que se quiere almacenar en memoria. Una luz ilumina la zona que se está barriendo, y los fotodiodos sensan la luz reflejada por la pequeña superficie que está debajo de cada uno. Considerando un fotodiodo cualquiera, durante su recorrido sobre la imagen barrerá una tira vertical angosta de la misma (figura 1.55), y detectará las distintas tonalidades que encuentre a su paso. Cuando pasa por zonas de color negro por el diodo prácticamente no circulará corriente eléctrica, y por zonas blancas dará lugar a una intensidad de corriente máxima. Entre estos extremos, las distintas tonalidades de gris reflejarán valores intermedios de luz, y en correspondencia por el diodo circulará una corriente proporcional al tono de gris sensado. Teóricamente existen infinitos tonos de grises, por lo cual la corriente que regula cada fotodiodo puede tomar infinitos valores entre los dos extremos para el negro y el blanco
Suponiendo un escáner de mano que se mueva a velocidad constante, si en un segundo recorre X cm, el diodo considerado al barrer la tira, indicada en grisado, dará lugar a una circulación de corriente, que en el transcurso del tiempo sufrirá las variaciones de intensidad indicadas.

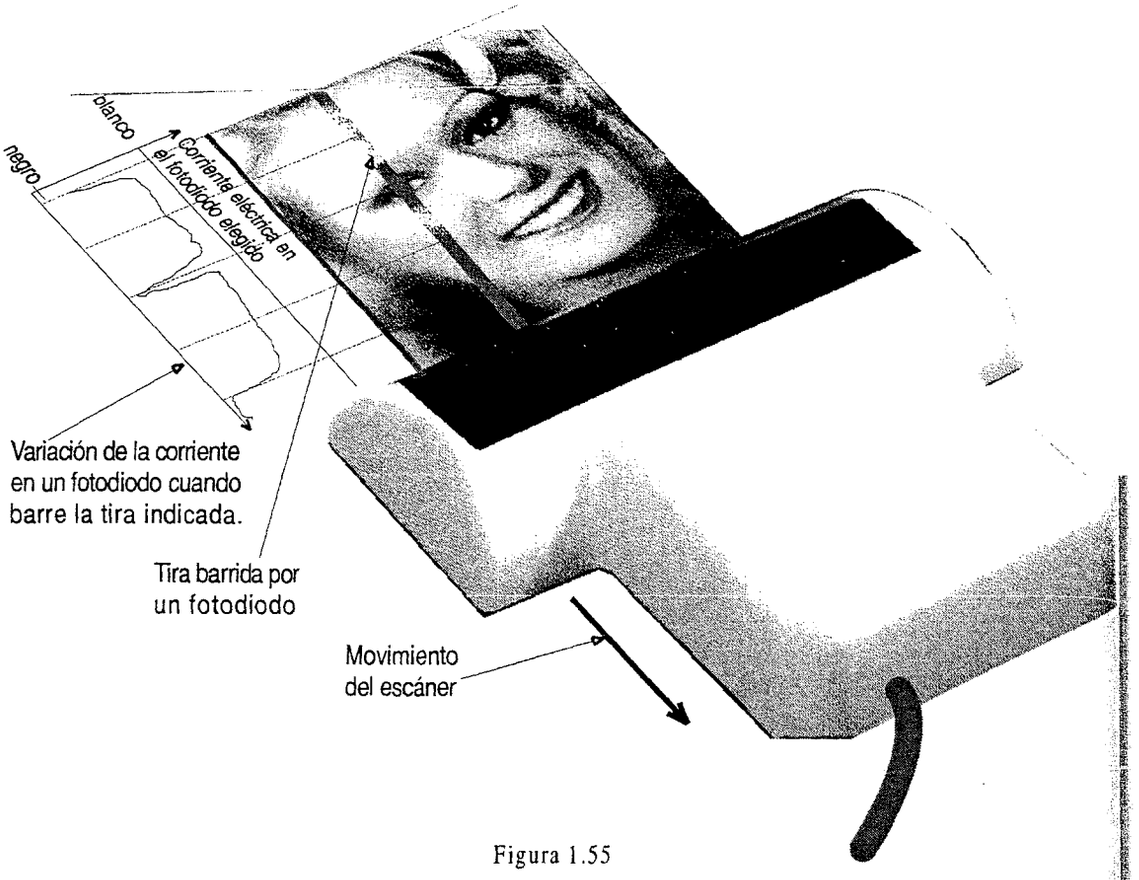


Figura 1.55

Según se vio, en las señales digitales binarias sólo tienen significado (0 y 1) dos valores (0 y 5 volts), que saltan abruptamente de un valor a otro. A diferencia, en las señales analógicas *cualquier valor de tensión representa información* acerca de los tonos de gris del área barrida por el diodo en ese segundo.

> En general, *una **señal eléctrica analógica** puede variar a lo largo del tiempo de una manera continuada, gradual, sin saltos bruscos, entre dos valores extremos que determinan un rango, y en un instante dado puede tener un valor cualquiera con significado informativo dentro de dicho rango.*[1]

A fin de explicar el principio de generación de las señales eléctricas analógicas que se transmiten por las líneas telefónicas como réplica de las señales que emiten nuestras cuerdas vocales, se describirá una experiencia fácil de realizar o imaginar, que permite además tener una primera aproximación al concepto de modulación.

Sea una habitación cuya lámpara eléctrica en lugar de estar simplemente gobernada por una llave "si-no", lo está mediante un potenciómetro regulador de intensidad luminosa. Si giramos éste hacia un lado y otro a un cierto ritmo, la intensidad luminosa fluctuará de igual forma. Esto sucede porque la corriente eléctrica que pasa por la lámpara está variando de intensidad de manera *análoga* a las variaciones a que sometemos el regulador.

Cuando hablamos, las vibraciones de nuestras cuerdas vocales impulsan el aire que las rodea produciendo variaciones de presión semejantes (como lo hace en mucho mayor grado el cono de un parlante al vibrar). Si dichas variaciones de presión del aire actúan sobre el micrófono de un teléfono, producen vibraciones *análogas* a las generadas por las cuerdas vocales sobre una pieza móvil del micrófono. Por formar parte el micrófono del circuito eléctrico telefónico, actúa de manera parecida al regulador luminoso citado: hace variar la corriente eléctrica que circula de modo análogo a las vibraciones de las cuerdas vocales.

---

[1] Teóricamente, puesto que entre dos valores próximos es factible encontrar otro intermedio, una señal analógica podría tener infinitos valores en su rango

Esta corriente llega por un cable hasta el receptor del otro aparato telefónico que está en comunicación, merced a la conexión establecida en la central telefónica.
De esta forma, una señal eléctrica analógica sirve para *transmitir* variaciones de presión (también analógicas) de un extremo a otro de una línea telefónica. Si estas señales llegan a un parlante o auricular, harán que este vibre al compás de sus variaciones, las cuales se convertirán en audibles, por lo que se llaman **señales de audio**. Otras señales analógicas como las de radiofrecuencia y TV no pueden ser escuchadas de esta forma, por ser de rápida variación

La comunicación entre dos computadores por vía telefónica supone la existencia en cada extremo de un dispositivo periférico modulador/demodulador (**módem**). Cuando un computador transmite datos a otro, el módem del primero actúa como modulador. Esto es, el módem hace variar la corriente eléctrica que envía al otro extremo, como ocurre cuando se está hablando por teléfono; pero se transmiten tonos de audio simples (como los que se escuchan al discar), cuyas variaciones representan unos y ceros (proceso de conversión digital a analógico denominado **modulación**). En un parlante ubicado en la plaqueta del módem del computador que está recibiendo se pueden escuchar dichos tonos de audio enviados en esta "conversación" entre computadores.

¿Qué implica una conversión anaiógica-digital (A/D) y en qué periféricos tiene lugar ?

Como se trató, a los efectos de *contar* unidades o entes separables, individualizables, usábamos los números naturales 1, 2, 3, ... 85, ... 205, ... que simbólicamente reemplazaron al ábaco para representar los objetos contados. La acción de *medir* magnitudes que varían dentro de un rango continuo de valores es diferente a la acción de contar. Una medición en general se realiza *cuantificando* mediante *números* la salida de un sistema analógico, o sea la señal analógica que varía de manera similar a la magnitud física a medir Una forma de cuantificar es usar una escala graduada (que es recta en un termómetro, y circular en un reloj), dividida en unidades adecuadas, la cual presenta números en ciertos puntos calibrados. Divisiones menores entre los mismos permite apreciar temperaturas como 38,4 °C. En esencia está representando la propiedad de dilatación de la materia mediante números fraccionarios, resultando una medida aproximada

> En general, cuando nos valemos de números para medir cuantitativamente una magnitud analógica, estamos efectuando una **conversión analógico-digital (A/D)**.

Esta conversión no siempre es necesaria. Así, cuando se aprecia el nivel de combustible o de aceite de un auto, no hace falta que la escala presente números indicadores.

Dado que en el interior de un computador *sólo* pueden existir señales digitales, y que señales provenientes del exterior son analógicas, como las que se generan cuando un escáner barre una imagen, o las enviadas vía módem por una línea telefónica, *se requiere además que en dichos periféricos se lleve a cabo una conversión A/D*. Igualmente será necesario convertir en señales eléctricas digitales el movimiento de la bolita de un mouse, o las señales analógicas de audio y video que llegan a una plaqueta para multimedia.

En relación con la digitalización de estas últimas –semejante a la que se hace antes de grabar música en un CD de audio– trataremos una forma de conversión A/D empleada para este tipo de señales eléctricas de variación relativamente lenta. La figura 1.56 ilustra los conceptos centrales relacionados con esta forma de conversión A/D. En ella se supone que en los instantes t_1, t_2, t_3 ... se toma una muestra, o sea se mide el valor de una señal eléctrica analógica hipotética que varía entre 0 y 15 volts.
La medición realizada en t_1 es de **0101** = 5 volts, valor entero aproximado al valor real que es 5,4 volts[1].
Del mismo modo, en t_2, t_3 ... se han determinado los valores enteros **1001** (9 en vez de 8,8), **1100** (12 en vez de 11,6) .... etc. Si todos estos *números binarios* se guardan ordenados en memoria, la curva continua (variable analógica) de la figura 1.56 quedará memorizada dentro del computador como el conjunto de puntos separados de la figura 1.57. Si se unen éstos, resulta una figura de forma parecida a la original.
Se comprende, que si en lugar de tomar 16 niveles de valor 0000 a 1111, se determinan 32 (de 00000 a 11111), existirá un menor error de cuantificación. En general, cuantos más bits se utilicen para representar cada medición, y cuanto menos separados estén los instantes t_1, t_2, t_3 ... –o sea cuantas más muestras

[1] Se denomina "*error de cuantificación*" a la diferencia 5,4 – 5 = 0,4

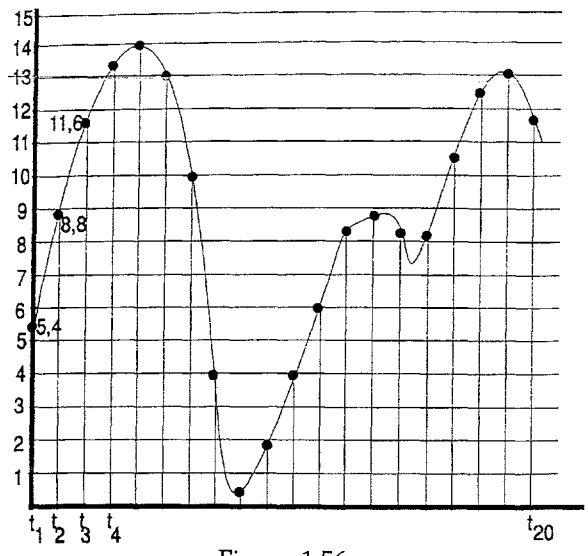


Figura 1.56

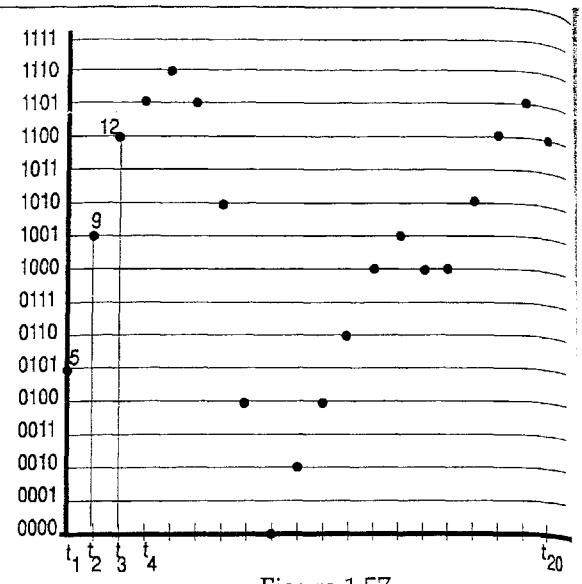


Figura 1.57

se tomen de la señal– mayor será la correspondencia entre la señal analógica y los números que representan puntos cuantificados de ella en memoria, requiriéndose mayor espacio en ésta para guardarlos.
La conversión A/D que tiene lugar en un módem se llama "**demodulación**" no siendo un proceso de cuantificación como el recién ejemplificado. Consiste en *detectar* en la señal eléctrica analógica –que llega al módem receptor por línea telefónica– los unos y ceros que fueron codificados por el módem transmisor (citado en la pregunta anterior), y generar en correspondencia los dos niveles de tensión de la señal binaria digital que puede manipular el computador. Por ejemplo, si dos módems intercomunicados intercambian información por modulación de frecuencia, al módem que está recibiendo llegará una señal analógica que presenta dos frecuencias distintas (figura 1.70), y detectará una frecuencia como portando, representando, ceros, y la otra frecuencia, unos.

En un mouse común la conversión A/D es en parte mecánica: el movimiento de la bolita se convierte y descompone en movimientos análogos en dos direcciones perpendiculares que comunica a dos rueditas dentadas o ranuradas que pueden girar. (figura 1.60). Los dientes de cada ruedita al girar van dejando pasar o cortando un haz de luz que incide sobre ellos, generada por un dispositivo fotoemisor. En el giro de cada rueda dentada, cada paso-corte-paso del haz es sensado por un fotodetector, que así genera un pulso eléctrico con cada diente de la rueda[1]. De esta forma, el movimiento continuado de la bolita transmitido a las dos rueditas se convierte en dos series de señales digitales, que permiten determinar la dirección y velocidad de ese movimiento. Se ha realizado así una conversión A/D.

¿Qué implica una conversión digital-analógica (D/A) y qué periféricos la llevan a cabo ?

Mientras que una conversión A/D está relacionada con la *entrada* de datos desde el exterior hacia un computador, una conversión D/A se requiere para ciertos tipos de salida desde éste hacia el mundo exterior.
Merced a ella, una señal eléctrica digital que representa unos y ceros, se transforma en una señal analógica. Se trata de procesos inversos a los esquematizados en las figuras 1.60, 1.64 y 1.70.
Así, una conversión D/A se realiza en una plaqueta de vídeo (figura 1.67), para que el monitor –hoy día analógico– pueda brindar una amplia graduación de colores, a fin de obtener una imagen más real.
En un módem, la acción de convertir los unos y ceros provenientes del computador en variaciones de una señal eléctrica analógica que se envía por línea telefónica, se denomina "**modulación**".
También se requiere conversión D/A para la información que recibe un periférico graficador ("plotter").

---

[1] En la figura 1.60 la luz emitida por un fotoemisor pasa entre dos dientes de una rueda, la cual activa al fotodetector correspondiente; mientras que la luz que emite el otro es interceptada por un diente, por lo que el fotodetector ligado al mismo no recibirá luz.

¿Qué hardware encontramos para la entrada/salida de datos, desde los periféricos hasta la porción central de un computador ?

Según dijimos, la designación "periféricos" proviene de la ubicación de estos dispositivos en la periferia de un computador, en relación a la **porción central** constituida por la UCP y la memoria principal. También se denominan *unidades de entrada o salida* según sea su función.

Anteriormente habíamos establecido que la función primera de los periféricos es *convertir* señales que representan datos externos en internos en las operaciones de entrada, o a la inversa en operaciones de salida.

Esto es, un periférico oficia de *"frontera"* entre el exterior y el interior de un computador para la conversión de señales. Del mismo modo, nuestros ojos, oídos, piel, etc., sensan señales externas y las transforman en señales eléctricas que van hacia nuestro cerebro o médula; y por ejemplo, nuestras cuerdas vocales y las cavidades asociadas realizan un proceso opuesto para comunicarnos con el exterior.
En una PC existen periféricos que están fuera del gabinete del computador (teclado, impresora, mouse, etc.) y otros que están dentro del mismo (unidades de disco – flexibles, rígidos, CD ROM– o un módem interno)

Siempre indisolublemente ligado al periférico existe *un medio* que representa el mundo exterior: el papel en la impresora, las teclas en el teclado, la pantalla en el monitor, la superficie donde se desplaza la bolita del mouse, la imagen que recorre el escáner, el disco fijo o inserto en la unidad correspondiente, etc.

En una **operación de entrada los datos** que llegan del exterior a un periférico tienen como destino la memoria principal, mientras que en una **operación de salida** el movimiento es el contrario.

Si se observa en una PC el encadenamiento de subsistemas conectados desde un periférico hasta la "motherboard", que contiene la porción central, en general podemos distinguir las siguientes etapas de hardware (figuras 1.60 a 1.65), cuyas funciones luego se detallan:

1. En el periférico encontramos circuitos electrónicos que constituyen lo que denominaremos **"electrónica del periférico"**.[1]
2. Un cable conteniendo varios conductores une eléctricamente el periférico en cuestión con un conector correspondiente a una plaqueta **interfaz**[2], insertable en la "motherboard"[3]. Para tal fin, dicho cable en sus extremos presenta conectores apropiados.
3. La **plaqueta interfaz**[4] porta un nivel intermedio de electrónica –entre la electrónica de un periférico citada y la porción central– que llamaremos **"electrónica intermediaria"** (que para el teclado está en la "motherboard"). Circuitos con memoria de ésta constituyen **registros** denominados **ports** (*"puertos"*).
4. A un bus de E/S de la plaqueta principal ("motherboard") están conectados varios zócalos donde se insertan plaquetas interfaces, a razón de una por zócalo. De este modo *un bus oficia de "camino" común, para permitir que varios dispositivos se puedan conectar a circuitos de la parte central.* Como se verá, en la "motherboard" de una PC puede haber distintos tipos de buses de E/S que cumplen esta función, pero con distinta velocidad de transferencia de datos.[5]

---

1 Elegimos esta denominación genérica, para evitar nombres como "controlador/a de periférico, dado que también se designa con este último nombre a plaquetas que se insertan en la plaqueta principal, a las cuales se conecta el cable del periférico correspondiente. Así, en una PC se habla de "controladora IDE o SCSI (se pronuncia "scasi") para denominar los circuitos que están junto a la caja que contiene el disco rígido o a la del CD. Pero también por ejemplo se llama "controladora", a la plaqueta (o tarjeta, como se diga) de vídeo a la que se conecta el monitor, y a los circuitos integrados de la "motherboard" que constituyen la denominada "controladora" del teclado, siendo hoy común encontrar también en la "motherboard" controladoras de disco y disquetes. Vale decir, que la palabra **controlador/a** designa la electrónica que gobierna las acciones de un periférico como el disco rígido, así como la que sirve como intermediaria para el pasaje de datos entre el periférico y la parte central ".

2 **Interfaz** es una palabra que en computación designa en general un hardware intermediario, ubicado *entre* dos subsistemas independientes, que sirve para comunicarlos y *adaptarlos* eléctricamente. Así, puede decirse que un periférico es una interfaz entre el exterior y el interior de un computador, o que un bus o un cable es una interfaz entre los dispositivos que conecta, etc. En las PC cuando se habla de interfaz (*"interface"* en inglés), se asume que se trata de una plaqueta adaptadora, insertable en la plaqueta principal, cuyos circuitos por un lado están conectados a la electrónica de un periférico (a través de un cable) y por otro a la porción central (a través de un bus). Dichos circuitos contienen registros de almacenamiento transitorio (ports), donde programas dejan órdenes para el periférico, y para retener datos en tránsito entre este último y la porción central. Otros circuitos sirven para generar señales de interrupción.

3 El teclado se conecta directamente a la "motherboard", siendo que los circuitos de su interfaz ("controladora") están en dicha plaqueta.

4 Esta plaqueta también se llama controladora, o plaqueta controladora, por lo que vale lo dicho en el pie de página anterior.

5 El primero en aparecer fue el bus ISA (Industry Standard Architecture) desarrollado por IBM para las XT, también llamado *bus del sistema o bus de expansión*. Este bus hoy existe –por razones de compatibilidad con plaquetas y periféricos corrientes– en las

¿De qué forma intervienen las cuatro etapas de hardware citadas en operaciones de E/S, desde o hacia distintos periféricos ?

Ejemplificaremos para periféricos corrientes[1], la función que desempeñan los cuatro subsistemas descriptos en una operación de entrada, desde que un periférico en cuestión detecta datos, hasta que llegan a memoria principal; y en una operación de salida, en el movimiento inverso de datos.[2]

Si bien en el presente casi todos los periféricos tienen el circuito de su interfaz y ports integrado en el chip o chips que constituyen el chipset que acompaña al procesador en la motherboard (figura 1.5), se ha preferido mantener el esquema anterior con la interfaz visible en la tarjeta enchufable en el bus correspondiente, a los efectos de que se vea más claramente el camino que sigue la información. Esto es, independientemente de que una interfaz esté en una tarjeta o en el chipset de la mother, lo que interesa es su función, que será siempre la misma, sin que interese dónde se encuentra físicamente. Vale decir, que ningún periférico puede conectarse directamente a un bus (sea ISA, PCI, etc.), sino que por intermedio de una interfaz, la cual también presenta registros ports para que en uno de ellos el periférico reciba órdenes, y que por otro envíe o reciba datos.

Empezando con el **mouse** (figura 1.60), en el mismo tiene lugar la conversión A/D antes descripta, como comienzo de cada operación de entrada. Los pulsos generados –que representan las componentes del movimiento de la bolita según dos ejes perpendiculares– son enviados *en serie por la electrónica del mouse,* a través de un conductor contenido en el cable de salida. Este cable se conecta a la plaqueta "multifunción"[3] donde está la interfaz "port serie"[4], elegida para el mouse (electrónica intermediaria).
Un circuito de ésta es un **registro port de datos**, siendo como todo registro, *hardware dedicado al almacenamiento temporario* de los datos que llegan desde la electrónica del mouse.
Del port, dichos datos salen *en paralelo, byte por byte*, a través de las líneas de datos de un bus de E/S, y así llegan al registro AX de la UCP (microprocesador). Por último, del registro AX pasan a memoria, donde un programa hace que el cursor del mouse aparezca en pantalla.

En el caso de la **impresora** (figura 1.61) la operación de salida tiene lugar cuando pasan de memoria principal al registro AX datos a imprimir, a razón de un byte por vez. Estos bits de datos desde AX, y a través del bus ISA, llegan simultáneamente a un registro port de datos[5] de la interfaz, que los guarda temporariamente. Este port forma parte de la circuitería de la interfaz "port paralelo" también ubicada en la plaqueta "multifunción" (nivel electrónico intermediario). De este port, cada byte a imprimir pasa en paralelo –a través del conector y cable que lo vinculan con la impresora– a la memoria de almacenamiento temporario de este periférico, que forma parte de su electrónica[6]. Esta también se encarga de convertir la información binaria a imprimir, contenida en su memoria, en señales gráficas en tinta negra o color sobre un soporte de papel, que oficia de mundo exterior.

La electrónica del periférico **teclado** está contenida en una pastilla con un microprocesador dedicado (figura 1.62).
El mismo, entre otras funciones, detecta qué tecla se pulsó o liberó. Luego envía en serie, hacia el registro port del teclado, el código binario de dicha tecla por una línea, la cual forma parte del cable que se enchufa a un conector de la "motherboard". Por ser un periférico muy estándar, el teclado no se conecta a una plaqueta insertable en la "motherboard". La electrónica intermediaria (interfaz con port) del teclado, conocida como *controladora* de teclado, está en la plaqueta principal. Del port citado, los 8 bits del código de la tecla siguen en paralelo por el bus ISA, hasta el registro AX, de donde van a memoria principal.

---

1 Cuyo funcionamiento se describe en la Unidad 2

2 Datos se emplea acá en un sentido general, que puede incluir instrucciones y resultados, según el caso.

3 Esta plaqueta contiene varias interfaces: para disco rígido, CD ROM, disquete, port paralelo (para impresora), y una interfaz port serie

4 Designado **COM1** aó **COM 42** en una PC. **Registro "port" es sinónimo de "port".**

5 **No debe confundirse un port,** que es un circuito con memoria que oficia de registro, **con el conector de la parte trasera de un gabinete,** que simplemente sirve para conectar un cable por el cual pueden llegar o salir datos desde o hacia dicho port.

6 Esta electrónica hoy día es tan compleja que incluye un microprocesador dedicado a la impresora, con un programa en ROM que maneja el funcionamiento de la misma. En las impresoras láser dicho microprocesador puede ser uno veloz de tipo RISC.

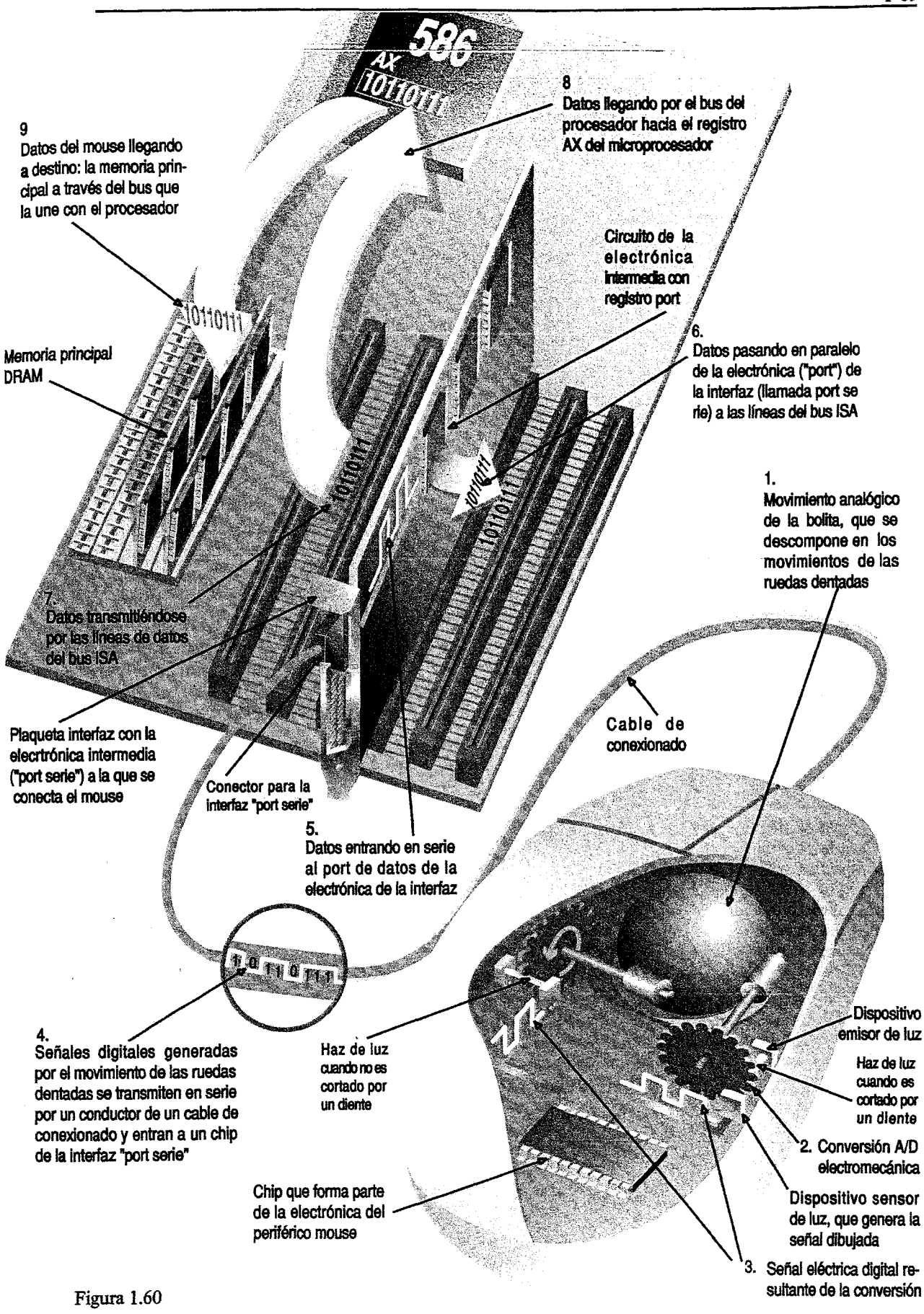

586
AX
10110111
8
Datos llegando por el bus del procesador hacia el registro AX del microprocesador
9
Datos del mouse llegando a destino: la memoria principal a través del bus que la une con el procesador
Circuito de la electrónica intermedia con registro port
6.
Datos pasando en paralelo de la electrónica ("port") de la interfaz (llamada port serie) a las líneas del bus ISA
Memoria principal DRAM
1.
Movimiento analógico de la bolita, que se descompone en los movimientos de las ruedas dentadas
7.
Datos transmitiéndose por las líneas de datos del bus ISA
Cable de conexionado
Plaqueta interfaz con la elecrtrónica intermedia ("port serie") a la que se conecta el mouse
Conector para la interfaz "port serie"
5.
Datos entrando en serie al port de datos de la electrónica de la interfaz
4.
Señales digitales generadas por el movimiento de las ruedas dentadas se transmiten en serie por un conductor de un cable de conexionado y entran a un chip de la interfaz "port serie"
Haz de luz cuando no es cortado por un diente
Dispositivo emisor de luz
Haz de luz cuando es cortado por un diente
2. Conversión A/D electromecánica
Chip que forma parte de la electrónica del periférico mouse
Dispositivo sensor de luz, que genera la señal dibujada
3. Señal eléctrica digital resultante de la conversión

Figura 1.60

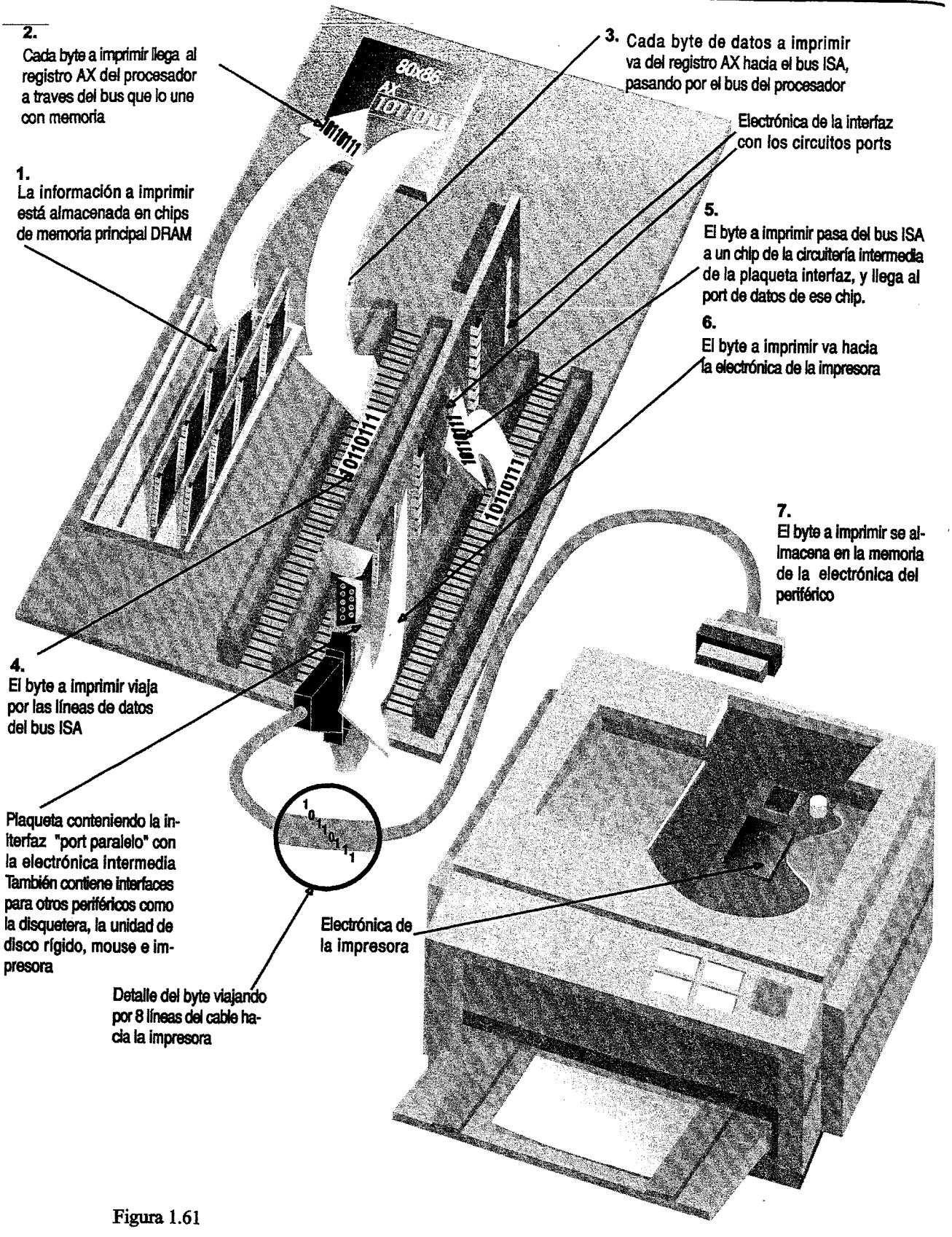

2.
Cada byte a imprimir llega al registro AX del procesador a traves del bus que lo une con memoria
3. Cada byte de datos a imprimir va del registro AX hacia el bus ISA, pasando por el bus del procesador
Electrónica de la interfaz con los circuitos ports
1.
La información a imprimir está almacenada en chips de memoria principal DRAM
5.
El byte a imprimir pasa del bus ISA a un chip de la circuitería intermedia de la plaqueta interfaz, y llega al port de datos de ese chip.
6.
El byte a imprimir va hacia la electrónica de la impresora
7.
El byte a imprimir se almacena en la memoria de la electrónica del periférico
4.
El byte a imprimir viaja por las líneas de datos del bus ISA
Plaqueta conteniendo la interfaz "port paralelo" con la electrónica intermedia También contiene interfaces para otros periféricos como la disquetera, la unidad de disco rígido, mouse e impresora
Electrónica de la impresora
Detalle del byte viajando por 8 líneas del cable hacia la impresora

Figura 1.61

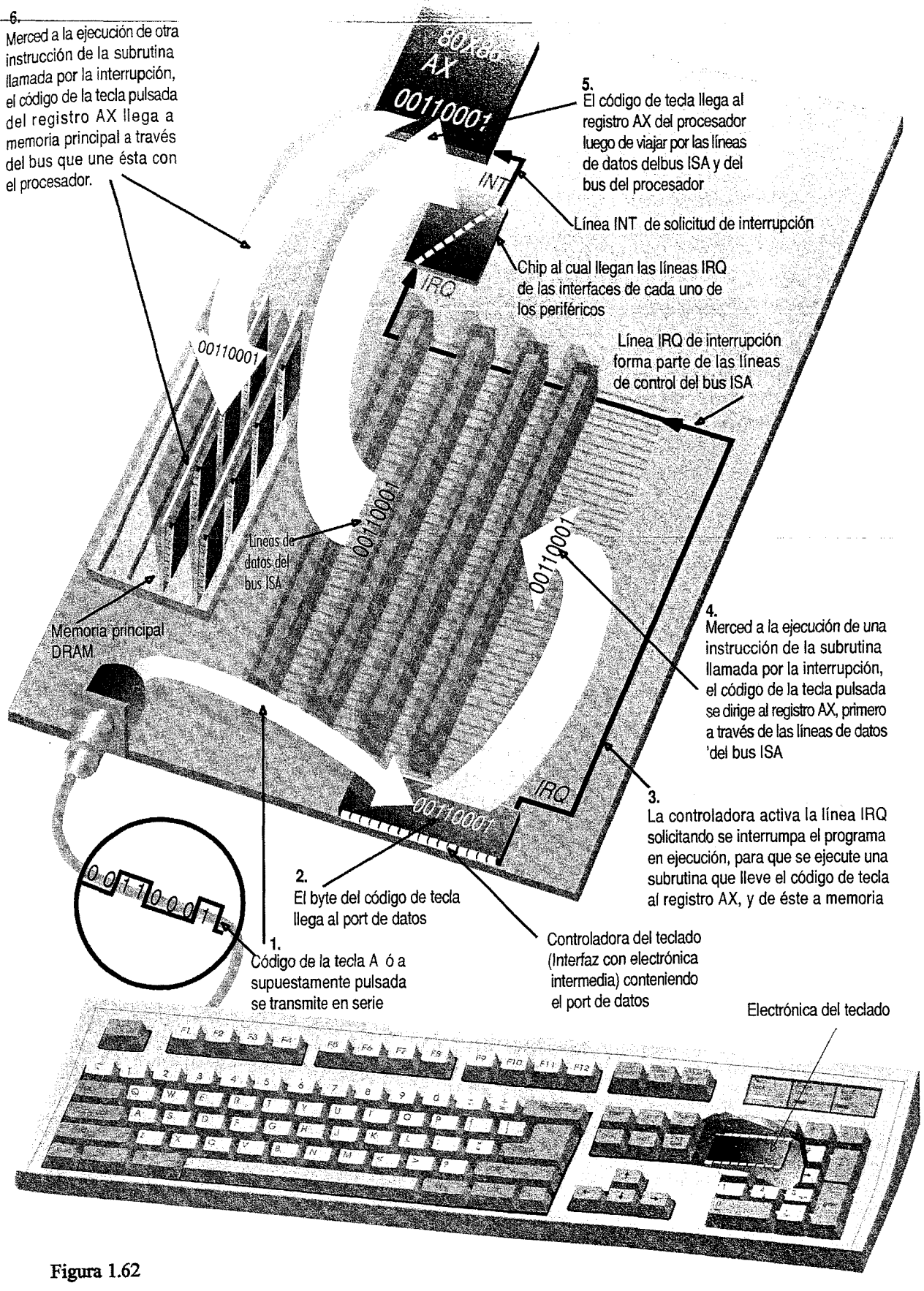

6.
Merced a la ejecución de otra instrucción de la subrutina llamada por la interrupción, el código de la tecla pulsada del registro AX llega a memoria principal a través del bus que une ésta con el procesador.
AX
00110001
5.
El código de tecla llega al registro AX del procesador luego de viajar por las líneas de datos delbus ISA y del bus del procesador
INT
Línea INT de solicitud de interrupción
Chip al cual llegan las líneas IRQ de las interfaces de cada uno de los periféricos
IRQ
Línea IRQ de interrupción forma parte de las líneas de control del bus ISA
00110001
Líneas de datos del bus ISA
Memoria principal DRAM
4.
Merced a la ejecución de una instrucción de la subrutina llamada por la interrupción, el código de la tecla pulsada se dirige al registro AX, primero a través de las líneas de datos 'del bus ISA
3.
La controladora activa la línea IRQ solicitando se interrumpa el programa en ejecución, para que se ejecute una subrutina que lleve el código de tecla al registro AX, y de éste a memoria
2.
El byte del código de tecla llega al port de datos
1.
Código de la tecla A ó a supuestamente pulsada se transmite en serie
Controladora del teclado (Interfaz con electrónica intermedia) conteniendo el port de datos
Electrónica del teclado

Figura 1.62

Un periférico incluido en el gabinete de una PC, como la **unidad de disco** (duro, flexible, CD ROM), presenta una electrónica[1], que en una operación de entrada (hacia memoria) sensa los bits grabados *en serie* en un sector de una pista de un disco, uno a uno, mediante el cabezal (figura 1.63). Luego de memorizar transitoriamente en un registro estos bits del sector leído, la electrónica del periférico los envía *en paralelo*, por tandas, a través de un cable plano. Este cable se conecta mediante un conector con "agujas" a la plaqueta multifunción, donde existen interfaces para varios periféricos. De ésta, los datos leídos siguen camino –por las líneas de datos del bus al cual ella está conectada– hacia la porción central con destino a la memoria principal.[2]
Una operación de entrada implica un movimiento de datos inverso al descripto.
En la plaqueta multifunción están los ports que forman parte de la interfaz de la disquetera, mientras que en un disco rígido o en un CD tipo IDE, dichos ports están en la electrónica del periférico.

Actualmente (figura 1.5) están en la "mother" de una PC, la electrónica y conectores que antes estaban en la plaqueta dibujada en la figura 1.63, con interfaces para unidades de disco (duro y CD), disqueteras (3 ½ y 5 ¼), y las interfaces "port serie" y "port paralelo", típicamente usadas para el mouse y la impresora.

El periférico **módem** interno[3] junto con su interfaz "port serie" están en una plaqueta insertable (figura 1.64), a la cual se conecta una línea telefónica.
En una operación de entrada, la señal analógica modulada llega por dicha línea al circuito demodulador –que forma parte de la electrónica del periférico módem– que la convierte en una serie de pulsos (*conversión A/D*). Estos entran a la electrónica intermediaria, o sea a la interfaz del módem[4], en donde son convertidos en 8 bits en paralelo (*conversión serie a paralelo*) y pasados a un port de datos. Desde éste se transferirán a través de un bus al registro AX de la UCP, y luego por otro bus llegarán a memoria.
De manera inversa, en una operación de salida (figura 1.70), otro port de la interfaz recibe desde AX por el bus, un byte en paralelo, para que sea convertido en bits en serie, los cuales son modulados y transmitidos por la línea telefónica. De este modo, a través de ports de su interfaz, el módem puede enviar o recibir datos –hacia o desde memoria– usándose el registro AX como escala intermedia en la transferencia de datos entre ports y memoria principal. En la figura 1.70 se trata con mayor detalle el módem y su interfaz "port serie".

Por las particularidades que se verán en relación con aspectos comunes de los periféricos tratados, dejamos en último término el periférico **monitor** (figura 1.65). Por empezar, la denominada "plaqueta de video"[5] –que se enchufa a la "motherboard"– además de portar la electrónica intermediaria, contiene una pequeña porción de la memoria principal, denominada **VRAM** (Vídeo RAM[6]), donde se almacena la información a visualizar. Esta llega a la VRAM merced a la ejecución de un programa por el microprocesador, y es convertida en señales analógicas por la electrónica intermedia. Por tres conductores del cable que une la plaqueta de vídeo con el monitor, estas señales[7] (y las de sincronismo) llegan a la electrónica del monitor, que se encarga de convertirlas en una imagen visible en pantalla (soporte que conforma el mundo exterior, detallado en la Unidad 2). Obsérvese que de la VRAM, y luego de una conversión D/A, la información a visualizar llega directamente a la electrónica del monitor, sin que medie entre ésta y memoria principal, ningún registro port intermediario para datos, como sucedía en relación con los periféricos antes citados.

---

1 En un disco duro o un CD, esta electrónica, que forma parte de la unidad de disco se conoce como "**controladora**" inteligente, que hace que un sistema operativo pueda "ver" un disco conforme al sistema de archivos que maneja. Hoy día las más usadas se conocen con las siglas **IDE** y **SCSI** Comprende un microprocesador dedicado, registros, y una memoria buffer para almacenar temporariamente información en tránsito desde o hacia el disco. También puede contener una memoria "cache" de disco.

2 En el caso de datos provenientes del disquete, ellos no pasan por el registro **AX** (como en el caso del teclado), sino que un subsistema electrónico se encarga de llevarlos directamente a la memoria, sin intervención de la UCP, acción conocida como acceso directo a memoria - ADM. Actualmente, en una PC resulta más veloz que la interfaz de un disco rígido envíe los datos al registro AX, y de éste van a memoria, dado que los buses utilizados para hacer ADM no son suficientemente rápidos en relación con la velocidad del procesador.

3 Existen módems externos, que se conectan a una plaqueta con interfaz "port serie".

4 Esta interfaz es del tipo "port serie". La plaqueta de un módem en el presente contiene un microprocesador dedicado.

5 Conocida también como "controladora de vídeo".

6 Esta RAM, a la par que es escrita por el microprocesador (UCP) puede ser leída por la electrónica intermediaria.

7 Que gobiernan los tres cañones electrónicos que forman cada punto de color al barrer la pantalla, merced a la mezcla de tres colores: rojo, azul y verde, siendo que cada cañón al impactar genera uno de estos colores. A diferencia de otros cables que conectan un periférico con su electrónica intermediaria, por este cable se transmiten señales analógicas.

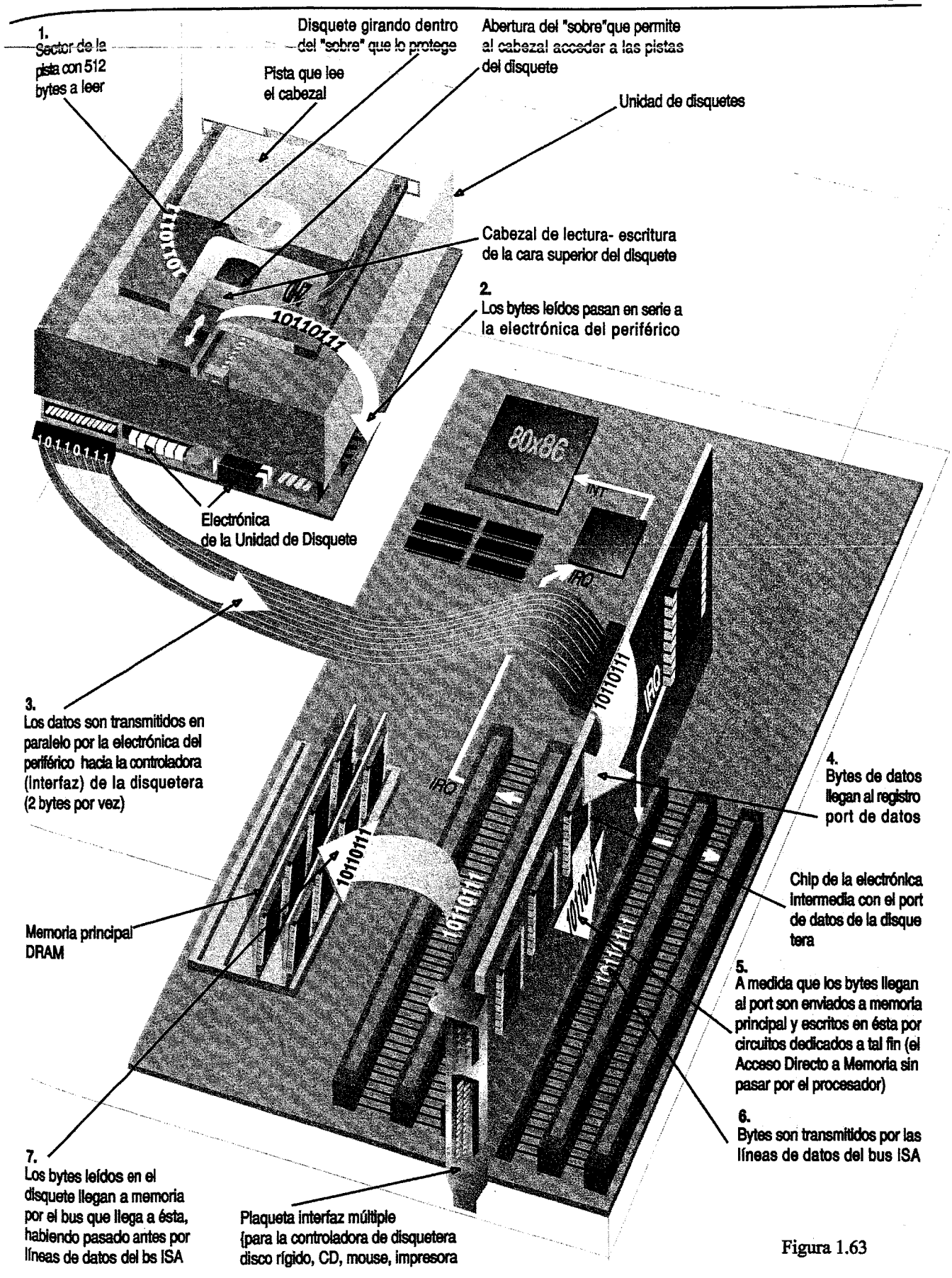

1.
Sector de la pista con 512 bytes a leer
Disquete girando dentro del "sobre" que lo protege
Abertura del "sobre"que permite al cabezal acceder a las pistas del disquete
Pista que lee el cabezal
Unidad de disquetes
Cabezal de lectura- escritura de la cara superior del disquete
2.
Los bytes leídos pasan en serie a la electrónica del periférico
10110111
Electrónica de la Unidad de Disquete
80x86
INT
IRQ
3.
Los datos son transmitidos en paralelo por la electrónica del periférico hacia la controladora (interfaz) de la disquetera (2 bytes por vez)
4.
Bytes de datos llegan al registro port de datos
Chip de la electrónica intermedia con el port de datos de la disquetera
Memoria principal DRAM
5.
A medida que los bytes llegan al port son enviados a memoria principal y escritos en ésta por circuitos dedicados a tal fin (el Acceso Directo a Memoria sin pasar por el procesador)
6.
Bytes son transmitidos por las líneas de datos del bus ISA
7.
Los bytes leídos en el disquete llegan a memoria por el bus que llega a ésta, habiendo pasado antes por líneas de datos del bs ISA
Plaqueta interfaz múltiple (para la controladora de disquetera disco rígido, CD, mouse, impresora

Figura 1.63

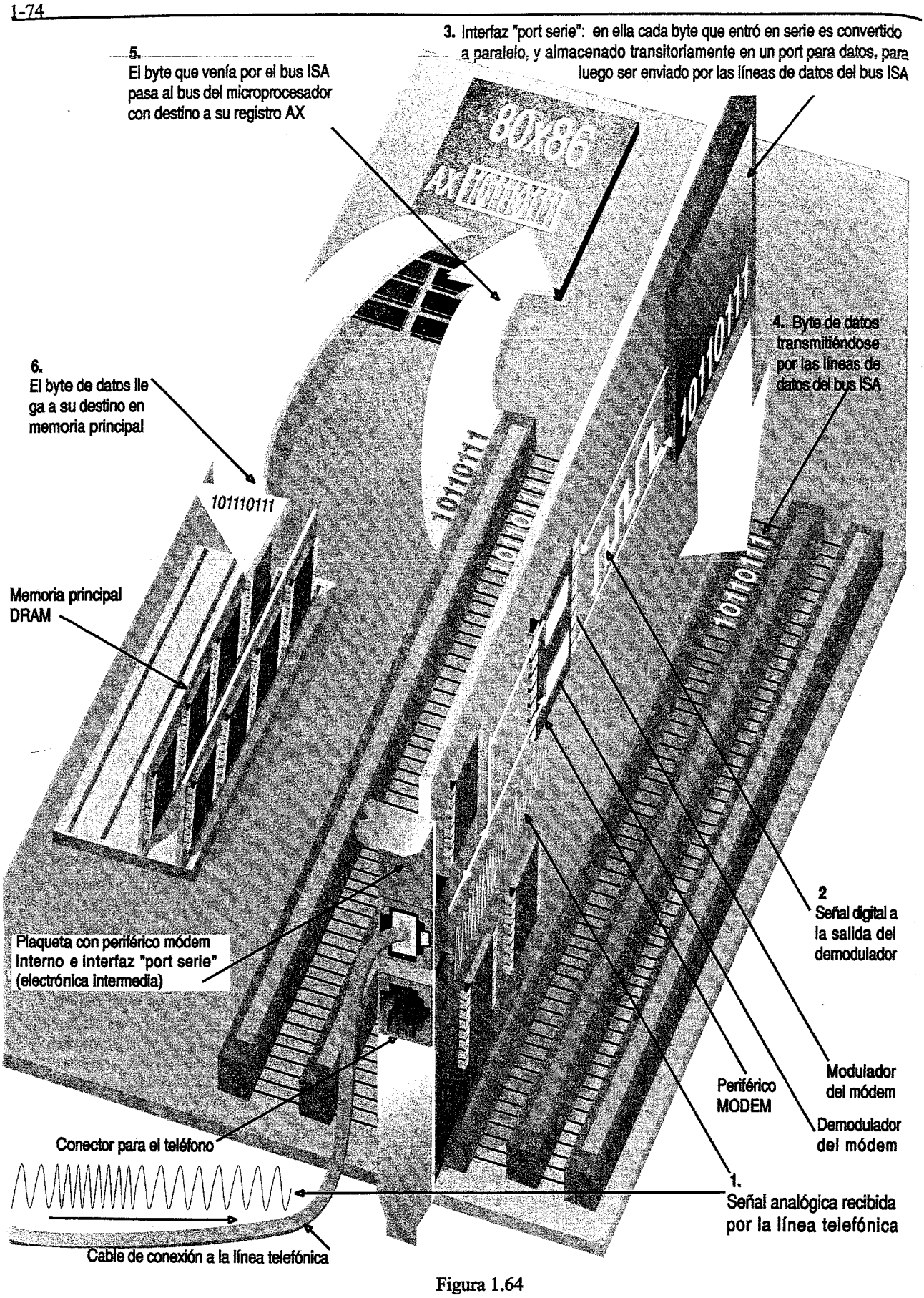

3. Interfaz "port serie": en ella cada byte que entró en serie es convertido a paralelo, y almacenado transitoriamente en un port para datos, para luego ser enviado por las líneas de datos del bus ISA
5. El byte que venía por el bus ISA pasa al bus del microprocesador con destino a su registro AX
4. Byte de datos transmitiéndose por las líneas de datos del bus ISA
6. El byte de datos llega a su destino en memoria principal
101110111
Memoria principal DRAM
Plaqueta con periférico módem interno e interfaz "port serie" (electrónica intermedia)
2 Señal digital a la salida del demodulador
Periférico MODEM
Modulador del módem
Demodulador del módem
Conector para el teléfono
1. Señal analógica recibida por la línea telefónica
Cable de conexión a la línea telefónica

Figura 1.64

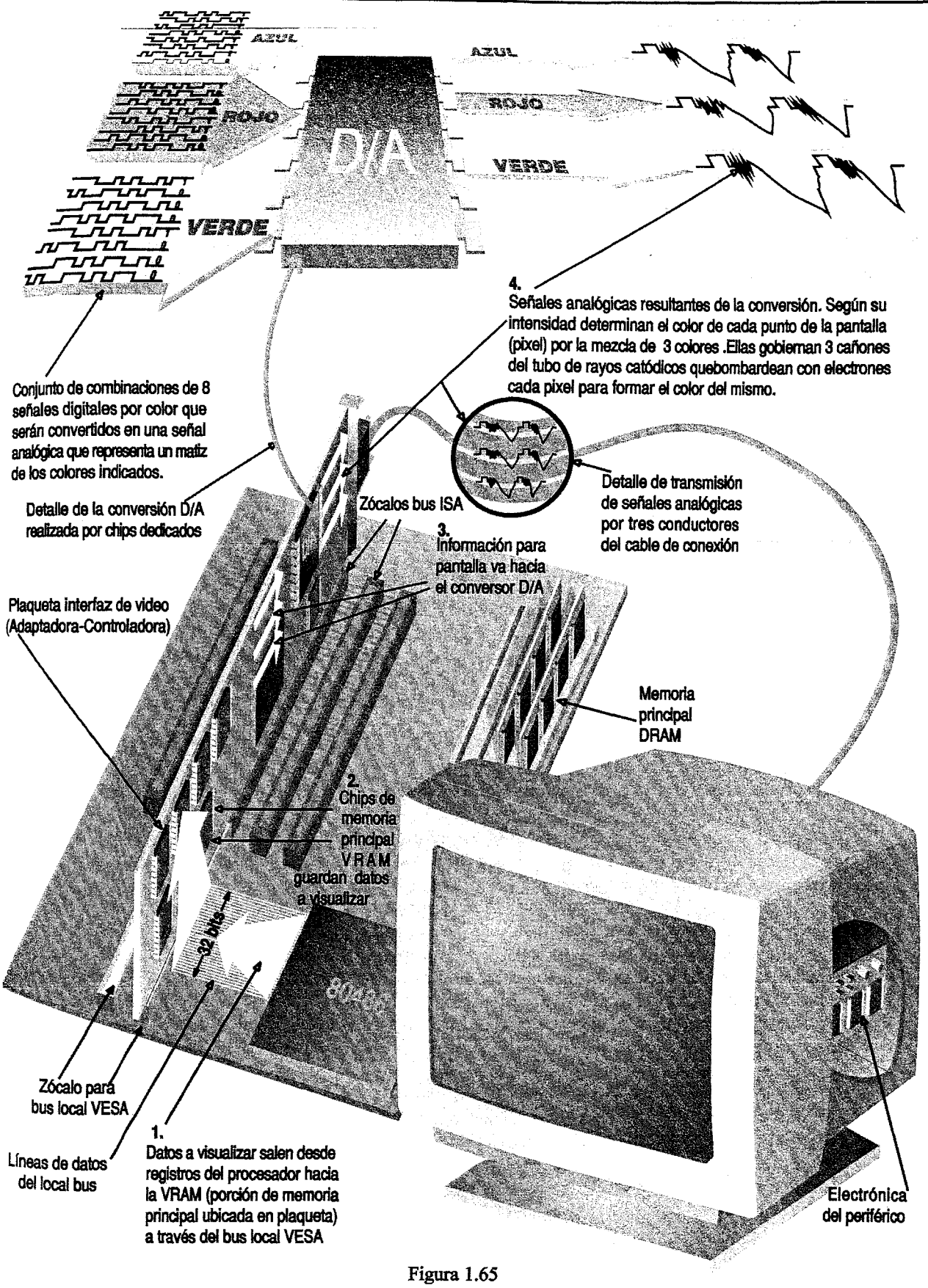

AZUL
ROJO
VERDE
D/A
AZUL
ROJO
VERDE
4.
Señales analógicas resultantes de la conversión. Según su intensidad determinan el color de cada punto de la pantalla (pixel) por la mezcla de 3 colores .Ellas gobiernan 3 cañones del tubo de rayos catódicos quebombardean con electrones cada pixel para formar el color del mismo.
Conjunto de combinaciones de 8 señales digitales por color que serán convertidos en una señal analógica que representa un matiz de los colores indicados.
Detalle de la conversión D/A realizada por chips dedicados
Detalle de transmisión de señales analógicas por tres conductores del cable de conexión
Zócalos bus ISA
3.
Información para pantalla va hacia el conversor D/A
Plaqueta interfaz de video (Adaptadora-Controladora)
Memoria principal DRAM
2.
Chips de memoria principal VRAM guardan datos a visualizar
32 bits
Zócalo para bus local VESA
Líneas de datos del local bus
1.
Datos a visualizar salen desde registros del procesador hacia la VRAM (porción de memoria principal ubicada en plaqueta) a través del bus local VESA
Electrónica del periférico

Figura 1.65

¿Qué es un port ?

En descripciones anteriores hemos usado la denominación "registro port" y "port" a secas como sinónimos[1]. Cuando se habla de port se asume que se trata de un **registro** temporario, que está en la electrónica intermediaria contenida en una plaqueta interfaz[2], o en chips de la "motherboard", *dedicado a guardar datos en tránsito entre periférico y la porción central, en una operación de entrada o salida, según sea.* Por su posición intermedia en relación con un bus de E/S y la electrónica de un periférico, *un port sólo opera con información digital*, ya sea cuando recibe (en una escritura del mismo) *o transmite* (cuando es leído).

Obsérvese que la palabra "registro" es bastante genérica. Hace mención a un circuito capaz de guardar, memorizar, un número reducido de bits (8, 16 ó 32) durante un lapso de tiempo requerido. Existen registros en la UCP, en la electrónica intermediaria, en periféricos; y también a las posiciones de memoria se las puede llamar registros.
O sea puede decirse que una memoria consta de registros. Por ejemplo, es correcto afirmar que los registros de la UCP conforman una pequeña memoria RAM, o que una memoria de un megabyte son 1.048.576 de registros de un byte. Si bien un registro posee memoria, esta última palabra se usa para denotar un lugar de almacenamiento de muchos bytes, compuesto en correspondencia por muchos registros.

Para evitar aclarar "registro de la UCP", "registro port", "registro de memoria", etc, es costumbre que:

- "registro" a secas se refiera a un registro de la UCP;
- "port" designa un registro que está en la electrónica intermedia (interfaz)
- no se denomine registro a una posición de memoria

¿Por qué operan como "buffers" ciertas zonas de memoria, los ports de una interfaz, la memoria caché y otras memorias ?

Una traducción de la palabra "**buffer**" es amortiguador, dispositivo capaz de absorber rápidamente energía proveniente de un brusco desnivel recorrido por la rueda, para luego expulsarla lentamente, de modo de no producir saltos bruscos en el chásis. Lo anterior supone una retención temporaria de la energía absorbida.

> Como se ejemplificará, en general, un **buffer** *adapta* dos velocidades distintas, permitiendo *independencia* entre un transmisor y un receptor, con el fin de que ambos puedan estar simultáneamente ocupados, sin que el más rápido pierda tiempo esperando al otro.

Un lavarropas automático cuando se desagota, envía rápidamente agua hacia una pileta (buffer), que retiene temporariamente toda el agua enviada, de modo que la rejilla pueda tomarla a un ritmo apropiado para no desbordar. Esto permite que mientras el agua pasa a la rejilla, el lavarropas realice otra operación, sin que tenga que enviar el agua lentamente, para adecuarse a la velocidad con que toma la rejilla. O sea que se *independiza* una velocidad de otra.
En el ejemplo anterior el transmisor es rápido y el receptor lento. Una situación opuesta ocurre cuando una canilla envía poca agua por segundo, y colocamos un balde para que se vaya llenando. Mientras esto tiene lugar podemos realizar otra tarea, conforme a nuestro ritmo de trabajo, sin estar inactivos esperando.
Una vez que el balde se llenó podemos usar el agua contenida, arrojando de una sola vez 10 ó 20 litros en un segundo. El balde es un buffer entre la canilla y la persona.
Igualmente un buzón acomoda, independiza, el ritmo de envío de cartas de las personas, con la velocidad del servicio del vehículo recolector, que recoge de una sola vez la bolsa con cartas a una determinada hora.
En computación, se empezó a usar la denominación **memoria buffer** para zonas de la memoria principal donde se guardan datos *en tránsito*, desde o hacia periféricos. Por ejemplo, permanecen en una de estas zonas datos que van llegando a una cierta velocidad provenientes de un disco, para ser procesados luego, a gran velocidad, mediante la ejecución de algún programa por la UCP. Mientras los datos van arribando a memoria, la UCP puede dedicarse a ejecutar otro programa.

---

1 En algunas publicaciones se llama port al conector al cual se enchufa el periférico.

2 El port del teclado está en la controladora del mismo, situada en la motherboard.

Otra zona de la memoria está destinada a datos a imprimir, que un programa deja rápidamente en ella, para que sean enviados a una impresora a una velocidad en promedio menor. El teclado también tiene reservada en memoria una zona buffer para los códigos de teclas pulsadas.
En general, cada periférico puede tener en memoria principal su zona buffer.

También puede designarse buffer a un registro o registros que almacenen temporariamente información en tránsito, con el fin de *adaptar, independizar,* velocidades.
Con este sentido, son buffers los registros que forman parte de cada electrónica intermedia, ubicada entre su periférico y la porción central, como se estableció. *O sea que los ports son buffers.*

Desde la primer generación de computadoras *los periféricos **no** se conectan directamente a memoria principal,* para entrar o sacar datos de ella[1], *sino que reciben o envían datos desde o hacia memoria a través de registros ports intermediarios,* que almacenan transitoriamente los mismos.
Dada la gran diferencia de velocidades de operación y transmisión de datos, entre la parte central de un computador (totalmente electrónico) y sus periféricos (con partes con movimientos mecánicos o accionados por personas, como el teclado), se impone necesariamente la existencia de buffers (ports) entre ambos, para poder almacenar la información en curso, durante el tiempo necesario.
Por ejemplo, en una PC se usa el bus común ISA[2] para entrar datos desde el teclado, o enviarlos hacia una impresora. Por ese bus es factible transmitir como máximo 1 MByte/seg. = 1 byte/ microsegundo, o sea también *una letra cada microsegundo. Si se piensa los pocos caracteres (bytes) por segundo que se pueden tipear desde un teclado, o que una impresora con cabezal móvil puede imprimir 200 a 300 caracteres por segundo, se comprende la diferencia abismal de velocidades que puede llegar a existir entre el mundo interior electrónico de un computador y el mundo exterior.*

Otra situación se da en el caso de periféricos que en muy breves lapsos pueden transmitir datos a velocidades cercanas a las que tienen lugar en la porción central.
Suponiendo una memoria principal DRAM con 70 nanoseg. de tiempo de acceso, sería factible realizar lecturas o escrituras sucesivas, a razón de una cada 100 nanoseg = 0,0000001 seg.[3]
O sea que en dicha DRAM teóricamente sería posible realizar un máximo de 10 millones de lecturas o escritura por segundo. Si en cada una se leen o escriben 2 bytes, en un segundo se podrían leer o escribir 20 millones de bytes desde o hacia memoria principal. Esto es, desde o hacia esa DRAM podrían transmitirse hasta cerca[4] de 20 MBytes/seg = 20 Bytes/microseg.[5] en un sentido u otro, según se lea o escriba.
Cuando en una lectura de un disco rígido la cabeza pasa sobre un sector de una pista, puede enviar hacia la electrónica de ese periférico una copia de los bits almacenados a una velocidad de transferencia de por ejemplo 5 Bytes/microseg. Si se usa bus local para enviarlos hacia la porción central, la velocidad de transferencia por dicho bus puede ser hoy de por lo menos 3,5 MBytes/seg = 3,5 Bytes/microseg.
Estas velocidades se acercan a los 20 Bytes/microseg. recién calculados, con que se pueden leer o escribir datos en forma continua en memoria. Por razones de disponibilidad de ésta, los datos que llegan desde la electrónica del disco deben aguardar unos breves instantes antes de pasarlos a memoria. Entonces, temporariamente hay que *retener* datos, hasta que la memoria pueda recibirlos. Esto podría asemejarse a la situación en que un jugador de fútbol recibe la pelota que otro le pasa, y debe retenerla un instante, hasta que un tercero pueda recibirla.

*En una plaqueta interfaz, además del port para datos en tránsito, existen otros registros para retener transitoriamente las órdenes (***comandos***)* que provienen de la ejecución de programas (de la ROM BIOS). Estos programas ordenan las operaciones que debe realizar el periférico conectado a dicha plaqueta, o sea relacionadas con el **control** del mismo. Por tal motivo se conocen como **"ports de control"**

Así, a la interfaz (controladora) de una unidad de disquetes debe indicársele, como ser: que active el motor, el cilindro, pista y sector a acceder, y si debe leer o escribir. Los códigos de estos comandos

---

1 El monitor, por ser un dispositivo totalmente electrónico, es el único periférico que no tiene port de datos como buffer adaptador de velocidades. Los datos a visualizar de la zona de memoria principal reservada para video se convierten en señales para el monitor.

2 Industrial Standard Architecture, o bus del sistema.

3 En una memoria DRAM se debe esperar entre una lectura o escritura y la siguiente, por lo cual se ha estimado una cada 100 nseg en lugar de una cada 70 nseg.

4 Recordar que se define 1 MB como 1024x1024 bytes y no 1.000.000 de bytes.

5 Puesto que un segundo es un tiempo muy largo en un computador, y que la memoria tiene unos pocos megabytes de capacidad, conviene pensar la velocidad anterior en su equivalente 20 Bytes/microsegundo. Del mismo modo, no da mucha idea decir que podemos arrojar 600 litros de agua en un minuto, en vez de su equivalente real de 10 litros por segundo, suponiendo baldes llenos de esta capacidad.

llegan velozmente desde el registro AX de la UCP, merced a la ejecución de instrucciones (tipo OU de programas de la ROM BIOS[1]. Estos comandos permanecen en los registros destinados a ellas en interfaz, denominados **ports de control**, hasta que la electrónica del periférico los lleve a cabo.
A su vez, la electrónica del periférico informa del **estado** (*"status"*) del mismo (si está listo o ocupado o si hay algún problema (en una impresora: falta de papel, fuera de línea, etc) en un registro port de interfaz, que es leído por dichos programas antes de transferir datos, conocido como **"status register**

Otro buffer "inteligente" que está presente en las PC actuales, es la **memoria "caché"** (a tratar interpuesta entre la memoria principal y la UCP (figura 1.77) El caché recibe de memoria princip DRAM, a una cierta velocidad, las instrucciones que serán ejecutadas próximamente por la UCP, los datos que serán procesados por ellas. Como ser, al triple o más de velocidad, la UCP pued tomar del caché dichas instrucciones y datos, dado que se trata de una memoria SRAM (Stat RAM), de acceso más rápido que la DRAM de memoria principal. A ésta pasarán más lentamen desde el caché los resultados que la UCP escribió velozmente en el mismo.

También en la electrónica de periféricos, como las unidades de discos, la impresora, y el teclado, existe *memorias buffers*, para guardar temporariamente información, en tránsito hacia o desde el medio exterior

¿Qué son las direcciones de los ports de una interfaz, y cómo se vincula ésta con la porción central a través del bus al cual se conecta?

Como se discutió, una plaqueta interfaz cumple funciones de *intermediación*, no sólo respecto a los datos en tránsito que retiene brevemente, *sino también en relación con las órdenes hacia el periférico, y con la indicación del estado de operatividad de éste.*

Los registros **ports para datos** y **ports para control**[2] de una plaqueta interfaz (*"electrónica intermedia"*) constituyen una parte esencial de la misma, y están vinculados directamente con la electrónica del periférico conectado a ella.
El número de estos registros en general no llega a diez, y está en relación con la cantidad de comandos a enviar a la electrónica del periférico, y a la cantidad de información que envía ésta hacia registros ports de la interfaz que indican el "status" del periférico (**port de status**)
Al tratar luego los ports serie y paralelo se dan detalles de estos registros, presentes en todas las interfaces o "controladoras", como quiera llamarse.

Cada interfaz (figura 1.66, que es un detalle de la figura 1.7) puede verse en parte como una pequeña memoria RAM, cuyos registros ports se pueden leer o escribir *del mismo modo que si fueran posiciones de memoria, para lo cual cada port tiene asignado en una interfaz un número fijo, que es su dirección.*

Un cable de lectura o escritura (L/E = 1/0) que llega de la UCP, ordena la lectura o escritura de un port. En la figura 1.66 aparecen las direcciones 0278, 0279, 027A de los ports de una interfaz "port paralelo".

Al igual que la memoria, una interfaz está ligada a la UCP (microprocesador) a través de un bus (figuras 1.7 y 1.66), con líneas para direcciones, datos y control. Entre estas últimas se tiene las líneas **L/E** e **IRQ**. Por ejemplo, la interfaz "controladora" de las unidades de disquetes ("drives"[3]) de 3 ½" y 5 ¼", por un lado se comunica con el bus ISA en cuyos zócalos se inserta (figura 1.63), y por otro –a través de un cable plano– con la electrónica del periférico, que controla el movimiento del motor, cabezal lectura/escritura del disquete, y otras acciones.

---

[1] Es importante recalcar al respecto que **la UCP no da órdenes a los periféricos. De la ejecución que la UCP realiza de programas para E/S (entradas o salidas) llegan a estos ports reservados para órdenes, las operaciones que debe realizar el periférico correspondiente. La UCP no comanda ni controla por hardware a los periféricos. La UCP ejecuta los programas que ordenan dichas operaciones mediante comandos que llegan a registros ports destinados a ellas, ubicados en cada interfaz.**

[2] La denominación "**controladora**" para algunas de estas plaquetas, tiene relación con las órdenes de funcionamiento y operación que llegan a estos registros, generadas durante la ejecución de programas, mayormente residentes en la ROM BIOS

[3] No confundir con *"driver"*, que es un programa para manejar un determinado periférico.

Otra función esencial de una plaqueta interfaz, es generar por una línea la señal que se envía a la UCP para solicitarle interrupción del programa en curso de ejecución (**IRQ –interrupt request–** en las PC). Por ejemplo, cuando en el periférico ligado a una de estas plaquetas se concretó algo: en el teclado se apretó una tecla, una impresora está lista para recibir más datos, una unidad de disquetes terminó de leer o escribir, etc.

En las figuras 1.62 y 1.63 se muestra la línea IRQ saliendo de la electrónica de una interfaz, para luego continuarse primero en la correspondiente línea de control del mismo nombre del bus ISA, y luego en la línea que llega hasta un chip, al cual llegan todas las líneas IRQ de las distintas interfaces. Este chip oficia de *"árbitro de interrupciones"*, para el caso que ocurran dos o más simultáneamente, y su línea de salida va a la entrada INT del microprocesador, a fin de solicitar interrupción transitoria del programa en ejecución.

¿Cómo se escribe o lee desde un microprocesador 80x86 un registro port mediante las instrucciones IN y OUT ?

Según se vio (figura 1.15), el código de máquina A10050 de la instrucción $\mathbf{I}_1$ antes usada, comprende el código de operación A1, siendo que 0050 permite determinar la dirección de memoria (5000) donde está el dato que debe ir al registro AX de la UCP. O sea, que para una dirección XXXX cualquiera, el código de máquina genérico será A1XXXX, siendo A1 el código que ordena la operación a realizar.
Para evitar tratar con códigos numéricos, esta orden A1, de mover un dato de memoria hacia AX se abrevia **MOV** en lenguaje assembler, que codifica abreviadamente la operación que ordena ese tipo de instrucción.
Del mismo modo, en forma general, el código de operación de sumar (adicionar) es **ADD**, restar es **SUB** (sustraer), saltar es **JUMP**, etc.

En las **operaciones de entrada** de las figuras 1.60, 1.62 y 1.64, datos son enviados por la electrónica de un periférico hacia el port de datos de la interfaz, a la cual el mismo está conectado. Luego *el dato pasa del port al registro AX*, a través del bus ISA, y del bus del procesador. *Este movimiento se realiza cuando la UCP ejecuta lo así ordenado por el código de una instrucción abreviada* **IN** (de "input") en assembler, que forma parte de una subrutina, por ejemplo una de la ROM BIOS.
Una vez que un dato llegó al registro AX, la operación de entrada se completa mediante la ejecución de otra instrucción de movimiento, como $\mathbf{I}_4$ analizada, que ordena escribir en memoria dicho dato que está en AX.
De esta forma se completa la operación de entrada, que comenzó cuando el periférico envió el dato al port.

Las **operaciones de salida** de las figuras 1.61 y 1.65 empiezan con un movimiento desde memoria hacia AX, que se realiza merced a la ejecución de una instrucción de movimiento, como $\mathbf{I}_1$ tratada. Luego *el dato va desde AX hacia el port de datos* de la interfaz a la que se conecta el periférico. *Para que este movimiento tenga lugar, debe ejecutarse el código de una instrucción de abreviatura* **OUT** (de "output")
Luego que el dato llegó al registro port, la electrónica del periférico lo tomará (leerá) del mismo.

Por lo tanto, la ejecución de instrucciones del tipo **IN** sirve para transferir datos desde un registro port hacia el registro AX, o sea para *leer* el contenido de un port; y las del tipo **OUT** para movimientos en sentido contrario, vale decir para *escribir* un registro port.

Antes que tenga lugar una operación de entrada o salida, se le deben enviar comandos a la electrónica del periférico involucrado, a fin de indicarle qué debe hacer[1]. Para ello *estos comandos deben llegar hasta los registros ports* (para comandos) de la interfaz a la cual el periférico se conecta.

Esto se logra, como en una operación de salida, pasando primero el comando desde memoria al registro AX –ejecutando una instrucción tipo $\mathbf{I}_1$– seguida de una instrucción tipo **OUT**, para que el comando pase de AX a un port de comandos. Esto se ejemplifica en la figura 1.67)
Para ejecutar una instrucción **IN** (lectura de un port de dirección XXXX indicada en la instrucción, y envío del contenido del port al registro AX), la UC ordena las siguientes acciones básicas (figura 1.68):

- Llevar al valor **1** el cable L/E de la UC que llega a todas las interfaces.
- Enviar por las líneas de datos la dirección XXXX (0279 H en la figura 1.68).
- Destinar al registro AX el dato contenido en el registro port, enviado por las líneas de datos del bus.

[1] Así, para las unidades de discos o disquetes se debe indicar el número de cilindro, pista y sector; y si se lee o escribe el sector.

Se trata, pues, de una operación muy semejante a la de lectura de una posición de memoria (figura 1.11)

La ejecución de una instrucción **OUT** (escritura de un registro port de dirección XXXX indicada en la instrucción, con una copia del contenido del registro AX), supone las siguientes acciones ordenadas por la UC:

- Por las líneas de direcciones del bus enviar la dirección XXXX. (0278 H en la figura 1.66).
- Enviar al port, por las líneas de datos, una copia del contenido del registro AX.
- Poner en **0** el cable de control L/E que llega a la interfaz donde está el port, para ordenar escritura.

Esta instrucción ordena un movimiento desde AX al port, o sea opuesto al ordenado por **IN**.

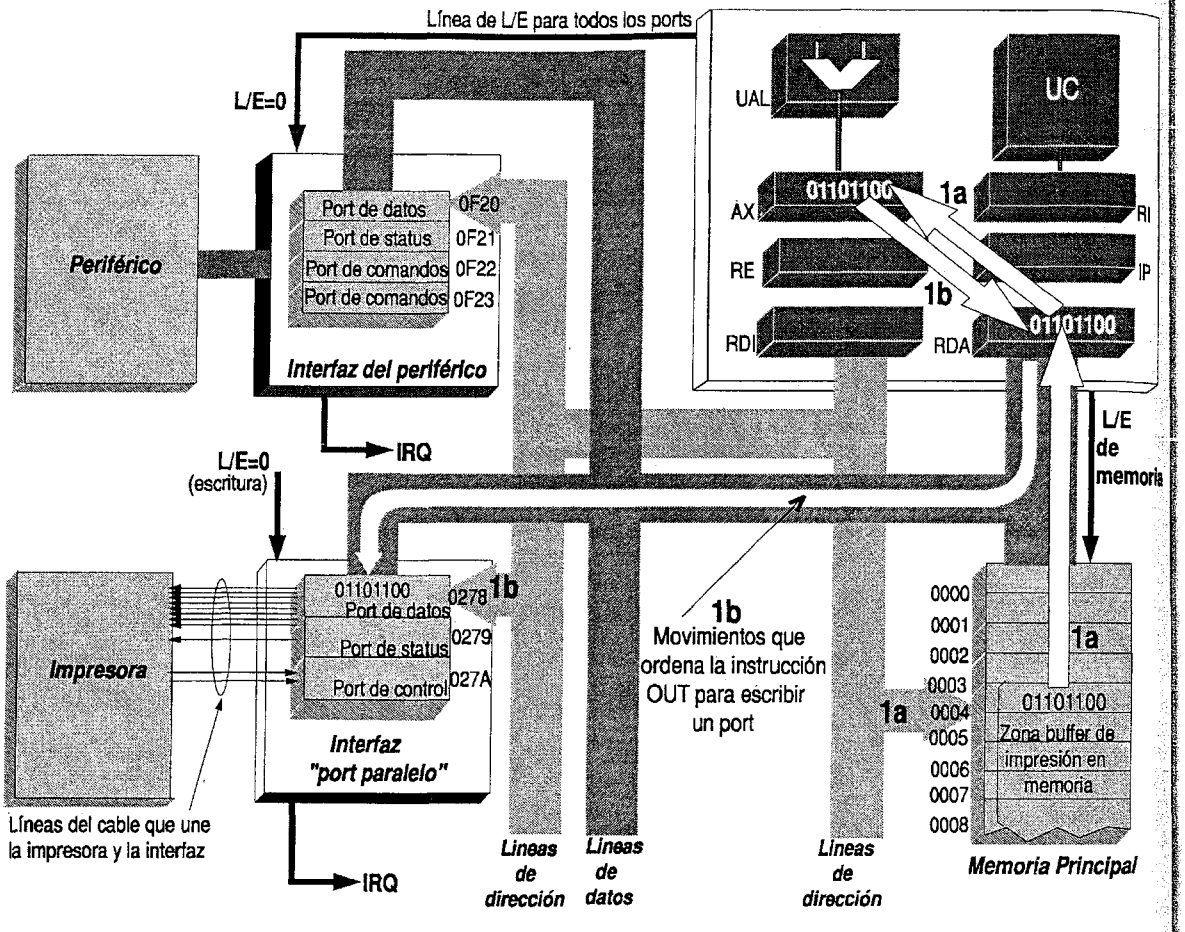


Figura 1.66

Obsérvese *que la "jurisdicción" de la UC llega hasta los registros ports,* siendo que *hasta ellos llega la línea de control L/E* que sale de la UC, pudiéndose así leer el contenido de un port (o cambiarlo en una escritura). A los periféricos no llega ninguna línea de control de la UC.
La UC ***no*** *tiene comunicación directa mediante línea alguna con ningún periférico.*

Esto es, la UC no ordena qué debe hacer un periférico mediante líneas que salen de ella. Esto se realiza al ejecutar la UC instrucciones como la **OUT**[1] citada, que envía una combinación binaria (comando) hacia un port de la interfaz del periférico, para ordenar qué debe efectuar la electrónica del periférico.[2]

A través de los ports la UC se comunica con los periféricos para enviarles comandos y datos.

[1] En una PC las instrucciones tipo OUT ó IN en general forman parte de subrutinas escritas en la ROM BIOS.

[2] **Del mismo modo que cada instrucción (software) que llega al registro de instrucción ordena qué debe hacer la UCP, cada comando que llega a un port de control (generado mediante la ejecución de software) ordena qué debe hacer un periférico determinado**

Hasta los ports puede llegar directamente la UC en relación con los periféricos. Cada port es seleccionado por la UC mediante la dirección que lo identifica, la cual es enviada por las líneas de direcciones del bus ligado al port.

*Los registros ports de todas las interfaces constituyen una suerte de "**frontera**", hasta la cual la UC puede enviar comandos y datos a velocidad electrónica, sólo limitada por las características del bus.*

Igualmente a tal velocidad puede tomar datos de un port. A su vez, dada la función buffer antes descripta de un port de datos (o de comandos), la electrónica del periférico lee un dato (o un comando) a la velocidad con que opera el mismo, en general varios órdenes de magnitud menor, por depender de sistemas mecánicos que gobierna (salvo el monitor que es totalmente electrónico)[1].

¿Qué se denomina "port serie" y "port paralelo" ?

Como se verá, *una y otra denominación en realidad se refieren a un tipo de interfaz*, a la que pueden conectarse distintos tipos de periféricos[2], constituida por *varios* registros ports direccionables.
La primer distinción básica, que hace a su denominación, es que el **"port serie"** recibe (o transmite), bit por bit, *por un solo cable* los datos que transmite (o recibe) la electrónica del periférico conectado al mismo[3].
En cambio en un **"port paralelo"**, esta transmisión de datos entre ambos subsistemas tiene lugar a través de 8 cables al mismo tiempo (figura 1.67), o sea puede realizarse más rápido.
En lo referente a su comunicación con el bus al cual se vincula – o sea donde se conecta la plaqueta que lo contiene– tanto el "port serie" como el "paralelo" *se conectan en paralelo al mismo*, típicamente a través de 8 líneas de datos, según el tipo de bus. Esto debe ser así, pues en cualquier caso, la porción central envía o recibe rápidamente información (datos o comandos según sea) hacia o desde un registro port.

Como se anticipó, los denominados "port serie" y "port paralelo" *en realidad son interfaces* conformadas por **varios** registros ports direccionables de la forma vista. En el "port serie" los datos se transmiten por un cable, bit a bit, al registro port de datos, pero los comandos para el periférico correspondiente llegan en paralelo desde la porción central, según se verá.
Por la relativa reducida velocidad de transmisión con que operan, el mouse y el módem son los periféricos que típicamente se conectan a un "port serie" distinto cada uno. Periféricos más rápidos como la impresora, reciben datos de a 8 bits desde un "port paralelo". También es común conectar a éste una unidad de cinta magnética para copias de resguardo, o puede conectarse una computadora tipo notebook para enviar directamente archivos a otra, estando ambas próximas.
Asimismo, dado que por razones de ruidos electromagnéticos inducidos, cables en paralelo sólo pueden transportar señales hasta unos 4 mts, si se quiere conectar directamente una impresora a un computador a mayor distancia (hasta unos 20 mts), se debe usar un "port serie", a costa de una menor velocidad de transmisión de datos hacia ella.

La interfaz "port serie" es mucho más compleja que la "port paralelo" (puede contener hasta 10 registros port direccionables), por tener que manejar el protocolo RS232C, como se verá al tratar esta interfaz.

---

1 En este sentido la denominación "port", traducible como "puerto", permite realizar el siguiente paralelo conceptual. Un puerto de un país está en la frontera entre la tierra y el mar, y se comunica con el resto del país a través de carreteras (buses en un computador). Por ellas las mercaderías pueden transportarse hasta el puerto en vehículos a decenas de km/h. Luego las mismas se almacenan temporariamente en el puerto, y se transportan en embarcaciones viajando a velocidad mucho menor. Igualmente vale la analogía para el movimiento desde el mar hacia el territorio, con escala en el puerto

2 El "port" paralelo se conoce también como interfaz **"Centronics"**, marca de impresoras que lo popularizó; y al "port" serie puede aparecer como port **RS232C**, siglas de las normas que especificaron por primera vez una interfaz serie. Sería correcto llamarlos **interfaz serie** e **interfaz paralelo**, respectivamente.

3 Así se transmiten por ejemplo los 8 bits de un carácter codificado en ASCII, con bits adicionales de comienzo y final. Un "port serie" es *asincrónico* en el sentido que el primero de esos bits puede llegar en cualquier instante, dado que el mismo no necesita estar en sincronismo con ninguna otra señal que vaya por otro cable extra, como ocurre con cada bit que envía en serie el teclado a su port. Para poder identificarlos, tanto el primer bit como los diez o doce que se usan para transmitir un carácter *deben ser de igual duración* en el tiempo.

¿Cuál es la estructura interna de una interfaz "port paralelo", y qué protocolo cumple una impresora conectada a ella ?

Este tipo de interfaz (figura 1.66) consta de tres registros ports: para datos, "status" y control (por ejemplo de respectivas direcciones 0278_H, 0279_H y $027A_H$ en una PC)[1]
Estos tres ports, por un lado, se pueden comunicar con el registro AX, a través de solamente 8 líneas de datos del bus, para ser leídos o escritos mediante la instrucción **IN** o **OUT**, respectivamente, según sea.
Por otro lado (figura 1.66), estos ports están conectados –conector por medio– a conductores del cable que une la electrónica de la impresora con la plaqueta (típicamente la "multifunción", donde existe una interfaz "port paralelo". En el presente el conector y esta interfaz están en la "motherboard"(figura 1.5).

En lo que sigue, se detalla el funcionamiento típico de la interfaz "port paralelo": conectada a una impresora. Las instrucciones indicadas, así como el código del comando enviado a la *interfaz forman parte de una subrutina de la ROM BIOS de memoria principal. En las figuras sólo aparece el movimiento de datos.*

1. Desde la zona buffer de impresión de memoria principal de la impresora llega al registro AX un byte a imprimir, merced a la ejecución de una instrucción que ordena este movimiento (**1a**), como la I_1 antes vista. La ejecución de la instrucción siguiente –tipo **OUT**– ordena escribir dicho byte en el *port de datos* de dirección 0278_H (movimientos **1b** de la figura 1.66). Este port guarda temporariamente el byte a imprimir.[2] (Sigue en figura 1.67).

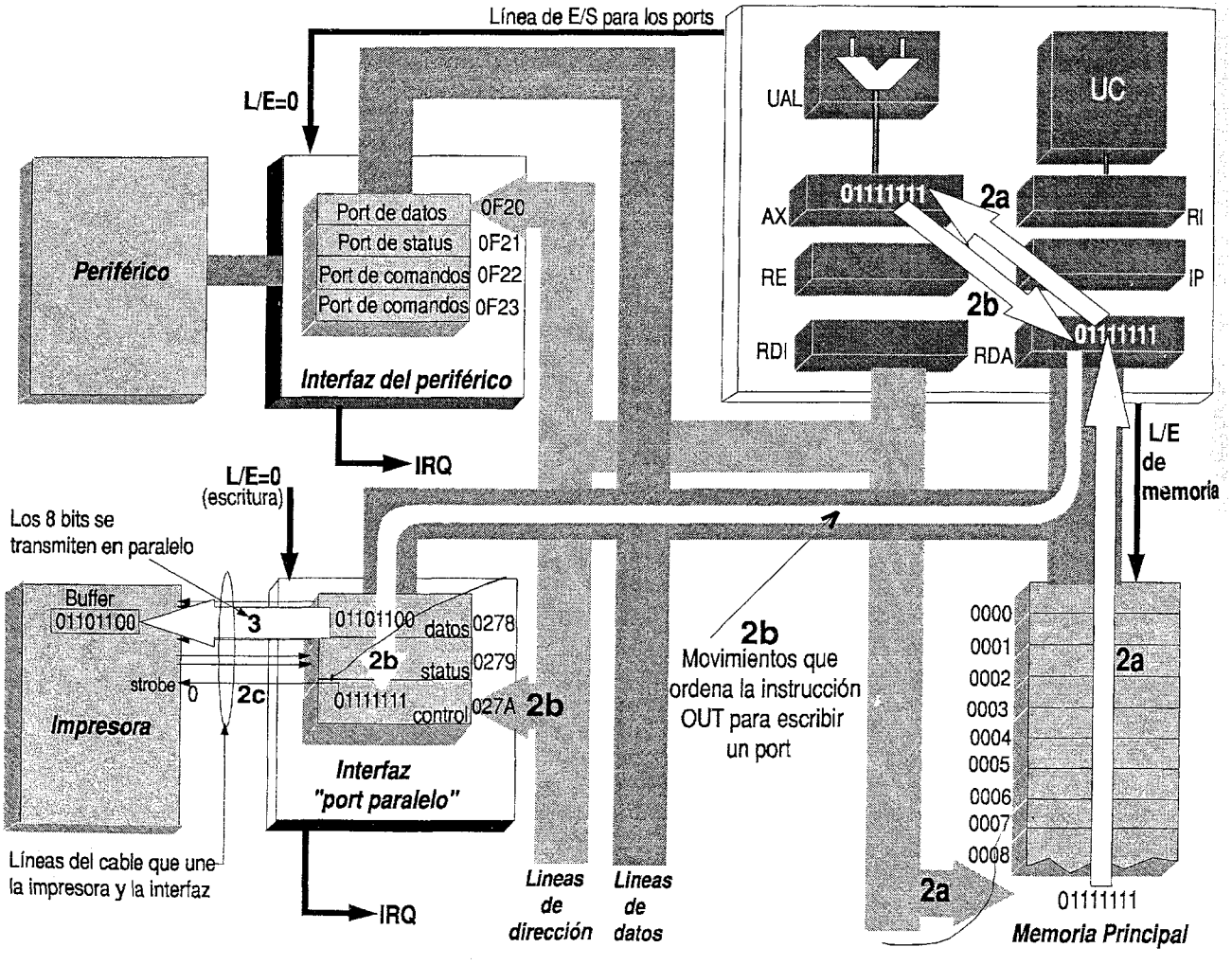


Figura 1.67

[1] En una PC puede existir hasta 4 de estas interfaces designadas con el nombre lógico LPTx, siglas de Line Printer.
[2] Si se quiere entrar datos desde otro computador usando esta interfaz, se debe usar el port de status para entrar los datos.

2. A continuación (como ser una millonésima de segundo después), se ejecuta nuevamente una instrucción de movimiento tipo I_1, que ordena pasar al registro AX el byte *de comando* 01111111 (**2a**). Este byte –merced a la ejecución de otra **OUT**– viaja por las líneas de datos del bus, y se escribe (**2b**) en el *port de control* de dirección 027A . El primer bit de este byte ligado a la línea STROBE llega (**2c**) con el valor **0** [1].

3. Al detectar la electrónica de la impresora que STROBE=0, toma del port de datos los 8 bits llegados en el paso 1. Estos viajan por los 8 conductores (**3**) del cable de conexionado, transmitiéndose así (en otra millonésima de segundo) hacia un buffer, que forma parte de dicha electrónica.

4. Inmediatamente (**4a**), la impresora envía a la plaqueta con la interfaz: un **0** por la línea ACK(nowledge), reconociendo que recibió el byte enviado en el paso 1, y un **0** por la línea BUSY, para indicar que está ocupada. Estas dos líneas llegan a los correspondientes bits del *port de "status"*, por lo que ambos ceros quedan escritos en el mismo (**4b**), resultando la combinación 00111111.[2]

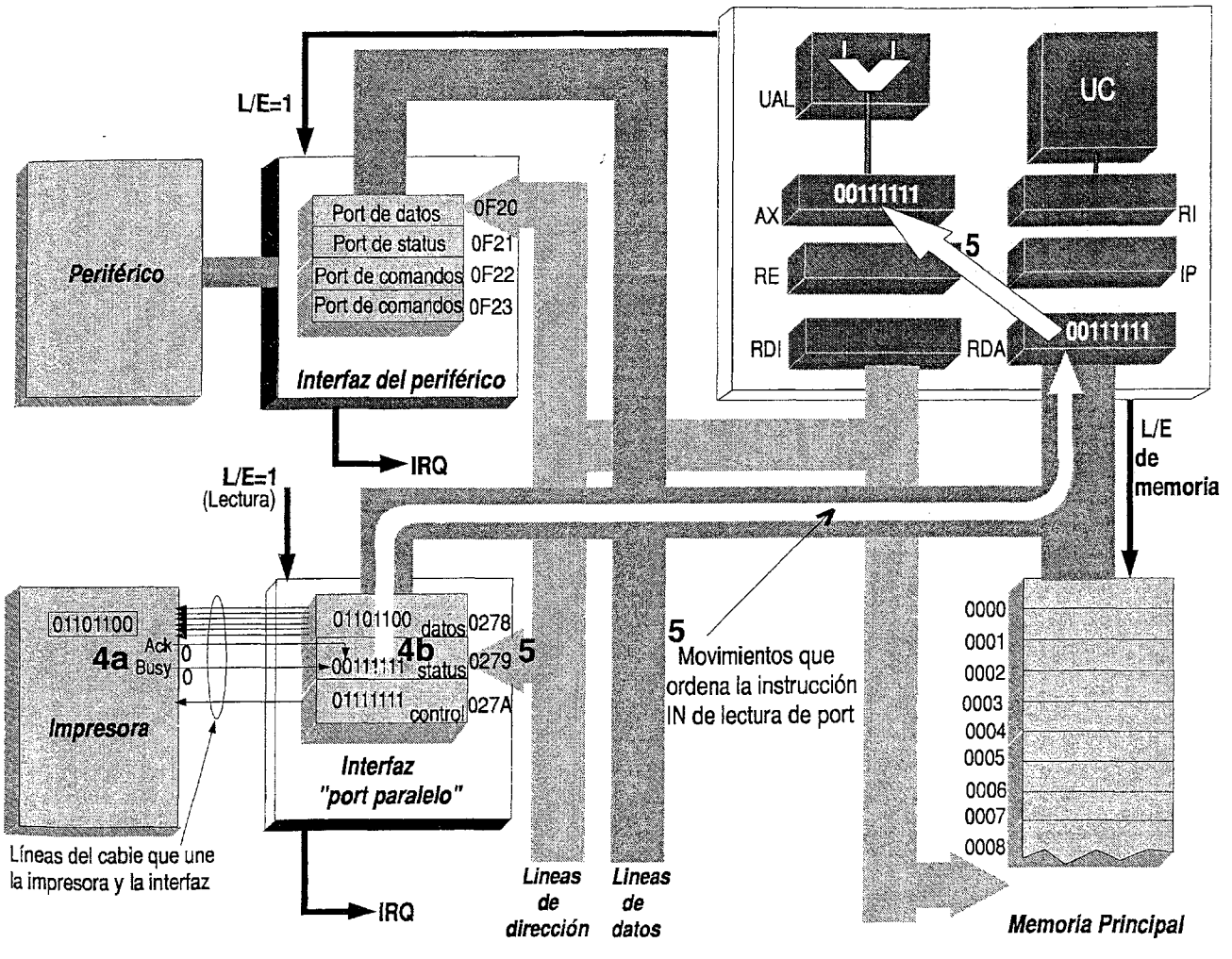


Figura 1.68

[1] Además de STROBE pueden enviarse al port de control otros bits de comando, como: un bit de **AUTO FEED**, que ordena cambiar de renglón luego de recibir la orden CR (Carry return); el bit de **SeLeCT IN** que ordena poner fuera de línea ("off line") la impresora; el bit de **INIT** ordena reinicializar la impresora (por ejemplo en modo texto).

[2] Otros bits de estado de la impresora, que la electrónica de ésta envía al port de status son: **PE** (paper error) indica falta de papel; **ERROR** indica que no se puede imprimir por otro problema distinto de PE; **SeLeCT** indica al procesador con *1* si la impresora está en línea (on-line) o no (off) según el estado de la llave correspondiente ubicada en el gabinete de la impresora.

5. Por la rapidez con que el microprocesador ejecuta instrucciones, se debe controlar de no enviar otro byte al port de datos (mediante el paso **1.**) antes que la impresora haya tomado el anterior (paso 3 finalizado). A tal efecto, la subrutina del BIOS contiene instrucciones para leer repetidamente el port de "status", de dirección 0279 (ejecutándose **IN**). Los 8 bits de este port (figura 1.68) llegarán por el bus de datos al registro AX (**5**), para determinar si cambiaron del valor 00111111 (indicador que llegó el byte enviado en el paso 1, pero que aún no se puede enviar otro) al valor 11111111. Esto último implica que la impresora puso en **1** el bit BUSY del port de "status", por que puede guardar otro byte a imprimir en su buffer)
6. La determinación anterior (situación no dibujada) permite conocer cuándo la impresora puso en **1** el bit BUSY del port de "status" (que pasará a contener 11111111), lo cual implica que está lista para guardar otro en su buffer, por lo que se le puede enviar un nuevo byte a imprimir, mediante al paso **1.**

Esta secuencia preestablecida de señales a enviar y recibir (*"handshaking"*) entre la subrutina y la impresora, constituye el "protocolo" para impresión por el "port paralelo".

En general un **protocolo** es un sistema de reglas y procedimientos que gobierna la comunicación entre dos o más dispositivos.

Puede estimarse en 10 microsegundos el tiempo que insume el protocolo, entre el envío de un byte a imprimir y el siguiente. Esto implica la posibilidad de enviar por un "port paralelo" unos 100.000 bytes/seg, cifra que en la práctica es mucho menor cuando opera una impresora.

El "port paralelo" presenta –como toda interfaz– una línea de solicitud de interrupción (**IRQ**), que forma parte de las líneas de control del bus. Ella se activa cuando se puede enviar otro byte al port de datos.
En una PC el envío de cada byte a imprimir no se hace por interrupción, sino por *"polling"*: instrucciones de la subrutina del BIOS leen repetidamente el valor del bit BUSY, hasta detectar que cambio de valor. Entonces se envía otro byte al port

¿Cuál es la estructura interna de una interfaz "port serie", y cómo está preparada para conectar un módem usando el protocolo RS232C ?

Por la **interfaz serial** denominada **"port serie"** o **de comunicación** (COM 1, 2, 3 ó 4 en una PC) pueden *entrar* bits *en serie* desde el exterior y transmitirse *en paralelo* (8 bits juntos) *hacia la porción central;* o bien desde ésta salir hacia ella 8 bits *en paralelo* y transmitirse *hacia el* exterior *en serie.*
Esta interfaz está preparada para cumplir con las normas **RS232C** en lo referente a conectores, señales eléctricas, protocolos de transmisión y verificación de errores. Por lo tanto, ella *puede ser interfaz de un módem, o de cualquier dispositivo que envíe o reciba datos en serie conforme a dichas norma*
Si bien el módem es el periférico que utiliza la mayoría de las líneas de señal que indica el protocolo RS232C, otros dispositivos pueden ser conectados a la interfaz "port serie". Así, puede conectarse el mouse (que sólo envía datos en serie), o puede conectarse una impresora que está en una habitación distante del computador. También pueden conectarse dos computadoras a través de esta interfaz.
En lo que sigue, supondremos conectado a la interfaz un módem. Si éste es interno, estará en una misma plaqueta junto con su interfaz (figura 1.64). En caso de un módem externo, el mismo se puede conectar a cualquier interfaz "port serie" disponible, a través del conector correspondiente (figura 1.71).

El funcionamiento y control de la interfaz "port serie" se basa (figura 1.69, del mismo tipo que las figuras 1.66 a 1.68) en un chip denominado "Universal Asynchronous Receiver-Transmiter" (**UART**).
Este chip presenta una línea de entrada de datos RD (para recibir en serie del módem), y otra línea salida de datos TD (para transmitir en serie hacia el módem), y contiene hasta 10 registros port direccionables
Cuando a una interfaz "port serie" se conecta un periférico de entrada, como el mouse, sólo se usa la línea RD, dado que únicamente se realizan operaciones de entrada.

La línea RD entra a un registro que convierte a paralelo el byte de datos que entraron en serie, y los pasa al *port de datos recibidos*[1] (que puede comprender un buffer de 16 bytes –para conectar módems rápidos–).

[1] Denominado *Receiver Buffer Register*

Este port puede ser leído como cualquier port (según se ilustra en la figura 1.68), de a un byte por vez, mediante la ejecución de una instrucción como **IN**, la cual ordena que una copia del contenido del port pase al registro AX, a través de las líneas de datos del bus.

En una operación de salida, mediante la ejecución de **OUT**, (como en la figura 1.66) un byte contenido en AX pasa por las líneas de datos del bus al *port de datos a transmitir*[1] (que puede constar de un buffer de 16 bytes); y de éste pasa en paralelo a otro registro (no dibujado). Este es el encargado de convertir el byte de paralelo a serie, para que salgan, bit a bit, por la línea TD.

Debe tenerse presente, que conforme a las normas RS232C, además del *grupo* de bits de datos que se transmiten en serie (típicamente 7 ú 8)[2] deben existir bits adicionales de control, antes y después de los mismos como ilustra la figura 1.70, con la presencia o no un bit de paridad[3]. Luego siguen uno o dos bits de "stop" de valor **1**, que indican fin del grupo.
La UART se encarga de agregar los bits citados en una transmisión, y de quitarlos antes que lleguen a memoria, en una recepción (entrada). Asimismo, la UART transmite los bits a la velocidad establecida por programa.
La línea TD ó la RD permanecen en el nivel alto hasta que tenga lugar una transmisión de datos. Esto se detecta en una recepción, por que en correspondencia con el bit de comienzo (siempre de valor 0), la línea pasa al nivel bajo durante un tiempo igual al de duración de cada bit. Esta forma de conformar los datos, pensada para poder distinguir un bit del siguiente (como se planteó en la figura 1.50), vale tanto para los datos recibidos, como para los transmitidos por una interfaz "port serie". O sea, que por ejemplo, cada byte de datos que envía el mouse a su "port serie" debe cumplir con este protocolo, conocido como de "comien zo-final". La UART verifica este protocolo de transmisión y también el de detección de errores por paridad.

Además de los dos ports para datos citados, la UART contiene 8 registros ports de un byte, direccionables, para comandos, protocolo y "status", que también se pueden escribir o leer. Ellos son:

Cuatro registros ports direccionables *para comandos* que se escriben (ejecutando la instrucción **OUT**) antes de que se transmitan datos entre dos módems intercomunicados.
Dentro de estos cuatro, existen dos ports designados de "bauds rates", para fijar (indirectamente, mediante un número de 2 bytes, Hi y Low) la velocidad (en bauds) con que se recibirán y transmitirán los bits.

El registro *port para control de línea* permite fijar por programa (figura 1.70):
- La cantidad de bits en serie (5, 6, 7 ú 8) que se recibirán o transmitirán por vez.
- Si al final de los bits en serie recibidos o transmitidos. habrá 1, 1 ½ o 2 bits de final ("stop")
- Si se detectará o no si hubo un bit errado en la transmisión, usando un bit extra de paridad.
- Si cada grupo de bits transmitidos o recibidos con paridad tendrá un número par o impar de unos.

El *port de habilitar interrupciones* permite activar la línea de solicitud de interrupción (IRQ) en función:
- del estado (lleno o vacío) de los ports de datos
- del cambio de valor de cualquier bit de los ports para protocolo y "status"

Dos registros ports direccionables de la UART se usan para llevar a cabo el **protocolo RS232C:**
- El *registro de control del módem* (escrito mediante OUT) genera hacia éste las señales RTS y DTR
- Un *registro de estado del módem* (leído mediante IN) guarda el valor de las líneas DSR, CTS, RI y RLSD que provienen del mismo.

Estas señales se relacionan con el protocolo que debe suceder antes de que el computador envíe datos al módem. Antes de enviar un byte al módem, debe darse la siguiente secuencia de señales (figuras 1.69 y 1.70):

1. Merced a la ejecución de **OUT**, se direcciona (03FC) el *port de control de módem*, para escribir 00000010. El "uno" de 00000010 activa la línea DTR, que avisa al módem que se le enviará un byte.

---

1 Designado *Transmitter Holding Register*

2 Por ejemplo, los bits de un carácter codificado en ASCII (código tratado en el Apéndice A.1)

3 Como en la memoria (sección 1.4 –donde a diferencia se transmiten bytes en paralelo– la paridad sirve para detectar si un bit está errado. Por ejemplo, si se envían 7 bits de datos en serie, el valor del bit de paridad (el octavo) se halla contando el número de unos que hay en los 7 bits de datos. Si este número es impar, el octavo bit vale **1**; y si es par, vale **0** (suponiendo que la paridad total de unos deba ser siempre par). Cuando esos 8 bits se reciben, debe cumplirse que tengan una paridad par de unos. Caso contrario se asume que algún bit está errado, pudiéndose solicitar retransmisión de esos 8 bits

2. El módem responde que está conectado, activando la línea DSR[1], con lo cual se escribe un "uno" en el *port de estado del módem*, formándose la combinación 00100000. Merced a la ejecución de IN se direcciona (03FE) y lee este port, y una copia de esta combinación llegará al registro AX (no dibujado).
3. Cuando 00100000 llega a AX, el programa detecta que el módem está activo, y ahora se escribe 00000011 al port de control del módem, mediante la ejecución de **OUT**. El último "uno" de 00000011 activará la línea RTS que llega al módem, para preguntarle si está listo ("ready") para recibir.
4. Si el módem está listo, activa la línea CTS, con lo cual se escribe otro "uno" en el *port de estado del módem*, formándose la combinación 00110000 pues DSR debe seguir activada. Merced a la ejecución de **IN** se direcciona (03FC) y lee este port, y una copia de esta combinación llegará al registro AX
5. Luego que AX recibió 00110000, el programa detecta que se puede enviar un byte, pasándose a ejecutar una instrucción **OUT**, que direcciona (03F8[2]), y escribe (**5a**) el *registro de transmisión de datos* con el valor supuesto 11100110. Entonces la UART calcula que el valor del noveno bit (de paridad) que se le debe agregar es **1**, si se ha establecido por programa que las combinaciones a enviar deben tener un número par de unos; y la UART también inserta los bits de comienzo y final, conforme a la figura 1.70. Después de agregar estos bits, la UART realiza una conversión de paralelo a serie, y transmite uno a uno los bits hacia el módem[3], por la salida TD (**5b**).

Mientras CTS siga activa pueden enviarse más bytes, que pasan rápido al buffer del módem (no dibujado). Cuando éste se llena, el módem desactiva CTS. Para volver a enviar otro byte hay que repetir los pasos 4 y 5 De este modo, se adapta la rápida velocidad de envío del computador a la de operación del módem.

Los pasos 1 a 5 se refieren al protocolo RS232C entre el módem y el procesador. Una vez que un byte entró al módem se guarda en un buffer de éste, para que sus circuitos de modulación realicen (**6**) una conversión de señales digitales a analógicas. Por simplicidad se ha supuesto que se modula en frecuencia. A medida que se obtiene cada valor de la señal modulada, el mismo se envía por la línea telefónica (7)

*Para entrar datos desde el módem u otro periférico **no** se requiere este protocolo*: mientras DTR esté activa se le puede enviar datos a la interfaz "port serie", pues cada byte que entra puede ser tomado rápidamente (mediante IN) del port de datos recibidos.

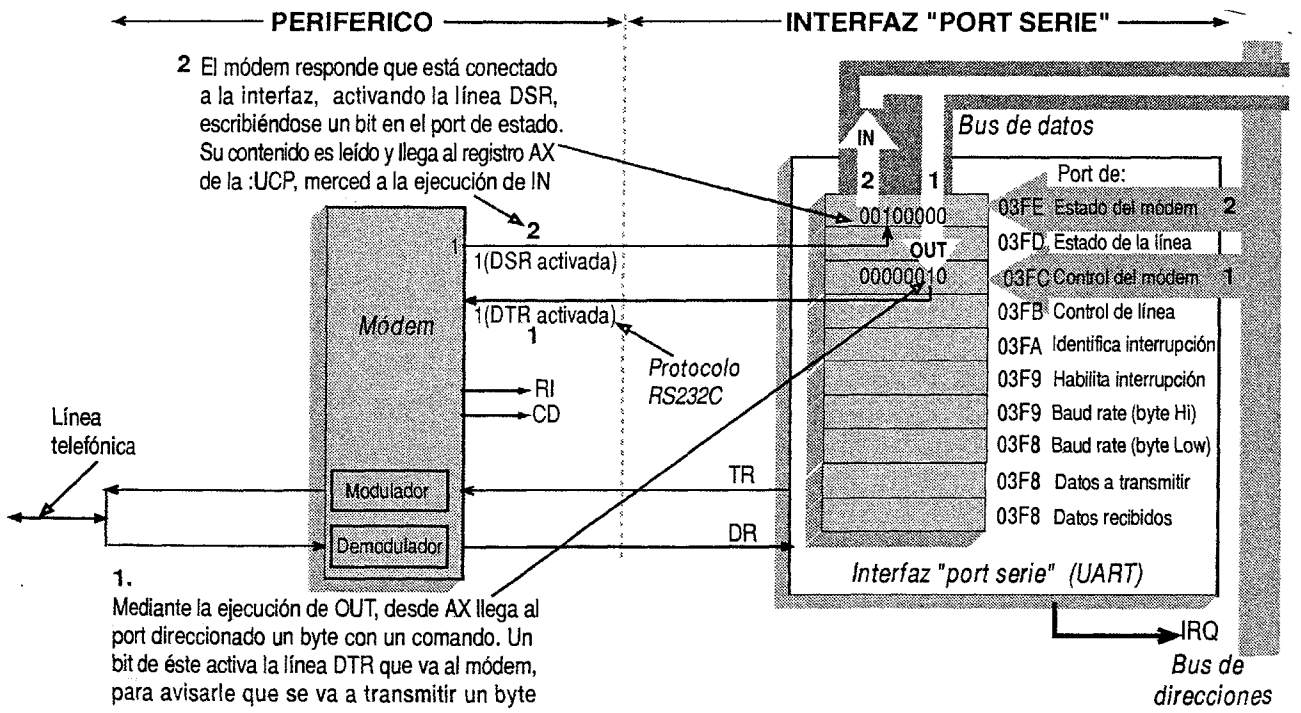
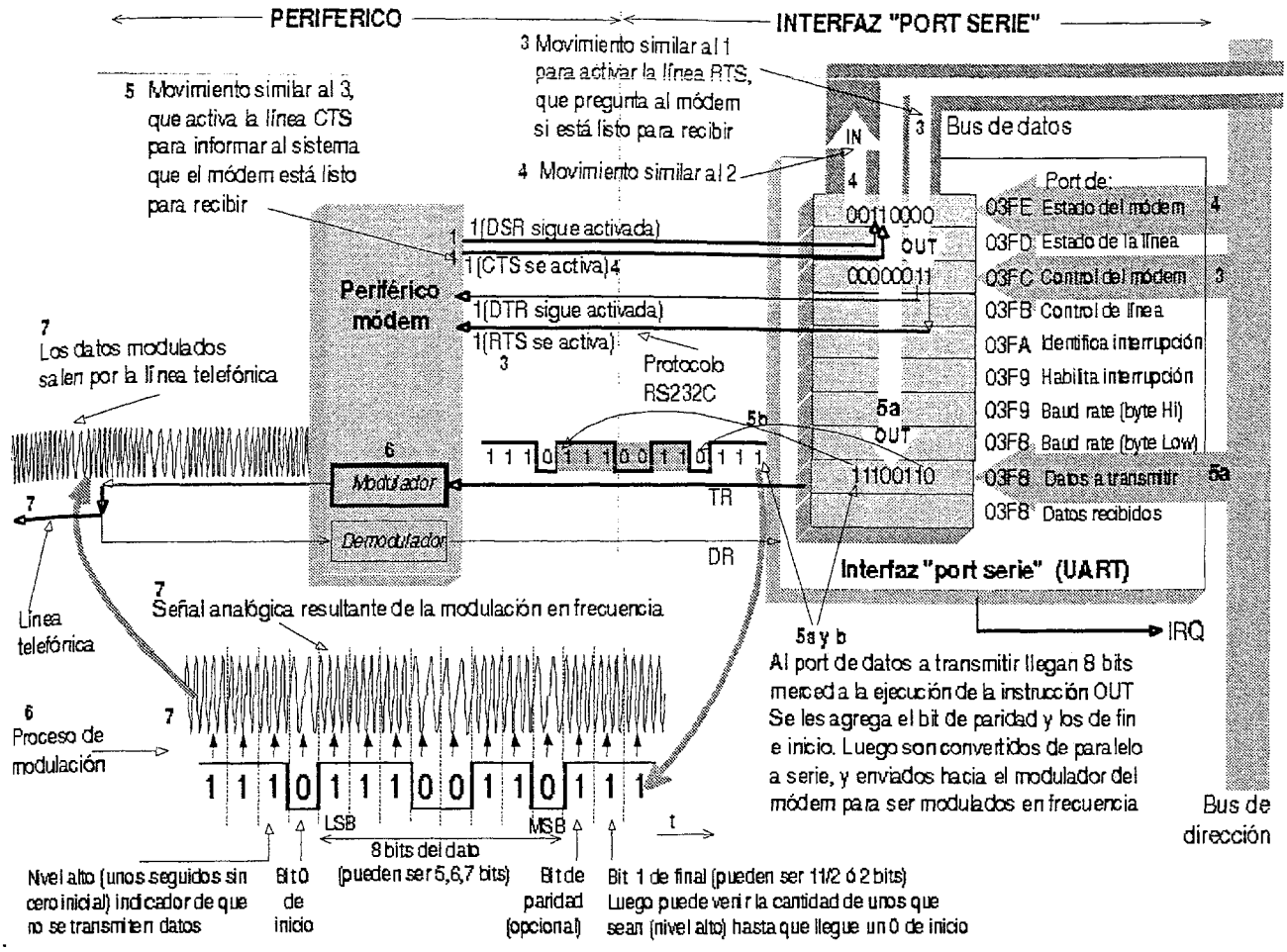


Figura 1.69

[1] DTR y DSR deben permanecer activas durante toda la comunicación.

[2] Aunque hay ports de igual dirección, mediante el estado de otro bit, el hardware determina cuál es el port direccionado.

[3] De los bits de datos primero sale el menos significativo (LSB –Less Significant Bit), que en un número es el dígito extremo derecho, y en último lugar sale el mas significativo (MSB – Most Significant Bit), o sea el dígito extremo izquierdo



Normas a cumplir en la transmisión serie según el protocolo RS232C

Figura 1.70

Mediante la línea RI(ng) el módem avisa que el teléfono suena, para que desde el computador se maneje el protocolo que hace que el módem atienda el llamado.
Si se activa la línea RLSD o CD (detectora de portadora), significa que el módem detectó la onda portadora de señal de otro módem que se quiere comunicar, el cual puede ser atendido o no.

El *port de "status"* de la UART permite conocer, al ser leídos los valores de sus bits individuales, si:
- en el registro port de datos recibidos hay un byte para entrar
- un nuevo byte entró al port de datos recibidos antes que el anterior fuera entrado
- el byte que entró al port de datos recibidos tiene un bit errado (error de paridad)
- hubo error en la forma en que llegaron desde el exterior los bits en serie
- en el port de datos a transmitir el byte aún no fue transferido
- el registro de desplazamiento que pasa de paralelo a serie contiene datos a transferir
- una línea para protocolo que llega desde el periférico (módem) está siempre fija en un valor

Por último, luego que la plaqueta interfaz serie solicita interrupción (mediante su línea **IRQ**), otro registro port identificador de interrupciones –al ser leído por la subrutina que atiende estas interrup ciones– permite determinar la causa de la misma, pues como se vio, puede haber varias posibles.

Hasta acá se ha supuesto una interfaz "port serie" ubicada en una plaqueta –que contiene sólo a ella o varias interfaces (plaqueta multifunción)– con un conector (figura 1.60) para el conexionado de un mouse (como aparece en dicha figura), un módem externo, o cualquier dispositivo que cumpla con las normas RS232C.
Este conector no existe en el caso de una plaqueta que contenga esta interfaz junto con un módem interno, en cuyo caso debe existir un conector para el conexionado de la línea telefónica (figura 1.64).

¿Cuáles son las características de otras Interfaces, como la de unidad de disquete, la de unidad de disco rígido y la de vídeo ?

Estas interfaces están diseñadas para ser conectadas a ellas *exclusivamente los periféricos indicados*. Por la complejidad de los periféricos que controlan, presentan un gran números de ports direccionables. Unos reciben comandos para dichos periféricos, y otros al ser leídos indican el estado de los mismos, de la forma vista para las interfaces "paralelo" y "serie", cuando se ejecutan las correspondientes subrutinas del BIOS.
Para disquetes, disco rígido y CD ROM, los comandos son del tipo: posicionar el cabezal, leer o escribir un sector, y otros de detalle, que serán tomados por la electrónica que controla las cabezas y los movimientos mecánicos.[1] A su vez ésta informa mediante códigos en los ports de "status" si hubo algún error: como ser falló el posicionamiento del cabezal, el sector no se encontró, error de lectura, etc
La plaqueta interfaz "controladora" de vídeo es una de las más complejas. Como se estableció, contiene una parte de la memoria principal (**VRAM**) donde la UCP escribe datos codificados en binario que luego se verán como texto o gráfico en la pantalla. Esto implica *que no presenta ports intermediarios para datos*. Contiene varios ports direccionables, mediante los cuales se puede tener acceso a decenas de registros identificables por un número. En ellos, subrutinas del BIOS para vídeo escriben una serie de comandos que permiten establecer la resolución, gama de grises o colores, manejo del cursor y otros atributos que tendrá la imagen que se verá en la pantalla. También es factible en muchos casos leer su estado.

Otra característica de estas interfaces, es que para las PC existen versiones de plaquetas para ser conectadas a distintos tipos de buses. Así, se tiene plaquetas multifunción y de vídeo para ser conectadas al bus tradicional ISA, con 16 líneas para datos; otras para ser conectadas al bus local VESA, de 32 líneas para datos (procesadores 386 ó 486) y otras para la conexión al bus local PCI. También estas alternativas se dan en una "motherboard" con un Pentium, siendo que su bus local es de 64 líneas de datos. Con estos buses de 32 y 64 líneas no sólo es posible enviar el doble o cuádruple de datos en paralelo que con el bus ISA, sino que también la velocidad de transmisión usando bus local puede ser varias veces mayor que la del bus ISA.

¿En qué se diferencia una E/S por acceso indirecto a memoria (AIM), de una E/S por acceso directo a memoria (ADM) ?

Como se trató recién, las operaciones de E/S de datos se hacen a través de interfaces con registros ports para datos[2]

El pasaje de datos del registro *port de datos* de una interfaz hasta memoria principal, o en sentido contrario constituye la **fase de transferencia** de una operación de entrada o salida (E/S)

La figura 1.72 ilustra la fase de transferencia de dos operaciones de *entrada*, realizadas de distinta forma, para el caso del teclado y de la unidad de disquete. Se plantea en cada caso una manera diferente de transferir datos, desde el port de datos correspondiente hacia memoria principal.
Si bien a los fines comparativos ambas fases se han dibujado juntas, en el tiempo primero debe suceder una y luego la otra.

Para la entrada de datos del teclado (al igual que para el mouse y el módem) la fase de transferencia será:

Port de datos ⇒ registro AX ⇒ memoria principal.

Esta "triangulación" *se realiza mediante la ejecución de instrucciones* del BIOS.
Desde el port los datos pasan[1] al registro AX del procesador –a través del bus de E/S–, mediante la ejecución de una instrucción tipo **IN**; y desde AX llegan a la zona buffer de memoria reservada para ellos –a tra

[1] Debe mencionarse que en los discos rígidos o CD ROM actuales con interfaz IDE Intelligent Device Electronic), esta electrónica no se encuentra en la plaqueta multifunción, sino que se halla junto la caja que contiene uno u otro tipo de unidad de disco. Mediante sendos cables plásticos planos (en general de color gris) se enchufan al bus a través de la plaqueta multifunción o directamente a través de un conector de la motherboard.. Esta plaqueta contiene las interfaces para los disquetes "A" y "B", y los "port serie" y "port paralelo" (usadas típicamente por el mouse y la impresora, respectivamente). A ella también se conectan directamente al bus las interfaces IDE del disco rígido y del CD ROM.

[2] Salvo el caso de la salida hacia el monitor, que se hace desde la porción de memoria principal para vídeo, sin pasar por un port.

vés del bus que une memoria con el procesador– mediante una instrucción de movimiento, como L_4 vista Esta forma la llamaremos E/S con **Acceso Indirecto a Memoria** (**AIM**), indicada en figuras 1.60, 1.62, y 1.64

Cuando en una operación de entrada se leen los 512 bytes de un sector de un disquete[2] (figura 1.63), la fase de transferencia se realiza por **Acceso Directo a Memoria** (**ADM** ó DMA en inglés):

Port de datos ⇒ memoria principal

Para realizar ADM se requiere una circuitería complementaria a una interfaz, que pueda leer o escribir datos en memoria principal *como lo hace el microprocesador*, dado que *éste no interviene* en el ADM, puesto que para llevar a cabo el mismo *no se ejecutan instrucciones*, como ocurre en el AIM citado.
En una PC esta electrónica se denomina "**Controlador de ADM**" (figura 1.72).

Básicamente, la lectura de un sector de un disquete, en general supone las siguientes acciones principales, ordenadas por instrucciones de subrutinas que están en la ROM BIOS. En la figura 1.72 aparece el movimiento por ADM de un dato desde el port de datos hacia memoria. La escritura de comandos o direcciones en ports de la interfaz de disquete o del Controlador de ADM, es similar a las figuras 1.67 a 1.70

1. Mediante la ejecución de una instrucción tipo **OUT**, se ordena, mediante un comando a un port de control (comandos) de la interfaz de la disquetera, que su motor se ponga en marcha.
   Luego la subrutina espera un tiempo fijo, hasta que el motor alcance su velocidad de giro estipulada.
2. A continuación, dicha subrutina direcciona y escribe, también mediante instrucciones **OUT**, registros de control de la interfaz, indicando número de unidad de disquete seleccionada (A ó B), y número de pista al que se quiere acceder.
3. Mientras el cabezal de lectura/escritura se posiciona sobre la pista indicada, una subrutina del BIOS direcciona y escribe –con instrucciones **OUT**– los registros port del Controlador de ADM. En el *registro de "cuenta"* se indica la cantidad de bytes a escribir en memoria ($FF_H = 255_D$); y en el *registro de dirección*, la dirección (supuesta 4033_H) de la zona de memoria a partir de la cual se deben escribir sucesivamente los 256 bytes. Estos datos llegan por las líneas de datos del bus conectado al Controlador.
4. Ports de control de la interfaz de la disquetera (se muestra uno solo) son direccionados, y se escriben en ellos comandos –mediante instrucciones **OUT**– que indican entre otros: la cara, número de sector.
5. Cuando la cabeza está sobre el sector a leer, se activa la línea de requerimiento de DMA (**DREQ**) que va de la interfaz al Controlador de ADM, para que éste *comience la fase de transferencia*, a la par que los bits del sector leídos en serie por la cabeza lectora pasan a un buffer en la electrónica de la disquetera.
6. El Controlador de ADM toma el control de bus (ISA en una PC), ordena escritura por la línea de L/E que llega a memoria, y su registro de dirección envía por las líneas de dirección (igual que la UCP) su contenido (4033), que es la dirección donde el primer byte será escrito en memoria.
   Activando la línea DACK que llega a la interfaz, le dice a ésta que coloque en las líneas de datos del bus el byte (o dos bytes) que llegó en paralelo a su port de datos (a través del cable plano –figura 1.63– que une el buffer de la electrónica de la disquetera con la plaqueta interfaz). De esta forma, dicho byte (26), es escrito en memoria en la primer dirección (4033) citada.
7. Luego la circuitería incrementa uno el registro de dirección y resta uno al registro de cuenta.
8. Mientras el registro de cuenta no llegue a cero (indicación que se escribieron 256 bytes) se vuelve al paso 6., y hasta que no se escribió el bloque de 256 bytes, la UCP no puede leer o escribir la memoria.
9. Cuando el registro de cuenta llega a cero, el Controlador activa la línea End of process (EOF) que llega a la interfaz, para que ésta a su vez active su línea IRQ de solicitud de interrupción, para que la UCP ejecute una subrutina que verifique que la transferencia ordenada por ADM fue hecha correctamente.
10 Se ordena en un port de control de la interfaz, que se detenga el motor (si no se debe acceder a otro sector).

---

[1] En realidad, pasa al lugar de destino una copia de la información binaria contenida en el lugar de origen. Típicamente de a uno o dos bytes por vez, pudiendo ser de 4 ú 8 bytes usando el bus local, como en el caso del disco rígido.

[2] El ADM en general se utiliza en relación con periféricos que deben transferir velozmente bloques de muchos bytes de datos, de modo de poder pasarlos como una "ráfaga" ("*burst*") continua hacia o desde memoria en breve tiempo. Si en una entrada se deben transferir 512 bytes, y si se escriben dos bytes en cada acceso, se requiere realizar 256 escrituras sucesivas en posiciones consecutivas de memoria (y 256 lecturas en una operación de salida). En las grandes computadoras por la cantidad de terminales conectadas, en general resulta un caudal muy grande de datos, por lo que el ADM es el método de transferencia usado para *todos* los periféricos. En los modelos tipo IBM/370 y en grandes computadores, todas las interfaces pueden hacer ADM, constituyendo dispositivos denominados "**canales**".

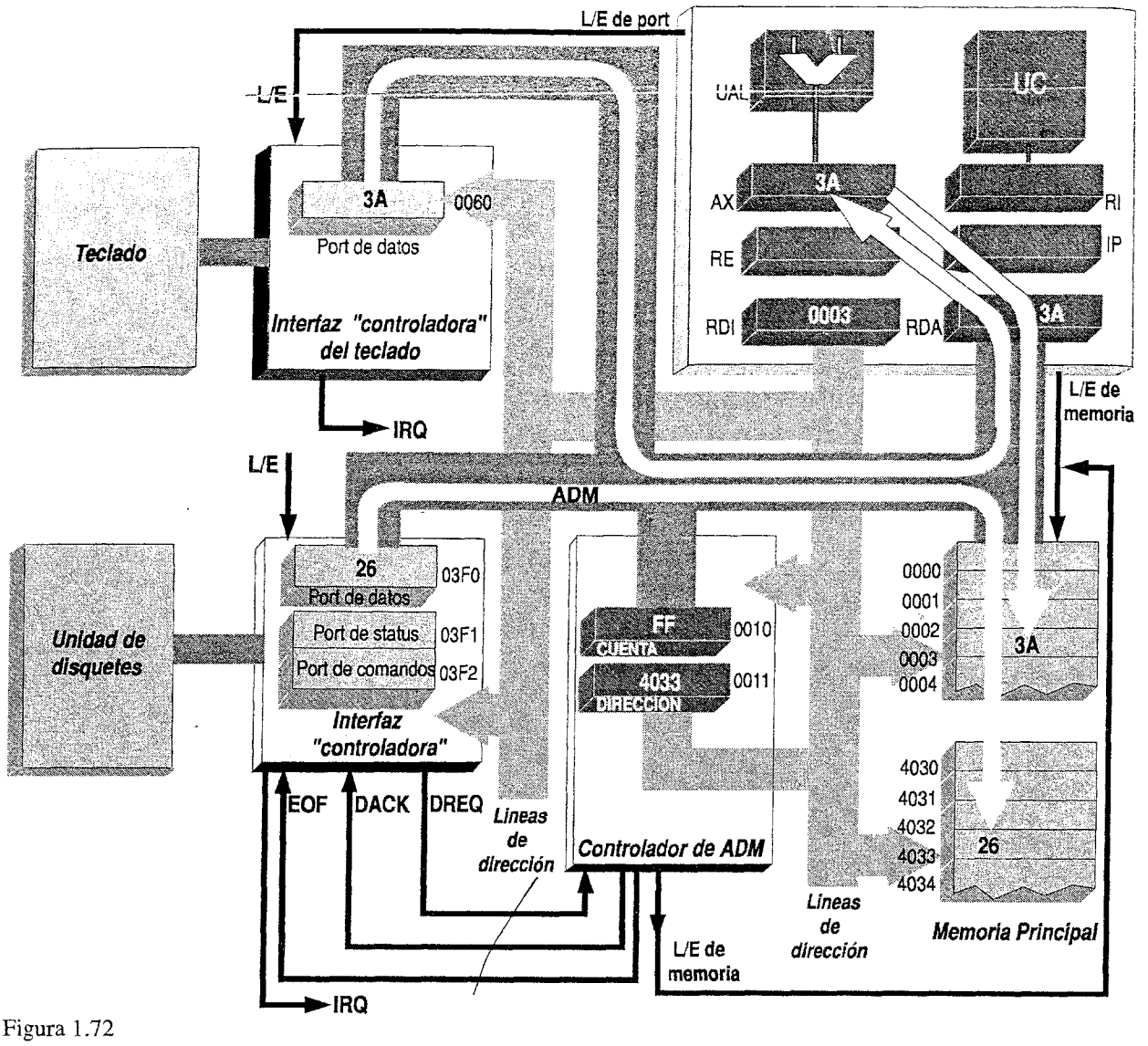


Figura 1.72

Se han tratado las dos formas de realizar la fase de transferencia en una operación de entrada. De manera análoga, en una operación *de salida* existen dos formas de transferir datos de memoria a un port:

Por AIM:

memoria principal ⇒ registro AX ⇒ port de datos

Esto es, en una operación de salida, los datos que están en una zona buffer de memoria pasan –a través del bus de memoria– al registro AX del microprocesador (mediante una instrucción tipo I_1), y desde AX llegan al port direccionado (mediante una instrucción tipo **OUT**). Luego los datos siguen hacia el periférico.
La figura 1.61 ilustra estos movimientos.
Por ADM:

memoria principal ⇒ port de datos

De lo anterior resulta que el AIM *es un proceso controlado por software*, dado que es ordenado mediante instrucciones, en cuya ejecución debe estar ocupada la UCP; mientras que el ADM *es controlado por circuitería* (hardware), sin ejecución de instrucciones.

En una PC predominan las E/S con transferencias por AIM.
De no existir limitaciones en la velocidad de transferencia del bus utilizado –como ocurre en las PC– *una E/S con transferencia por ADM es mucho más rápida que otra por AIM*

1.11 LAS INTERRUPCIONES PC HARDWARE: "TIMBRES" PARA LLAMAR A SUBRUTINA

¿Qué son las interrupciones ?

Una **interrupción** supone la suspensión temporaria de la ejecución de un programa, para pasar a ejecutar una *subrutina de servicio de interrupción, la cual en general no forma parte del programa* (por pertenecer típicamente al sistema operativo, o al BIOS –Basic Input Output System–)[1].
Luego de ejecutarse dicha subrutina, debe reanudarse la ejecución de dicho programa.

Así, en la figura 1.73, luego que se termina de ejecutar la instrucción $\mathbf{I_2}$ de un programa antes visto, se suspende transitoriamente la ejecución de $\mathbf{I_3}$, y de las instrucciones siguientes de dicho programa (1).
Se supone que ello es debido a que una interfaz, por ejemplo la del teclado (figura 1.62), activó su línea de solicitud de interrupción IRQ (2), para que se pase a ejecutar la subrutina de la ROM BIOS que atiende al teclado (3).
Luego que se termina de ejecutar esta subrutina (4), se vuelve a la secuencia interrumpida (5), por lo cual se ejecutará la instrucción $\mathbf{I_3}$ que había quedado en suspenso, y las siguientes (6)

Por lo tanto, las interrupciones *son la forma en que se llama*, para su ejecución, *a subrutinas de un sistema operativo y de la ROM BIOS.*

Se debe tener bien en claro que un sistema operativo no se ejecuta por que "él quiere", ni en simultaneidad con el programa en ejecución, sino que la UCP pasa a ejecutar una subrutina o módulo de un sistema operativo sólo cuando una interrupción así lo determina.

Existen **interrupciones por software** (mediante una instrucción) e **interrupciones por hardware.**

Las interrupciones por hardware **externas** (a la UCP) son las del tipo descripto hasta ahora: tienen lugar cuando una interfaz activa su señal de solicitud de interrupción (denominada **IRQ** –Interrupt Request– en las PC), que por una línea de control del bus llega a un chip
Cada línea IRQ que sale de una interfaz tiene un subíndice **n** (del 0 al 15 en una PC) que la identifica. Todas las IRQ entran a un chip *"árbitro de interrupciones"*, que decide cuál interfaz interrumpirá primero, para el caso que se activen varias IRQ simultáneamente. Este chip activa una línea que llega al procesador (designada INTR en los 80x86, como muestran las figuras 1.62 y 1.63).
Las interrupciones por hardware **internas** (designadas a veces **"excepciones"**) ocurren en el interior de la UCP, si hay algún problema mientras se ejecuta una instrucción.

Obsérvese que tal como se la ha definido, la palabra "interrupción" designa la suspensión *momentánea* de un programa o subprograma, realizada por medio de una instrucción de máquina específica (software) o por hardware (originada en el interior de la UCP o mediante la línea IRQ citada),debiéndose luego retomar la ejecución del programa. Todo ello **sin intervención humana.**
No se trata, pues, del mero hecho de que un usuario decida cambiar o suspender, mediante el teclado o el mouse, la ejecución de un programa por otro a su elección. En general, los efectos de una interrupción no son visibles para un usuario, dado que llaman a subrutinas del sistema operativo o del BIOS.

[1] También es factible que programas como procesadores de texto, hojas de cálculo y otros tengan sus propias subrutinas para interrupción, siendo además que un usuario puede crear sus propias rutinas de servicio de interrupción.

¿Cómo opera una interrupción por hardware externa ?

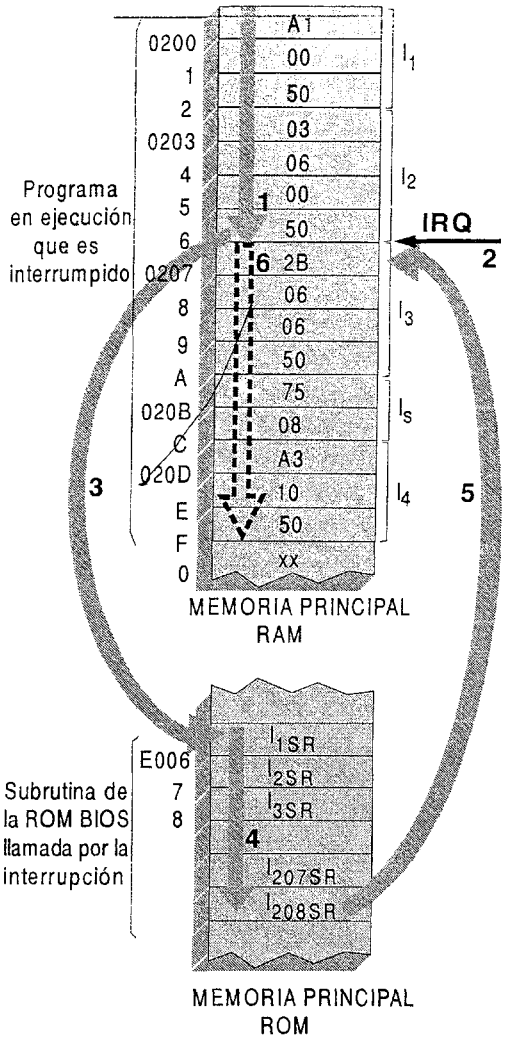


Figura 1.73

Anteriormente se planteó, que otra función muy importante de una plaqueta interfaz es *generar la señal IRQ que se envía por un cable* hacia un subsistema ligado a la UCP, para solicitarle a ésta la *interrupción del programa en curso de ejecución.*
Se trata claramente de una interrupción *que tiene su origen en el hardware*: la activación de un cable de control del bus.
Esta activación tiene lugar en principio cuando en el periférico conectado a una plaqueta interfaz se concretó algo: en el teclado se apretó una tecla, una impresora está lista para recibir más datos, una unidad de disquetes terminó de leer o escribir, etc.

La activación de dicha señal IRQ por una interfaz (*que equipararemos a un toque de timbre*), causará que en algún momento posterior *se interrumpa momentáneamente la ejecución de un programa, y que la UC pase a ejecutar la subrutina de la ROM BIOS o del sistema operativo*[1]*, preparada para atender a dicha interfaz*[2].
Al finalizar la ejecución de esta subrutina, podrá seguir la ejecución del programa interrumpido. A continuación se darán dos ejemplos.

Cada vez que en una PC se pulsa o libera una tecla, el código de ésta llega al port de datos de la interfaz del teclado, lo cual hace activar su señal IRQ (figura 1.62). Esta IRQ sirve de *indicación de que en una operación de entrada puede realizarse la fase de transferencia por acceso indirecto a memoria* (AIM), para que dicho código llegue a la zona buffer de memoria reservada para los códigos de teclas.
A fin de realizar el acceso indirecto a memoria, luego de la activación del IRQ, la UC interrumpe el programa en curso de ejecución, y pasa a ejecutar una subrutina del BIOS. En ésta existirá una instrucción IN (que pasa un byte del port al registro AX), seguida de otra del tipo I_4 (que pasa dicho byte de AX a memoria principal), como ya se trató
Luego se reanuda la ejecución del programa.

Cuando finaliza un acceso directo a memoria (ADM) –como el que ocurre en la fase de transferencia de las operaciones de entrada y salida de un disquete, entre su interfaz y memoria (figuras 1.63 y 1.72)– la interfaz involucrada activa su IRQ, para que se interrumpa el programa en ejecución, y se pase a ejecutar la subrutina que verifica que el ADM fue bien hecho. Caso contrario se ordenará intentarlo varias veces. Si el problema subsiste, dicha subrutina hará aparecer un cartel en pantalla indicando dicho error de hardware.
Para la impresora, en una PC suele usarse la señal IRQ de su interfaz para indicar falta de papel, lo cual origina que la subrutina que atiende este evento indique el mismo con un aviso en pantalla.

Otra fuente de interrupción externa proviene de la interfaz con los contadores usados para generar las horas y minutos que aparece en pantalla. A tal fin, cada segundo dicha plaqueta genera 18 interrupciones, siendo que con cada una de éstas, la subrutina que sirve a esta interrupción actualiza la hora indicada cada minuto.

Obsérvese que estos eventos (tecla pulsada, fin de ADM, reloj, etc.), por un lado, *no pueden ser previstos en un programa cuándo sucederán;* y por otro, cuando en memoria hay varios programas que alternan su ejecución (*"multitasking"*) también puede ocurrir una interrupción en un programa a causa de un periférico que en ese momento está trabajando para otro programa.

[1] Por sus funciones puede considerarse a la ROM BIOS como parte de un sistema operativo, con flexibilidad para manejar una gama amplia de periféricos, provenientes de distintos fabricantes.

[2] La dirección donde empieza esta subrutina se halla en una zona de memoria principal, que en las PC se llama de "vectores interrupción"

De lo anterior también resulta –como se anticipó– que *hay un conjunto de líneas de solicitud de interrupción.* Así, por lo menos de cada interfaz sale una línea designada IRQ_n siendo n distinta para cada una. Hay IRQ compartidas, como la de la interfaz para disquetes y discos.

Todas estas líneas se dirigen a un chip "árbitro de solicitudes de interrupción", que en caso de activarse varias IRQ simultáneamente, da curso sólo a la que tiene subíndice n menor. Las restantes deben esperar. Una salida de este chip activa la línea de control INTR que entra a la UC, para indicarle que hay una solicitud de interrupción activa.

Cada vez que termina de ejecutar una instrucción, la UC sensa si la línea de control INTR está activa (indicadora de solicitud de interrupción). Si el programa en ejecución lo permite[1], el mismo es interrumpido. De no ser así, *dicha solicitud espera hasta ser atendida.*

En las dos primeras generaciones de computadoras este mecanismo de interrupciones que indica un evento externo que debe ser atendido no estaba incorporado. Un método que se usaba consistía en ejecutar periódicamente subrutinas que interrogaban (*"polling"*) a los periféricos, para determinar si tenían datos a enviar. Ello implicaba tiempo de la UCP gastado en su ejecución, aunque no hubiera nada para enviar.

¿Cómo operan las Interrupciones por software ?

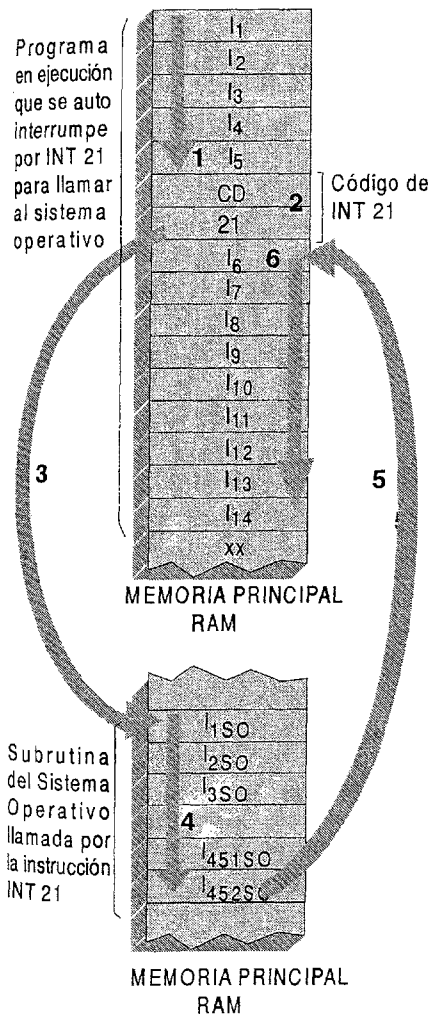


Figura 1.74

Una interrupción por software se realiza *ejecutando el código de una instrucción que ordena* **llamar a subrutinas del sistema operativo o de la ROM BIOS** cuando necesita el servicio de una de ellas.

Este tipo de instrucción en el lenguaje assembler de una PC se simboliza **INT xx**[2]), donde el número **xx**, identifica la subrutina llamada y también permite localizarla en memoria principal).

A diferencia de las interrupciones por hardware, una interrupción por software queda establecido en qué momento de la ejecución de un programa sucederá, pues *se trata de una instrucción que está en determinado lugar de un programa, que llama a dichas subrutinas.*

Por ejemplo, cuando un programa necesita hacer una operación corriente relacionada con los recursos que maneja el sistema operativo: típicamente abrir/cerrar un archivo, acceder a un disco, imprimir, visualizar información en pantalla, etc.

En todos los casos se trata, pues, de un llamado a subrutinas que los programas de usuario requieren en su auxilio, cuando se necesita realizar operaciones típicas, como las citadas.

La figura 1.74 –en esencia similar a la 1.73– da cuenta del mecanismo de una interrupción por software, que podemos esquematizar en los siguientes pasos:

1. Un programa que se venía ejecutando, luego de su instrucción $\mathbf{I_5}$ necesita llamar a una subrutina del sistema operativo (SO), por ejemplo para leer un archivo en disco.
2. A tal efecto, luego de $\mathbf{I_5}$ existe en el programa la instrucción de código de máquina CD21, simbolizada **INT 21** en assembler, que realiza el requerimiento del paso 1. Puesto que no puede seguir la ejecución de las instrucciones $\mathbf{I_6}$ y siguientes del programa hasta que no se haya leído el disco y esté en memoria principal dicho archivo, virtualmente el programa se ha *autointerrumpido,* siendo además que luego de INT 21, las instrucciones que se ejecutarán no serán del programa, sino del sistema operativo.

---

1 Mediante una instrucción especial se puede hacer esperar (*"enmascarar"*) solicitudes de interrupción.

2 En otras computadoras como las PDP y la IBM/370 esta instrucción se llama TRAP y SVC nn (Supervisor Call), respectivamente.

3. La ejecución de INT 21 permite hallar la subrutina del SO
4. Se ejecuta la subrutina del SO que prepara la lectura del disco.
5. Luego de ejecutarse la subrutina del SO, y una vez que se haya leído el disco y verificado que la lectura es correcta, el SO ordenará reanudar la ejecución del programa autointerrumpido en espera.
6. La ejecución del programa se reanuda

Es importante tener siempre presente que tanto las interrupciones por hardware o software son mecanismos para llamar a subrutina del SO o de la ROM BIOS.

Esto es, **los programas del sistema operativo no se ponen en ejecución por sí mismos, sino que son llamados por las interrupciones, que son los "timbres" que llaman al sistema operativo.** De ahí su importancia.

Dado que tanto los programas o subrutinas del sistema operativo o del BIOS que ejecuta la UCP se originan en interrupciones por software o hardware, éstas en esencia marcan, *articulan, la secuencia automática de procesos que va realizando un sistema de computación,* de modo de *utilizar la UCP en relación con las necesidades de atención de dichos procesos en el tiempo.*

¿Cómo se retorna al programa interrumpido ?

En cualquier caso que exista una interrupción, antes de que se pase a ejecutar la subrutina del sistema operativo o del BIOS que da servicio a dicha interrupción, la UC *guarda automáticamente en memoria la dirección de retorno,* donde está la instrucción a partir de la cual continúa el programa interrumpido De esta forma, luego que se ejecutó dicha subrutina, se podrá retornar a ejecutar el programa.

¿Qué es la zona de memoria principal denominada "pila" ?

La dirección de retorno mencionada en la respuesta anterior se guarda en una zona elegida de memoria principal denominada **"pila"** (*"stack"*). Esta se utiliza para retornar al programa que llamó a una subrutina o que fue interrumpido, una vez que se ejecutó la subrutina llamada, *en cualquier llamado a subrutina (del sistema operativo, del BIOS o de una subrutina propia del programa),* sea por interrupciones o por instrucciones de llamado a subrutina.

La pila es necesaria, por que es común que la subrutina llamada por una interrupción *a su vez sea interrumpida* para llamar a otra subrutina, y esta segunda también puede sufrir interrupciones, y así sucesivamente.

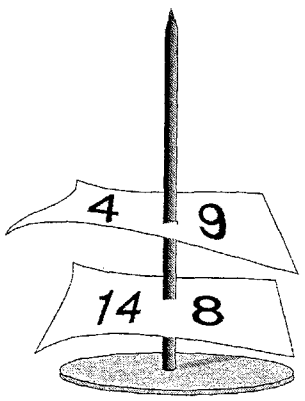


Figura 1.75

Para entender esto utilizaremos un libro de cocina. Suponiendo que se elabora una receta de su página 14, que en el renglón 8 remite a la elaboración de la página 4, y que en el renglón 9 de ésta a su vez se requiere hacer otro sub-procedimiento indicado en la página 54. Se podría cuidar el orden de elaboración como sigue.
Una vez que se llegó al renglón 8 de la página 14 se anotaría en un papel estos números, el cual se insertaría en un pincha-papeles. Luego se elaboraría el procedi-miento de la página 4 hasta llegar al renglón 9 en el cual se interrumpe. Estos números se anotan en otro papel que se apila en el pincha-papeles sobre el papel anterior, y se pasa a realizar el subprocedimiento de la página 54. Terminado el mismo, para retornar a terminar el anterior, se leen en el papel que está más arriba del pincha-papeles los números 4 y 9, que permiten reanudar la receta de esa página. Realizada ésta, se leen en la hoja apilada debajo de la anterior en el pincha-papeles, los números 14 y 8, que permiten encontrar la receta principal, para terminar la receta..
Obsérvese que el último de los procedimientos interrumpidos es el primero en completarse luego, siendo que la pila de papeles del pincha-papeles permite cuidar el orden necesario. Se trata de una pila "ultimo en entrar, primero en salir" (*"last-in-first-out"–LIFO)*

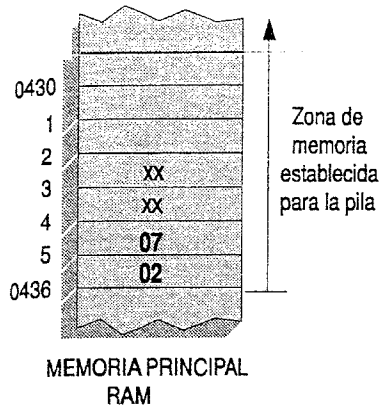


Figura 1.76

De forma análoga, a fin de guardar *ordenadamente* la dirección de retorno de un programa interrumpido, y de las sucesivas subrutinas que fueron interrumpidas posteriormente, con cada interrupción se va escribiendo en posiciones sucesivas decrecien-tes de memoria cada dirección de retorno (más los datos necesarios para poder reanudar cada subrutina interrumpida sin que nada se pierda)
Esto es, con cada interrupción o llamado a subrutina, se apila en la cima de la pila la información que permite retornar al programa o subrutina interrumpida[1].
Cuando no suceden más interrupciones, y una vez que se termina de ejecutar la última subrutina llamada, se retorna a la subrutina cuya dirección de retorno está en la cima de la pila, o sea a la que se guardó más recientemente. Por consiguiente, la última subrutina que fue interrumpida es la primera a la cual se retorna. Luego se vuelve a la anteúltima interrumpida, y así sucesivamente hasta retornar al programa interrumpido en primer lugar. Para lograr este orden, en la pila se empieza a desapilar cada bloque de información almacenada, en orden inverso al que fue apilado.
En la figura 1.76 se supone una pila definida a partir de la dirección de memoria 0436, para el programa de la figura 1.73. Este programa luego de la interrupción debe proseguir en la instrucción localizable mediante la dirección de retorno 0207H.
No bien la UCP acepta la solicitud de interrupción, guarda automáticamente en la pila (en este caso a partir de la dirección 0436H) dicha dirección 0207H.
Sobre este valor se supone que guarda otro valor, indicado XXXX, que debe ser del Registro de Estado.

Debe mencionarse que en la UCP existe un registro "**puntero de pila**" ("*stack pointer*"–**SP**), que contiene la dirección de la pila que en el presente corresponde a su cima, siendo el mismo actualizado también *en forma automática* por la UC cada vez que se lee o escribe la pila.
En el ejemplo de la figura 1.76, se ha supuesto que al sobrevenir la interrupción se podía apilar a partir del comienzo de la pila (dirección 0436H), o sea que en la pila no había nada apilado anteriormente.
Entones, el SP en una PC estaría con el valor 0437H antes de la interrupción, para alcanzar el valor 0433H luego que ella tiene lugar, indicando así el SP la nueva dirección de la cima de la pila.
De sobrevenir otra interrupción o llamado a subrutina, situación no planteada en la figura 1.76, los valores a resguardar se apilarían en la dirección 0432H de la pila, sobre XX, variando en correspondencia el SP.
Cuando se termina de ejecutar la subrutina que atiende la interrupción por hardware, solicitada por la interfaz, se retorna al programa interrumpido leyendo 0207H de la pila, y luego SP vuelve automáticamente al valor 0437H.

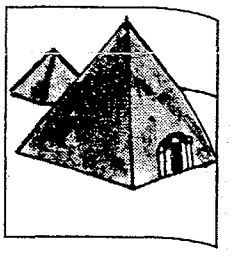
¿ Qué es la zona de memoria de "vectores interrupción"

En la descripción anterior no aclaramos cómo se localiza la subrutina llamada por una interrupción por hardware o por software. Para ambas existe en memoria una zona llamada de "*vectores interrupción*" donde para *cada* número de IRQ y de INT existen dos celdas consecutivas de memoria que indican la dirección de la subrutina que atiende a *esa* interrupción. Por tal motivo a cada par de estas celdas se las llama "vector interrupción", en el sentido que contienen un número que "apunta" hacia donde está dicha subrutina, y a las interrupciones que encuentran su subrutina de esta manera, se las llama "interrupciones vectorizadas".
A su vez la dirección donde están las dos celdas (vector) que corresponde a cada IRQ o INT se localiza en función del número de IRQ o de INT.

En la Unidad 3 de esta obra, páginas 44 a 53 de la 2da edición, se tratan en detalle las interrupciones por software y hardware.

[1] La pila y el stack pointer se tratan en detalle en la unidad 3 de esta obra (Assembler)

1.12 MEMORIA CACHE Y JERARQUÍA DE MEMORIAS EN UN COMPUTADOR



¿Qué es una memoria caché ?

Este tipo de memorias, junto con el pipeline y el advenimiento de las arquitecturas RISC constituyen una de las mejoras sucesivas al modelo de Von Neumann (sección 1.14).
El bajo costo y crecimiento vertiginoso en la velocidad de los procesadores en relación con los Mhz de sus pulsos reloj, no ha sido acompañado históricamente en igual medida por las memorias a las que ellos acceden. Así en el periodo en que los Mhz de las UCP pasaron de 10 a 50, las DRAMS sólo mejoraron de 100 a 70 nseg Se necesita que las memorias a las que las UCP acceden sean baratas, de gran capacidad de almacenamiento, y que puedan suministrar instrucciones y datos tan prontamente como las UCP pueda procesarlos (en el tiempo de un ciclo reloj de la UCP). También deben permitir gran "ancho de banda" (bytes transmitidos por segundo).

Las memorias "**caché**" ubicadas entre la UCP y la memoria principal (fig. 1.77.a), de pequeña capacidad en relación con ésta, pero varias veces más rápidas, son una solución de compromiso entre costo y velocidad. Contienen las instrucciones y datos de memoria a los que la UCP accedió últimamente, siendo también a los que más probablemente accederá próximamente, como se discute a continuación "Caché" significa oculta, escondida, en el sentido que la UCP envía una dirección por el bus de direcciones *desconociendo la existencia del caché*, que es un hardware también oculto al programador.

En general en los distintos niveles de los procesos de datos se trata que el acceso a la información requerida sea rápido y económico. Para tal fin en los procesos de datos se tiene en cuenta el *principio de localidad o proximidad*, el cual estipula que si ya se consultó información, es muy probable:

1. que la misma ***pronto*** sea consultada *otra vez* (*localidad o proximidad temporal*).
2. que ***pronto*** se consulte información cercana, vecina a ella (*localidad o proximidad espacial*).

Por ejemplo, la apertura de archivos de un computador ha sido concebida de acuerdo con este principio. Así, cuando en el Word abrimos un archivo, y luego lo cerramos, si enseguida o en otra oportunidad queremos abrir un archivo, el nombre del último archivo cerrado será el primero en aparecer en una lista ordenada de los más recientes archivos consultados. De este modo se estima que es muy factible que volvamos a consultarlos (proximidad temporal). Por otro lado, si abrimos un archivo de una carpeta, y luego pedimos abrir otro, el Word automáticamente mostrará dicha carpeta (proximidad espacial), pues en la mayoría de los casos el archivo buscado es uno de la carpeta en uso. (proximidad espacial). En ambos casos se evita que el usuario abra carpetas innecesariamente, pudiendo así acceder más rápidamente a la información que necesita.
Vale decir, que del conjunto de todos los archivos almacenados en el disco rígido *no se accede a todos ellos con igual probabilidad*, sino que generalmente se consulta un subconjunto que pertenecen a una misma carpeta o a unas pocas en relación con todo el disco; y además se accede a ellos repetidamente
También en el caso de la memoria principal, en cualquier lapso breve de tiempo los accesos sucesivos a ella ocurren en un espacio limitado de direcciones consecutivas (proximidad espacial), y es corriente acceder repetidamente a esas direcciones (proximidad temporal).
Los programas constan de secuencias de instrucciones que están en direcciones consecutivas de memoria, siendo frecuente que secuencias se vuelvan a ejecutar n veces, dando lugar a los "ciclos". Lo primero implica que es muy probable que las direcciones de dos instrucciones que se ejecutan una tras otra sean muy cercanas.
A su vez, los datos que estas instrucciones procesan, si constituyen elementos ordenados de un vector o de una matriz, o los cercanos a la cima de una pila, se encuentran vecinos en una zona acotada de memoria, y los programas ordenan operar una y otra vez sobre estos datos.

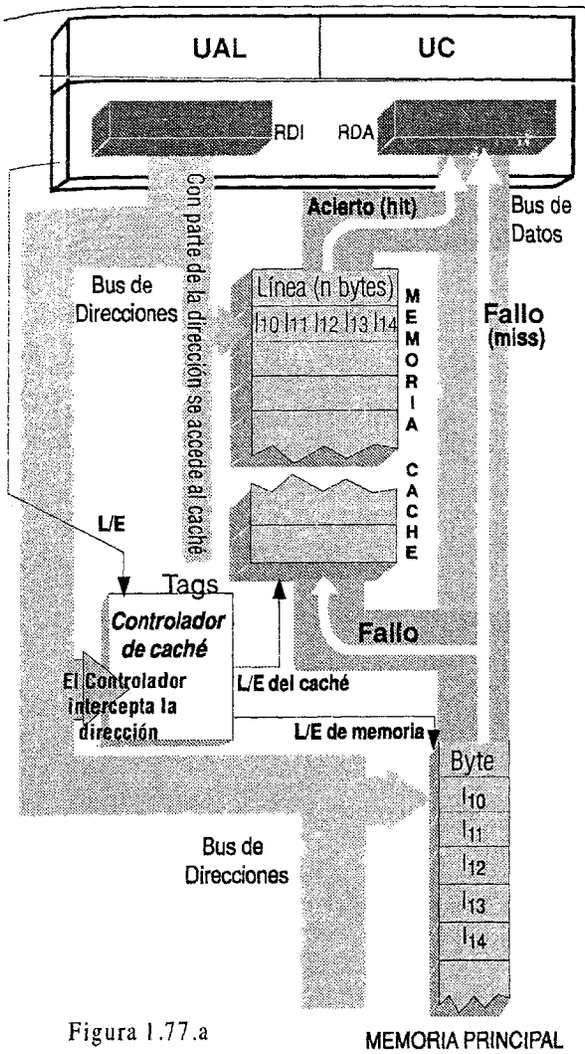


Figura 1.77.a

Por lo tanto, la ejecución de un programa se va realizando en secuencias de instrucciones relativamente cortas, que mayormente se ejecutan muchas veces, siendo que también los datos que cada secuencia procesa están en direcciones próximas, los cuales asimismo en general son accedidas repetidamente. En este caso también resulta para un conjunto de instrucciones y datos a procesar almacenados en memoria, que sus elementos no son accedidos con igual probabilidad; sino que durante cada lapso del proceso de datos que realiza la UCP sólo se accede con mayor probabilidad a dos subconjuntos: uno de instrucciones y otro de datos (que ellas operan), localizados en dos zonas reducidas de memoria principal (cuyos espacios de direcciones varían a medida que se ejecuta el programa). En especial se accede a bloques de datos de gran longitud en el procesamiento de textos, planillas y multimedia.

Los contenidos de estas dos zonas limitadas de memoria DRAM que almacenan instrucciones y datos que la UCP necesita acceder *por estar procesándolos o para procesarlos próximamente*, se copian en una memoria SRAM pequeña (como ser 512 KB ó 1 MB frente a 128 MB de la DRAM) pero de acceso mucho más rápido para la UCP que la DRAM: el "**cache**" o "antememoria". De este modo la UCP leerá del caché instrucciones y datos más rápidamente que de memoria, sin necesidad de acceder a ella.

Desde otra perspectiva, un caché permite *simular* una memoria principal DRAM de gran capacidad, pero con un tiempo de acceso semejante al de la SRAM de la caché. Este tiempo no debe superar el lapso T que media entre un pulso reloj y el siguiente, a fin de que la UCP no se quede "esperando" inactiva más de un pulso reloj. Cada pulso extra de espera (*"wait state"*) en cada lectura o escritura, implica una pérdida de performance de la UCP. Esto se manifiesta en un mayor tiempo en la ejecución de los programas.

Así, mientras los procesadores como el 80286 no superaban los 10 Mhz, o sea periodo T = 100 nseg, (sección 1.7), dado que el tiempo de acceso de una DRAM entonces era de unos 90 nseg, en general no superaba dicho tiempo T. Pero ya en el 80386 de 33 Mhz, o sea T = 30 nseg, el acceso a la DRAM apenas bajó a 70 nseg, superando el tiempo T (sin contar el tiempo que suma el controlador de memoria), por lo que se usó un caché externo (Level 2 = L2) para minimizar los "wait states". Con el 80486 rondando los 100 Mhz (T = 10 nseg) el acceso a un caché L2 superaba dicho T, debiéndose incorporar un caché dentro del 486 (Level 1 = L1), usando también un L2 para acceder más rápido a éste en lugar de la memoria, si la información a acceder no estaba en L1. Estos dos niveles de caché se usan hoy día para un mejor rendimiento, pudiendo existir más niveles.

Un caché guarda en las celdas que constituyen sus "**líneas**" un subconjunto de bloques de la memoria. Cada línea almacena un bloque de bytes consecutivos de la memoria (por ejemplo 16 ó 32.

El caché requiere un subsistema circuital complejo denominado "***controlador de caché***" que cumple varias funciones, siendo las más importantes:

a. Intercepta la dirección que envía la UCP por el bus de direcciones a fin de determinar si el contenido correspondiente a ella está (acierto = "hit") o no (fallo = "miss"). En el primer caso permitirá que la UCP acceda al caché, y en el segundo a la memoria con la consiguiente pérdida de tiempo. Para tal determinación cuenta con una rápida memoria auxiliar: la "tag memory".
b. En los cachés con correspondencia asociativa, decide cuál bloque será reemplazado por otro proveniente de memoria cuando ocurre un fallo.
c. Implementa la forma en que los resultados obtenidos por la UCP se guardarán en memoria.

¿Cómo es la estructura de un caché de correspondencia directa ?

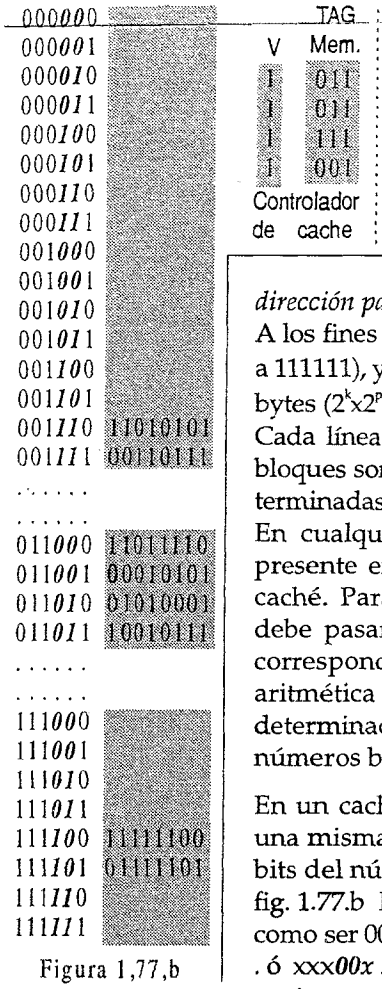


Figura 1,77,b

Mientras que un chip de memoria está organizado como una sucesión de celdas independientes que pueden guardar un byte, identificable cada una por su dirección (fig. 1.77.a), un caché se organiza en un conjunto de líneas que típicamente guardan un bloque de 16 ó 32 bytes cada una. Cada línea se identifica y localiza en el caché por su número de "entrada" o "índice", *que viene a ser como su dirección para localizarla* en la SRAM. El número de líneas es una potencia de dos.

A los fines didácticos supondremos (fig. 1.77.b) una memoria de 2^6 bytes (direcciones 000000 a 111111), y un caché con capacidad para 8 bytes estructurado en 2^k=4 líneas que guardan 2^p=2 bytes ($2^k x 2^p = 2^2 x 2^1 = 8$) con 2^k=4 entradas numeradas de *00, 01, 10, 11* (números de k=2 bits) Cada línea guarda un "bloque" fijo de bytes *consecutivos* de memoria. En este ejemplo los bloques son de 2 bytes (2 campos) el de la derecha de la línea para direcciones de memoria terminadas en 1 (p=0), y el de la izquierda para las terminadas en 0 (p=1).

En cualquier instante *una copia* de un subconjunto de bloques de la memoria está presente en el caché, siendo que su número es mucho mayor que el de líneas de un caché. Para establecer en que número de línea se asigna un bloque de memoria que debe pasar al caché en reemplazo de uno residente en él, se utiliza un algoritmo de correspondencia. Este opera con la dirección de memoria que produjo el fallo usando aritmética del módulo. En el caso de las direcciones se reduce a una correspondencia determinada por un subconjunto de bits de cada una de ellas, por ser las direcciones números binarios sucesivos cuya cantidad es una potencia de dos.

En un caché de *correspondencia directa,* cada bloque de memoria se adjudica siempre a una misma línea, cuyo número de entrada (dirección en el caché) es un subconjunto de bits del número binario que es la dirección de cualquiera de los bytes del bloque. Así, en la fig. 1.77.b los contenidos de todas las direcciones de memoria cuyos bits en "italica" son *00*, como ser 000*00*0 ó 000*00*1 ó 001*00*0 ó 001*00*1 ó 010*00*0 ó 010*00*1 ó 011*00*0 ó 011*00*1 ó... ó xxx***00x*** ..... ó 111*00*0 ó 111*00*1 **sólo** se pueden guardar en la línea *00* del caché. Según terminen en 0 ó 1 irán a la izquierda o derecha, respectivamente.

Análogamente, los contenidos de direcciones con bits indicados *01* tales como 0000*01*0 ó 000*01*1 ó 001*01*0 ó 001*01*1 ó 010*01*0 ó 010*01*1 ó 011*01*0 ó 011*01*1 ó... ó xxx***01x*** ..... ó 111*01*0 ó 111*01*1 se guardarán siempre en la línea *01*.

Direcciones del tipo xxx***10***x y xxx***11***x deben guardarse en las líneas *10* y *11*, respectivamente.

En general, si un caché tiene 2^k líneas (4 en este ejemplo) y cada una guarda 2^p bytes (2 en este caso), para cada dirección se usarán los últimos *p* bits de la derecha para indicar en que posición de la línea se guardará el byte correspondiente a esa dirección; y los siguientes *k* bits (a la izquierda de esos *p* bits) se emplean para indicar el número de línea dónde estará dicho byte. En el ejemplo, k = 2 bits de línea (líneas *00, 01, 10, 11*), y p=1 bit (el byte puede estar en la posición 0 ó 1 dentro de una línea).

Como se verá, para cada dirección de memoria, los *t* bits que están a la izquierda de los *k* bits de línea (en este caso t=3) sirven para que el controlador determine si el contenido de dicha dirección está o no en el caché. Dichos *t* bits constituyen el "*tag*" (etiqueta) de una dirección. Así, 010 es el tag de la dirección 010*011*.

Puesto que cada línea guarda un bloque de 2^p bytes consecutivos de memoria, *las direcciones de los bytes de un bloque que está en una línea tendrán igual tag,* ya que ellas sólo difieren en sus últimos *p* bits de la derecha.

Determinaremos t, k, p para una memoria DRAM de 2^{32} bytes (4 GB) que usa un caché SRAM de 512 KB = 2^{19} bytes organizado en 32768 líneas = 2^{15} líneas que guardan 16 bytes = 2^4 bytes (se verifica que $2^{15} x 2^4 = 2^{19}$ bytes). Puesto que $2^{15} = 2^k$ resulta que k=15 (números de línea de *000000000000000* a *111111111111111*), y p=4 (en cada línea cada una de las $2^p=2^4$=16 posiciones que guardan un byte se identifica por los últimos p=4 bits de su dirección: 0000 a 1111). Dado que cada dirección tiene 32 bits, resulta: t = 32 - (15 + 4) = 13 bits para cada tag.

Por ejemplo, en una dirección de 32 bits como 10110100111001*1011101110101010*1101, debe considerarse que 1011010011100 es el tag; ***1011101110101010*** es el número de línea, y 1101 indica dentro de ésta la posición del byte correspondiente a la dirección dada. Son comunes cachés de 32 bytes por línea.

Por lo tanto la estructura del caché determina los números de bits de **k** y **p**, siendo que los de **t** son los que restan del número *n* de bits de cada dirección de memoria: **t = n - (k + p)**.

El controlador de caché posee una "tag memory": una SRAM que contiene un tag por cada línea del caché, que es el mismo para todas las direcciones de los 2^p bytes que ella guarda (fig. 1.77.b), pues las mismas sólo difieren en sus últimos *p* bits de la derecha (p =1 en la fig. 1.77.b).
Cuando luego de un fallo se transfiere un bloque de bytes consecutivos de memoria hacia una línea del caché, el controlador escribe en su "tag memory" el tag de las direcciones de esos bytes.
Si luego la UCP ordena leer el contenido de una dirección de memoria, ella será interceptada por el controlador, que rápidamente comparará el tag de la misma con el tag de la "tag memory" vinculado al número de línea correspondiente a esa dirección. Si el circuito comparador indica igualdad de tags, es un acierto, o sea que en la línea del caché correspondiente a ese tag está el byte que la UCP direccionó, y por lo tanto dicho byte será proporcionado por el caché casi al mismo tiempo de la indicación de acierto.
De no ser así resulta un fallo, por lo que el controlador permitirá leer en la memoria el contenido de dicha dirección (fig. 1.77), el cual llegará a la UCP (en la que ocurrirán "wait states"), y el mismo también pasará a la línea del caché citada -que antes no lo contenía- junto con el byte de la dirección adyacente, reemplazándose así el anterior contenido de esa línea. Asimismo, en la "tag memory" se cambia el tag de esta línea por el tag del nuevo bloque.
Trataremos de concretar esta descripción en el caché 'básico" de la fig. 1.77.b, para lo cual supondremos que sucedieron las siguientes asignaciones, siendo que al hacer cada asignación se escribió el tag de cada línea -entre paréntesis- en la tag memory del controlador.

línea *00* : contenidos 11011101 y 00010101 de las direcciones 011***00*** y 011***00***1 (011)
línea *01* : contenidos 01010001 y 10010111 de las direcciones 011***01*** y 011***01*** (011)
línea *10* : contenidos 11111100 y 01111101 de las direcciones 111***10*** y 111***10***1 (111)
línea *11* : contenidos 11010101 y 00110111 de las direcciones 001***11*** y 001***11***1 (001)

Supongamos ahora que la UCP quiera leer la memoria, para lo cual por el bus de direcciones envía la dirección 011010, que será interceptada por el controlador. Así se determina que se trata de la línea *01*, y se compara su tag (011) con el de la dirección. Como ambos son iguales resulta un acierto, por lo que el caché proporcionará el contenido 01010001 correspondiente a la posición 0 de esa línea, siendo 0 el último bit de la dirección. A tal fin a la salida del caché existe un multiplexor que mediante el valor de este último bit permite seleccionar si por sus salidas aparecerá el byte de las posiciones (campos) 0 ó 1.
Si la dirección hubiese sido 101*010*, al comparar para la línea *01* su tag existente (011) con el 101 de esa dirección, resultaría un fallo. Entonces se accederá a memoria para obtener los contenidos de las direcciones 101010 y 101011, los cuales reemplazarán a los anteriores 01010001 y 10010111. Luego el nuevo contenido de la posición 0 de dicha línea pasará a la UCP. En la "tag memory" se reemplazará 011 por 101.
Como en el bloque traído de memoria *también* está el byte de la dirección *siguiente* (101011) a la solicitada, muy probablemente dicho byte será pedido por la UCP en el próximo acceso, por lo cual en el mismo ocurrirá un acierto. Teniendo en cuenta que en la práctica un bloque puede tener 16 ó 32 bytes, al ser llevado al caché junto con el byte solicitado se guardarán bytes de las direcciones consecutivas siguientes a la que ocasionó el fallo. De este modo al llegar más bytes que los solicitados, *en forma automática* se trata de aprovechar la proximidad espacial, pues es dable esperar que los bytes que llegaron "de más" anticipadamente en el bloque, sean los próximos en ser accedidos en el caché, resultando una sucesión de aciertos.

Lo anterior también sirve para subrayar el hecho de que *un caché recibe información desde memoria solamente cuando ocurre un fallo*, o sea en el caso que el byte direccionado por la UCP no se encuentra en él. Ello implica por un lado, que la información fluye de memoria hacia el caché en forma discontinua, pues la UCP accede continuamente al caché. Por lo tanto el tráfico desde memoria hacia la UCP se reduce notablemente, posibilitando a su vez que la memoria pueda ser accedida con menos conflictos con la UCP por parte de dispositivos de E/S que realizan ADM: acceso directo a memoria (fig. 1.72). Por otra parte se verifica que la próxima información a la que seguramente la UCP accederá llega *en forma automática*: como se accede a un bloque de direcciones consecutivas de memoria, junto con los bytes que se direccionaron (que provocaron el fallo) vienen a la línea del caché que se reemplaza los bytes de las direcciones siguientes que próximamente serán accedido por la UCP.

Además de seguido, por lo general los controladores también piden el bloque con las direcciones siguientes al bloque que contenía la dirección del fallo, el cual irá a otra línea del caché. Así en éste por lo menos existirán dos bloques (líneas) con direcciones consecutivas.

En la "tag memory" SRAM se determina si un contenido de ella (tag) *está o no*, para lo cual se *compara* el tag de la dirección generada por la UCP con el tag de la línea del caché accedida, utilizando un circuito comparador por línea.
Este funcionamiento es propio de las denominadas "**memorias asociativas**", en las cuales se puede así determinar si un número -en este caso el de un tag- se encuentra o no en ellas.
Para no perder tiempo, a la par que se accede al tag de una línea mediante el campo de *k* bits de la dirección (*00, 01, 10 ó 11* en el ejemplo), también se accede a una copia del bloque contenido en esa línea, para que aparezca en las salidas del caché, haya acierto o fallo. Si hay acierto, parte del mismo (seleccionado con los *p* bits de la dirección) irá a la UCP. En caso de fallo simplemente no pasa a la UCP.

Un caché con más bytes por línea, responderá mejor cuando la UCP accede a una zona con buena proximidad espacial, como ser una larga secuencia de instrucciones consecutivas. En cambio cuando ocurre un fallo (< 10 % de los accesos) tendrá que acceder a un mayor número de bytes sucesivos en memoria que un caché con menos bytes por línea.

En el controlador (fig. 1.77.b) además de los "tags" existe para cada línea un bit de validación (V) que sirve al controlador para saber si los contenidos de una línea son válidos. Por ejemplo, cuando se enciende el computador todas las líneas tendrán su V=0, de modo que si la UCP genera una dirección que por casualidad está en el caché así se indica que los contenidos de la línea correspondiente son inválidos, puesto que son "basura" accidental. Luego, en los primeros instantes de la ejecución de un programa, las primeras direcciones de instrucciones y datos no estarán en la "tag memory", ocurriendo un gran número de fallos; pero con cada nuevo bloque que es traído de memoria, el mismo contendrá bytes de direcciones que siguen a la que la UCP quiere acceder, con lo cual comenzará a funcionar la proximidad espacial y temporal. Asimismo, para cada línea donde hubo reemplazo de bloque, el controlador pondrá su V=1.

Los caches de correspondencia directa si bien presentan una circuitería sencilla, el hecho de que se adjudique cada bloque de memoria siempre a la misma línea (por ej cada dirección xxx*01*x irá a la línea *01*) puede ocasionar a veces una disminución en la tasa de aciertos. Tal sería el caso en que se ejecutan instrucciones que direccionan en forma repetitiva datos de dos bloques de memoria, los cuales por sus direcciones deban ser adjudicados, casualmente, a una misma línea del caché. Por lo tanto, en ese lapso constantemente el controlador debería ir cambiando el bloque que está en esa línea, pasando al cache en forma alternada uno y otro de esos dos bloques de la memoria. Estas situaciones se solucionan con compiladores más inteligentes, que no generen este tipo de accesos.

Los problemas del tipo anterior no pueden ocurrir en los cachés con *correspondencia totalmente asociativa* donde *cualquier* bloque puede ser adjudicado por el controlador a *cualquier* línea, lo cual permite aprovechar mejor el principio de proximidad en el acceso al caché, posibilitando una mayor tasa de aciertos.
En la fig 1.77.b, por ejemplo el bloque de direcciones 101010 y 101011 sería factible asignarlo a una cualquiera de las líneas del caché (suponiendo igualmente 8), según mejor convenga. y no sólo a la línea *01*.
El controlador de este caché cumplimenta un algoritmo que permite determinar a cuál línea (por ej *la menos accedida últimamente*) se asignará el bloque de memoria que debe entrar en el caché, en reemplazo de otro bloque. O sea que el número de línea no se obtiene a partir de la dirección que produjo un fallo.
Para determinar si hay o no acierto se requiere comparar el tag de la dirección a la que la UCP quiere acceder, con el tag de cada una de las líneas en la "tag memory". Puesto que se requiere rapidez habrá que hacer múltiples comparaciones simultáneas, resultando compleja y limitada en velocidad la circuitería de comparación. Además los comparadores deben tener muchas entradas, dado que cada tag comprende bastantes bits, pues son los de cada dirección menos los **p** bits de la derecha (para la fig. xxx, el tag sería de t=5 bits, y p=1). Por ello sólo esta correspondencia sólo es viable en cachés relativamente pequeños, con pocas líneas.
A diferencia de esto, en el caché de correspondencia directa tratado, cada bloque a asignar iba una línea predeterminada, fija, según su dirección. En él se reemplaza al bloque existente, sin considerar si éste fue o no recientemente accedido, o sea que no se explota a fondo la proximidad temporal de los contenidos.

Además de seguido, por lo general los controladores también piden el bloque con las direcciones siguientes al bloque que contenía la dirección del fallo, el cual irá a otra línea del caché. Así en éste por lo menos existirán dos bloques (líneas) con direcciones consecutivas.

En la "tag memory" SRAM se determina si un contenido de ella (tag) *está o no*, para lo cual se *compara* el tag de la dirección generada por la UCP con el tag de la línea del caché accedida, utilizando un circuito comparador por línea.
Este funcionamiento es propio de las denominadas "**memorias asociativas**", en las cuales se puede así determinar si un número -en este caso el de un tag- se encuentra o no en ellas.
Para no perder tiempo, a la par que se accede al tag de una línea mediante el campo de *k* bits de la dirección (*00, 01, 10 ó 11* en el ejemplo), también se accede a una copia del bloque contenido en esa línea, para que aparezca en las salidas del caché, haya acierto o fallo. Si hay acierto, parte del mismo (seleccionado con los *p* bits de la dirección) irá a la UCP. En caso de fallo simplemente no pasa a la UCP.

Un caché con más bytes por línea, responderá mejor cuando la UCP accede a una zona con buena proximidad espacial, como ser una larga secuencia de instrucciones consecutivas. En cambio cuando ocurre un fallo (< 10 % de los accesos) tendrá que acceder a un mayor número de bytes sucesivos en memoria que un caché con menos bytes por línea.

En el controlador (fig. 1.77.b) además de los "tags" existe para cada línea un bit de validación (V) que sirve al controlador para saber si los contenidos de una línea son válidos. Por ejemplo, cuando se enciende el computador todas las líneas tendrán su V=0, de modo que si la UCP genera una dirección que por casualidad está en el caché así se indica que los contenidos de la línea correspondiente son inválidos, puesto que son "basura" accidental. Luego, en los primeros instantes de la ejecución de un programa, las primeras direcciones de instrucciones y datos no estarán en la "tag memory", ocurriendo un gran número de fallos; pero con cada nuevo bloque que es traído de memoria, el mismo contendrá bytes de direcciones que siguen a la que la UCP quiere acceder, con lo cual comenzará a funcionar la proximidad espacial y temporal. Asimismo, para cada línea donde hubo reemplazo de bloque, el controlador pondrá su V=1.

Los caches de correspondencia directa si bien presentan una circuitería sencilla, el hecho de que se adjudique cada bloque de memoria siempre a la misma línea (por ej cada dirección xxx*01*x irá a la línea *01*) puede ocasionar a veces una disminución en la tasa de aciertos. Tal sería el caso en que se ejecutan instrucciones que direccionan en forma repetitiva datos de dos bloques de memoria, los cuales por sus direcciones deban ser adjudicados, casualmente, a una misma línea del caché. Por lo tanto, en ese lapso constantemente el controlador debería ir cambiando el bloque que está en esa línea, pasando al cache en forma alternada uno y otro de esos dos bloques de la memoria. Estas situaciones se solucionan con compiladores más inteligentes, que no generen este tipo de accesos.

Los problemas del tipo anterior no pueden ocurrir en los cachés con *correspondencia totalmente asociativa* donde *cualquier* bloque puede ser adjudicado por el controlador a *cualquier* línea, lo cual permite aprovechar mejor el principio de proximidad en el acceso al caché, posibilitando una mayor tasa de aciertos.
En la fig 1.77.b, por ejemplo el bloque de direcciones 101010 y 101011 sería factible asignarlo a una cualquiera de las líneas del caché (suponiendo igualmente 8), según mejor convenga. y no sólo a la línea *01*.
El controlador de este caché cumplimenta un algoritmo que permite determinar a cuál línea (por ej *la menos accedida últimamente*) se asignará el bloque de memoria que debe entrar en el caché, en reemplazo de otro bloque. O sea que el número de línea no se obtiene a partir de la dirección que produjo un fallo.
Para determinar si hay o no acierto se requiere comparar el tag de la dirección a la que la UCP quiere acceder, con el tag de cada una de las líneas en la "tag memory". Puesto que se requiere rapidez habrá que hacer múltiples comparaciones simultáneas, resultando compleja y limitada en velocidad la circuitería de comparación. Además los comparadores deben tener muchas entradas, dado que cada tag comprende bastantes bits, pues son los de cada dirección menos los **p** bits de la derecha (para la fig. xxx, el tag sería de t=5 bits, y p=1). Por ello sólo esta correspondencia sólo es viable en cachés relativamente pequeños, con pocas líneas.
A diferencia de esto, en el caché de correspondencia directa tratado, cada bloque a asignar iba una línea predeterminada, fija, según su dirección. En él se reemplaza al bloque existente, sin considerar si éste fue o no recientemente accedido, o sea que no se explota a fondo la proximidad temporal de los contenidos.

La *correspondencia asociativa por conjunto* (de 2^k líneas) *con c conjuntos o vías* (que implican c alternativas de asignación), es una solución de compromiso entre la correspondencia directa y la totalmente asociativa. Como en la fig. 1.77.b el número de línea donde irá un bloque está determinado por *k* bits comunes a todas las direcciones de los bytes de ese bloque. Pero hay c líneas (vias) con igual número de línea. El controlador conforme a un algoritmo, adjudica el bloque a una de dichas c líneas: la menos accedida últimamente.

Reformaremos (fig. 1.77.c) el cache de 4 líneas de la fig.1.77.b de modo de tener para cada número de línea 2 líneas que guardan un bloque de 2 bytes cada una. Así se forman c=2 conjuntos de líneas designados vía 0 y vía 1. En cada vía la posición de cada byte se identifica con 0 y 1.
Todas las direcciones del tipo xxxx*0*x irán a la línea ***0***, y las del tipo xxxx*1*x a la línea ***1***, siendo que el último bit de la derecha indica si irá a la posición 0 ó 1 de la línea. Ahora cada tag tendrá 4 bits.

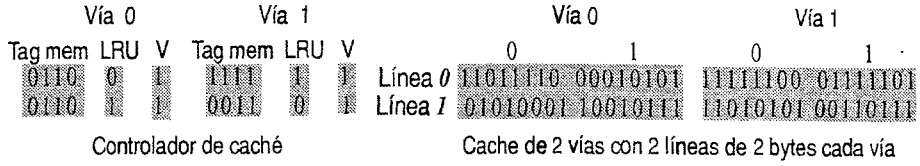


Figura 1.77.c

Así, el bloque de direcciones 011*00* y 011*01* (tag 0110) irá siempre a la línea ***0***, y el controlador decidirá si será la línea ***0*** de la vía 0 ó la línea ***0*** de la vía 1. Para tal fin además del tag, cada línea de cada vía, junto con su bit de validez V, tiene un bit "LRU" (least reciently used, o sea usada menos recientemente). Cada vez que una línea de una vía es accedida, el controlador pone en 1 su bit LRU, y 0 el bit LRU de igual número de línea de la otra vía.
En la fig. 1.77.c para un caché de igual tamaño que el de la fig. 1.77.b, aparecen reubicados los mismos bloques de esta última como se indica, con un valor supuesto para el bit LRU de cada uno.
Supongamos que hubo un fallo al querer asignar el bloque de direcciones 010*00* y 010*01* (de contenidos 10000101 y 11110000, no dibujados en la fig. 1.77.b). El mismo deberá ir al número de línea ***0***, y se asignará a la vía cuyo bit LRU vale 0, pues ello implica que el bloque que está en dicha vía (0 en este caso) ha sido accedido antes que el bloque de la otra vía, dado que éste con LRU=1 es más probable que sea accedido próximamente. Asimismo se colocará el tag (0100) en lugar del tag del bloque reemplazado. La nueva situación del caché y del controlador aparece en la fig. 1.77.d
Si luego la UCP quiere acceder otra vez a la dirección 010*00*, el controlador comparará el tag 0100 con los dos tags de los dos bloques de número de línea ***0*** para determinar en que vía está el bloque. En cada acierto, un multiplexor -ubicado en las salidas del caché- seleccionará por su tag el bloque accedido entre los dos posibles para un número de línea, el cuál aparecerá en dichas salidas.

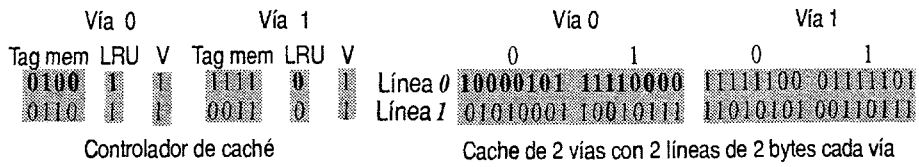


Figura 1.77.d

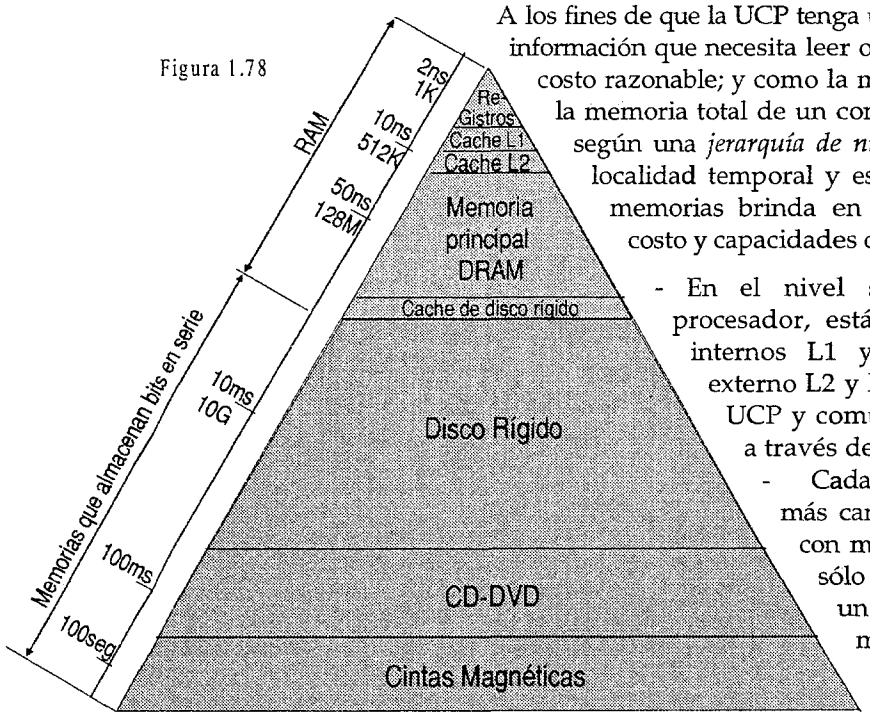
Con este esquema se evita el conflicto antes citado en la correspondencia directa, cuando dos bloques que iban a un mismo número de línea eran accedidos en forma alternada. Ahora estarían en igual número de línea, pero cada bloque en una vía distinta. La alternancia en el acceso sólo produciría el cambio de valor de los bits LRU de los dos bloques de igual número de línea.
Si bien estos cachés necesitan más comparadores que los de correspondencia directa y en consecuencia un mayor tiempo de acceso, son menos complejos y más rápidos que los de correspondencia totalmente asociativa, por lo que resultan una buena solución de compromiso entre ambos.
Si se tiene 4 vías, el controlador tiene 4 posibilidades para asignar un bloque a un número de línea.

La estrategia de usar dos cachés multinivel, uno muy pequeño dentro del procesador (level 1 - L1) y otro externo (L2), busca que L1 sea de acceso sea extra rápido, mientras que L2 se encargue de manejar los fallos de L1, de manera de minimizar los "wait states" de la UCP. Cuando el caché L2 accede a un bloque de memoria principal debido a un fallo, lo hace en modo "ráfaga" ("burst"), esto es el tiempo de acceso al primer byte del bloque (por ej. de 32 bytes) es mayor, pero los subsiguiente bytes por estar en direcciones consecutivas se acceden varias veces más rápido.

Hasta ahora hemos tratado las lecturas de instrucciones y datos en un caché, siendo que ellas predominan sobre las escrituras. Pero también la UCP requiere en menor grado guardar en memoria principal resultados de datos que ha procesado cuando una instrucción así lo ordena.
El controlador de caché también maneja la estrategia de escritura del caché y memoria principal.
Una forma denominada *"write through"* ("a través" del caché) consiste en escribir (actualizar) simultáneamente ambos, lo cual conlleva tiempo. Para mejorar esto, suele usarse un "buffer" de escritura, que es escrito rápidamente junto con el caché, y que guarda resultados hasta que sean escritos en memoria. En caso que la dirección a actualizar no se encuentre en el caché, sólo se escribe la memoria.
Otra forma, *"write back"*, o "post-escritura" consiste en marcar en el caché las líneas que la UC escribió en él, de forma que cuando la línea así marcada sea reemplazada, el bloque que contiene sea escrito en memoria principal. Esto último supone que durante un lapso de tiempo, el caché y la memoria tendrían información diferente, situación conocida como *"incoherencia"*. Puede ocurrir entonces que una porción de memoria tengan información desactualizada. Para hacer por ejemplo una lectura de esa porción hacia el disco rígido, se debe acceder al caché, lo cual hace más complejo el hardware.

¿Qué es una jerarquía de memorias ?

Figura 1.78



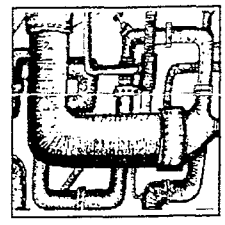
A los fines de que la UCP tenga un tiempo de acceso a la cantidad de información que necesita leer o escribir lo más bajo posible y a un costo razonable; y como la memoria más lenta es la más barata, la memoria total de un computador está organizada (fig 1.78) según una *jerarquía de niveles de memorias*. Ella combina la localidad temporal y espacial con lo que la tecnología de memorias brinda en cuanto a velocida-des de acceso, costo y capacidades de almacenamiento. Así se tiene:

- En el nivel superior (fig. 1.78), dentro del procesador, está la memoria más rápida (cachés internos L1 y registros). Le siguen el caché externo L2 y la memoria principal, cercanos a la UCP y comunicados directamente con la UCP a través de buses externos.
- Cada nivel es más pequeño, más rápido, más caro por byte almacenado, y accedido con más frecuencia que el nivel inferior; y sólo puede intercambiar información con un nivel adyacente. Se busca tener la mayor capacidad con el menor costo por byte almacenado (como proveen los discos), a la par que el tiempo de acceso de un nivel inferior a otro superior vecino sea lo menor posible, en especial para la UCP.
- La información contenida en un nivel también está en el nivel siguiente inferior.

Al tratar los cachés se vio que una UCP con una memoria principal DRAM de 64MB y tacc = 50 nseg. y un caché SRAM de 512KB y tacc = 20 nseg., pueden hacer que la UCP en más del 90% de los accesos que éstos sean de 20 nseg. O sea, es como si la UCP tuviera una memoria de 64MB y tacc = 20 nseg., siendo que resultaría muy caro tener 64MB de SRAM.
Con igual técnica, entre esa memoria DRAM y el disco duro se puede simular mediante el sistema operativo una *memoria virtual* de muchos GB, con un tacc muy aceptable y sin costo adicional.
Igualmente los CDs amplían económicamente la capacidad del rígido.
Se trata siempre, entre dos niveles adyacentes, de soluciones de compromiso entre costo y velocidad.

En síntesis, para cada computador se busca construir una jerarquía de memorias tal que para el usuario simule una memoria con una capacidad de almacenamiento auxiliar virtualmente ilimitada, y con un tiempo de acceso tan rápido hoy día, como el primer nivel de memoria caché incorporado al procesador.

1.13 DETALLES DE LOS BUSES



¿Que características tienen los buses compartidos ?

Un **bus** es un camino eléctrico constituido por líneas conductoras, que permite intercambiar información binaria entre dos o más dispositivos. Si es compartido (fig. 1.79.b) por más de dos, éstos pueden comunicarse entre sí a través del bus a razón de dos por vez, siendo que en un instante dado uno solo puede transmitir. Cualquier señal que éste transmite puede ser recibida por cualquier otro dispositivo del bus, pero sólo responderá el que fue seleccionado.

Figura 1.79.a

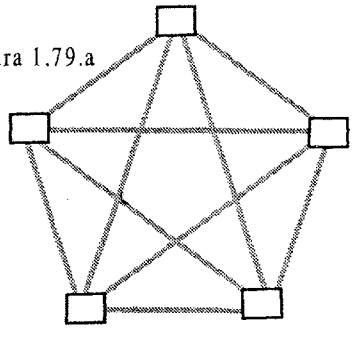
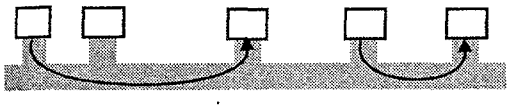


Figura 1.79.b



En general se trata de que un bus permita económicamente que se conecten a él dispositivos de distinto tipo con tal que cumplan con especificaciones mecánico-eléctricas y de señalización temporal.
El hecho de compartir líneas implica una gran economía de cableado y espacio, en relación con los mismos dispositivos conectados de a dos, "punto a punto", con un bus independiente *dedicado* entre cada par (fig. 1.79.a). Un bus compartido tiene la limitación que dispositivos conectados al mismo deben esperar para recibir o transmitir datos, pues ello se realiza *sólo* entre dos dispositivos por vez; y que se necesita circuitería adicional de *control del bus*, que asegure esto último, para evitar interferencias.
Lo dicho puede equipararse a teléfonos internos que comparten una misma línea. Antes de comunicarse uno con otro debe constatarse que el indicador luminoso de que 2 están hablando no esté encendido.
En un bus compartido se necesita un *protocolo de arbitraje* que especifique la secuencia de pasos (señales) que deben ocurrir para que un dispositivo pueda comunicarse con otro. Con los teléfonos sería: ver si el Led está apagado, sino esperar; levantar el tubo; digitar el número del otro interno, etc.).
Así se determina, *controlador del bus* mediante, cuál es el próximo dispositivo ("*master*") que va a iniciar una transferencia con otro del bus. El "*master*" selecciona el dispositivo "*slave*" o "*target*" con el cual realizará la transferencia de datos en un sentido u otro.
En un bus *serie* la información se transmite bit a bit por un solo conductor (dos en el USB), mientras que en un bus *paralelo* esto se realiza mediante un cierto número de conductores para transmitir simultáneamente *n* bits. Para más detalles eléctricos es conveniente ver la sección 1.10.
Cada dispositivo se identifica y selecciona por un número que es su *dirección* (el número de interno en los teléfonos). Entre dos dispositivos comunicados se establece un *enlace* o camino entre ellos.

Estructura de un bus paralelo:
En un bus paralelo la selección se realiza mediante *líneas de dirección*, y la transmisión de datos usando *líneas de datos*, cuyo número (8, 16, 32, 64, 128) define el *ancho* del bus. La fig. 1.79.c ilustra estas líneas en un bus ISA que interconecta una UCP 8088, su memoria y slots para tarjetas de interfaces para periféricos
Las líneas de dirección y datos se indican como caminos grisados. Se supone que el 8088 envía una dirección (que llega a todos los dispositivos) para seleccionar una celda de memoria con orden de lectura (línea de control L/E=1). Sólo responde la memoria enviando un byte por las líneas de datos.
Es factible que un mismo conjunto de líneas primero se use para direccionar y luego para transmitir.
Las *líneas de control* transmiten señales si/no para: peticiones (por ejemplo la línea IRQ solicita interrupción), órdenes (por ejemplo línea L/E para leer/escribir), avisos, confirmaciones y tipo de información enviada. Así controlan el acceso y la ocupación de las líneas de dirección y datos, y de hecho *indican cuándo son válidos los valores* presentes en esas líneas.

Si el bus es **sincrónico** (como el ISA o el PCI) una de las líneas de control designada "clock" o Mhz recibe pulsos regularmente espaciados en el tiempo, generados por un oscilador, los cuales marcan los instantes en que se pueden activar o desactivar líneas de control, o enviar direcciones o datos.

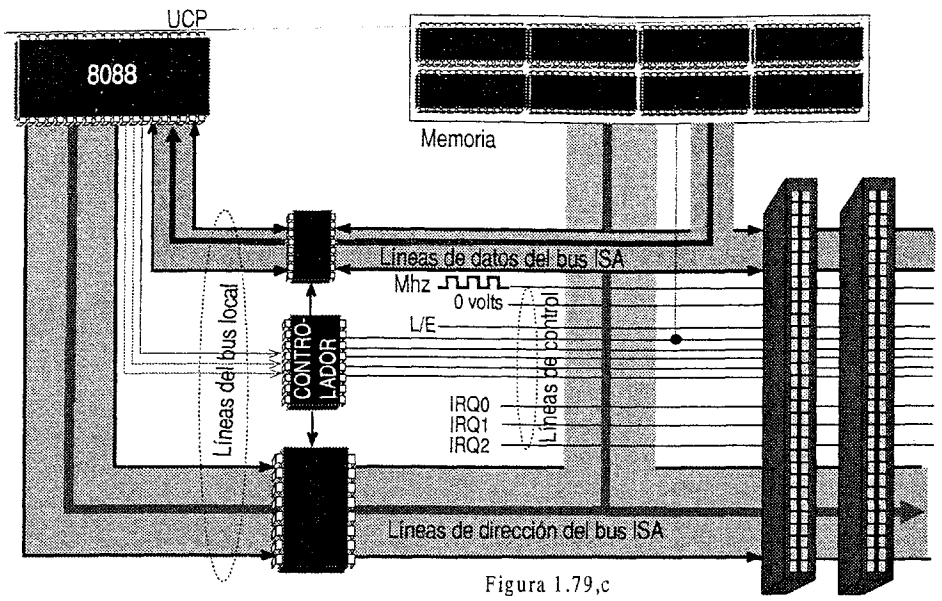


Figura 1.79,c

En un bus **asincrónico** a partir de la activación de una línea por un "master" en un instante no regido por un clock, se sucede en respuesta la activación de otra línea por el "slave", a la cual puede seguir la activación de otra, etc. Así la activación de una línea depende de la activación de otra, pero no del clock: cada nuevo evento sólo es respuesta al evento anterior al mismo. Este bus (como el SCSI) es más complejo de implementar, pues los dispositivos deben ser auto-temporizados.

Una *transacción* en un bus presenta dos fases: envío de una dirección seguida del envío de datos.

La velocidad de transmisión (Mbytes/seg) o "*ancho de banda*" de un bus está limitada por su longitud y por el número de dispositivos conectados al mismo. Su causa es la inercia en la velocidad de cambio de las señales eléctricas digitales debido en esencia al tiempo de carga/descarga de capacitores parásitos distribuidos a lo largo de las líneas de un bus, cuyo efecto aumenta con cada dispositivo conectado. Por tal motivo el bus local vinculado al procesador es de corta longitud frente a buses de E/S como el ISA, el SCSI o el USB, que por ser más lentos permiten longitudes mayores y más dispositivos conectados.

Jerarquía de buses:

Muchos computadores se hicieron con un único bus, como ilustran las figs 1.79.c, 1.67 y 1.72 para ser compartido por los dispositivos arriba indicados y los controladores de ADM. Pero a medida que crece el número de dispositivos conectados aumenta la puja de los mismos por usar el bus, produciéndose "cuellos de botella". Si bien puede mejorarse la velocidad de transferencia (como ser con más líneas para datos, realizando transferencias de datos mediante grandes bloques de bytes sucesivos, arbitrajes rápidos y utilización de buffers) para disminuir el tiempo de ocupación por dispositivo, dicha velocidad se ve limitada entre otras cosas por la capacidad parásita que agrega cada dispositivo. Ello no se condice con las cada vez mayores exigencias de velocidad de dispositivos para gráficos y vídeo.

A los efectos de descongestionar el tráfico que tiene lugar si se usa un solo bus, hoy día los computadores presentan una *jerarquía de buses* de distinta velocidad y longitud, que por un lado minimiza la competencia por el acceso a memoria. Por otro permite conectar muchos periféricos reduciendo conflictos y tenerlos fuera del gabinete. Así (fig. 1.80), el bus que une el caché L2 (sección 1.12) con la UCP concentra todo el tráfico, que cuando el mismo no existía iba de memoria principal a la UCP por el bus que los une. Este ahora queda más despejado para transferir por ADM (sección 1.10) datos entre periféricos y memoria. Asimismo, las interfaces de los periféricos no se conectan al bus del sistema PCI, sino a otros **buses de E/S** como el ISA, USB o SCSI, de modo que al PCI sólo están conectados los adaptadores inteligentes o puentes ISA/PCI, USB/PCI, SCSI/PCI que aíslan al PCI de un *número grande* de periféricos de *velocidad dispar*, conectados a dichos buses. Esto permite conectarlos con buses de mayor longitud (SCSI, USB) fuera del gabinete, sin ocupar el número limitado conectores ("slots") existentes. También dichos adaptadores evitan conflictos en relación con la configuración de IRQ y ports de las interfaces (sección 1.11 y 1.10).

El nivel más alto de la jerarquía está constituido por el "local bus" que sale de la UCP, de muy alta velocidad y corta longitud, con *muy pocos y veloces* dispositivos conectados a él. Como éstos operan con velocidades altas y no muy dispares, es recomendable que el bus sea sincrónico, por ejemplo para transferir entre memoria y el caché L2 (que en un Pentium actual están conectados por un bus interno

dedicado). Mediante un chip "puente" este bus se comunica con el bus PCI, al cual están conectados los buses de E/S citados, a los que se conectan los restantes componentes del computador (fig. 1.80).
El bus PCI (fig 1.80), está ubicado en una jerarquía intermedia entre los buses ultra rápidos que salen de la UCP y los buses de E/S. Por él pasan datos desde o hacia la UCP, la memoria, y dispositivos periféricos.
Debe mencionarse que el denominado Port Acelerado para Gráficos (**AGP**), que funciona a los Mhz de la UCP, está pensado como intermediario para las transferencias entre memoria y la tarjeta de vídeo, de modo que se efectúen a máxima velocidad (528 Mbytes/seg para 2X, y > 1Gbyte/seg para 4X, siendo la velocidad máxima del PCI cercana a los 100 Mbytes/seg), a través del chip que contiene al puente PCI.

¿ Cómo funciona el bus PCI ?

El bus **PCI** (*Peripheral Component Interconnect*) creado por Intel, por su independencia del procesador y subsistema de memoria usados, por su versatilidad de conexionado con otros buses para periféricos existentes (ISA, SCSI, USB, etc), por sus 64 líneas para datos, y por su característica "plug'n play" (**PnP**), se ha constituido en un estándard para los fabricantes de motherboards.
Buses como el VESA local bus para transferencias rápidas en aplicaciones gráficas, por estar directamente conectado al bus local de la UCP dependía del mismo. En cambio el PCI es independiente, es estándard.
Es un bus sincrónico de 66 (ó 33) Mhz: su línea "reloj" durante los 15 nseg que dura cada uno de los 66 millones de pulsos (ciclos) que ocurren por segundo, permanece durante 7,5 nseg en 0 volts, y en los otros 7,5 nseg. en 3,3 volts, marcando instantes en que se pueden enviar datos, direcciones, o comandos (fig. 1.80.b).
Si se trabaja con 64 líneas de un PCI, se puede enviar juntos 8 bytes en cada ciclo, por lo que teóricamente se podría enviar 8 bytes 66 millones de veces por segundo, o sea 66x8 = 528 Mbytes/seg, que no es suficiente para usar el PCI como un bus para leer o escribir datos en memoria.
En promedio un bus PCI opera a 100 Mbytes/seg., pues entre otras cosas parte de los ciclos se usan para preparar las transferencias, sin que haya transmisión efectiva de datos.
Para comparar, el bus ISA es de 8,33 Mhz y tiene 16 líneas, lo que en teoría da 8,33x2 = 16,66 Mbytes/seg.; y el EISA de 8,33 Mhz y 32 líneas alcanzaría los 8,33x4 = 33,32 Mbytes/seg.
Dado que antes de transferir se requiere en cada lectura a través del PCI ocupar 2 ciclos preparatorios, en realidad debe estimarse una velocidad promedio de transferencia de datos bastante menor que la calculada, la cual aumenta a 266 Mbytes/seg cuando se transfieren en ráfaga ("burst") muchos bytes consecutivos.
A diferencia de otros buses, en el PCI un "burst" no tiene longitud fija: continua hasta que uno de los 2 dispositivos intervinientes indique fin de transferencia, o si otro de alta prioridad debe usar el bus.

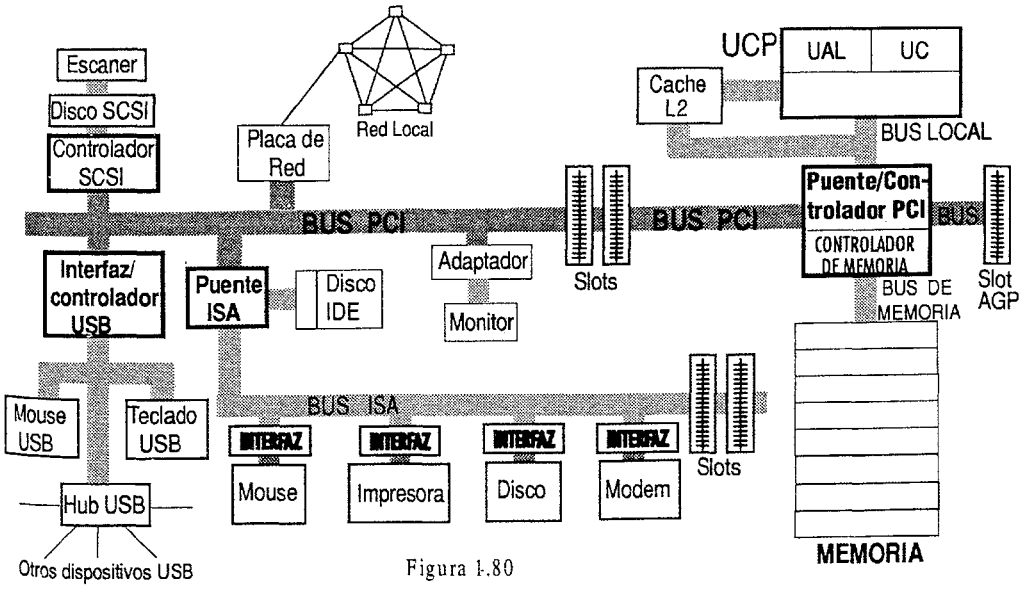


Figura 1.80

El bus PCI se comunica mediante el **puente PCI** o controlador PCI (fig. 1.80) con el bus local que va a la UCP, y con el bus de memoria que va a ésta.
En el chip que contiene al puente PCI también se encuentra el **árbitro del bus PCI**, y el controlador de memoria, que tranforma las órdenes hacia la memoria DRAM en una secuencia de señales para ella (Sección 1.4).

Dicho puente, conforme a su nombre, *permite la comunicación entre dos de los tres buses citados por vez.* Además contiene buffers inteligentes, para el re-envío de datos que temporalmente guarda según distintas necesidades. Así permiten que la UCP y el puente trabajen en paralelo de manera que se intercambien datos entre dos dispositivos del PCI mientras la UCP direcciona memoria. También detecta

si se realizan sucesivas L/E que no sean en ráfaga a posiciones consecutivas de memoria (por ej. a la memoria de vídeo), en cuyo caso junta en un buffer los datos llegados al puente en forma aislada, y los reenvía en ráfaga a máxima velocidad de transferencia, compatible con las necesidades de la UCP.
Al encenderse el equipo, el puente PCI puede configurar automáticamente ports e IRQ correspondientes a distintos periféricos, para lo cual rastrea dispositivos conectados al bus PCI, y luego asigna una única dirección base para sus ports, y un determinado nivel *n* de IRQn para interrumpir.

Al bus PCI (fig. 1.80) están conectados el **puente ISA** (el cual también conecta al PCI la electrónica IDE del rígido) y los adaptadores inteligentes para los buses USB, SCSI, otro PCI, así como para conexión a red. Pueden conectarse al PCI a través de zócalos de expansión interfaces de periféricos, los cuales por su velocidad no pueden conectarse al ISA. Se trata, pues, de un bus intermediario entre el bus local y otros buses. También puede existir en la motherboard conectados al PCI adaptadores de vídeo para manejar monitores. El bus PCI también puede dar apoyo en multiprocesamiento, siendo que a un bus PCI puede conectarse otro PCI (PCI a PCI) mediante otro puente.

De los dispositivos conectados al PCI incluidas las interfaces, sólo 2 por vez pueden comunicarse a través del mismo: el *iniciador* o bus master (**M**) -al cual un árbitro le otorgó el control del bus entre varios dispositivos que pujaron por dicho control para ser **M**- y el *target* (destinatario) o slave (**S**).
El arbitraje si bien se realiza sincronizado por el reloj del bus, *no requiere la utilización de ciclos extras*, dado que tiene la ventaja de que se efectúa paralelamente al uso del bus por otro **M**. Para tal fin, cada posible **M** (incluso la UCP) se comunica con el árbitro mediante dos líneas dedicadas (fig 1.80.a).

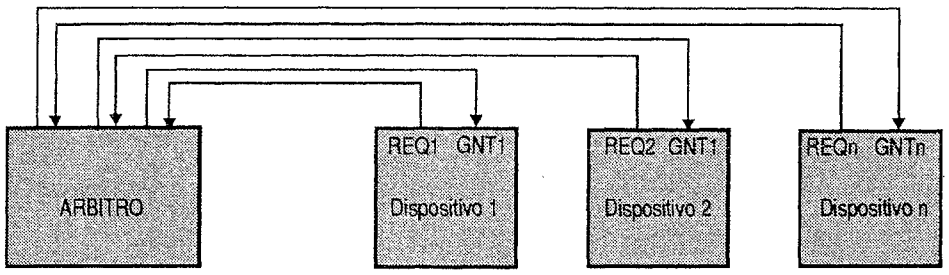


Figura 1.80.a

Una señal de requerimiento (REQ#) sale de cada posible **M** hacia el árbitro, y otra de otorgamiento (GNT#) va en sentido opuesto.
Se busca beneficiar a las transferencias largas (modo ráfaga), aunque el árbitro puede obligar al **M** a ceder el bus en el próximo ciclo si existe prioridad. Dicho **M** puede ganar el bus para una nueva transferencia si no llegaron otros requerimientos al árbitro.
Puede establecerse que el primer **M** en solicitar es el primero en ganar el bus, o una cesión cíclica, o prioridad (por ej, el que opera en modo ráfaga tenga prioridad sobre otro en modo simple).
Un bus PCI presenta 64 (ó 32) líneas A/D (adress/data) que en un ciclo pueden usarse para direccionar un dispositivo PCI, y en el siguiente(s) para transferir datos ahorrando espacio.

Las órdenes (tipo de transacción) se dan por códigos de 4 bits en 4 líneas designadas C/BE (Comando/Byte Enable). El código del comando se envía por estas 4 líneas en el mismo ciclo en que una dirección se pone en las líneas A/D. En los ciclos en que por las líneas A/D se envían datos, por las 4 líneas citadas mediante otra combinación binaria se indica cuáles de los 8 bytes que van por las 64 líneas A/D deben seleccionarse.
Suponiendo que un **M** gana el control del bus, y que el mismo se efectiviza más tarde en un ciclo 1 (fig. 1.80.b), dicho **M** activa (hasta el inicio de la última fase de la transacción) la línea Frame de transacción en curso, y envia una dirección de inicio (de memoria o de un port) a través de las líneas A/D, a la par que ordena una transacción mediante un código en las 4 líneas C/BE#. Dicha dirección llegará a todos los dispositivos del bus. El **S** que reconoce como una de sus direcciones esa dirección, la guardará junto con la orden. Ejemplos de las 16 transacciones que ordena un comando son:

- *lectura/escritura de un port* (E/S),
- *lectura/escritura de la memoria,*
- *secuencia IntA*: envío de la dirección del vector interrupción a la UCP (Unidad 3 de esta obra),
- *lectura/escritura de configuración:* cada dispositivo dispone de 256 bytes para configurarlo en la inicialización del sistema ("espacio de configuración") que otros dispositivos pueden leer activando la línea IDSEL Dicho espacio puede ser leído por un Sistema Operativo para determinar qué dispositivos están conectados, y así brindar el servicio "plug 'n play" (conectar y operar).

si se realizan sucesivas L/E que no sean en ráfaga a posiciones consecutivas de memoria (por ej. a la memoria de vídeo), en cuyo caso junta en un buffer los datos llegados al puente en forma aislada, y los reenvía en ráfaga a máxima velocidad de transferencia, compatible con las necesidades de la UCP.
Al encenderse el equipo, el puente PCI puede configurar automáticamente ports e IRQ correspondientes a distintos periféricos, para lo cual rastrea dispositivos conectados al bus PCI, y luego asigna una única dirección base para sus ports, y un determinado nivel *n* de IRQn para interrumpir.

Al bus PCI (fig. 1.80) están conectados el **puente ISA** (el cual también conecta al PCI la electrónica IDE del rígido) y los adaptadores inteligentes para los buses USB, SCSI, otro PCI, así como para conexión a red. Pueden conectarse al PCI a través de zócalos de expansión interfaces de periféricos, los cuales por su velocidad no pueden conectarse al ISA. Se trata, pues, de un bus intermediario entre el bus local y otros buses. También puede existir en la motherboard conectados al PCI adaptadores de vídeo para manejar monitores. El bus PCI también puede dar apoyo en multiprocesamiento, siendo que a un bus PCI puede conectarse otro PCI (PCI a PCI) mediante otro puente.

De los dispositivos conectados al PCI incluidas las interfaces, sólo 2 por vez pueden comunicarse a través del mismo: el *iniciador* o bus master (**M**) -al cual un árbitro le otorgó el control del bus entre varios dispositivos que pujaron por dicho control para ser **M**- y el *target* (destinatario) o slave (**S**).
El arbitraje si bien se realiza sincronizado por el reloj del bus, *no requiere la utilización de ciclos extras*, dado que tiene la ventaja de que se efectúa paralelamente al uso del bus por otro **M**. Para tal fin, cada posible **M** (incluso la UCP) se comunica con el árbitro mediante dos líneas dedicadas (fig 1.80.a).

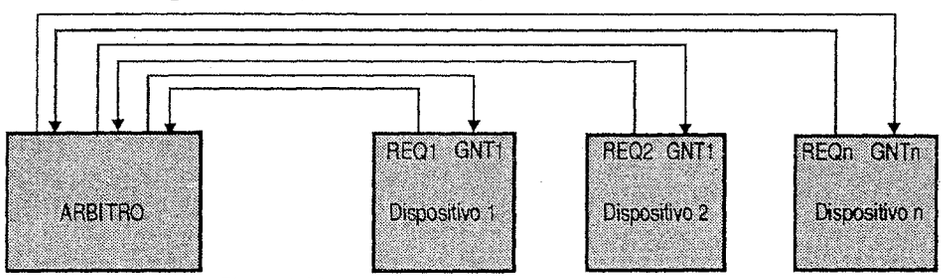


Figura 1.80.a

Una señal de requerimiento (REQ#) sale de cada posible **M** hacia el árbitro, y otra de otorgamiento (GNT#) va en sentido opuesto.
Se busca beneficiar a las transferencias largas (modo ráfaga), aunque el árbitro puede obligar al **M** a ceder el bus en el próximo ciclo si existe prioridad. Dicho **M** puede ganar el bus para una nueva transferencia si no llegaron otros requerimientos al árbitro.
Puede establecerse que el primer **M** en solicitar es el primero en ganar el bus, o una cesión cíclica, o prioridad (por ej, el que opera en modo ráfaga tenga prioridad sobre otro en modo simple).
Un bus PCI presenta 64 (ó 32) líneas A/D (adress/data) que en un ciclo pueden usarse para direccionar un dispositivo PCI, y en el siguiente(s) para transferir datos ahorrando espacio.

Las órdenes (tipo de transacción) se dan por códigos de 4 bits en 4 líneas designadas C/BE (Comando/Byte Enable). El código del comando se envía por estas 4 líneas en el mismo ciclo en que una dirección se pone en las líneas A/D. En los ciclos en que por las líneas A/D se envían datos, por las 4 líneas citadas mediante otra combinación binaria se indica cuáles de los 8 bytes que van por las 64 líneas A/D deben seleccionarse.
Suponiendo que un **M** gana el control del bus, y que el mismo se efectiviza más tarde en un ciclo 1 (fig. 1.80.b), dicho **M** activa (hasta el inicio de la última fase de la transacción) la línea Frame de transacción en curso, y envia una dirección de inicio (de memoria o de un port) a través de las líneas A/D, a la par que ordena una transacción mediante un código en las 4 líneas C/BE#. Dicha dirección llegará a todos los dispositivos del bus. El **S** que reconoce como una de sus direcciones esa dirección, la guardará junto con la orden. Ejemplos de las 16 transacciones que ordena un comando son:

- *lectura/escritura de un port* (E/S),
- *lectura/escritura de la memoria,*
- *secuencia IntA*: envío de la dirección del vector interrupción a la UCP (Unidad 3 de esta obra),
- *lectura/escritura de configuración:* cada dispositivo dispone de 256 bytes para configurarlo en la inicialización del sistema ("espacio de configuración") que otros dispositivos pueden leer activando la línea IDSEL Dicho espacio puede ser leído por un Sistema Operativo para determinar qué dispositivos están conectados, y así brindar el servicio "plug 'n play" (conectar y operar).

Si es una transacción de *escritura*, en el ciclo 2 (fig. 1.80.b), el **S** que en el ciclo 1 guardó la dirección enviada activará la línea Devsel (Dispositivo seleccionado), con los cual el **M** que direccionó se entera que **S** recibió la dirección, y **S** también activará Trdy (target preparado para recibir).
Las líneas se activan en 0 volts. El **M** también activará la línea Irdy (Iniciador operando) y enviará por las 64 (o 32) líneas A/D el dato a escribir, a la par que codificará en las 4 líneas C/BE cuáles de los 8 (ó 4) bytes que envía por A/D deben ser considerados.

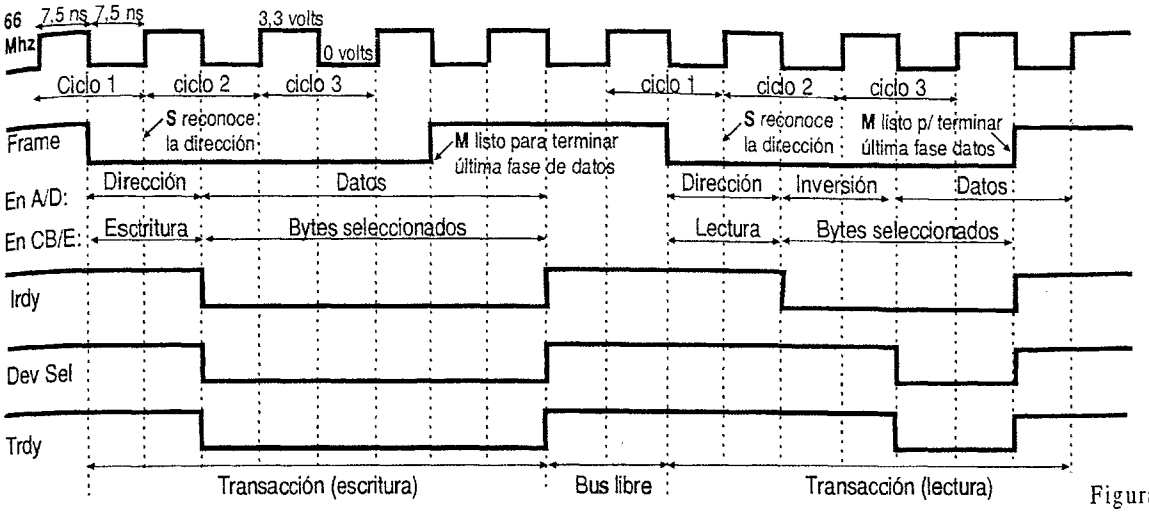


Figura 1.80.b

Al comenzar el ciclo 3 (flanco creciente del pulso correspondiente) el buffer del **S** (suponiendo que sea el puente PCI u otro chip) habrá tomado los datos de las líneas A/D sin esperar que se escriban en el destino final (como ser la memoria principal) de modo de no demorar la transferencia de los datos que **M** puede enviar en el ciclo 3. Para ello durante este ciclo **M** mantiene activadas a Frame e Irdy, a la par que por A/D envía los nuevos datos a escribir y por C/BE indica los bytes que deben ser considerados. El S también seguirá con Devsel y Trdy activadas, indicando que están preparados para tomar nuevos datos por A/D. De existir algún problema (buffers llenos en **M** o en **S**) se desactivarán las líneas Irdy ó Trdy. Entonces hasta que ambas no vuelvan a estar activas no se transmitirán datos durante uno o varios ciclos ("wait states").
El **M** se desentiende de la manera cómo el **S** o los bloques vinculados al mismo (como el controlador de memoria) van incrementando la dirección de escritura a partir de la dirección de inicio.
En el ciclo siguiente al que **M** terminó de enviar todos los datos desactiva Frame e Irdy indicando que deja el bus libre para el **M** que ganó el control del bus, con lo cual el S desactiva Devsel y Trdy Las líneas de control citadas, son compartidas por todos los dispositivos del bus.
Si la transacción es una *lectura* el ciclo 1 será igual (salvo la información que se envía por las líneas A/D, pero recién en el ciclo 3 ocurrirá lo que sucedió en el ciclo 2 de la escritura, dado que **S** no puede enviar datos hacia **M** por las líneas A/D (aunque **M** active Irdy), siendo que S está diseñado para que al inicio de una lectura deba perderse un pulso para invertir el sentido en que transmiten estas líneas. Por lo tanto la velocidad en una lectura resulta menor que en una escritura.

¿Cómo funciona la conexión Interfaz-Bus SCSI para periféricos?

El SCSI (siglas de Small Computers System Interface) no es un bus como el bus local, el PCI, el ISA u otros que consisten en un conjunto de líneas conductoras de corta longitud, fijos en una placa. Se trata de un bus exterior al gabinete de un computador, *uno de cuyos extremos se conecta al bus PCI* mediante un adaptador-controlador SCSI. A éste se pueden encadenar por conexionado "daisy channel" (fig.1.81) hasta 7 periféricos, prolongándose con cada periférico agregado la longitud del bus (máxima 6 mts.) Cada conjunto de cables que conecta un dispositivo con el siguiente es un tramo del bus SCSI que se conecta con el tramo anterior. La salida del último dispositivo debe terminar en resistencias o en un circuito según la longitud y el tipo de SCSI.
Este bus surgió en los 80 para conectar periféricos en Macintosh sin usar zócalos. Es apto para conectar con buena performance unidades de CDs, discos duros, cintas, scanners, RAID, y multimedia.

El original SCSI-1 supone 8 líneas para datos y velocidad de transmisión dentro del bus de hasta 5 Mbytes/seg. En los 90 apareció la SCSI-2 para 16 ó 32 líneas, y con hasta 20 ó 40 Mbytes/seg.

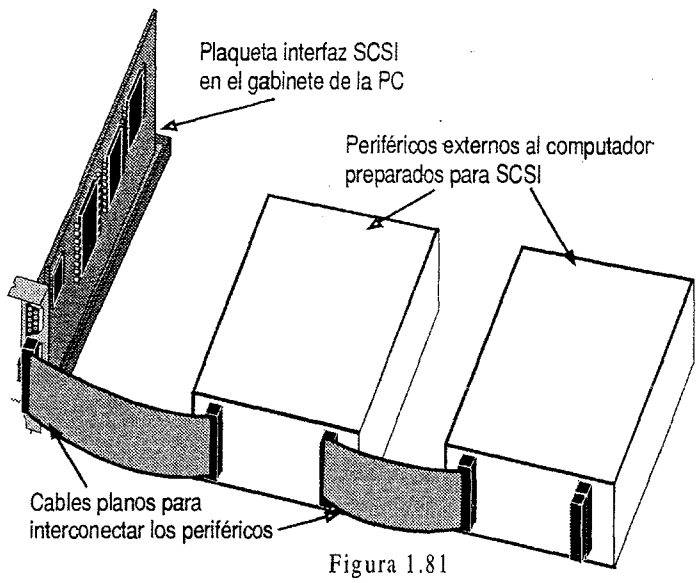


Figura 1.81

Normalmente el bus SCSI funciona en forma totalmente asincrónica, pudiendo hacerlo sincrónicamente durante la fase de transferencia de datos. Esto debe acordarse entre los dos dispositivos que se han comunicado. Se considera al adaptador SCSI a PCI como una unidad SCSI, por lo que quedan disponibles sólo 7 unidades para conectarse al bus SCSI. Cada unidad tiene una dirección (**ID**) entre 0 y 7 configurable mediante jumpers.
Es factible conectar a un bus SCSI por ejemplo 4 adaptadores SCSI, los cuales pueden conectarse a 4 computadoras, que así comparten dicho bus, y pueden intercambiar información entre sí.
Si bien lo común es que la información se transfiera entre memoria principal y algún periférico vinculado al bus SCSI (a través del bus PCI y del adaptador SCSI), *el bus SCSI permite intercambiar información entre las unidades conectadas al mismo*. Así pueden comunicarse un disco rígido con una unidad de cinta, sin pasar por la memoria principal y *sin la intervención de la UCP* ejecutando programas.

Cualquier unidad SCSI puede tomar el control del bus SCSI actuando como iniciador (**M**) y enviar por el bus la dirección de la unidad "target" (**S**) con la que quiere comunicarse (para enviarle comandos y transferir datos La línea de control BSY del bus indica si éste está ocupado, siendo que sólo se pueden enviar datos dos dispositivos por vez. Las direcciones, comandos y datos viajan por las mismas 8 líneas en distintos momentos.
El bus SCSI permite por ejemplo que si el adaptador SCSI/PCI (**M**) le envía a la unidad SCSI de disco rígido (**S**) la orden de acceder a un sector para leerlo, mientras el cabezal se posiciona **S** libera el bus, a fin de que mientras dura el tiempo de acceso al sector, el bus pueda ser usado por otras unidades. Cuando la unidad de disco puede transmitir la información pedida, pide seleccionar el adaptador y se la envía. El bus SCSI presenta 9 líneas de control:

BSY (Busy): indica bus ocupado.
SEL (Select): usada por un **M** para seleccionar un destinatario **S**, o para que éste seleccione al que fue su **M** para reconectarse a él en una fase de reselección (en cuyo caso además debe activarse la línea I/O)
I/O (Input/Output): indica en que sentido se transfiere los datos.
C/D (Control/Data): para indicar si por las 8 líneas de datos pasan códigos de control (órdenes, status , o mensaje) o datos.
REQ (Request): el **S** seleccionado solicita que se transfieran datos de **S** hacia M o en sentido contrario.
ACK (Acknowledge): el **M** acepta que datos sean enviados de **S** a **M** o en sentido contrario.
MSG (Message): generado por un **S** para indicar que la información que se transmite es un mensaje
ATN (Attention): cuando un **M** tiene un mensaje para su **S**.
RESET: resetea todos los dispositivos del bus, que pasa a la fase de bus libre.

La actividad del bus se da en una secuencia de hasta 8 fases, dado que al arbitraje puede seguir una fase de selección o reselección. Luego se pueden repetir *n* veces las ultimas 4 fases agrupables en una:

1. *Bus libre*: cuando no hay ningún dispositivo usando el bus. Se detecta si BSY y SEL no están activas
2. *Arbitraje descentralizado*: si uno o más de los 8 dispositivos quieren ganar el control del bus y ser M activan BSY y colocan en las 8 líneas de datos su ID (cada una activada sola es un ID distinto). Cada uno lee estas líneas para ver si hay un ID más alto; de ser así deja de activar BSY. El ganador activa SEL.
3. *Selección*: el **M** selecciona su **S** activando 2 de las líneas de datos: la de su ID y la del ID de S. El **S** debe activar BSY al detectar su ID (pasado un lapso el bus es libre), indicando que la operación puede hacerse.

4. *Reselección* (opcional): permite que un **S** lento libera el bus (ejemplo anterior del acceso a la unidad de disco), y cuando termina la parte lenta de la operación ordenada se vuelva a conectar con su iniciador **M**.
5. *Ordenes*: el **M** transmite al **S** por las líneas de datos una o varias órdenes encadenadas. Mediante REQ seguido de ACK se van transmitiendo byte a byte las órdenes a ejecutar.
6. *Transferencia*: desde una unidad a otra según sea, activándose REQ seguido de ACK con cada byte transmitido por las líneas de datos. Las líneas S/D y BSY deben estar activas.
7. *De estado*: Para que el S solicite al **M** enviar información acerca del desarrollo de la transferencia.
8. *Mensaje*: Para que **S** solicite enviar a **M** uno o más mensajes, como ser de desconexión o de orden cumplida, con lo cual se vuelve a la fase de bus libre.

¿ Cómo funciona el Universal Serie Bus (USB) ?

El número de zócalos en el bus ISA o PCI para insertar tarjetas interfaces de periféricos que se quiere agregar es limitado. Si bien con la tecnología PnP se evita configurar mediante microllaves o puentes el valor de *n* de un IRQn, se requiere igualmente para tal fin abrir el gabinete.
Por otra parte, el ISA o el PCI por ser buses de transmisión paralelo necesitan protección por las posibles interferencias entre sus líneas, tienen grandes limitaciones en su longitud, ocupan mucho lugar en la "mother" y requieren conectores grandes y relativamente costosos. Ello se justifica para el envío de datos desde o hacia el procesador y la memoria, cuando son necesarias altas velocidades de transmisión Pero en una PC para el envío de datos desde periféricos no muy rápidos o lentos hacia sus ports o en sentido inverso, técnicamente no se justifica el uso de dichos buses. El ISA ya obsoleto, subsiste en razón del gran número de usuarios que tienen periféricos y tarjetas adquiridos para ese bus.
En cambio, un bus serial usa sólo 2 líneas para enviar datos, direcciones y control, y otras 2 con +5 y 0 volts para los periféricos. Así resulta pequeño, económico y apto para computadores pequeños. Asimismo, a medida que surgen nuevos dispositivos multimedia hacen falta más ports por cada uno de ellos, y hay que resolver conflictos resultantes de compartir líneas IRQ. Como se verá un bus serie permite compartir mejor y disminuir el número de puertos y líneas IRQ requeridas.

Para conectar periféricos de baja velocidad (teclado, mouse, scanner, joysticks, parlantes, cámaras, teléfonos digitales, etc.) y de velocidad media (modems, unidades de CD, impresoras) en los años 90 las principales empresas de hardware y software acordaron crear un bus estándar denominado **USB** (Universal Serial Bus) que los fabricantes de motherboards incluyeron desde entonces en ellas, y que entre otras cosas permite:

- Conectar dispositivos mientras el equipo está funcionando, sin usar puenteadores (jumps) ni microllaves para configurarlos, sin reconfigurar el sistema, y sin tener que abrir el gabinete.
- Instalar hasta 127 dispositivos en un solo equipo, mediante cables de 4 conductores, con conectores que evitan errores de conexionado.
- Facilitar la conexión de dispositivos que operen en tiempo real (teléfono, audio).

A diferencia del SCSI, el USB no permite transferir directamente datos entre periféricos conectados a él.
Cuando se conecta un nuevo dispositivo al USB con el computador funcionando, el adaptador USB/PCI (figs. 1.80 y 1.82.a) detecta el evento y se produce una interrupción que llama al sistema operativo. La subrutina llamada lee en el dispositivo insertado sus características, tipo de dispositivo y la velocidad de transmisión que utiliza. Si es compatible con el bus USB le asigna una dirección entre 1 y 127, y ordena escribir ésta en los registros de configuración del dispositivo. Los dispositivos sin configurar tienen escrito por defecto dirección cero para poderlos direccionar la primera vez que se conectan.

Se distinguen dos tipos de software ejecutados por la UCP: el "Client Software" correspondiente a un dispositivo, y el USB System Software, que es software de un sistema operativo que soporta al USB.
Para el USB existen dos clases de dispositivos: las *funciones* (periféricos), y los *hubs* (concentradores).

El bus "raíz" del USB inserto en la "mother" está conectado en uno de sus extremos (figs. 1.80 y 1.82.a) al **Controlador-Adaptador** USB/PCI (**CAUSB**) *inteligente*, vinculado al bus PCI y que forma parte de un chip de la "mother". Este adaptador, que tiene software asociado, permite regular velocidades de transferencia hasta 1,5 Mbytes/seg. para dispositivos lentos, hasta 12 Mbytes/seg para veloces, y hasta 60 Mbytes para más veloces.

Este bus "raíz" en su otro extremo tiene por lo menos un conector USB para enchufar un dispositivo de E/S, o un cable de expansión, o un hub USB, conforme a una topología de conexionado tipo árbol irregular, con su raíz-tronco en la "mother". Usando hubs se conectan hasta 127 dispositivos.

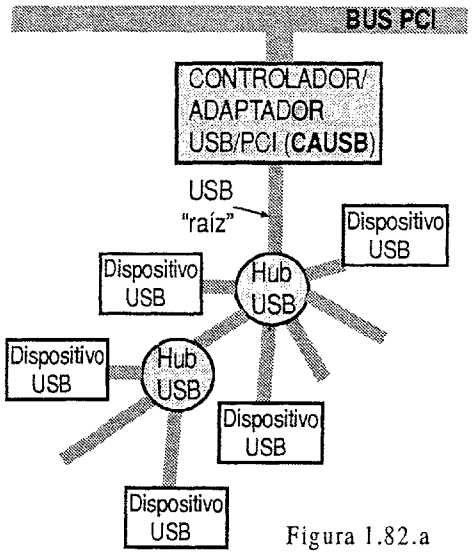


Figura 1.82.a

Un **hub** por un lado se conecta al bus "raíz" o a otro hub, y por otro lado presenta varios conectores hembra, para concentrar el conexionado de cables USB que se dirigen hacia él como centro de una conexión "estrella" (al igual que los rayos de una rueda que van hacia su soporte central), siendo que en el otro extremo de cada uno de esos cables hay un dispositivo, u otro hub. Así se logra ampliar el número de dispositivos que se pueden conectar al USB, los cuales funcionan paralelamente y en forma independiente.

El USB *no tiene líneas para direcciones o control.* Como en una red, por las dos líneas de información se transmiten uno tras otro bits de control, de direcciones y de datos, formando **paquetes** de bits. Se usan dos líneas para rechazar el ruido. Cada bit viaja codificado en **NRZI** (fig 1.82.b).
El **CAUSB** decide, auxiliado por el software, cuál dispositivo puede enviar o recibir datos a través del conexionado. Para ello envía, cuando el bus no es utilizado, un paquete de bits que llega a todos los dispositivos, en forma directa o a través de hubs que lo re-envían, para que el dispositivo elegido detecte que debe enviar o recibir datos. O sea que el **CAUSB inicia todas las transferencias**, para lo cual accede a la lista de transacciones preparadas por un software, y **las traduce en paquetes** que envía por el bus.

Los hubs complementan este proceso, dado que cada uno tiene incorporada inteligencia para:
- Detectar, en sus conectores (ports)[1], la conexión/desconexión de dispositivos y configurarlos.
- Controlar transferencias de datos y órdenes vinculadas a los dispositivos conectados a él.
- Proporcionar tensión de alimentación a dispositivos vinculados al mismo.
- En el caso de un Hub USB 2.0, intercambiar información a alta velocidad con el adaptador USB/PCI o con otro hub, según las normas USB 2.0 que permiten transferir a mayores velocidades

Se requiere que los dispositivos puedan disponer de un USB Peripheral Controller, y que los hubs contengan un USB Hub Controller, que contienen las siguientes funciones circuitales:
- *Transceivers*: para conformar la información binaria que viaja por el conexionado.
- *Serial Interface Engine* (SIE) convierte en serie la información y transcodifica la información binaria nativa del dispositivo en codificación NRZI que viaja por el bus (y viceversa), a la par que se encarga de concretar protocolos, detectar errores, y de secuenciar los paquetes de información.
- *Buffers* tipo FIFO, para recibir y enviar datos, cuya transmisión puede ser masiva o isócrona.
- *Functiones de monitoreo* del estado del controlador y los buffers.

Tanto en memoria principal como en los dispositivos se requieren buffers para transacciones pendientes.

Los movimientos de datos que ocurren a través del USB (u otros buses) se originan en el software, cuando desde un programa se necesita transferir datos de memoria a un dispositivo periférico o en sentido contrario. Para tal fin se debe pasar a ejecutar un "Client software" (**CSW**) que "ve" a cada dispositivo conectado al USB como una entidad lógica identificable por un número, denominada "*función*". Los sistemas operativos contienen software para manejar el USB: el USB "System Software" (**SSW**), para el cual una "función" es un "USB logical device". El CSW se comunica con el SSW -vía su USB Driver interface (**USBD**)- para mover datos entre memoria y una "función", generando una solicitud denominada I/O request packet (**IRP**).
Una vez que dicho movimiento se ha realizado, el CSW recibe una notificación del software driver asociado al **CAUSB**, a fin de que puede acceder por ejemplo a los datos transferidos a memoria. Este driver convierte cada IRP en transacciones que debe concretar el **CAUSB** mediante paquetes, y lleva una lista de transacciones a realizar así como de su estado de progreso (dada por el **CAUSB**).
En el USB cada *transferencia* de datos entre memoria principal y un dispositivo (pasando por el **CAUSB**) consta de una o más *transacciones*. Cada transacción puede realizarse dentro del lapso de una unidad de tiempo de 1 mseg denominada *frame*, mediante el envío de uno o varios paquetes.

[1] Acá el sentido que se da a "port" no es el de un registro (como en la interfaz de un periférico), sino el de un punto de conexionado.

Se dan 4 tipos de transferencias, solicitadas mediante el paquete "token" (indicación) correspondiente:

Isocrona: se transmite una secuencia de paquetes de tamaño fijo, a intervalos regulares que dependen de la aplicación en tiempo real que se trate. Así se garantiza velocidad de transferencia constante, como se requiere en telecomunicaciones (por ejemplo, para vídeo, micrófonos o telefonía digitalizados).

- *Masiva* (bulk): las transmisiones de los paquetes no son periódicas, pero se efectúan a altas velocidades, a fin de enviar rápidamente importantes cantidades de datos (caso de impresoras, scanners y transmisión de imágenes corrientes). Estas transferencias pueden realizarse entre dos isocronas.
- *Interrupt data*: dado que el USB *no reconoce interrupciones*, como la usada para manejar el evento de apretar/soltar una tecla del teclado, si se conecta éste al USB se requiere que una subrutina del sistema operativo cada 50 mseg pregunte si sucedió tal evento a fin de que se transfiera el código de la(s) tecla(s).
- *De Control*: típicamente son transferencias entre el **CAUSB** y un dispositivo del bus, para leer en los registros de éste información necesaria para su configuración (setup).

Las características de un dispositivo que interesa determinar y registrar en su configuración automática son: identificación del fabricante, clase de dispositivo, y capacidad de manejo de potencia. Cuando se configura un dispositivo que se conecta al bus, el SSW debe determinar si es compatible con el USB, para lo cual examina el descriptor del dispositivo localizado en el mismo, a fin de determinar por ejemplo si su velocidad de transferencia es compatible con el bus. Luego debe verificar si el tiempo máximo para realizar una transacción no supera un milisegundo (frame).

El CSW se vincula con una función (dispositivo) considerándola compuesta de una hasta 16 subentidades lógicas, cada una denominada "*endpoint*", identificables con un número de 4 bits en los paquetes "token". Este número se encuentra en el campo ENDP que sigue al de la dirección de 7 bits del dispositivo (ADDR). Ambos números (fig. 1.xxx) son necesarios para identificar el "endpoint", y se asignan en el momento en que el dispositivo se conecta al USB.
Por ej. la unidad de disquete es de entrada/salida, puede ser vista por el CSW como dos endpoints del USB con los cuales se comunica a través de dos "*pipes*" (canal lógico): uno para entrada y otro para salida, dado que un endpoint sólo puede enviar o bien recibir datos, o sea es unidireccional.
Un endpoint presenta propiedades que definen el tipo de transferencia que puede hacer con el CSW, como ser:

- Frecuencia de acceso al bus y demoras máximas permitidas en el pasaje por el bus.
- Velocidad de transferencia requerida.
- Tamaño máximo de paquete que el endpoint puede recibir o transmitir.
- Tipo de transacción que realiza.
- Manejo de errores requerido.

Cualquier paquete que se envía (ver más abajo) debe comenzar con un campo SYNC de 8 bits para sincronismo seguido por otro, el PID (Packet Identification) de 8 bits, 4 de los cuales codifican entre otros:
SOF (Start Of Frame): comienzo de envío de paquetes.
ACK (Acknowledge): acuse de datos recibidos sin errores por el **CAUSB** o un dlspositivo.
NACK (Not Acknowledge): señal de un dispositivo al **CAUSB** que no pudo recibir o transmitir datos.
IN: enviar datos desde el dispositivo.
OUT: enviar datos hacia el dispositivo.
SETUP: configurar dispositivo.
DATA0: bloque de datos en orden par.
DATA1: bloque de datos en orden impar.
STALL: enviado por un dispositivo, indicando que no puede recibir o transmitir datos.

La mayoría de los paquetes termina en un campo **CRC** (Cyclic Redundance Code) que sirve para detectar errores en la transmisión de un paquete, en cuyo caso se retransmite el paquete.
Las transferencias isocronas no admiten retransmisiones por errores en los datos recibidos, dado que no hay tiempo para que el receptor emita paquetes ACK o NACK, por lo que debe decidir luego qué hacer.
Existen paquetes de indicación ("token"), paquetes de datos, y paquetes de diálogo ("handshaking").

El paquete SOF sirve *para marcar tiempos*. Es emitido cada mseg. por el **CAUSB** hacia todos los dispositivos, incluyendo los hubs, como si fuera un reloj sincronizador. Luego del mismo pueden

enviarse otros paquetes, dentro del mseg. así definido, entre un paquete SOF y el siguiente. Al PID le sigue un campo de 11 bits para indicar número de frame, al cual le sigue un CRC (5 bits). Dicho número interesa en especial a dispositivos isocronos. Un SOF no exige paquete de respuesta; sólo marca tiempos. Su formato es:

SYNC (8)	SOF (8)	Nro (11)	CRC (5)

Los paquetes "token" cuyo PID es IN, OUT o SETUP intervienen en una transferencia de Control para enviar datos a un endpoint de un dispositivo recién conectado. El endpoint que recibe un paquete SETUP, luego puede recibir datos y responder con un ACK si fueron correctos. Su formato es:

SYNC (8)	PID (8)	ADDR (7)	ENDP (4)	CRC (5)

Los paquetes de datos (siguen a paquetes "token") tienen DATA como PID, y a este campo le sigue otro con los bytes transmitidos (de 0 a 1023), y luego otro para CRC de 16 bits. La transmisión puede ser isocrona, masiva o interrupt data. En cualquiera de éstas una transferencia supone una o más transacciones DATA. Para las transferencias tipo Interrupt Data las normas USB no establecen formatos específicos, siendo 64 y 8 bytes la máxima cantidad de datos a transmitir por paquete en full-speed, y en low speed, respectivamente. En el momento de la configuración del dispositivo el SSW determina dicho máximo, el cual es mantenido.

SYNC (8)	DATA (8)	(1 a 1023 bytes de datos)	CRC

Los paquetes de diálogo ACK y NACK se transmiten sólo con el campo PID (además del SYNC):

SYNC (8)	ACK (8)

SYNC (8)	NACK (8)

Según se describió antes, el USB tiene dos líneas en lugar de una para enviar cada bit de información. En cada instante se sensa la diferencia de tensión entre esas dos líneas (tensión *diferencial*) a fin de compensar los ruidos electromagnéticos que pudieran generarse. Cuando ocurre un ruido éste afecta por igual a las dos líneas, o sea que ambas durante cortos lapsos pueden subir o bajar igual valor de tensión en relación a masa, pero estas variaciones no modifican el valor de la diferencia neta de tensión entre los dos conductores.
La información se envía en el código binario **NRZI** (Not Return Zero Inverted), que tiene la ventaja de que el ritmo fijo con que se envían por el bus unos y ceros (frecuencia reloj, o "clock") está inserto en la información, por lo que el receptor no necesita una tercer línea para identificar cuándo hay uno o cero.

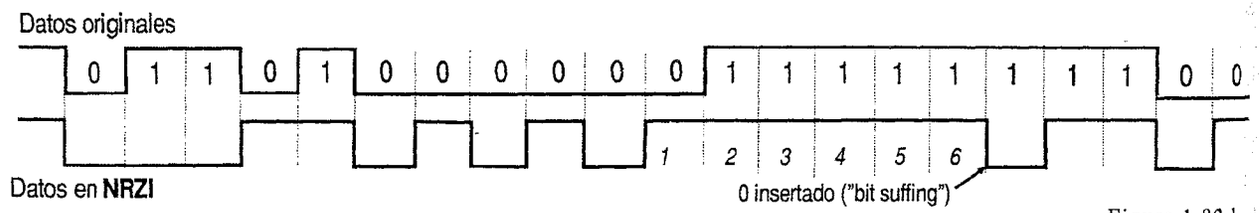


Figura 1.82.b

En NRZI (fig. 1.82.b) los bits de la información binaria que se quiere codificar en NRZI se van generando igualmente espaciados en el tiempo ("bit time"), siendo que una sucesión de ceros hace que la señal NRZI vaya cambiando con cada cero, mientras que una sucesión de unos hace que la señal NRZI no cambie. Un cero debe insertarse luego de cada 6 unos consecutivos ("bit suffing") para forzar una transición de nivel. Este cero es reconocido y descartado por el receptor cuando pasa de NRZI al código binario originario.

USB 2.0

La descripción desarrollada corresponde a los buses USB que cumplen con las normas USB revisión 1.1. La norma 2.0 define USB con velocidades de transferencia de hasta 60 Mbytes/seg con lo cual pueden conectarse dispositivos de vídeo y de almacenamiento, entre otros. Se establecen 3 tipos de dispositivos:

- Low Speed (LS): hasta 10.000 Bytes/seg (teclado, mouse, joysticks, realidad virtual)
- Full Speed (FS): de 50.000 Bytes/seg a 1 Mbyte/seg (audio, micrófonos)
- High Speed (HS) de 3 a 50 Mbytes/seg (vídeo, almacenamiento)

USB 2.0 permite aprovechar dispositivos y hubs USB anteriores, pero para funcionar en 2.0 se requiere un **CAUSB** para 2.0 y hubs 2.0, amén de cables de conexionado para operar en **HS**.
Este genera un paquete SOF cada 125 µseg (*un microframe* = 1 µframe) que llega a los hubs, y éstos por cada 8 de éstos generan un SOF cada mseg (125 µseg x 8 = 1 mseg) que llega a *todos* los dispositivos FS o LS. Para éstos todo sucede en esencia como si trabajaran con la norma 1.1, siendo cada hub una interfaz entre el adaptador y los dispositivos. Un dispositivo **HS** conectado a un hub USB 2.0 es como si estuviese conectado al **CAUSB** 2.0.
Cuando el **CAUSB** 2.0 en un µframe determinado quiere ordenar una transacción (fig. 1.82.c), como ser que un dispositivo "X" low o full le envíe (IN) datos al **CAUSB** 2.0; luego de enviarle éste *al hub* un paquete SOF, le enviará dos más: uno denominado *Start Split Transaction Token* (SSPLIT), indicado a continuación, al cual seguirá un IN "token" como el anteriormente definido en USB 1.1.

PID de Split (8)	Direcc. hub 2.0 (7)	S/C (1) 0 Start 1 Com plete	Direcc. hub 1.1 (7) (si existe)	Speed 0 full 1 low (1)	E en Start complementa a S en full speed (1)	ET (2) End Point Tipe	CRC (5)

El hub reenviará el "token" IN al dispositivo "X", el cual durante los frames siguientes (de 1 mseg) enviará paquetes de datos al hub, que los guardará en su buffer, y le responderá con un paquete ACK al dispositivo "X" ("X" es un dispositivo full o low speed, pero no high speed).
A todo esto, el **CAUSB** 2.0 luego de haber enviado los paquetes SOF, SSPLIT e IN en **un** µframe, en el siguiente puede iniciar otra transacción con otro dispositivo "Y", enviando SOF, SSPLIT y otro token, sin esperar a que el dispositivo "X" envíe los datos solicitados en el µframe anterior. Del mismo modo el **CAUSB** 2.0 puede iniciar otras transacciones en cada µframe. Luego de varios µframes posteriores, el adaptador enviará al hub el *Complete Split Transaction Token* (CSPLIT) -token Split como el de SSPLIT pero con S/C=1 - seguido del mismo IN token enviado anteriormente, para requerir que el hub le envíe el resultado de la transacción completada con "X".
El hub por las dos líneas de alta velocidad que lo unen con el adaptador, enviará paquetes con los datos enviados por "X", que estaban en su buffer.

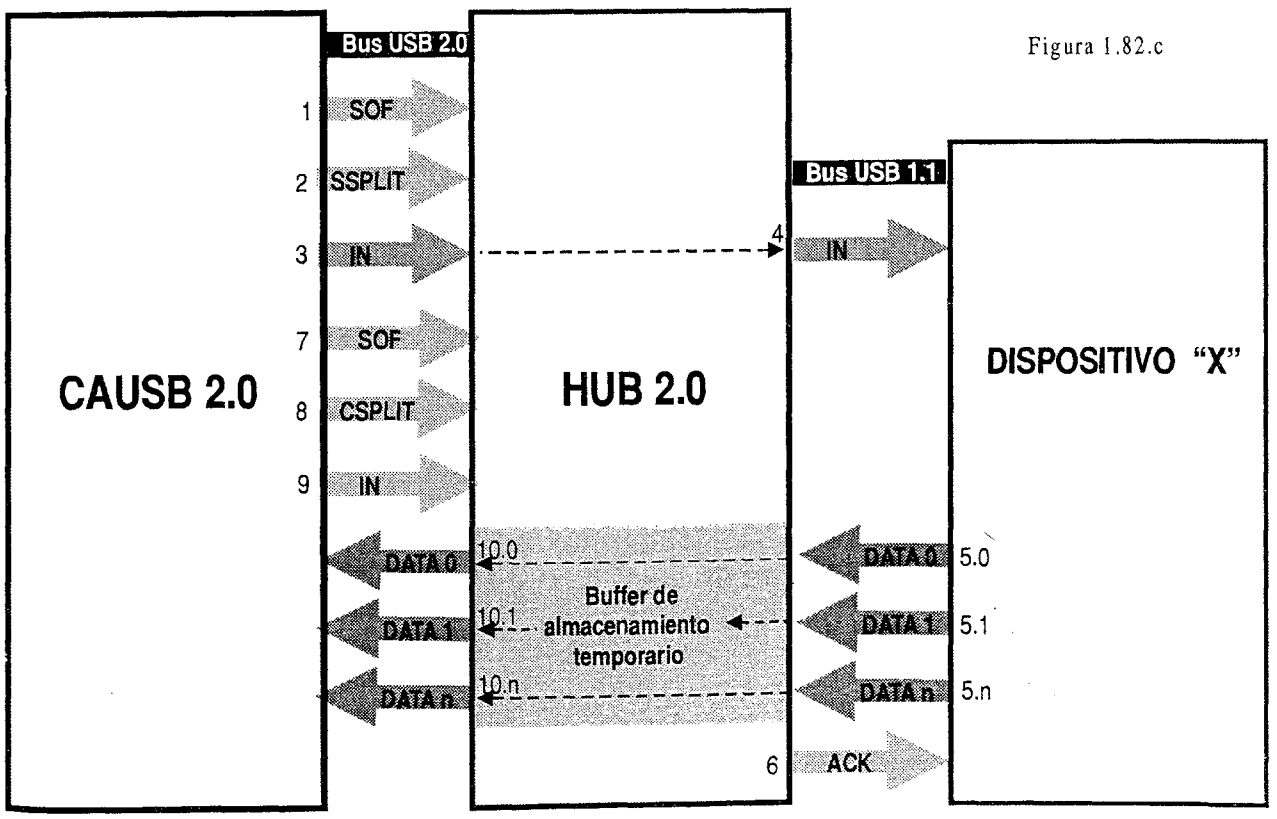
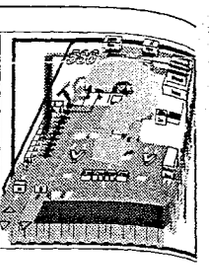


Figura 1.82.c

1.14 PARA ENTENDER LOS PENTIUM® I, II, III, 4, XEON® Y LOS PROCESADORES RISC



¿Qué es el "modelo de Von Neumann", y en qué medida los procesadores actuales lo cumplen ?

Si bien con importantes mejoras en la velocidad de prcocesamiento, *la mayoría de los procesadores actuales procesan en base al esquema de la figura 1.7,* denominado "**modelo de Von Neumann**"[1], que supone:

- Existe *una* sola UCP, que procesa *en secuencia* una instrucción tras otra.
  Ejecuta *una sola instrucción por vez* mediante una serie de pasos.
- Las instrucciones a ejecutar y los datos a procesar, codificados en binario, deben almacenarse en una rápida memoria *interna* (memoria principal) *antes* de realizar el procesamiento de los mismos.
- Existen *instrucciones de "salto"*, (figura 1.35) que ordenen a la UC discontinuar o no (según se alcance o no un resultado interno) la secuencia de instrucciones que viene ejecutando, para pasar a ejecutar otra secuencia, cuya primer instrucción se debe poder localizar.

En la figura 1.27 se indicaban cinco pasos o etapas básicas para ejecutar una instrucción.
Una de las primeras mejoras en velocidad para el modelo, fue efectuar el paso 5 mientras se espera el dato a operar (paso 3), quedando así 4 subprocesos típicos por los que pasa la ejecución de cada instrucción: los cuales progresan con cada pulso reloj, según se ha visto (figuras 1.30, que se repite en la 1.84).

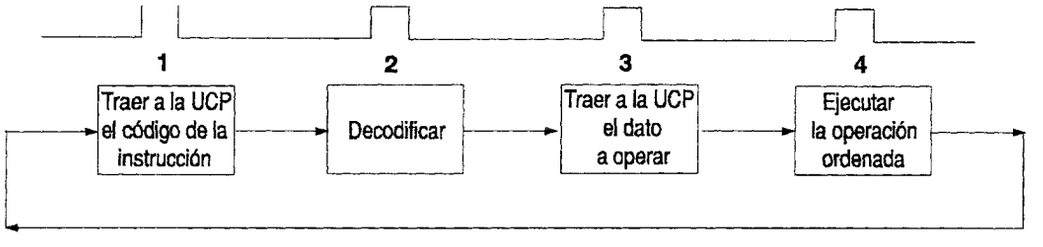


Figura 1.84

La figura 1.85 (izquierda) es similar a la 1.84, salvo la dirección (diagonal) en que avanza el proceso, para poder comparar el procesamiento según el modelo original de Von Neumann con otro más eficaz.
La ejecución de una instrucción progresa de un renglón al siguiente con cada pulso reloj, por lo cual los pulsos se han dibujado en sentido vertical.
Primero se termina de ejecutar totalmente una instrucción ($\mathbf{I_1}$), y luego la siguiente ($\mathbf{I_2}$), insumiendo la ejecución de ambas instrucciones 8 pulsos reloj.

¿Qué mejora en la velocidad presentan los procesadores actuales con "pipe line" ?

Hoy día, para aumentar la velocidad de procesamiento se ha mejorado el modelo original, dando lugar a otro que podemos denominar modelo de Von Neumann con "solapamiento de procesos", o con "**pipe line**" o "**segmentado**" –como quiera llamarse– que se pasa a exponer conceptualmente.
Intel en 1978 ya adoptó el "pipe line"[2] en el procesador 8086.

---

[1] En el presente también se denomina "escalar o "secuencial", generalizable a cualquier computador que opera en forma secuencial sobre los datos.

[2] Traducible "como por un tubo"), término originado en el proceso de fabricación en serie de autos, adoptado por Ford en 1919.

Esta mejora sustancial en la cantidad de instrucciones que se procesan por segundo se basa en las líneas de producción en serie de las fábricas de autos.. En ellas se divide el proceso de fabricación en una serie de subprocesos que se pueden realizar en forma independiente. En una cadena de este tipo, cuando se termina un subproceso de fabricación de una unidad (como ser el de pintura), la misma es desplazada al lugar donde se realiza siguiente subproceso de la cadena, a la par que otra unidad –también en proceso de fabricación– ocupa el lugar de la primera, para ser sometida al mismo subproceso realizado sobre la unidad anterior.
De esta forma *se realizan simultáneamente todos los subprocesos* independientes que requiere el armado de un auto, *pero aplicados a distintos autos* en curso de fabricación. Cuando se termina de producir un automóvil, los que fueron entrando a la cadena estarán parcialmente construidos.
Para plantear didácticamente la mejora habida apelaremos a un proceso conocido: el lavado de autos. Un lavadero simple tiene una persona a cargo de todas las etapas del lavado. Entra un auto por vez, y después de un tiempo, en el cual se sucedieron dichas etapas, el auto sale limpio. Luego entra el auto siguiente a lavar, y así de seguido.
Esto es semejante al procesamiento de cada instrucción en el modelo original de Von Neumann (figura 1.85 izquierda), siendo que la siguiente instrucción recién se puede comenzar a ejecutar luego de transcurrido el número de pulsos que requiere la ejecución de la anterior.

En un lavadero semiautomático en el cual el proceso se hace en 4 etapas de 5 minutos (entrada y pago del ticket → cepillado automático → limpieza de ruedas e interior → limpieza de vidrios y secado final[1]) se pueden ir procesando 4 autos simultáneamente. Cada auto tardaría 20 minutos en salir, pero puede salir **un** auto terminado cada 5 minutos. Esto es, aumenta la cantidad de autos lavados por hora, lo cual redunda en un menor precio de lavado, pero *cada cliente debe esperar las 4 etapas* (20 minutos).

Si al modelo de Von Neumann se le agrega "pipelining", la UCP mantiene su esquema básico, pero se le debe agregar circuitería adicional, del mismo modo que un lavadero automático requiere más personal, maquinaria y espacio interno para espera, en comparación con un lavadero manual unipersonal.
Así, se necesita un buffer para almacenar por orden de llegada los códigos de varias instrucciones (como ser 4 ó 5) pedidas a la memoria (o al caché), y otros buffers intermedios entre etapas. Estos sirven para que no se pierda el código de una instrucción en curso de ejecución, o datos y resultados relacionados con ella.[2]

La figura 1.85 (derecha) ilustra cómo un "pipe line" permite procesar simultáneamente diversas etapas de distintas instrucciones, completándose en cada etapa una parte de la ejecución de cada instrucción.
Se ha supuesto a los fines comparativos que el "pipe line" se realiza con las 4 etapas y tiempos (dados por pulsos reloj, designados t_1, t_2,. ...) de la figura 1.84 ó 1.30, y que todas las instrucciones requieren para su ejecución 4 pulsos). Entonces la UC ordenará:

En t_1, la primera de estas instrucciones que corresponde ejecutar (I_1), pasa del buffer al registro RI.
En t_2 el código de I_1 es decodificado, y al registro RI pasa a contener el código de I_2.
En t_3 se trae[3] el dato a operar para I_1, se decodifica I_2, y a RI llega desde el buffer el código de I_3.
En t_4 termina de ejecutarse I_1, se trae el dato a operar para I_2, se decodifica I_3, y llega a RI el código de I_4

Así de seguido se llevan a cabo en paralelo los procesos indicados en diagonal en la figura citada, cada uno independiente del otro. De esta forma, al cabo de 8 pulsos se habrán terminado de ejecutar 4 instrucciones, o sea, 4 veces más que con el modelo sin "pipe line" que aparece a la izquierda de la misma figura.
En general, si se tiene un "pipe line" de **n** etapas, teóricamente[4] se puede procesar hasta **n** veces más instrucciones por segundo que sin "pipe line", suponiendo que todas las instrucciones requieran **n** etapas. Esto implica también una situación ideal, con todas las instrucciones de igual complejidad, ejecutándose en 4 pulsos reloj. Así, *con cada pulso entra una instrucción al "pipe line", y se termina de ejecutar otra.* Resulta, que si bien *no se reduce el tiempo de ejecución de una instrucción* (cada una requiere 4 pulsos reloj), en cada pulso reloj se está ejecutando una etapa de 4 instrucciones distintas, lo cual permite ejecutar varias veces más rápido (4 en este caso) las instrucciones de un programa que en un modelo sin "pipe line".

---

[1] Suponiendo que esta última etapa sea la que dura 5 minutos y otras mucho menos, ella determina el ritmo de lavado.

[2] Del mismo modo, en el lavadero citado puede requerirse un lugar entre dos subprocesos, donde un automóvil que sale de un sub-proceso permanezca en él demorado, antes de pasar al siguiente, so pena de llevarse por delante el auto que aún está en este subproceso.

[3] Desde la memoria caché, si está en ella (sino habrá que pedirlo a la memoria principal) o desde un registro de la UCP.

[4] Un "pipe line" sin circuitos para "predicción de saltos condicionados" puede cortarse, si por ejemplo I_1 es una instrucción de salto condicionado (figura 1.35), que obligue que la siguiente que corresponda ejecutar no sea I_2; o si tiene lugar una interrupción por hardware. O demorarse un pulso reloj por que el dato a operar no está el caché y hay que pedirlo a memoria. Asimismo, la circuitería extra para el "pipe line" hace que cada instrucción se ejecute con pulsos de mayor duración en relación con un modelo sin "pipe line"

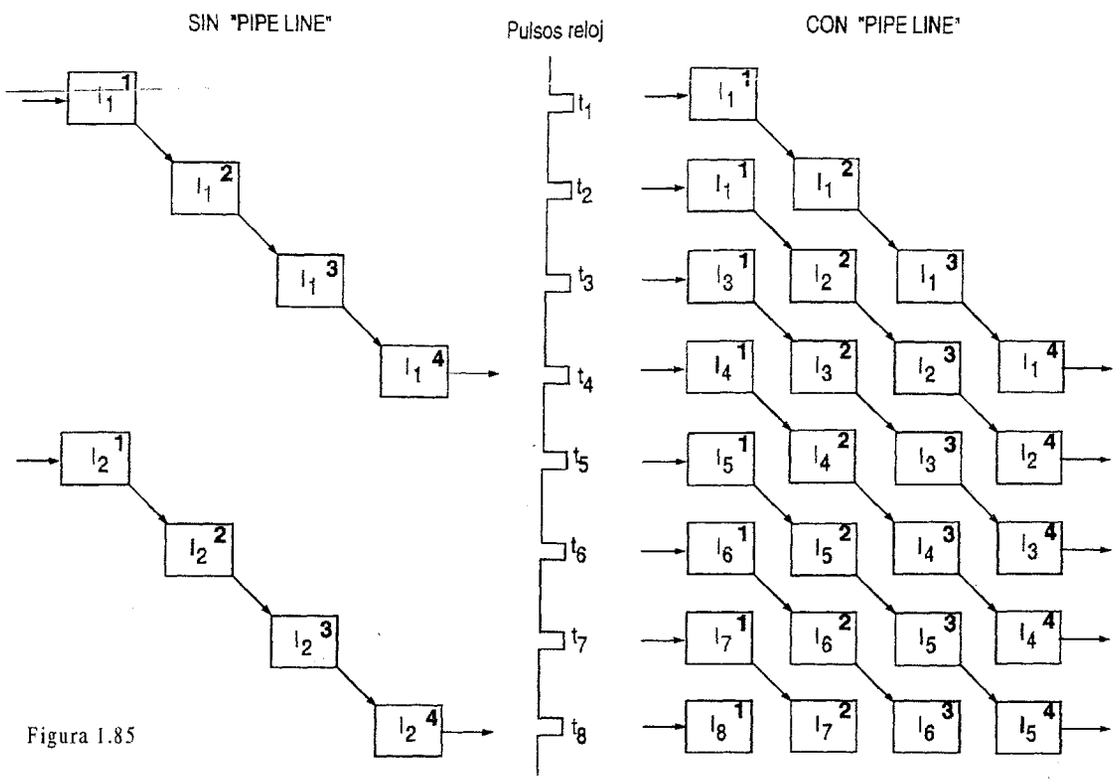


Figura 1.85

¿Qué es el multiprocesamiento o procesamiento en paralelo ?

Los requerimientos actuales de velocidad de procesamiento hicieron necesario el desarrollo de máquinas designadas "**no Von Neumann**", en el sentido de que existen **varios procesadores** operando juntos, **en paralelo.** Así se puden ejecutar ejecutar, en forma independiente, varias instrucciones de un mismo programa, o varios programas independientes, u operar con diversos datos a un mismo tiempo. Esto se conoce como "**multiprocesamiento**", contrapuesto al "uniprocesamiento" de Von Neumann. De existir varios lavaderos que trabajen "en paralelo" sería factible que varios autos salgan terminados simultáneamente y que además se ayuden mutuamente. En las arquitecturas "no Von Neumann", varias UCP pueden terminar de ejecutar juntas varias instrucciones por pulso reloj.

No debe confundirse *multiprocesamiento* con "**multiprogramación**" (*"multitasking"*, también traducible como *"multitarea"*), consistente en la ejecución alternada por una UCP de varios programas que están en memoria principal. Dada la velocidad de procesamiento, *puede parecerle al usuario como simultánea la ejecución de dos o más programas cuya ejecución en realidad se alterna muy rápidamente.*

¿Cómo funciona básicamente un microprocesador 486 ?

A continuación describiremos los principales bloques que están en el interior de un procesador 486 (figura 1.86), y las funciones que cumplen. Luego se concretará de qué modo los mismos participan, paso a paso, en la ejecución de la conocida secuencia de instrucciones I_1, I_2, I_3, I_4 desarrollada en las figuras 1.23 a 1.26. Este procesamiento, que fue realizado en dichas figuras conforme al modelo "original" de Von Neumann, fue acelerado ya en procesadores de Intel anteriores al 486, como se desarrolló en una respuesta anterior. En la figura 1.86 aparecen los siguientes sub-bloques y bloques:

- Los registros de direcciones (RDI) y de datos (RDA), pertenecientes a la *"Unidad de Interconexión con el Bus"* (**BIU** en inglés), encargada de la comunicación con el exterior a través de las 32 líneas de datos y 32 líneas de direcciones del bus, conectadas a las correspondientes patas del procesador ("local bus"). En relación con éste, los registros RDI y RDA cumplen las mismas funciones que en la figura 1.23, siendo que ahora instrucciones y datos leídos en memoria pasan al caché interno de 8 KB del procesador

- La *Unidad de caché* de 8 KB guarda las instrucciones y datos que seguramente serán requeridos próximamente. Por una parte, a través de un bus de 128 líneas, se pueden leer del caché 128/8 = 16 bytes que pasan a un buffer de la Unidad de pre-carga de instrucciones. Corresponden en promedio a unas 5 instrucciones a ejecutar, que así llegan juntas para entrar al "pipe line". Por otra, el caché puede ser leído para que se envíen 32 bits de datos a la UAL, o a un registro de la UCP ó 64 bits de datos a la Unidad de Punto Flotante (FPU en inglés). En una escritura van hacia el caché 32 ó 64 bits, respectivamente
- La *Unidad de Pre-carga* proporciona las direcciones de las próximas instrucciones a ejecutar, y guarda las mismas en orden en dos buffers de 16 bytes, para que luego cada una sea decodificada
- La *Unidad de Decodificación* realiza dos decodificaciones de cada instrucción, según se verá.
- La *Unidad de Control* (UC) mediante líneas que salen de ella (dibujadas en figuras 1.87 a 1.89), activa las operaciones que con cada pulso reloj deben realizar los distintos bloques de la UCP (U. de pre-carga, U. Decodificadora, UAL, UPF y UC), conforme lo establecen microcódigoa de la ROM de Control.
- La *Unidad de segmentación, paginación y protección de memoria*, conocida como "Unidad de manejo de memoria" (**MMU** en inglés) se encarga de proporcionar las direcciones físicas de memoria que utiliza un programa Para tal fin esta unidad convierte la referencia a la dirección del dato –que viene con la instrucción– en la correspondiente dirección física. Puesto que la memoria de una PC se divide en segmentos, y éstos –de ser necesario– pueden subdividirse en páginas (por ejemplo si se usa el sistema operativo Unix). Esta unidad se encarga de ello, así como de la protección contra escrituras no permitidas en zonas reservadas de memoria. Conviene aclarar que el nombre de esta unidad no tiene mucho que ver con la traducción castellana de "pipe line" como "segmentación", razón por la cual se prefirió usar dicha palabra inglesa.

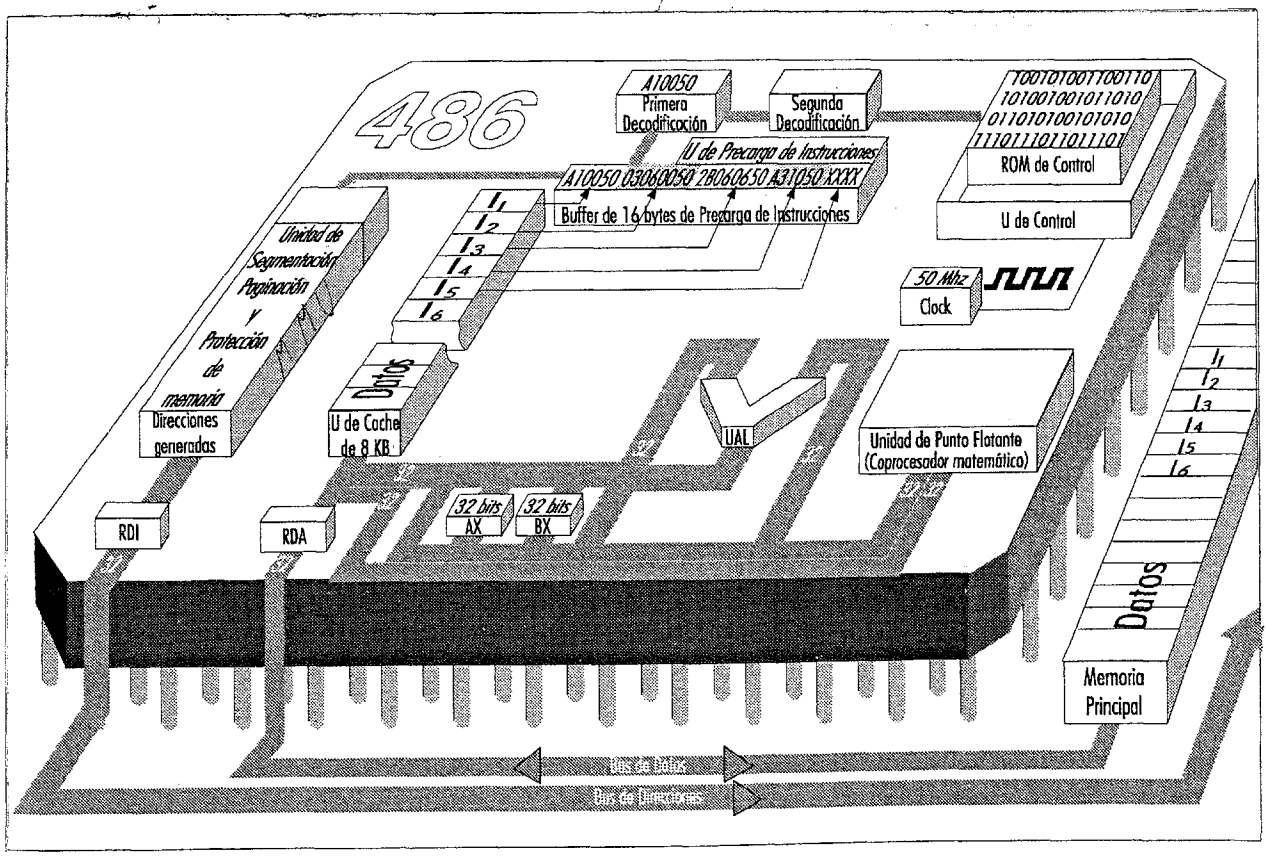


Figura 1.86

Todas estas unidades participan en el "pipe line" de instrucciones, que en el 486 consta de 5 etapas, que progresan con cada pulso reloj, al compás de sus millones de ciclos por segundo:

1. **Pre-carga** ("pre-fetch") consiste en la llegada de los códigos de las próximas instrucciones que entrarán al "pipe line" a dos buffers (de 16 bytes cada uno) de la Unidad de Pre-carga, , para formar una "cola". En la figura 1.86 se ha supuesto que a uno de estos buffers han llegado desde el caché 5 instrucciones (promedio de instrucciones que entran en los 16 bytes de este buffer) en forma simultánea. Los códigos de ellas son los mismos que hemos usado en la figura 1.15 para I_1, I_2, I_3, I_4, que en hexa son A10050, 30600500, B0600650, y A31050, respectivamente. La instrucción I_5, aparece con un código XXXX. De no haber estado estas instrucciones en el caché, primero se hubiera pedido I_1 a la memoria principal[1], y llegaría una copia de su código al buffer de pre-carga para que entre al "pipe line", y otra copia del mismo al caché. Inmediatamente llegarán luego al caché desde memoria, uno tras otro, los códigos de I_2, I_3, I_4[2], que pasarán a la cola del buffer. De esta forma, sólo se pierde tiempo en obtener del exterior a I_1

2. **Primera Decodificación**: a la Unidad de Decodificación llegan los primeros 3 bytes de cada instrucción, para separar –entre todos los bytes que forman su código de máquina– su código de operación, del número que hace referencia a la dirección del dato (Los códigos de operación pueden tener de 1 a 3 bytes). Así, en la figura 1.86, al primer decodificador llegan los bytes A10050H, que en este caso son todos los bytes de la instrucción I_1, identificándose **A1** como el código de operación, y **0050** como la referencia a la dirección del operando, número que pasará a la Unidad de segmentación y paginación, que formará la dirección del dato a operar, de modo que pueda ser leído del caché (si está en éste).

3. **Segunda Decodificación:** en la figura 1.87, el código de operación **A1** identificado en el paso anterior es ahora decodificado. Esto permite determinar la secuencia de microcódigo contenida en la ROM de Control. Merced a esta secuencia la UC generará las señales de control, que enviará por las líneas (insinuadas con flechas) que salen de ella, para que cada unidad que controla, ejecute una parte de la instrucción con cada pulsos reloj (como en la figuras 1.31 y 1.32). Si la instrucción es simple se ejecuta en un solo pulso. Al mismo tiempo que I_1 pasa por esta etapa del "pipe line", tres bytes (030600H) del código de I_2 (03060050H) entran a la etapa de primera codificación, siendo que 0050 –dirección traspuesta del dato– pasará a la U. de segmentación, para leer luego el dato del caché.

4. **Ejecución:** en la figura 1.88 el dato que debe transferirse al registro AX –como ordena I_1– hay que leerlo en la dirección (5000H) que la U. de Segmentación dejó en el registro RDI, la cual permite leer el dato a operar en el caché. Suponiendo que el dato está en la U de caché, el mismo llegará al registro RDA[3]. Paralelamente con la acción recién descripta para I_1, el código 0306 de I_2 pasa a la segunda decodificación, a la par que los bytes 2B0606 del código 2B060650 de I_3 van a la primera decodificación.

5. **Almacenamiento de resultados**: a esta etapa final del "pipe line" llega I_1, completándose su ejecución, para lo cual el dato (1020H) pasará al registro AX (paso incluido en la figura 1.88). Al mismo tiempo se tiene que: I_2 entra en la etapa de ejecución, obteniéndose del caché el dato 1020H, que pasa al RDA. Este dato se suma en la UAL con el contenido (1020H) de AX (figura 1.88), conforme ordena el código de dicha instrucción.
   El código 2B06 de la instrucción I_3 entra a la segunda decodificación, y los bytes A31050, o sea todos los que conforman el código A31050 de I_4 son sometidos a la primer decodificación.

En la figura 1.89 se ha incluido cómo progresa el "pipe line" con otro pulso reloj, a fin de terminar de ejecutar I_2, que pasa a la quinta etapa. En ésta, el resultado de la UAL (2040H) debe guardarse en AX, así como los "flags" SZVC que ella también genera, resultantes de la operación, en el registro de estado (no dibujado). Paralelamente, I_3, I_4 e I_5, pasan por las etapas 4, 3 y 2 del "pipe line".

---

1 Si como es corriente, existe un segundo nivel de caché exterior (por ejemplo de 256 KB), se buscaría I_1 primero en este caché rápido, y de no encontrarse en el mismo, se obtendría I_1 de memoria principal.

2 Cuando no hay un contenido en un caché, su controlador solicita a la memoria el mismo y los que están en las direcciones siguientes

3 Por razones didácticas se ha buscado continuidad con el modelo de Von Neumann (figuras 1.23 a 1.26), aunque la ejecución de I_1 pueda realizarse en un paso menos en el 486. Esta simplificación puede traer algunas inconsistencias en el paso 5.

1. **Pre-carga** ("pre-fetch") consiste en la llegada de los códigos de las próximas instrucciones que entrarán al "pipe line" a dos buffers (de 16 bytes cada uno) de la Unidad de Pre-carga, , para formar una "cola". –En la figura 1.86 se ha supuesto que a uno de estos buffers han llegado desde el caché 5 instrucciones (promedio de instrucciones que entran en los 16 bytes de este buffer) en forma simultánea. Los códigos de ellas son los mismos que hemos usado en la figura 1.15 para I_1, I_2, I_3, I_4, que en hexa son A10050, 30600500, B0600650, y A31050, respectivamente. La instrucción I_5, aparece con un código XXXX. De no haber estado estas instrucciones en el caché, primero se hubiera pedido I_1 a la memoria principal[1], y llegaría una copia de su código al buffer de pre-carga para que entre al "pipe line", y otra copia del mismo al caché. Inmediatamente llegarán luego al caché desde memoria, uno tras otro, los códigos de I_2, I_3, I_4[2], que pasarán a la cola del buffer. De esta forma, sólo se pierde tiempo en obtener del exterior a I_1

2. **Primera Decodificación**: a la Unidad de Decodificación llegan los primeros 3 bytes de cada instrucción, para separar –entre todos los bytes que forman su código de máquina– su código de operación, del número que hace referencia a la dirección del dato (Los códigos de operación pueden tener de 1 a 3 bytes). Así, en la figura 1.86, al primer decodificador llegan los bytes A10050H, que en este caso son todos los bytes de la instrucción I_1, identificándose **A1** como el código de operación, y **0050** como la referencia a la dirección del operando, número que pasará a la Unidad de segmentación y paginación, que formará la dirección del dato a operar, de modo que pueda ser leído del caché (si está en éste).

3. **Segunda Decodificación:** en la figura 1.87, el código de operación **A1** identificado en el paso anterior es ahora decodificado. Esto permite determinar la secuencia de microcódigo contenida en la ROM de Control. Merced a esta secuencia la UC generará las señales de control, que enviará por las líneas (insinuadas con flechas) que salen de ella, para que cada unidad que controla, ejecute una parte de la instrucción con cada pulsos reloj (como en la figuras 1.31 y 1.32). Si la instrucción es simple se ejecuta en un solo pulso. Al mismo tiempo que I_1 pasa por esta etapa del "pipe line", tres bytes (030600H) del código de I_2 (03060050H) entran a la etapa de primera codificación, siendo que 0050 –dirección traspuesta del dato– pasará a la U. de segmentación, para leer luego el dato del caché.

4. **Ejecución:** en la figura 1.88 el dato que debe transferirse al registro AX –como ordena I_1– hay que leerlo en la dirección (5000H) que la U. de Segmentación dejó en el registro RDI, la cual permite leer el dato a operar en el caché. Suponiendo que el dato está en la U de caché, el mismo llegará al registro RDA[3]. Paralelamente con la acción recién descripta para I_1, el código 0306 de I_2 pasa a la segunda decodificación, a la par que los bytes 2B0606 del código 2B060650 de I_3 van a la primera decodificación.

5. **Almacenamiento de resultados**: a esta etapa final del "pipe line" llega I_1, completándose su ejecución, para lo cual el dato (1020H) pasará al registro AX (paso incluido en la figura 1.88). Al mismo tiempo se tiene que: I_2 entra en la etapa de ejecución, obteniéndose del caché el dato 1020H, que pasa al RDA. Este dato se suma en la UAL con el contenido (1020H) de AX (figura 1.88), conforme ordena el código de dicha instrucción.
   El código 2B06 de la instrucción I_3 entra a la segunda decodificación, y los bytes A31050, o sea todos los que conforman el código A31050 de I_4 son sometidos a la primer decodificación.

En la figura 1.89 se ha incluido cómo progresa el "pipe line" con otro pulso reloj, a fin de terminar de ejecutar I_2, que pasa a la quinta etapa. En ésta, el resultado de la UAL (2040H) debe guardarse en AX, así como los "flags" SZVC que ella también genera, resultantes de la operación, en el registro de estado (no dibujado). Paralelamente, I_3, I_4 e I_5, pasan por las etapas 4, 3 y 2 del "pipe line".

---

1 Si como es corriente, existe un segundo nivel de caché exterior (por ejemplo de 256 KB), se buscaría I_1 primero en este caché rápido, y de no encontrarse en el mismo, se obtendría I_1 de memoria principal.

2 Cuando no hay un contenido en un caché, su controlador solicita a la memoria el mismo y los que están en las direcciones siguientes

3 Por razones didácticas se ha buscado continuidad con el modelo de Von Neumann (figuras 1.23 a 1.26), aunque la ejecución de I_1 pueda realizarse en un paso menos en el 486. Esta simplificación puede traer algunas inconsistencias en el paso 5.

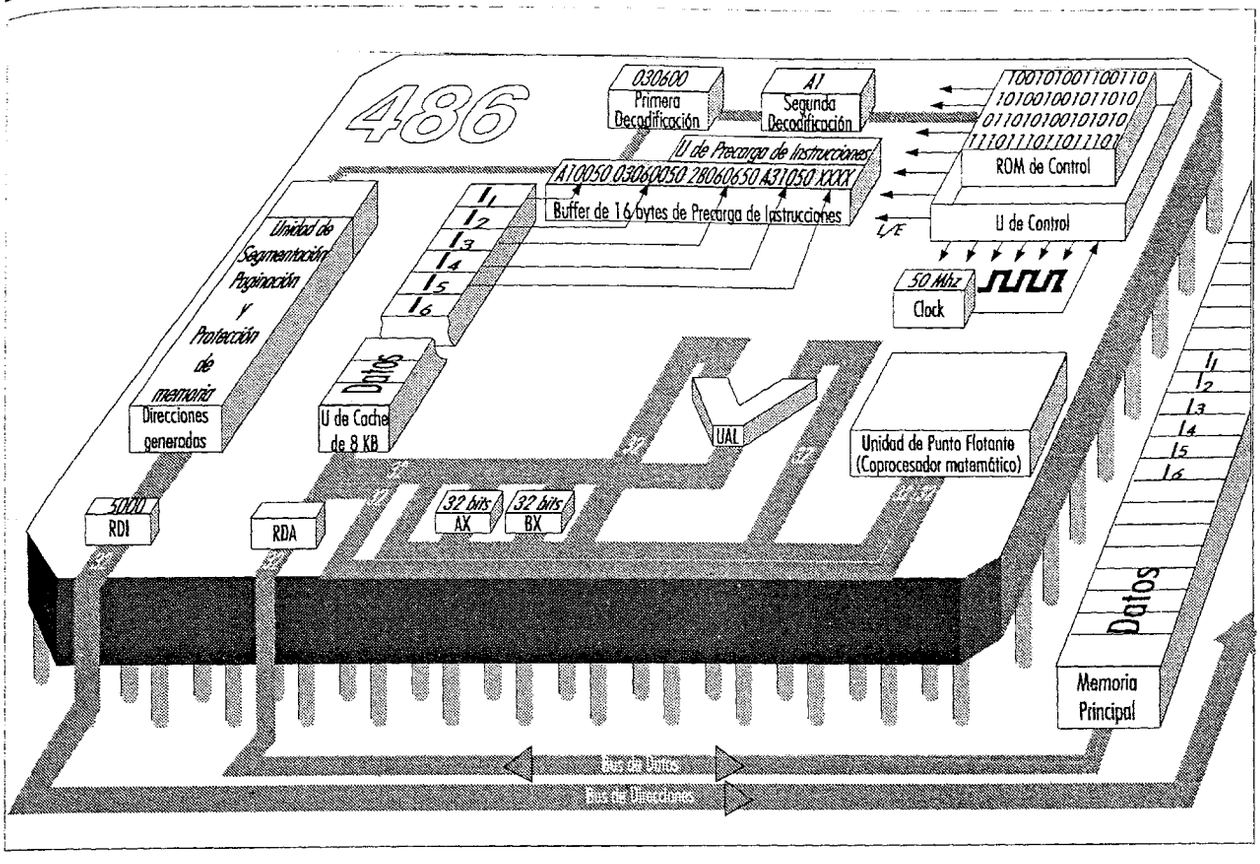


Figura 1.87

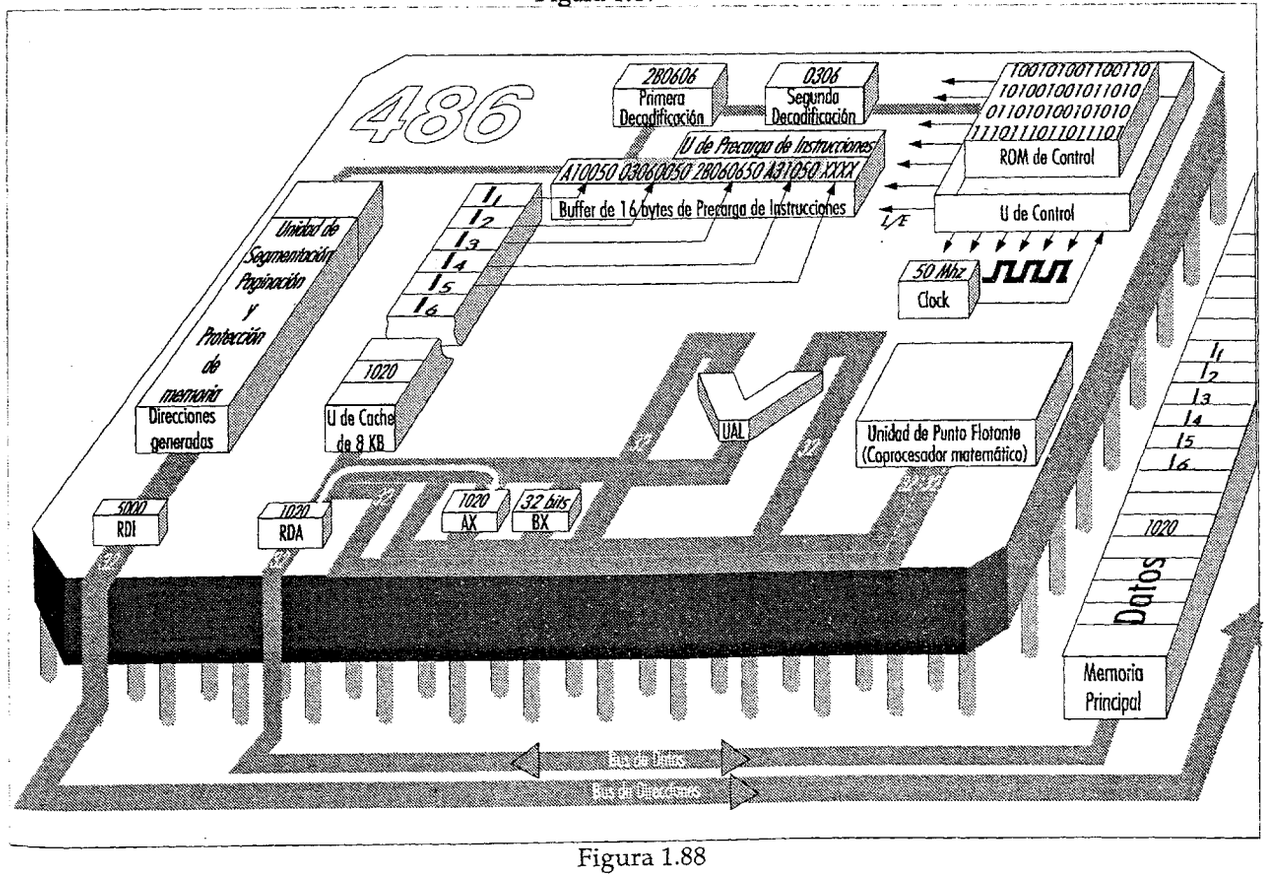


Figura 1.88

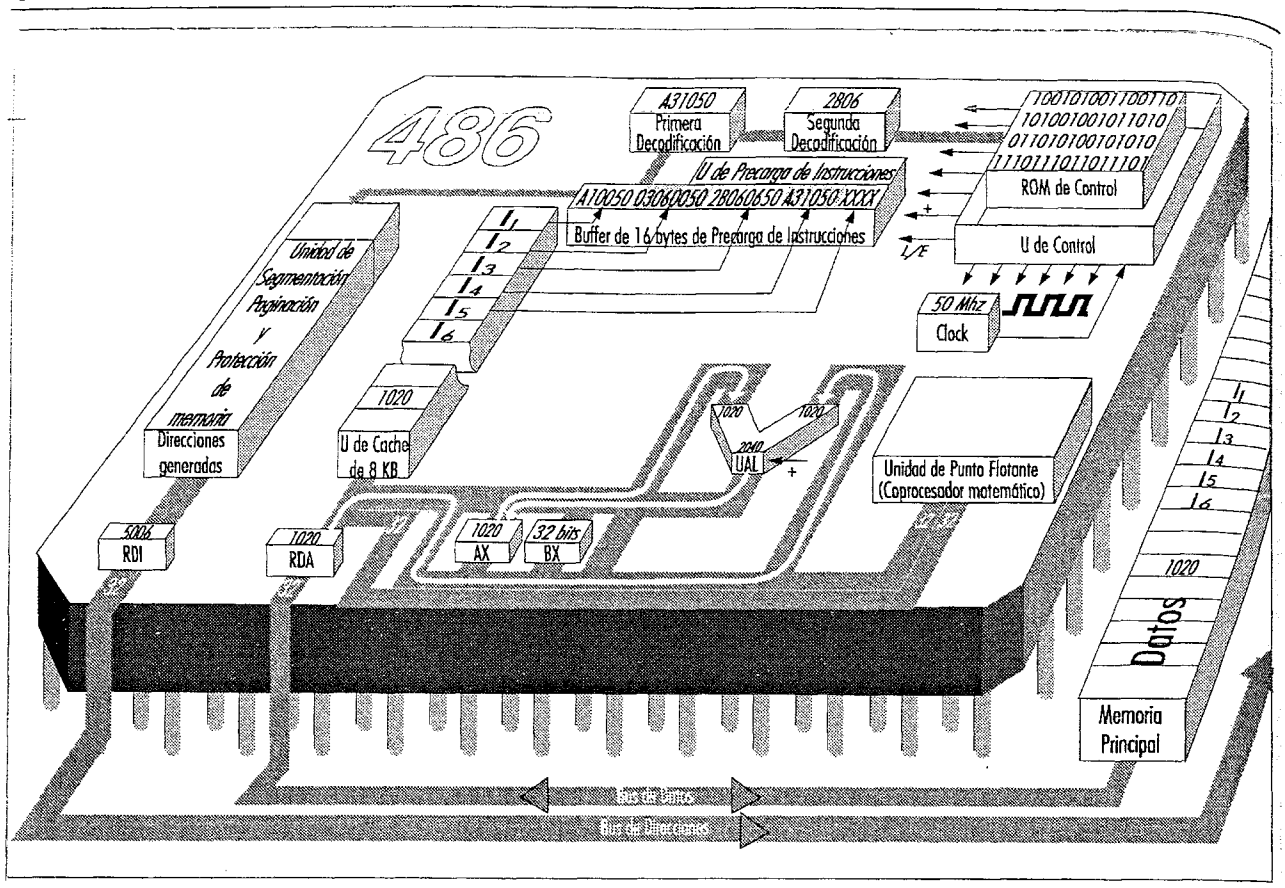


Figura 1.89

¿Qué tiene en común y cómo funclona el Pentium en relación con el 486 ?

La figura 1.90 da cuenta de un esquema básico de un Pentium basado en el del 486 de la figura 1.86. Como en el 386 y el 486, en el Pentium las instrucciones para enteros, siguen un "pipe line" de 5 etapas. Para la etapa de pre-carga en la figura se supone que en un caché interno de 8 KB se encuentran las próximas instrucciones a ejecutar, las cuales en el Pentium pasan en promedio de a diez juntas (de I_1 a I_{10}, pudiendo ser de un programa realizado para cualquiera 80x86, pues el Pentium es compatible con ellos) hacia un buffer de la U de pre-carga, que puede almacenar 32 bytes (existen dos de estos buffers). O sea en dicho caché se leen 32 bytes en un solo acceso. Los datos están en otro caché de 8 KB.
Tener dos memorias caché separadas permite que mientras por un lado se accede a las próximas instrucciones a ejecutar en un caché, al mismo tiempo en el otro, se accede a datos, sin tener que esperar. Por tener el Pentium un bus de datos externo e interno de 64 bits que llega a cada caché, se posibilita que en cada acceso al caché externo (para leer un dato o instrucción contenido en éste, y si no lo está se accede a memoria principal), cada caché reciba el doble de datos o instrucciones –según cual sea– que en el 486.
El Pentium, contiene dos "pipe line", para instrucciones, que operan con números enteros, a fin de poder procesar dos instrucciones en forma independiente (como una fábrica de autos con dos líneas de montaje). Esto lo hace "**superescalar**", capaz de terminar de ejecutar dos instrucciones en un pulso, como los procesadores RISC, por lo cual también como en éstos, se requiere un caché para datos y otro para instrucciones. Asimismo deben existir por duplicado: la unidad decodificadora (de modo de poder decodificar dos instrucciones por vez), la unidad de segmentación generadora de direcciones de datos, y la UAL.
Puesto que dos instrucciones en proceso simultáneo pueden necesitar acceder juntas al caché de datos para leer cada una su dato a operar, este caché tiene duplicado el número de líneas de datos y de direcciones.

A la primer decodificación entran dos instrucciones al mismo tiempo. Durante la misma se determina si ambas se procesarán juntas (una en cada "pipe line"), o si sólo seguirá una por el "pipe line". También se identifica la porción de cada instrucción que permite formar la dirección del dato (que pasa a la unidad de segmentación correspondiente), y cada código de operación (que pasará a la segunda decodificación).

Una instrucción para números en punto flotante opera con datos de 64 bits, que ocupan los dos "pipe lines" para números enteros (de 32 bits cada uno), por lo que ella no puede procesarse junto con otra instrucción[1].
Estas instrucciones pasan por las cinco etapas correspondientes a instrucciones para enteros, y además requieren 3 etapas de un "pipe line" exclusivo para punto flotante. Puede decirse que el Pentium presenta un "pipe line" de 8 etapas, siendo que las instrucciones para enteros se ejecutan en 5 etapas.
Las denominadas instrucciones "simples" para enteros, luego de haber pasado por la pre-carga, y las dos decodificaciones (pasos **1, 2 y 3**) se ejecutan en uno, dos o tres pulsos reloj, según sea su complejidad.
No requieren acceder a microcódigos de la ROM de Control: a partir de su código de operación, se generan las señales de control que ordenarán a los circuitos de la UCP intervinientes, qué operaciones realizarán.
Este tipo de instrucciones para enteros son las que pueden procesarse de a dos (una en cada "pipe line").
Instrucciones como $\mathbf{I_1}$, $\mathbf{I_2}$ e $\mathbf{I_3}$ (de la figura 1.86) que requieren una lectura del caché interno de datos (si el dato no está en él, hay que leer el caché externo, y si tampoco está en éste, leer la memoria principal), pueden ejecutarse en dos pulsos, luego de la segunda decodificación (paso 3); mientras que $\mathbf{I_4}$, que necesita una escritura en el caché de datos, se lleva a cabo en tres pulsos.
Las instrucciones muy simples, por ejemplo con datos a operar en registros de la UCP, y el resultado de la operación asignado a otro registro de la UCP, se ejecutan en un solo pulso reloj, luego de la 2da decodificación

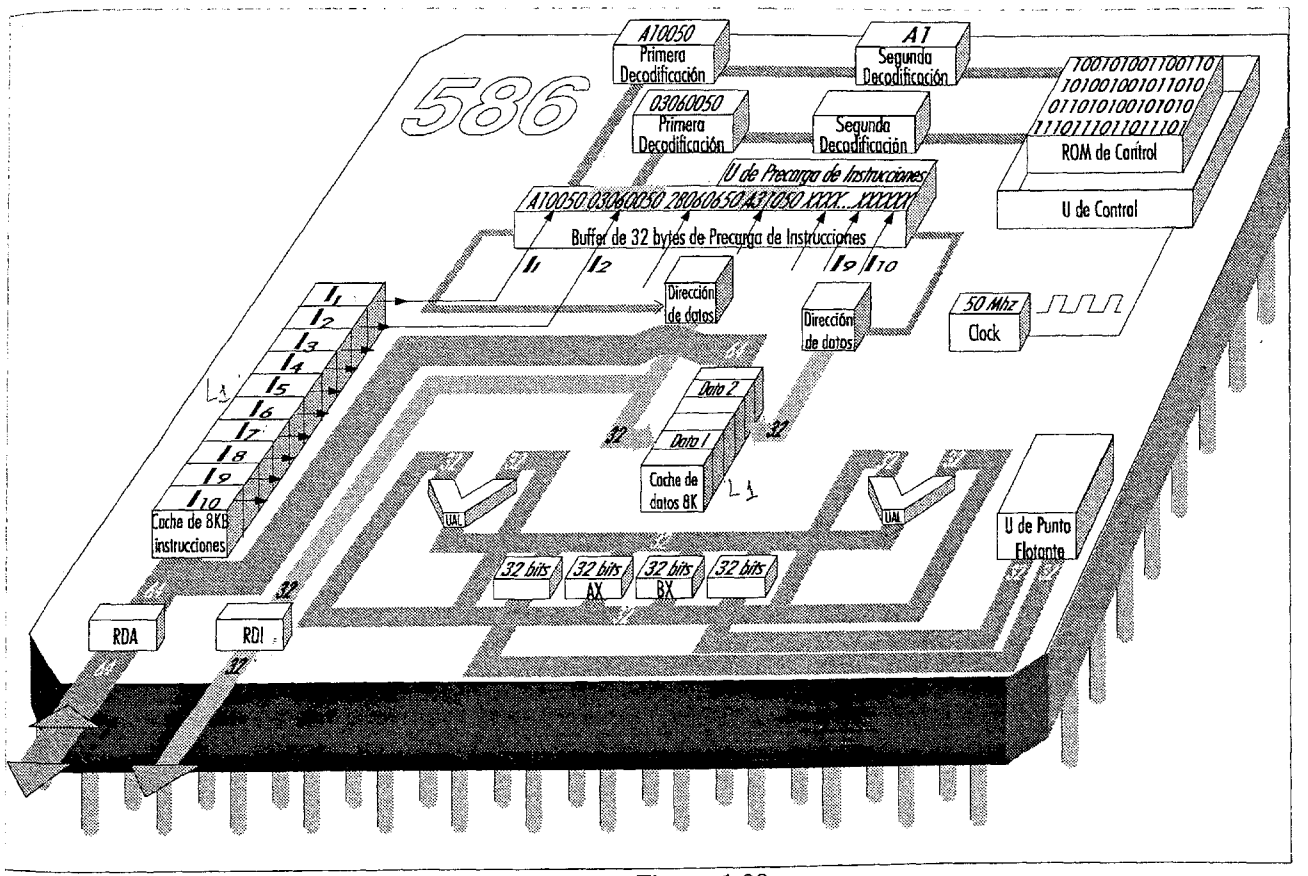


Figura 1.90

Si en la primer decodificación se determina que el par de instrucciones identificadas son simples, y que la segunda en orden no depende del resultado de la primera, cada una sigue su ejecución en uno de los dos "pipe-lines", o sea que se procesan en paralelo.
Y si además ambas se ejecutan en igual cantidad de pulsos reloj, entonces, al cabo del último de ellos, las mismas se terminan de ejecutar simultáneamente. Esta es la forma en que el Pentium puede ejecutar dos instrucciones en un pulso reloj, *lo cual significa que los resultados de las operaciones ordenadas se obtienen a un mismo tiempo.*

[1] Procesadores RISC como el Power PC, el Alpha, y el SuperSpark pueden procesar juntas una instrucción para enteros con otra para punto flotante, y además cambiar el orden de ejecución que las instrucciones tienen según el programa, lo cual el Pentium no puede hacer.

– Esto se ejemplifica en las figuras 1.91 y 1.92, para las 5 etapas citadas del "pipe line", siendo que el paso 2 corresponde a la primer decodificación.

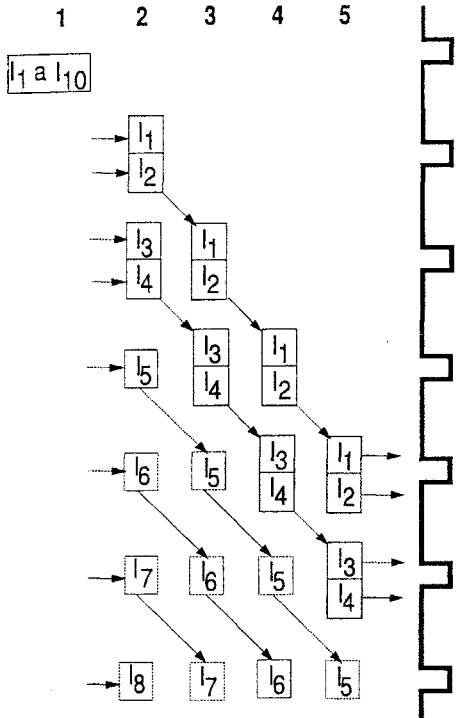


Figura 1.91

Se ha supuesto que en la etapa de pre-carga (paso 1) entraron al buffer diez instrucciones, siendo que las cuatro primeras en orden de ejecución se ejecutan de a pares, por ser sencillas, y requerir dos pulsos reloj luego de la segunda decodificación para terminar de ejecutarse.[1] Esto último se supone que no ocurre con I_5 e I_6, ni con I_6 e I_7, por lo que I_5 e I_6 siguen de a una por el "pipe line"

De esta forma, en el quinto pulso se terminan de ejecutar juntas I_1 e I_2, y en el sexto I_3 e I_4.

Si I_5 e I_6 son instrucciones de punto flotante, su ejecución recién finalizará en el décimo y undécimo pulso, respectivamente, por requerir 3 etapas más, (6, 7 y 8 no indicadas en el dibujo)

Puesto que cada "pipe line" posee su unidad de segmentación –para generar la dirección de un dato– y su UAL, y que desde ambos "pipe lines" se puede acceder simultáneamente al caché de datos al unísono con el otro "pipe line", no existen conflictos de recursos que demoren el procesamiento de dos instrucciones juntas. Asimismo, dado que éstas son "simples", no requieren acceder a la ROM de microcódigo, la cual por ello no necesita duplicarse.

En caso de que una de las dos instrucciones que se procesan juntas requiera menor cantidad de pulsos para su ejecución, ésta se detendrá uno o dos ciclos (según sea) para que los "pipe lines" sigan acoplados.

Cuando por no cumplirse los requisitos, dos instrucciones decodificadas no pueden ser procesadas juntas, primero sigue en uno de los "pipe lines" la que corresponde en orden según el programa. Al pulso siguiente la otra instrucción se volverá a decodificar junto con la que le sigue.

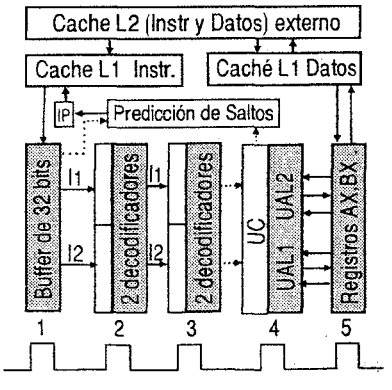


Figura 1.92

Supondremos otra vez que se ejecutan las instrucciones I_1 a I_5 ya conocidas (figura 1.15) y que en un pulso reloj entran a la primera decodificación –como en el 486– 3 bytes de los códigos de máquina de I_1 e I_2. Entonces se detectará que I_2 depende del resultado de I_1, pues I_2 ordena sumar un número que está en memoria al contenido del registro AX, y el valor de éste no puede conocerse hasta que no se termine de ejecutar I_1. Por lo tanto, al pulso siguiente sólo entrará a la segunda decodificación en uno de los "pipe lines" la instrucción I_1 (como sugiere su código de operación A1 decodificándose en esa etapa, mientras que el código de I_2 no aparece en ella). En este mismo pulso se decodificarán juntas I_2 e I_3, y por depender I_3 de I_2, al pulso siguiente sólo pasará a la segunda decodificación I_2; a la par que I_1 entrará en la fase de ejecución[2].

Otra innovación que ha incorporado el Pentium es la *"predicción de bifurcación"*. Cuando se procesa una instrucción de salto (que en general son el 50% de las instrucciones de un programa), si la condición del salto se cumple, las instrucciones que entraron después de ella al "pipe line" deben ser reemplazadas por la instrucción a la cual se ordena saltar, y las que le siguen. Aunque éstas se pueden preparar por las dudas en el caché (como lo hace el 486), si ocurre el reemplazo citado se produce una demora de dos o más ciclos.

El **BTB** (Branch Target Buffer) del Pentium es otro caché de 1 KB, que guarda la dirección hacia donde saltar ("target"), que indica cada instrucción de salto que entrará al "pipe line" junto con dos bits que indican con 00, 01, 10 ó 11, cuatro situaciones posibles en las dos últimas ejecuciones de una instrucción de bifurcación, siendo que 1 significa bifurcación realizada, y 0 no efectuada. BTB predice

---

[1] Si bien el Pentium es compatible con el software (en código de máquina) desarrollado para los 80x86, Intel recomienda para aumentar su performance un 30%, volver a traducir (con el compilador para Pentium) estos programas, a partir de sus versiones en Pascal, C, etc. De este modo estos programas, o los nuevos que se desarrollen, se acercarán más al óptimo ideal indicado en la figura 1.91 para I_1, I_2, I_3 e I_4

[2] Debido a que sólo se ha usado el registro **AX**, cada instrucción depende del resultado que debe generar la anterior. Si por ejemplo: I_1 ordenara llevar el número a sumar al registro **BX**; I_3 ordenara sumar AX con BX, I_4 ordenara llevar el número a restar al registro **CX**, se podrían haber ejecutado juntas I_1 e I_2, lo mismo que I_3 e I_4. Se comprende que aumentar el número de instrucciones simples puede conducir a una ejecución más rápida de una secuencia, usando los dos "pipe lines". Un compilador "inteligente" para el Pentium realiza esta tarea.

"dinámicamente" que si se tiene una combinación distinta de 00 ocurrirá una bifurcación. Entonces, uno de los dos buffers de pre-carga (de 64 bytes) que contiene la secuencia de instrucciones a la que se debe saltar, será el que suministrará los códigos de las mismas para que sean decodificados.

¿Qué características tienen los procesadores CISC ?

Las siglas CISC de "Complex Instruction Set Computer" hacen referencia a un computador con un repertorio complejo de instrucciones y una UC basada en una ROM de Control (fig. 1.34) cuyos primeros exponentes en los 70 fueron el IBM 360® y la VAX de DEC®. Siguieron los 68000 de Motorola® y los 80x80 hasta el Pentium®. *Desde el Pentium Pro y Pentium II en adelante, por razones de compatibilidad, Intel® mantiene su repertorio CISC, pero se ejecuta en un núcleo RISC,* siendo que en el interior de un Pentium las instrucciones CISC se traducen a RISC por medio de circuitos.

La palabra "complejo" alude a un repertorio entre 200 y 300 instrucciones, que según la cantidad de pasos (µcódigos) para ser ejecutadas (fig. 1.34) pueden ser simples, menos simples y complejas. A su vez cada instrucción puede aparecer en un programa de muchas formas (modos de direccionamiento) distintas (25 en la VAX), según que el dato que se quiere operar en la UAL esté en un registro de la UCP, en memoria, o se trate de una constante. Esta complejidad implica un *número muy variado de formatos de instrucciones de diferente longitud y disposición de la información en cada formato,* por lo que su etapa de decodificación no puede ser muy rápida por requerir una circuitería complicada.

Asimismo, todas las operaciones que hace la UAL pueden hacerse en modos que accedan a datos en memoria, pues los procesadores CISC tienen pocos registros, lo cual implica más pasos en la ejecución.

En los repertorios de instrucciones CISC, las complejas son aquellas que ordenan mover cadenas de caracteres de un lugar a otro de memoria, existentes en Intel; o CASE de la VAX, que emula la sentencia del mismo nombre usada en lenguajes de alto nivel. O sea que un procesador CISC además de las instrucciones que ordenan una sola operación tiene otras que ordenan una serie de operaciones, como lo hacen las sentencias que constituyen los programas en lenguaje de alto nivel.

Así, mientras que una instrucción de salto condicional común ordena decidir entre dos secuencias de instrucciones cuál de ellas se ejecutará (sección 1.9), la instrucción CASE ordena una decisión entre varias secuencias posibles, siendo que ello también puede hacerse usando varias instrucciones de comparación y de salto comunes. En esencia el procesador puede ejecutar directamente como instrucciones de máquina ciertas sentencias de alto nivel, sin que el compilador necesite traducirlas a una secuencia de instrucciones de máquina, con lo cual el compilador es más simple de construir.

En realidad la traducción la realiza el hardware, dado que la secuencia de pasos simples (µcódigos) que una UCP de núcleo CISC hace para ejecutar cualquier instrucción están memorizadas en la ROM de Control que forma parte de la UC, donde están guardados dichos pasos.

Como se describió (figs. 1.33 y 1.34), cada paso se lleva a cabo mediante una combinación binaria (*micro-código* o *microperación*), que con cada pulso reloj aparece en las líneas de control de la UC provista por la ROM de Control (no confundir con Rom Bios), la cual activa los circuitos que intervienen en ese paso. Dado que las instrucciones complejas ocupan muchos pasos necesariamente se requiere esta ROM.

Por lo tanto, *un procesador CISC necesariamente debe contener una ROM* con los microcódigos, para poder ejecutar las instrucciones complejas. Esta es una de las características CISC.

De esta forma, en principio podría ser más rápido obtener dicha secuencia de pasos a realizar de dicha ROM inserta en la UCP que pedir una secuencia equivalente de instrucciones de memoria principal.

Con el advenimiento masivo de las memorias caché y de los pipe line con pre-búsqueda de las instrucciones a ejecutar, dicha aparente ventaja se desvaneció.

En general, en la concepción CISC se busca una menor disparidad entre los lenguajes de alto nivel y el lenguaje de máquina, lo que se da en llamar "salto semántico". Recordar al respecto, que una sentencia como $\mathbf{Z = P + P - Q}$ se debe traducir a una secuencia de instrucciones de máquina como $\mathbf{I_1, I_2, I_3, I_4}$.

Suponiendo que en un cierto lenguaje de alto nivel dicha sentencia (u otra más común) se usara frecuentemente, se podría tener un CISC que hiciera corresponder a esa sentencia una sola instrucción $\mathbf{I_x}$, de máquina, que reemplazara a las 4 instrucciones citadas. Esto se conseguiría escribiendo en la ROM de Control una extensa secuencia de microcódigos para poder ejecutar $\mathbf{I_x}$.

Se comprende que esta concepción puede llegar al extremo de fabricar un CISC con instrucciones de máquina que sean equivalentes a sentencias muy usadas en un cierto lenguaje de alto nivel, pero que no se usarían, si se programa en otro lenguaje de alto nivel que no las utiliza.

Otro de los motivos de usar instrucciones complejas, fue que por ocupar menos espacio de memoria que varias simples que realizan la misma función, podría lograrse ahorrar espacio, en una época en que la memoria era un recurso caro. Por otra parte, estudios en los 80 indicaron que los compiladores con frecuencia no descubren las sentencias que pueden traducirse directamente en instrucciones complejas, y que *la mayoría de las instrucciones traducidas son simples.*

Se consideran "*simples*" las instrucciones que una vez obtenidas por la UCP desde la memoria, no requieren acceder otra vez a ésta ya sea para obtener un dato o para escribir el resultado de la operación. Esto ocurre con las instrucciones en "*modo registro*" [1] en las que se ordena operar dos números que están en dos registros de la UCP, y el resultado asignarlo a uno de esos registros o a un tercero; y con las instrucciones en "*modo inmediato*" [2] que ordenan operar el dato (una constante) que forma parte de la instrucción, y que por lo tanto llega con ella al registro RI de la UCP (fig. 1.23), cuando dicha instrucción fue obtenida. También son simples las instrucciones de salto ("*modo relativo*")[3], ya que viene con ellas el número que se debe sumar a la dirección de la instrucción siguiente para obtener la dirección a la cual se debe saltar (Unidad 3 de esta obra).

Las instrucciones tipo *Load* (cargar un registro desde memoria) o *Store* (guardar en memoria el contenido de un registro), una vez obtenidas por la UCP desde la memoria requieren acceder una vez más a ésta para leerla (*Load*) o para escribirla (*Store*). Por lo tanto hace falta un paso más que en una *simple* para ejecutarlas.

La dirección de memoria a acceder puede formar parte de la instrucción (*modo directo*), o estar en un registro (*modo indirecto por registro*), o se halla sumando dos registros o un registro y una constante (otros modos)[4].

De lo anterior resulta que las instrucciones *simples*, las del tipo *Load/Store* y las complejas necesitan distinta cantidad de pulsos reloj para ser ejecutadas. Pensando en su ejecución en un CISC con pipeline (como el 486), las instrucciones complejas, por necesitar muchos pasos, demorarán la ejecución de las de pocos pasos.

Simplificando el problema y a los fines conceptuales, supondremos un lavadero de autos en serie con 4 etapas, pensado para coches en los que cada etapa dura 5'. Entonces cada 5' saldría un auto terminado, siendo que cada uno está en el proceso 20'. Si también entran autos que requieren un lavado de 25' debido a que requieren más etapas o más tiempo para una de ellas, entonces no saldría un auto cada 5'.

En síntesis, un CISC tiene limitaciones en su productividad (instr/seg) por requerir una decodificación compleja que insume tiempo, y por ser su pipeline ineficiente si entran en él instrucciones simples y complejas.

Asimismo, dado que en un CISC predomina la ejecución de instrucciones simples, pero la Unidad de Control y la ROM de Control tienen una complejidad acorde a todos los tipos de instrucciones que se deben ejecutar, resulta que el tiempo de ejecución de las simples se ve perjudicado. Además dicha ROM que almacena las ordenes para llevar a cabo los pasos de ejecución de las instrucciones -indispensable sólo para las complejas por requerir mucho pasos- ocupa hasta el 50% del área de silicio de la UCP, impidiendo otras funciones.

En los CISC con escaso número de registros de uso general (16 en el 486 y Pentium) puede suceder que los programas tengan más instrucciones que las necesarias -y por lo tanto tarden más en ejecutarse- dado que los registros no alcanzan. Entonces primero deben agregarse instrucciones para salvar contenidos de registros en memoria a fin de usarlos en otro procedimiento, y luego otras instrucciones deben restaurar esos contenidos.

¿En qué se diferencian los procesadores RISC de los CISC ?

Los procesadores **RISC** (Reduced Instruction Set Computer) -computador con repertorio de instrucciones reducido- fueron desarrollados por D. Patterson y J. Hennessy en los 80, luego de analizar las limitaciones en productividad que presentaban los CISC, a partir de estudios estadísticos realizados con el software

Buscando optimizar la performance de los procesadores, se realizaron estadísticas de las instrucciones de máquina más usadas. Resultó que las instrucciones *más simples* –que sólo son el **20%** del repertorio de instrucciones de un procesador CISC– constituían el **80%** de programas típicos ejecutados.

También se determinó que en los programas predominaban las sentencias de *asignación*, siguiéndole en número los *If*, los *call*, y los *loop*. Las llamadas a procedimiento (*call*) son las que emplean más tiempo de máquina, y los ciclos repetitivos (*loops*) son los que consumen la mayor parte del tiempo de ejecución de los programas.

También en especial se constató que la mayoría de las instrucciones que traducía un compilador eran *simples.*

Se verificó que el rendimiento de una arquitectura depende fundamentalmente de la existencia de **compiladores inteligentes**, capaces de generar código de máquina que optimicen *asignaciones* y *loops*, y con algoritmos para generar secuencias que ayuden a la productividad del pipeline.

---

[1] Que en el Assembler de Intel (Unidad 3 de esta obra) pueden ser por ejemplo: **ADD AX, BX** ; **MOV CX, DX**; **AND CX, DX**, etc.

[2] Idem **MOV CX, 2000**; **ADD BX, 4** **CMP AX, 50**

[3] Idem **JZ 32A0** **JL 0500** **JMP 2800**

[4] Idem Modo directo: **MOV AX, [2000]** (Load); **MOV [5000], CX** (Store); Indirecto por Registro: **MOV AX, [SI]** , **MOV [DI], CX**

A tales fines estos compiladores procuran que los datos más utilizados en un lapso de tiempo estén en registros, minimizando los accesos a memoria. Conforme con ello, *todas las operaciones de la UAL deben ser hechas con instrucciones en modo registro* (como está pensado el repertorio de un RISC), usando las del tipo *Load/Store* sólo cuando sea imprescindible. Lo anterior implica que *un RISC debe presentar un gran número de registros.* Los programas en código de máquina generados por estos compiladores tienen por lo general un 20 a 30% más de instrucciones que un CISC, pero se ejecutan mucho más rápidamente, por ser las instrucciones simples.

Si bien el número de instrucciones de un RISC se reduce en relación con los CISC (los RISC "puros" menos de 100; el PowerPC tiene 225), ésta no es la esencia de un RISC. Lo determinante es que el repertorio de instrucciones es *sencillo*: todas **son igual de igual tamaño**, *con su cod-op y operandos en igual disposición y con pocos modos de direccionamiento.* Todo ello con el objetivo de **simplificar el Decodificador y la Unidad de Control**, de modo que cada paso en la ejecución de una instrucción sea lo más rápido posible.

Por constar de instrucciones cuya fase de ejecución requiere mayormente un paso, o a lo sumo dos (para load/store), *no se requiere una ROM de Control para guardar cada microcódigo que debe aparecer en las salidas de la UC con cada pulso reloj a los fines de comandar el procesador.*

Esto a su vez permite aprovechar el área que ocupa esta ROM en el chip, el que se usa para registros y pipelines. *Más registros implica menos accesos a memoria*, con la consiguiente ganancia de tiempo.

En un pipeline se trata de que con cada pulso reloj se termine de ejecutar una instrucción (la que está en el último paso del mismo), para lo cual, entre otras cosas se necesita que todas las instrucciones se ejecuten en igual número de pasos. Suponiendo que sea así, si el clock es de 100 Mhz (lo cual implica que cada paso dura 10 nseg,), saldrían ejecutadas del pipeline 100 millones de instrucciones por segundo (MIPS). Si se logra que cada paso por ser más sencillo dure 5 nseg,, se ejecutarían hasta 200 MIPS (con un reloj de 200 Mhz).
Conforme a lo anterior, puesto que no existen instrucciones complejas, y que la mayoría de las instrucciones son sencillas, ejecutables en igual cantidad de pasos (las del tipo *Load/Store* usan un paso más para acceder a memoria) aumenta considerablemente el rendimiento de un pipeline RISC en relación al de un CISC.

Mientras que la mayoría de los CISC si por ej. se suman los contenidos de dos registros, el resultado se asigna a uno de ellos, destruyendo su contenido original, en los RISC dicho resultado puede ir a un tercer registro, merced a una UAL ligada a 3 buses (fig. A4.6). Esto permite reutilizar operandos para dar mayor flexibilidad a los compiladores a fin de que minimicen dependencias en las instrucciones que generan, y sea menor su número
Nuevamente se ve en estos casos que la *complementación entre compilador inteligente y procesador, a los efectos de que éste sea más sencillo y su pipeline más eficaz.*

Para mejorar los tiempos involucrados en el llamado y retorno de subrutinas, el gran número de registros de un RISC puede ser usado para no tener que acceder a una pila definida en la memoria (U3 esta obra)
Así, en el RISC II existen 138 registros en círculo, con los cuales pueden formarse "ventanas" de 32, que es el número de registros que puede usar un programador. Cuando se llama a una subrutina se habilita otra ventana de 32 para uso de ella, siendo que con la ventana anterior comparte algunos registros para el pasaje de datos y resultados. Al finalizar la ejecución de la subrutina se vuelve a la ventana anterior; y si desde esta subrutina se llama a otra, para ésta se habilita la ventana siguiente que compartirá registros con la anterior.
Las variables globales (que comparten subrutinas) se reservan 10 registros para todas las ventanas.

¿ Cómo funciona la familia P6 (Pentium® Pro, II, III y el Celerón®)[1] ?

El Intel 486™ tiene un pipeline de 5 etapas y una sola UAL. Podía ejecutar hasta una instrucción por pulso reloj.
En la fig. 1.90 se mostró un esquema del Pentium I, sintetizado en la fig. 1.92, procesador con un pipeline de 5 etapas y parcialmente superescalar: con dos unidades de ejecución (**UE**), constituidas por 2 UAL, que podía terminar de ejecutar juntas en un pulso reloj dos instrucciones para enteros si una instrucción no requiere un valor calculado por la instrucción anterior. De existir tal dependencia, la segunda instrucción debe esperar los ciclos necesarios. En promedio del pipe line salían ejecutadas 1,3 instrucciones por ciclo reloj.

La ejecución fuera de orden en un **P6** permite aprovechar dicha espera para adelantar la ejecución de instrucciones que no estén en conflicto con ninguna instrucción pendiente. Así puede mejorarse el número de instrucciones terminadas de ejecutar por ciclo (3 en promedio) y por segundo, mediante un hardware más complejo.
También realizaba predicción dinámica de saltos para mejorar el rendimiento del pipeline.

[1] La estructura y funcionamiento del Pentium Pro tiene grandes similitudes con el **PowerPC 604**

A continuación se describirán aspectos centrales comunes del funcionamiento de la familia de procesadores de **arquitectura P6** de Intel introducida en 1995, que comprende sucesivamente el **Pentium Pro**, el **Pentium II**, el **Pentium III**, el **Xeon®** y el **Celerón®** (este último no tiene caché L2 integrado como los anteriores).

El núcleo de estos Pentium, es una organización superescalar de concepción RISC con varias **UE**, para llevar a cabo operaciones simultáneas, capaz de ejecutar en forma transparente las instrucciones fuera del orden en que están en el programa. Existen **UE** para enteros, punto flotante, MMX® y load/store, con el fin de lograr más instrucciones ejecutadas por segundo. Se supone que las **UE** para enteros permiten sumar dos registros y el resultado enviarlo a un tercero como en el esquema de la fig. A4.6). En dicho núcleo, se ejecutan instrucciones simples *de igual longitud* (como ser 118 bits) que designaremos "micro-operaciones tipo RISC" (**µops-R**).
Ellas provienen de la *traducción por hardware* realizada dentro de un Pentium®, de programas para procesadores 80x86 y Pentium® de Intel® que identificaremos con las siglas 80x86. Cada nuevo modelo de Pentium debe poder ejecutar programas con instrucciones de todos los procesadores 80x86 anteriores.
Esta traducción se realiza en paralelo con la ejecución de **µops-R** que acaban de ser traducidas.

Desde el P6 en adelante físicamente ya no existen más los registros AX, EAX, BX, EBX, SI, ESI, etc.
Al ser traducida cada instrucción 80x86 en **µops-R**, dichos registros luego son renombrados, en correspondencia con decenas de registros (R_1, R_2, ... R_n) existentes en el núcleo RISC citado. Vale decir que **todos los 80x86 y Pentium de Intel tienen repertorio de instrucciones CISC, pero el núcleo de los Pentium actuales es RISC.**

La fig. 1.93 ilustra las 11 etapas del pipeline de la familia P6 a las que nos referiremos y su relación con los cachés y unidades de predicción de saltos (**UPS**). Es un pipeline de 11 etapas contra 5 del Pentium I.
Lo que interesa es el número de instrucciones que se ejecutan por segundo, sin que importe el hecho de que una ejecución requiera una sucesión de muchos pasos, siempre que sean muy breves.
Si cada paso se ejecuta rápidamente, el tiempo de cada ciclo del reloj puede acortarse, por lo que puede usarse un reloj con más Mhz, o sea con mas ciclos por segundo. Entonces si además en cada ciclo se terminan de ejecutar una o más instrucciones juntas, aumenta la cantidad de instrucciones que se ejecutan por segundo..
Suponiendo un lavadero de autos en serie con 4 etapas en el lavado, si cada una dura 5' saldrá un auto listo cada 5', siendo que cada auto tarda 20' en terminarse. Si a costa de tener más maquinarias, espacio y personal, el lavado se divide en 8 etapas de 4' cada una, cada auto se terminará en 32', pero saldrá un auto cada 4'.

En la etapa de pedido, el IP le proporciona una dirección a la Unidad de Pedido (**UP**) en cada acceso de ésta al caché L1 de instrucciones, para pedir a partir de ella dos líneas del mismo (64 bytes) con instrucciones sucesivas próximas a ejecutar (en promedio 32) y analiza tiras de 16 bytes que guarda temporariamente en un buffer (UP1 en la fig. 1.93).
Por tener cada una de estas tiras instrucciones de distinta longitud debe ser analizada en UP2, a fin de determinar en qué byte comienza el código de operación (cod-op) de cada instrucción. Cada tres cod-op sucesivos así hallados van en orden a una Unidad Decodificadora (**UD**) donde cada uno se traduce en una o varias µops-R. *Todo sucede como si en las salidas de la UD cada instrucción CISC se ha dividido en una o varias pseudo instrucciones RISC.*
El número de µops-R por cada instrucción depende si ésta es simple, o de acceso a memoria, o compleja.
La **UD** recibe en orden grupos de 3 instrucciones del programa en ejecución, provenientes caché LI de instrucciones, y las traduce en µops-R (3 por ciclo reloj) de 118 bits, acordes al orden de esas instrucciones.
Mientras se realizan los procesos de pedido y traducción, las Unidades de Ejecución (**UE**) al mismo tiempo están ejecutando µops-R de instrucciones traducidas anteriormente.

Más en detalle:
Para la traducción, la **UD** tiene 3 decodificadores que operan *en paralelo* para traducir 3 instrucciones (UD1): 2 de ellos para traducir cod-ops de instrucciones 80x86 simples (que son las que predominan en los programas) cuyos cod-ops se traducen en una sola µop-R. El tercer decodificador maneja cod-ops que se traducen de 1 a 4 µops-R. Si se tiene una instrucción compleja que se traduce en más de 4 µops (son muy pocas) pasa a una ROM; y si hay varias complejas, UP3 no deja que lleguen a los 2 decodificadores para simples. UP3 opera un buffer rotatorio que enfrenta cada cod-op con el decodificador apropiado).
En la etapa UD2 los µops generados en UD1 pasan en el orden correspondiente al programa a una cola de µops, y las µops de salto se presentan a la Unidad de Predicción Estática de Saltos (**UPES**) a tratar.

Estas µops-R así ordenadas van a una etapa de Renombramiento de Registros (**RR**) para eliminar falsas dependencias entre instrucciones sucesivas debidas al escaso número de registros que presentan al programador los procesadores CISC de Intel, lo cual impide que se ejecuten juntas. Las dependencias verdaderas se mantienen, para ser resueltas por la ejecución fuera de orden. Se convierte una arquitectura 80x86 que sólo presenta 16 registros para usar con los programas (8 para enteros y naturales más 8 para punto flotante), en una arquitectura RISC con 40 registros.
Por ejemplo, si dos instrucciones 80x86 ordenan escribir un resultado en un mismo registro, no podrían ser ejecutadas juntas fuera de orden, a menos de que transitoriamente uno de los dos resultados se guarde en un

registro que sea "alias" del verdadero. Esta situación es muy probable que ocurra, dado que en los 80x86 con 8 registros para enteros, dos instrucciones cercanas pueden ordenar guardar un resultado en un mismo registro.

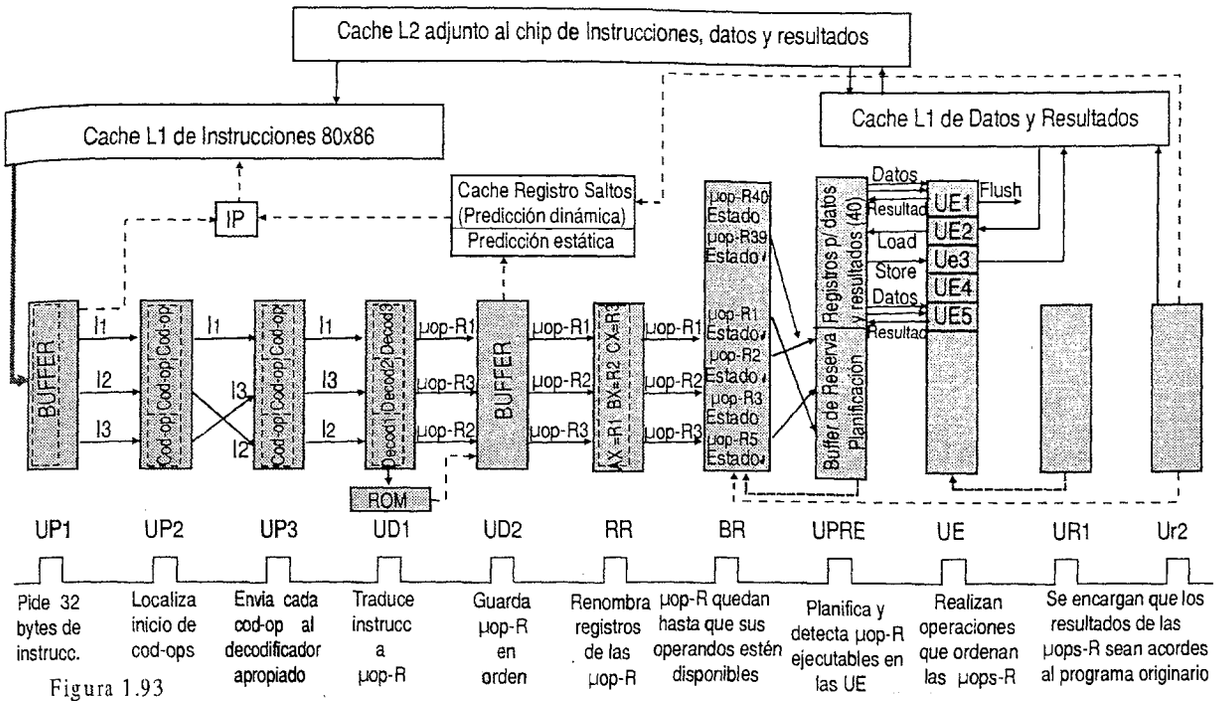


Figura 1.93

Las µops-R que entraron a **RR** pasan ordenadas al Buffer para Suministro y Reordenamiento de µops-R (**BSR**), conforme al orden que tienen en el programa las instrucciones traducidas en la decodificación de la cuales ellos provienen, a fin de que la Unidad de Retiro (**UR**) pueda rastrear este orden luego de ejecuciones realizadas fuera de orden.
El **BSR** es un buffer circular de trabajo que puede contener hasta 40 µops-R. Asimismo contiene los 40 registros físicos para los datos y resultados de las µops-R antes citados. Junto con cada µop-R además se guarda:

- la dirección de memoria de la instrucción que originó su traducción.
- el "alias" con que fueron renombrados sus registros 80x86 en relación con los 40 registros internos citados.
- la indicación ("status") *de si la µop-R está lista para ejecutarse por estar disponibles los datos que requiere su ejecución*, o si ya fue enviada para su ejecución, o si ya fue ejecutada y está lista para ser retirada.

Si la **UPRE** (Unidad de Planificación, Reserva y Envío para ejecución) lee en el **BSR** la indicación de que una µop-R esta lista para ser ejecutada, una copia de ésta y de sus operandos es tomada del **BSR** y pasan a un Buffer de Reserva de la **UPRE** (de hasta 20 µops-R), sin importar el orden en que estaba en el **BSR**. [1] Si está disponible una **UE** que realiza la operación ordenada por una de estas µops-R, ella es efectuada en dicha **UE**, y el resultado pasa al registro que corresponda en el **BSR**, o a un registro transitorio, según sea.

En cierto modo es como si este Buffer contendría los Registros de Instrucción con las µops-R cuya operación se concreta en la **UE** que corresponda, siendo que ciertos bits de cada µop-R ordenan que deben hacer dicha **UE**. En cada ciclo reloj sólo puede operar una **UE** por µop-R

En la arquitectura P6 hay 5 posibles **UE**. Dos **UE** están dedicadas a µops-R *store*, y otra **UE** para µops-R *load*.
La cuarta **UE** contiene -según se necesite- una UAL para enteros, circuitería (coprocesador) para µops-R de punto flotante, y circuitería SIMD para procesamiento vectorial MMX, no existente en el Pentium Pro (ver más adelante). Esta **UE** además tiene circuitería para limpiar el pipeline en el caso poco probable que falle la predicción de un salto, pues habrá µops-R en varias etapas del mismo cuya ejecución no debe proseguir. Cuando se ejecuta una µop-R de salto si su resultado no coincide con el pronosticado (ver más adelante) deben eliminarse todas las µops-R que siguen a la de salto, incluidas las del buffer de reserva citado, y deben ejecutarse las µops-R que correspondan.
La quinta **UE** también como la anterior puede operar según se sea: a enteros (con otra UAL), a números en punto

[1] En la ejecución en orden, la decodificación de instrucciones se realiza hasta que ocurre una dependencia o conflicto, reanudándose cuando éste se soluciona. De este modo no pueden llegar instrucciones posteriores a las del conflicto, lo cual impide ejecutarlas anticipadamente si no presentan dependencias con las que están en ejecución.. En cambio si se ejecuta fuera de orden, y el hardware permite un flujo continuado de uops-R hacia la **UPRE** que se almacenan para ser ejecutadas cuando convenga, se puede anticipar la ejecución de instrucciones sin dependencias; y mientras el buffer no se llene se le pueden seguir enviando instrucciones desde el decodificador. De esta forma puede darse un flujo continuo en la decodificación y en la extracción de uops-R desde la **UPRE** a las **UE**. Así se puede tener capacidad de anticipación para decidir qué uop-R pueden ejecutarse fuera de orden.

flotante y a operandos MMX. Así es factible ejecutar hasta 5 μops-R juntas y 3 en promedio.

Por lo tanto, la **UPRE** evalúa cuál μop-R es la más conveniente para ser ejecutada en el siguiente pulso del reloj, para lo cual se determina en cada ciclo las μops-R que tienen disponibles sus operandos, las dependencias que pueden ser resueltas y el estado de ocupación de las **UE**. En esencia **se lleva a cabo un análisis de flujo de datos** a fin de lograr la ejecución **óptima**. Las μops-R ahora están ordenadas en función de las dependencias de datos y no de su orden originario. Si varias μops-R pueden ejecutarse, un algoritmo FIFO determina cuáles irán a las **UE**, con prioridad de las μops-R de salto frente a las otras, dada su importancia en la prevención de las ramificaciones.
Si se termina de ejecutar una μop-R, la **UPRE** actualizará el **BSR** indicando tal evento, para que dicha μop-R sea retirada, y la **UE** correspondiente queda disponible para otra μop-R.
Aunque en una ejecución fuera de orden las operaciones no se efectúan en el orden indicado en el programa, los resultados de las mismas son almacenados temporariamente para ser luego asignados a memoria o a los registros indicados por las instrucciones del programa conforme al orden de éstas.
Si la **UPRE** tiene tiempo disponible, el mecanismo anterior también hace posible la **ejecución especulativa** de decenas de instrucciones que están más adelante en el programa, las cuales muy probablemente van a necesitarse. En caso de que esta predicción falle, se anulan los resultados obtenidos.

La Unidad de Retiro (**UR**) se encarga de que las μops-R sean retiradas en el orden de las instrucciones del programa de las cuales provienen, teniendo en cuenta los saltos habidos, y que los resultados sean asignados acorde con dicho orden. No asigna resultados de μops-R mal ejecutadas.
Envía resultados temporarios que recibe de las **UE** al lugar apropiado cuando son definitivos: al registro "alias" que corresponda o al caché de datos (L1), y también puede ser a una **UE** que está esperando ese resultado.
La **UR** posee registros para resultados temporarios de ejecuciones que aún no han terminado, esperando que finalicen ejecuciones que empezaron anteriormente. Por otra parte el procesador efectúa ejecuciones conforme a la alternativa más probable en un salto. Si la **UR** verifica que una μop-R ejecutada se originó en una instrucción que no debió ser ejecutada, los resultados de tal ejecución no son considerados.
Cuando todas las μops-R en que se ha traducido una instrucción se han marcado como listas para ser retiradas, y se han ejecutado la(s) μop(s)-R de la instrucción anterior, la **UR** determina que los resultados intermedios deben retirarse, para asignarse como definitivos en registros "alias" involucrados en la instrucción citada en primer lugar, o en el caché de datos, y saca del **BSR** la μop-R que corresponde ser reemplazada por otra de la cola de μops-R.
Si bien las μop-R se ejecutan fuera de su orden original, son retiradas en el orden de las instrucciones del programa de las cuales provienen, a la par que no se asignan resultados de μops-R mal ejecutadas.
A continuación, las μops-R ejecutadas (3 en un promedio, que coincide con las 3 citadas que generan los decodificadores, y hasta 5 por ciclo reloj) son reemplazadas por otras provenientes de los decodificadores.
Si este Buffer se llena, los decodificadores transitoriamente dejan de enviarle.

En definitiva, en promedio, desde la decodificación, en cada etapa del pipeline entran y salen **en paralelo** 3 μops-R (siendo que a ella entran 3 instrucciones a traducir), por lo que en cada pulso reloj se terminan de ejecutar 3 μops-R, lo cual se acerca a 3 instrucciones por pulso (ciclo). Es como si existieran 3 pipelines para 3 μops-R.
La ejecución fuera de orden potencia el paralelismo interno, pues si una μop-R espera por un operando que tiene que generar otra μop-R, o por un recurso circuital para ser ejecutada, otras μops-R que correspondan a instrucciones posteriores en el programa pueden terminar de ejecutarse.

En la descripción anterior se supuso implícitamente que el programa a ejecutar fue traducido por compiladores para modelos 80x86 CISC anteriores al Pentium I. Si un Pentium posterior ejecuta programas traducidos especialmente para el mismo por un compilador inteligente, éste buscará "ayudar" al hardware, para lo cual:

- evitará la existencia de instrucciones complejas en la traducción
- minimizará el número de instrucciones que leen o escriben la memoria
- las operaciones que ordenan las instrucciones serán entre registros
- minimizará las dependencias

De esta forma, si bien las instrucciones que resultan de la traducción usan los registros 80x86, y deben ser a su vez traducidas a μops-R, se consigue una buena mejora en el rendimiento del procesador.

Predicción de saltos:
Cada vez que la UAL genera un resultado, también indica mediante el valor de unos indicadores (flags **Z, S,** y otros dados en Unidad 4) si el mismo fue zero (Z=1) o no (Z=0), si fue de signo positivo (S=0) o no (S=1), etc
Recordemos también (sección 1.9) que una instrucción de salto condicional (ISC) decide -en función del valor de un resultado anterior, por el que se pregunta mediante los flags- cuál es la próxima instrucción que se ejecutará luego de ella entre dos instrucciones posibles: la que le sigue a continuación u otra cuya dirección ella permite determinar. Por ejemplo saltar si Z=1 (**condición**) y caso contrario no saltar, o sea seguir normalmente

con la instrucción que sigue a continuación, equivale a decir "saltar si el resultado anterior fue cero", y si es distinto de cero (Z=0) no saltar. Para saber si un resultado R tiene un valor X, la instrucción anterior a la ISC debe ordenar hacer R - X, de modo que si R = X el resultado será cero, y resultará Z=1.
Asimismo existen las instrucciones de salto incondicional (ISI) que "si o si" ordenan saltar ("go-to") a la dirección de otra instrucción, *sin* la alternativa de las ISC de seguir ejecutando las instrucciones siguientes.

En la descripción anterior no se consideraron las ISC que son el 20% del total de instrucciones de un programa, ni las ISI que son el 10%. Cuando entre las instrucciones sucesivas pedidas por la **UP** llega una ISI, las que le siguen no se ejecutarán. aunque fueron pedidas.
La situación se complica más si llega una ISC, pues de no existir una predicción de saltos, la **UP** conocería recién cuando la **UE** de saltos la ejecute, cuál de las dos secuencias alternativas se ejecutará a continuación.
Si ésta es la que está en el pipe line no hay problema. Pero si la secuencia a ejecutar después de la ISC es la otra, habrá que vaciar el pipeline ("flush") de 11 etapas para pasar a ejecutarla, lo cual hace perder entre 4 y 11 pulsos

Se requiere un algoritmo para determinar *anticipadamente* cuál de esas dos secuencias *es más probable* que se ejecute luego de la ISC. Así no se frena el flujo de recolección en la **UP** y el de salida de µops-R ejecutadas en las **UE**. De esta manera se predice las alternativas de ejecución en las ramificaciones múltiples de los programas.

Para cumplimentar este algoritmo (que acierta en más del 90%), existe un Cache de Registro de Saltos (**CRS** o **BTB** - Branch Target Buffer) con la historia de 256 ISC ejecutadas recientemente. Al **CRS** se entra con la dirección que tiene en memoria la ISC sobre la cual se necesita conocer su historia (campo indicativo de 4 bits) y acerca de la dirección hacia donde dicha ISC ordenó saltar la última vez que se ejecutó.

Durante la obtención de las instrucciones la **UP** determina si alguna de las instrucciones es de salto. Entonces envía su dirección de memoria al **CRS**.

Si la ISC ya ha sido ejecutada antes (como una ISC que al final de un loop ordena saltar a la instrucción donde comienza el mismo para repetirlo), dicha ISC figurará en el **CRS** con la indicación de si entonces se efectuó o no el salto, y la dirección dónde saltar. En caso afirmativo lo más probable es que el salto se vuelva a hacer, por lo que **la dirección dónde saltar** -que se calculó con la información contenida en la ISC- **sirve para que la UP pida anticipadamente del caché L1 la secuencia de instrucciones existentes a partir de dicha dirección.**
Si esta predicción "**dinamica**" (realizada sobre la marccha) falla, deben perderse de 4 a 11 ciclos para actualizar el pipeline de 11 etapas a fin de ejecutar la secuencia que sigue a la ISC.
Predicción "**estática**": si dicha ISC no está en el **CRS**, se escribe su dirección en una entrada del **CRS** (lo cual implicará reemplazar a otra entrada más antigua), y se asume que si se ordena saltar hacia atrás (caso frecuente de los loops), lo más probable es que el salto se producirá.

Dual Independent Bus:
Los Pentium Pro y II presentan el "Dual Independent Bus": dos buses independientes que pueden trabajar en paralelo. Uno va al caché L2 (incorporado al chip, y para operar al doble de velocidad que el de otros Pentium) y otro, el **bus del sistema** conectado entre el Pentium y memoria, y vinculado al bus PCI, para realizar transferencias entre memoria y periféricos. Este bus permite transacciones múltiples simultáneas en lugar de una por vez realizada en forma secuencial. Así se logra hasta 3 veces más ancho de banda que el bus simple.

Instrucciones MMX®:
El Pentium II difiere especialmente del Pentium Pro por presentar un conjunto de instrucciones MMX (multimedia extensions) para operar con datos de 8, 16, 32 bits de longitud, ordenados en vectores o matrices, como los de ciertas aplicaciones multimedia repetitivas. Así en audio los datos digitalizados suelen ser de 16 bits. En vídeo los puntos de la pantalla (pixels) que constituyen una imagen se componen combinan-do 3 colores (rojo, verde, azul), indicados por 8 bits, siendo cualquier imagen una matriz ordenada de puntos.
Las instrucciones MMX se ejecutan *en un solo ciclo reloj* (hasta dos por ciclo) y operan sobre elementos constituidos por números enteros de 8, 16, ó 32 bits, agrupables en formatos de 64 bits como sigue:

- *Byte empaquetado*: 8 grupos de 8 bits.
- *Palabra empaquetada:* 4 grupos de 16 bits (palabra para Intel)
- *Doble palabra empaquetada*: 2 grupos de 32 bits (doble palabra para Intel)

Mientras que las instrucciones corrientes operan un par de operandos por vez, una instrucción MMX puede realizar una misma operación aritmética sobre múltiples pares de dichos elementos *al mismo tiempo*. Así se obtiene hasta 10 veces más velocidad que con instrucciones corrientes, evitándose también el uso de coprocesadores para multimedia. Esta forma de procesamiento se conoce como **SIMD** (Single Instruction Multiple Data, o sea una instrucción para múltiples datos) y también como "procesamiento vectorial".
Por ejemplo, si dos formatos de 8 grupos de 8 bits representan dos intensidades de puntos de dos imágenes, su suma simultánea para formar una nueva imagen, se realiza en un solo ciclo reloj. Para estas aplicaciones de color

existe la aritmética "de saturación": si en una suma de enteros el resultado no entra en el formato, o sea hay overflow (Unidad 4 de esta obra), deberá ser que el resultado sea el menor o mayor valor representable
En el **Pentium II** existen 57 instrucciones MMX para sumar, restar, multiplicar, comparar, empaquetar, desempaquetar, transferir, convertir, y para hacer operaciones lógicas.
El **Pentium III** (500 Mhz) presenta las extensiones "Streaming SIMD" con 70 nuevas instrucciones para mejor tratamiento de imágenes, sonido, vídeo 3D, reconocimiento de voz, acceso a Internet, y control del caché.

¿Cómo funcionan el Xeon y Pentium 4 con Hyper Threading ?

Los últimos Xeon™, el Pentium® M y el Pentium 4 son exponentes de la nueva micro-arquitectura NetBurst ™ de Intel® que posibilita el llamado procesamiento Hyper-Threading ("**HT**") y que presenta innovaciones en relación con la arquitectura P6 antes descripta que le sirve de base.
Los actuales sistemas operativos y otros programas de usuario dividen la carga de trabajo en *procesos* que pueden estar formados por uno o varios "**threads**" o "**hilos**", que son porciones de programa en código que se ejecutan concurrentemente.
El "HT" permite que un único procesador físico, que ocupa un chip, aparezca ante el sistema operativo o programas de usuario como dos procesadores lógicos, capaces de ejecutar dos subprocesos en paralelo.
Conforme a ello se pueden programar procesos o subprocesos como si se dispusiera de 2 procesadores físicos.
Pensando en la microarquitectura, ello significa -como se verá- que instrucciones para ambos procesadores lógicos pueden coexistir y ser ejecutadas simultáneamente sobre recursos de ejecución compartidos.
En un procesador común (monoprocesador) los "threads" se ejecutan alternativamente. Este Xeon y Pentium 4 posibilitan procesar 2 "threads" de instrucciones simultáneamente, con un rendimiento que según Intel en un servidor puede superar hasta en un 25% el de un procesador común (aunque menor que el de 2 procesadores físicos separados), sin el costo que significa montar dos procesadores. La "motherboard" debe ser para Xeon o Pentium 4.
Cada día nuestra sociedad basada en la dinámica que imponen los mercados demanda procesadores más y más veloces, a costa de mayor gasto de potencia por disipación de millones de transistores de cada chip.
Para la tecnología actual de fabricación de chips, su disipación de calor se ha convertido en uno de los límitespara el desarrollo de chips más potentes. En los últimos años el número de transistores por chip y su disipación crecen muchísimo más que las mejoras logradas en la performance de los procesadores.
La tecnología para 'HT' busca ser una solución de compromiso entre estos factores antagónicos, Para tal finsólo se duplica un pequeño número de subsistemas menores, que implican el 5% del área de silicio. Los 2 procesadores lógicos comparten la mayoría de los recursos: cachés, buffers, unidades de ejecución y la predicción de saltos.

Se describirá el funcionamiento del Xeon con arquitectura NetBurst, muy similar al Pentium 4, con 42 millones de transistores, y un pipe line de 20 etapas, cada una de menor duración que las 11 de Pentium anteriores. Estode por sí posibilita duplicar los Mhz de funcionamiento, siendo que se debe operar a esta mayor velocidad paraque el procesador tenga la performance en instrucciones/segundo para la que fue ideado.
Dado que cada "thread" se ejecuta en un procesador lógico distinto, con muchos recursos físicos propios, en lo que sigue hablar de "thread" será sinónimo de procesador lógico requerido para ejecutarlo.

El esquema de la fig. 1.94 es un esquema reducido de las 20 etapas del pipeline real. En relación con la arqutectura P6 de los Pentium anteriores hay una diferencia esencial: en lugar del caché L1 para instrucciones 80x86, esta arquitectura presenta un "Trace Cache L1" (**TC**) de 8 vías, para guardar "**traces**". Estas son secuencias ordendas de µops-R provenientes de la traducción por parte de la Unidad de Pedido y Decodificación (**UPD**) de instrucciones 80x86 no complejas, o sea las de uso más frecuente, que son la mayoría que los programas contienen.
Debe asumirse como en el caché de la fig. 1.77.c que la línea y posición dentro de ella donde se encuentra cada µop-R del "trace" (guardado en dicha línea) en el **TC** se localiza por su tag, o sea a partir de la direción de memoria donde estaba la instrucción de la cual fue traducida, como se vio al tratar los cachés.
El **TC** puede guardar 12K de µops-R de igual longitud, y dar salida a 3 µops-R por cada ciclo reloj,siendo direccionado por la **CRS** (BTB en inglés). Esta fue definida en la arquitectura P6 en relación con la predicción de saltos y ramificaciones múltiples, importante de entender.

Es dable suponer que así como los programas están estructurados en secuencias de instrucciones consecutivas, y que de una secuencia se pasa a otra (o se vuelve al comienzo de la misma) por medio de instrucciones e salto condicional, que dan la dirección de la primer instrucción de dicha secuencia, lo mismo ocurre con lasµops-R en que se traducen las instrucciones. O sea que se ejecuta una secuencia de µops-R almacenada en el TC que corresponde a una secuencia de instrucciones del programa, y para localizar en el **TC** la secuencia deµops-R (correspondientes a otra secuencia de instrucciones) que se debe ejecutar a continuación, el **CRS** -que predice dinámicamente cuál será ella- proporciona la dirección que permite dicha localización, como se indica en la fig. .94.

Para "HT" existe un mecanismo para identificar líneas de los dos "threads".

Las *μops-R generadas por instrucciones complejas son aportadas por la ROM* vinculada al **TC** (fig. 1.94), por ser las mismas infrecuentes de aparecer en un programa.
También debe suponerse que cuando se direcciona al **TC** para localizar en una línea del mismo una próxima secuencia de 3 μops-R a ejecutar, éstas deben aparecer en las salidas del **TC**. Dichas μops-R (de igual longitud) irán a la etapa de Renombramiento de Registros (**RR**) de donde pasan según el orden originario a la "Cola de μops-R", junto con la dirección de la instrucción que los originó. Lo mismo ocurre con las μops-R generadas por la ROM.
Esta cola oficia de buffer intermediario entre los subsistemas que operan en orden y en desorden. Para "HT" este buffer se comparte mitad para cada "thread", pudiéndose identificar en esta cola las μops-R de cada "thread".
Si para 'HT" se debe acceder simultáneamente al **TC** para obtener líneas con μops-R de ambos "threads", durante un ciclo pedirá una línea de uno de ellos, y en el siguiente una línea del otro. Mientras uno de ellos esta detenido se podrá acceder durante ciclos sucesivos al **TC** para obtener líneas del otro. Por lo tanto el **TC** es un recurso compartido en 'HT", pudiendo un "thread" tener más líneas que el otro.
También es compartida la ROM de μops-R. Cuando del caché L2 llega una instrucción compleja, el **TC** le envía a la ROM el número que genera la **UPD** el cual es como la dirección de la ROM donde está la primera de una secuencia de μops-R, en que se traduce una instrucción compleja que llegó a la **UPD**. Para "HT" el hardware permite identificar a que "thread" corresponde dicha secuencia.
Si al direcionar el **TC** *hay un fallo* ("miss") se debe acceder a la jerarquía de memorias (en primer lugar al caché L2) para obtener dos líneas de instrucciones (64 bytes), las cuales serán traducidas por la **UPD** (de a una por vez) y enviadas como μops-R a la **UT** y también a la "Cola de μops-R".

Obsérvese que en esta arquitectura sólo se pierde tiempo en las traducciones cuando hay un fallo en el TC.

Antes de ir a la **UPD** los 64 bytes citados provenientes del caché L2, temporariamente se guardan en un Buffer L2 (**BL2**) hasta que puedan ser decodificadas las instrucciones. Para "HT" existen dos **BL2**, uno para cada "thread".
También existen dos **CRS** y dos buffers para instrucciones de retorno (de subrutina, por ejemplo).

En definitiva el **TC** mayormente provee las μops-R que se van a ejecutar próximamente, y cada vez que en el **TC** hay un fallo, la **UPD** traducirá en μops-R las instrucciones del **BL2** (*a razón de una por vez*) que irán al **TC** y también a la Cola de μops-R, salvo que aparezca alguna instrucción compleja. Esto es como en una UCP con caché: si hay un fallo, las instrucciones a ejecutar van al caché y a la UCP (en este caso dicha cola).

Si en "HT" hace falta decodificar instrucciones de los dos "threads", se sacan alternadamente secuencias de cada **BL2**.

El proceso de ejecución de μops-R se inicia con una asignación ("allocation") en la etapa **RR** que renombra los registros 80x86 indicados en las μops-R, que están en la "Cola de μops-R" en otros registros "alias". Para tal fin existen 128 registros para enteros y 128 para punto flotante, más buffers para reordenamiento y para operaciones load/store., que para "HT" son particionados en mitades, una para cada "thread", siempre identificables por el hardware del procesador, como en todos los casos.
En "HT" en la "Cola de μops-R" hay μops-R de los dos "threads", y con cada pulso reloj se alterna la asignación de recursos en la **RR**. Si un "thread" usa todos los recursos que tiene reservados, la asignación sigue para el otro.
Junto con la asignación se realiza el renombramiento de registros. Para ello existen dos "Tablas de Registros Alias", que indican para cada nueva μop-R que se va a ejecutar de cada "thread" en que registros "alias" estarán sus operandos (que pueden ser resultados de μops-R ejecutadas antes), y en qué registro "alias" irá el resultado.
Las secuencias de μops-R con sus registros renombrados pasan a dos colas": una para los provenientes de instrucciones load/store (que deben acceder a memoria), y otra para las que se tradujeron de las instrucciones que no ordenan acceder a memoria. Estas dos colas también pueden ser particionadas con la mitad de entradas para cada "thread". De estas colas salen μops-R hacia la **UPRE**, en forma alternada -una por "thread" con cada pulso reloj.

En la **UPRE** existen 5 "Lógicas Planificadoras" (**LP**) que determinan cuándo μops-R se pueden ejecutar. Cada **LP** tiene su propia cola de 8 a 12 μops-R que debe seleccionar para enviar a las **UE**. *Las LP no distinguen entre las μops-R de dos "threads"*. Las μops-R se seleccionan en función de la dependencia de datos y disponibilidad de **UE**. Para evitar problemas está limitado el número de entradas en uso que puede tener en cada cola un "thread".

Puesto que para cada μop-R se efectúa en una **UE** la operación que ordena con los datos apropiados y se deja el resultado en el destino indicado por ella; y dado que dichos datos están físicamente en registros que están definidos desde la etapa **RR**, al igual que el destino del resultado, y que para "HT" dichos recursos están separados para cada "thread", no hay problemas en encontrar los resultados que usarán las μops-R que se ejecutarán luego.
Asimismo las μops-R se pueden identificar, pues tienen asociado (incluso en la **UT**) un número que es la dirección en memoria de la instrucción originaria de la cual fueron traducidas.

Si bien en cada ciclo reloj las **LP** pueden hacer que las **UE** efectúen las operaciones de hasta 6 μops-R, como el máximo de μops-R que genera la **UT** es 3, *queda limitado a este número la cantidad de μops-R ejecutables por ciclo.*

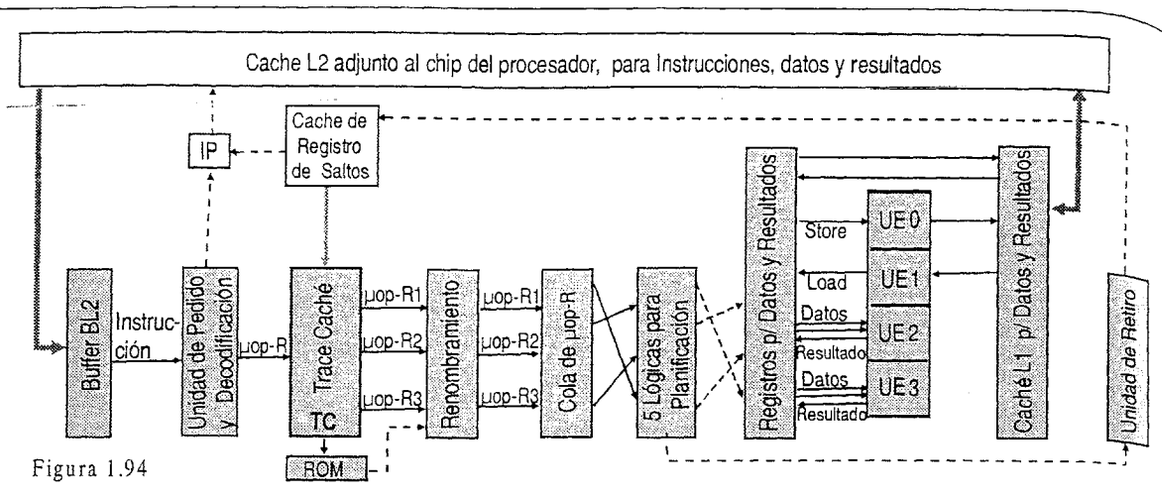


Figura 1.94

Existen 4 **UE**, siendo que las dos que contienen una UAL operan al doble de la frecuencia del procesador, o sea que en medio ciclo completan la operación, siendo por ello posibles hasta 6 operaciones por ciclo. Ellas son:
UE0 dedicada a µops-R "store", y **UE**1 para µops-R "load", a razón de una µop-R por ciclo.
UE2: en la primer mitad de un ciclo puede operar enteros en una UAL, o una instrucción de transferencia en punto flotante; y en la segunda mitad de dicho ciclo puede operar otra vez enteros en la UAL de dicha **UE**.
UE3: en la primer mitad de un ciclo puede operar enteros en una UAL, u operar (en el coprocesador) todas las operaciones aritméticas en punto flotante, pero no las de transferencia, o cualquier operación SIMD, o llevar a cabo una µop-R de salto; y en la segunda mitad de un ciclo puede operar nuevamente enteros en la UAL de dicha **UE**.

Para µops-R load/store existe un Buffer Conversor compartido en "HT" por los dos "threads", que es una memoria totalmente asociativa (definida al tratar cachés), que convierte direcciones a direcciones físicas de memoria.
Después de haberse realizado la operación ordenada por cada µop-R en la **UE** correspondiente, las µops-R pasan a un buffer de reordenamiento, que hace de intermediario entre la **UPRE** y la **UR**. Este buffer se parte en mitades si hay dos "threads".
La **UR** retira las µops-R en orden en forma alternada para cada "thread".
Luego que las µops-R ejecutadas se retiran, los resultados pasan desde registros hacia el caché de datos L1.
Este es un caché asociativo de 4 vías con líneas de 64 bytes, que siempre escribe "write-through" en el caché asociativo L2 de 8 vías con líneas de 64 bytes, con 128 bytes por línea. Este Xeon tiene un caché similar L3.
Los cachés L2 y L3 operan con direcciones físicas y el L1 con virtuales, pero con tags físicos.

De la descripción anterior resulta, que dos "threads" que se están ejecutando en paralelo comparten (en principio por partes iguales) la mayoría de los recursos físicos de un único procesador, ya sea por partición de los mismos o por alternancia (en un ciclo reloj uno y en el siguiente el otro)'y no se trata simplemente que se ejecuta un "thread" y cuando éste se detiene por algún motivo se pasa a ejecutar el otro.

Apéndice 1 de I

REPRESENTACIÓN DE NÚMEROS BINARIOS NATURALES Y OPERACIONES CON ELLOS

A1.1 SISTEMAS NUMÉRICOS POSICIONALES

¿Qué es un sistema numérico posicional ?

La necesidad de representar conjuntos de objetos ha llevado a las distintas culturas a adoptar diversas formas de simbolizar su valor numérico.
Una manera primera de representar el número de elementos que constituyen un cierto conjunto, es establecer una correspondencia con un número igual de símbolos.
Esto lo hacemos cuando contamos con los dedos, o si para representar, como ser, los días de la semana, dibujamos igual número de trazos: |||||||
Tal sistema de representación sería *"unario"*, pues se usa un solo tipo de símbolo. Su desventaja es que no permite simbolizar cómoda y rápidamente conjuntos con muchos elementos.

Cuando fue necesario designar la existencia de muchos elementos, se trató siempre de ***utilizar la menor cantidad de símbolos***, para lo cual se establecieron operaciones *implícitas* entre los mismos.
Los romanos utilizaron un sistema de signos de valor crecientes: **I, V, X, L, C, D, M**, etc.,
que se agrupaban de derecha a izquierda, sumándose o restándose entre sí, según siguieran o no el orden creciente:

CXVII = cien + diez + cinco + uno + uno
MCMV = mil + (mil – cien) + cinco

Esta codificación requería nuevos símbolos cuando se agotaban los de mayor valor, a la par que los cálculos por su complejidad convenía realizarlos con ábacos.
Fueron los pueblos orientales y americanos (mayas) los que desarrollaron los sistemas ***posicionales***, basados en un conjunto limitado y constante de símbolos, entre los cuales se encontraba el "cero", para indicar la ausencia de elementos. Miles de años antes, el ábaco, construido en la tierra o con madera fue el antecesor natural de estos sistemas, siendo que la ausencia de objetos en una posición o varilla implicaba de hecho el cero.

En estos sistemas, **cada símbolo**, además del número de elementos de un tipo que representa considerado aisladamente, **tiene un significado o peso distinto, según la posición que ocupa en el grupo** de caracteres del que forma parte.

De esta manera es posible representar sistemáticamente *cualquier* número, empleando en forma combinada un conjunto *limitado* de caracteres simbólicos.
Los caracteres se denominan ***"dígitos"***, y constituyen piezas de información digital (sección 1.10)

Relacionado con los diez dedos, el **sistema posicional decimal**, también denominado de *"base* o *raíz diez"* por utilizar diez símbolos (que forman la sucesión monótona creciente **0, 1, 2, 3, 4, 5, 6, 7, 8, 9**) permite representar cualquier número de elementos combinando dichos símbolos.

Este sistema servirá para comprender conceptualmente en qué consiste y cuál es la estructura de cualquier sistema numérico posicional.

Si bien en la práctica apreciamos mecánica e intuitivamente, sin pensar, la magnitud de un número decimal con solo visualizarlo, en realidad la cantidad de elementos que el número simboliza se determina sumando los productos que se obtienen de multiplicar el valor de las unidades que representa cada dígito por el *"peso"* (valor) de la posición que ocupa (unidades, decenas, etc.):

$9323 = 9x1000 + 3x100 + 2x10 + 3x1$ siendo: $10 = 1 \times 10 = 10^1$; $100 = 10 \times 10 = 10^2$; $1000 = 100 \times 10 = 10^3$

Resulta que **el peso de cada posición es el de la anterior multiplicada por la base** (diez, o sea el número de símbolos del sistema), *resultando así los pesos potencias crecientes de la base*. Esto es común a todos los sistemas posicionales. En caso de necesitarse representar números más grandes se usan siempre los mismos diez símbolos, y sólo es necesario agregar nuevas posiciones a la izquierda.
Un sistema numérico es una forma de fraccionar una totalidad en porciones, cuyos tamaños, a partir del valor uno, se van escalonando -a medida que se requieren nuevos tamaños- de forma tal que cada nuevo tamaño que se necesite es el anterior multiplicado por la cantidad de símbolos de la base.
El número máximo de porciones de cada tamaño lo determina el símbolo de mayor valor de la base en cuestión

Cuando leemos 109 personas ($109 = 1x100 + 0x10 + 9x1$) en relación a la totalidad de dichas personas, las mismas de hecho han sido divididas en forma *virtual*, de modo de conformar de manera única:
- un solo grupo de 100, (pudiendo existir hasta 9 grupos posibles de 100),
- con las restantes ($109 - 100 = 9$) no se puede formar ningún grupo de 10, y sí 9 grupos de una persona.

También puede pensarse en relación con el número 109, que en una balanza (figura A1.1) se debe pesar un objeto *de peso supuestamente desconocido*, pero que a los fines didácticos de simular su pesada debemos partir de que dicho peso es conocido (109 grs).

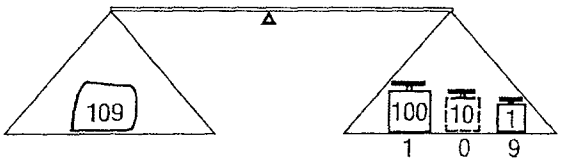


Figura A1.1

Los juegos de pesas a utilizar en base diez serán subconjuntos de 9 pesas múltiplos de diez:
- 9 pesas de **1** gramo
- 9 pesas de **10** gramos
- 9 pesas de **100** gramos
- 9 pesas de **1000** gramos . . . y así de seguido;

o sea que podemos tener más subconjuntos de 9 pesas que sean múltiplos de diez (teóricamente infinitos), según sea la magnitud del peso de los objetos que se quiere pesar.

Para pesar dicho objeto se usaría primero una pesa de 100 grs. (la de mayor peso posible para empezar a equilibrar, siendo que con dos pesas de ese tipo se excederían los 109 grs. a pesar: $100 + 100 = 200 > 109$). Luego si se prueba con una pesa de 10 grs. también se excederá el peso a ponderar: $100 + 10 = 110 > 109$). Finalmente agregando 9 pesas de 1 gr. se equilibra la balanza. En definitiva se usaron: **1** pesa de 100 grs, **0** pesas de 10 grs y **9** pesas de 1 gr. con lo cual el número 109 indica el peso del objeto en base diez. O sea:

100 10 1
1 0 9 = $1x100 + 0x10 + 1x9$

Es importante observar que tanto con las personas como con la balanza, se empieza siempre respectivamente por los grupos o pesas más grandes (de valor . . . 1000, 100, ...) que se puedan formar o emplear. De este modo se asegura que la representación simbólica es única.

¿Cuáles son las características de los sistemas numéricos posicionales?

Podemos sistematizar como sigue las características del sistema decimal, que también, como se tratará, *son comunes a todos los sistemas numéricos posicionales*, cualquiera sea su base (definida por el número de símbolos empleados para formar los números)

- Consta de un número finito de símbolos individuales distintos que constituyen la *"base o raiz"* del sistema (diez en el sistema decimal, dos en el binario, etc.).
- Cada símbolo aislado representa un número específico de *unidades*.
- Existe un símbolo (*cero*) para indicar la ausencia de elementos a representar.
- Los símbolos pueden ordenarse en forma monótona creciente.
- Formando parte de un número compuesto por varios símbolos, *un mismo símbolo tiene un significado o "peso" distinto según su posición relativa en el conjunto.*
- La posición extrema derecha corresponde a *unidades* (peso uno); a partir de ella, *cada posición tiene el peso de la que está a su derecha multiplicada por la base*, resultando siempre que *el peso de cada posición es una potencia de la base. Esta en todas las bases se simboliza* **10** ("uno cero").

A1.2 SISTEMAS NUMÉRICOS OCTAL BINARIO Y HEXADECIMAL

¿Qué símbolos se emplean en otras bases numéricas y cómo se representan números en ellas ?

A los fines de que resulte fácil simbolizar números en otros sistemas numéricos, los símbolos 0 y 1 existen en todos los sistemas con igual significado que en el decimal.
Si otros sistemas usan algunos o todos los símbolos decimales restantes del 2 al 9, se ha acordado que su significado es el mismo que en decimal. Así 7 representa siete unidades, ya sea en decimal, octal o hexadecimal.

El **sistema hexadecimal** tendrá **dieciséis** símbolos distintos que constituyen la base.
Del 0 al 9 coinciden en significado con los correspondientes decimales; para los seis restantes se crearon los símbolos de la **A** hasta la **F**, como aparecen en la figura A1.1 en correspondencia con sus equivalentes decimales y con los números de otros sistemas.

Binario	0	1	*10*														
Octal	0	1	2	3	4	5	6	7	*10*								
Decimal	0	1	2	3	4	5	6	7	8	9	*10*	11	12	13	14	15	16
Hexadecimal	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	*10*

Figura A1.2

Sistema en base ocho u octal

En base ocho los ocho símbolos van del 0 al 7 (fig. A1.2), con los cuales se puede formar cualquier número.
Con los mismos supuestos que planteamos para la pesada realizada en relación con la fig. A1.1, pesaremos (fig. A1.3) el mismo objeto cuyo peso en octal supondremos desconocido, siendo que en base diez pesa 109 grs.
Veamos cuáles son las pesas en base ocho. Si en base diez cada pesa en relación con la del tamaño anterior era diez veces más pesada, en octal lo será **ocho veces**. Juntando ocho pesas de un valor se construye una pesa del tamaño siguiente. Si para fines didácticos simbolizamos en base diez el peso de cada tipo de pesa octal, por lo que se tendría la serie de valores: **1** gr; 1 gr x 8d = **8d** grs; 8d grs x 8d = **64d** grs; 64d grs x 8d = **512d** grs., etc...
El símbolo "d" indica que se trata de base diez. El símbolo para representar 1 es el mismo en ambas bases.
De cada uno de dichos tamaños existen un total de 7 pesas octales (en base diez eran 9), siendo 7 el símbolo octal de mayor valor:
- 7 pesas de **1** gr.
- 7 pesas ocho veces mayores que la de 1 gr. (8d grs)
- 7 pesas dieciséis veces mayores que la de 1 gr. (**16**d gramos) y ocho veces mayor que el tamaño anterior.
- 7 pesas sesenta y cuatro veces mayores que la de 1 gr. (**64**d grs) y ocho veces mayor que el tamaño anterior.

y así de seguido, o sea que podemos tener más subconjuntos de 7 pesas que sean múltiplos de ocho (teóricamente infinitos), según sea la magnitud del peso de los objetos que se quiere pesar.

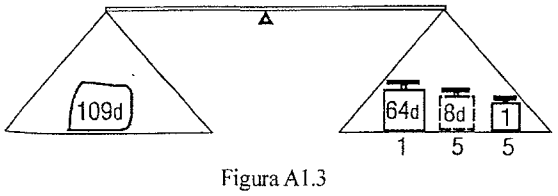


Figura A1.3

Volveremos a pesar el objeto que en base diez pesaba 109 grs.
Cuando se pesa un objeto en general se desconoce su peso, el cual hay que determinar. Para ello se requiere ir probando poniendo y sacando pesas (octales en este caso), hasta que con las pesas adecuadas la balanza alcance el equilibrio. La selección de las pesas adecuadas la simularemos haciendo cálculos simples en base diez, como se hizo en relación con la figura A1.1.

Suponiendo (fig. A1.3) que se empieza adecuadamente a equilibrar la balanza con **una** pesa sesenta y cuatro veces mayor que la de 1 gr en el platillo derecho. Pensando en base diez se habrán equilibrado 64d grs, por lo que faltará equilibrar 109 – 64 = 45d grs.
Si se coloca otra pesa del mismo tamaño que el anterior se tendría: 64d + 64d = 128d > 109d, y el peso del platillo derecho superaría al del izquierdo, por lo que no puede colocarse más que una pesa de dicho tamaño.

Probando ahora con pesas ocho veces mayores que la de 1 gr. si se colocan en el platillo derecho 5 pesas de 8d (lo cual suma 40d), de los 45d que faltaba balancear, ahora restarán balancear 45d – 40d = 5 grs., lo cual se realiza colocando 5 pesas de 1 gr.
En definitiva, en el platillo derecho se habrán colocado: **1** pesa sesenta y cuatro veces más pesada que 1 gr.; **5** pesas ocho veces más pesadas que las de 1 gr y **5** pesas de 1 gr. Esto quedaría simbolizado en octal (ó) como **155**.

100 10 1]d 64 8 1]d
Entonces 1 0 9d = 1 5 5ó (sobre los símbolos en base diez y base ocho se indican sus pesos en decimal)
(La igualdad anterior también implica que si en un plato de la balanza colocamos las pesas en base diez correspondientes al 109, y en el otro las pesas octales correspondientes al 155, obviamente se equilibrarán)

El mismo objeto que en la fig. A1.1 fue fraccionado artificialmente en porciones indicadas como 109d; en la fig. A1.3 lo hemos fraccionado también en forma virtual en porciones para simbolizar su peso en octal. Si a los fines de comparar lo efectuado en ambas casos expresamos el valor de las pesas octales en base diez, resulta que 155ó implica que se fraccionó dicho objeto en una porción de 64 grs, en 5 porciones de 8 grs. y en 5 porciones de 1 gr.

Asimismo, si pensamos en el conjunto de personas que en base diez fue dividido conforme indican los símbolos 109, dicha totalidad en base ocho fue dividida (pensando el tamaño de los grupos en base diez), en un grupo de 64, 5 grupos de 8, y 5 grupos de una persona, siendo que pueden existir hasta 7 grupos de cada tipo.

Nuevamente se verifica que un sistema numérico es una forma de fraccionar una totalidad en porciones, cuyos tamaños, a partir del valor uno, se van escalonando -a medida que se necesitan nuevos tamaños- de forma tal que cada nuevo tamaño que se necesite es el anterior multiplicado por la cantidad de símbolos de la base.
El número máximo de porciones de cada tamaño lo determina el símbolo de mayor valor de la base en cuestión.

¿Cómo se simbolizan los pesos en octal sin necesidad de estimarlos en base diez ?

Hasta acá nos hemos centrado en el objetivo de entender conceptualmente qué es un sistema numérico posicional, para lo cual por razones didácticas hemos valorado los pesos en base diez, mediante los cuales simbolizamos el peso de un objeto o de un número de elementos usando los símbolos en base ocho. Si bien se puso énfasis en que la balanza usaba pesas octales, y que el peso simbolizado 155 se podía lograr -sin conocer que es 109d- equilibrando la balanza con pesas octales, *de hecho los cálculos que hemos realizado en base diez para llegar al 155 implican un pasaje de base diez a octal, método que seguiremos usando.*
A fin de ratificar que cada sistema de numeración no surge del sistema decimal, sino que es independiente, como lo sugiere claramente la posibilidad de pesar usando pesas octales, expresaremos en octal el valor de las pesas o pesos de este sistema. Simplemente (fig. A1.4.a) si ponemos en el plato izquierdo de la balanza una pesa ocho veces mayor que 1 gr. (**8** grs. en base diez) para equilibrarlo hace falta una sola pesa ocho veces mayor que 1 gr, y ninguna (cero) pesa de 1 gr., o sea que esta pesa o peso en octal se simboliza **10** (léase uno-cero, y no "diez").

Conforme con esto, en la fig. A1.2 luego del símbolo mayor 7 sigue el **10** (1 0ó = 8d), así como en base diez después del 9 sigue el 10. Esto es general: **en cada base al símbolo más alto en valor le sigue en valor el 10.**
Del mismo modo (fig. A1.4.b), si en el plato izquierdo se coloca una pesa sesenta y cuatro veces mayor que 1 gr se equilibra con una sola de ese tamaño y ninguna de los tamaños menores, por lo que en octal este peso se simboliza 100ó (léase uno-cero-cero, y no "cien"). Igualmente una pesa 256 veces mayor que 1 gr se equilibra con solo una de ese tamaño, y ninguna de los tamaños subsiguientes, por lo que en octal este peso es **1000ó**, etc.

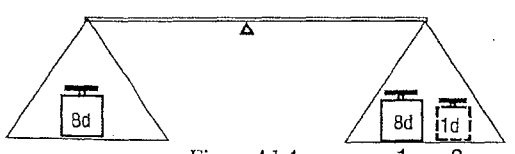


Figura A1.4.a 1 0

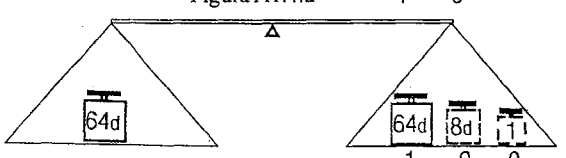


Figura A1.4.b 1 0 0

Por lo tanto, en octal las pesas se simbolizan 1, 10, 100, 1000 . . . (estos mismos símbolos representan los valores de las pesas que se usan en base diez, aunque en octal representan 1, **8**, 64 y 512, respectivamente).
O sea 100% en octal: 100 10 1]ó

1 5 5ó = 1x100 + 5x10 + 5x1]ó

Esta cuenta (que indica una pesa de 100, más 5 de 10, más una de 1) realizada en octal daría, obviamente, 155ó.
La apreciación de los pesos de 155 ó en base diez era 64 8 1 1 5 5ó = =[1x64 + 5x8 + 5x1]d = 109d ; esto es, pensado en base diez se usaron una pesa de 64 más 5 de 8, más 5 de 1, suma que da 109d.

Al apreciar los pesos en base diez en esencia se está pasando de octal a base diez, siendo que el significado de los símbolos 1 y 5 es el mismo en ambas bases. **Este método será el usado para pasar de cualquier base a decimal.**

Ejercicios:
Con la balanza con pesas octales pesar un objeto que en base diez pesa 100d. *Respuesta:* 144ó
Idem otro que pesa 120d *Respuesta:* 170ó

Sistema en base dos o binario

En base dos los símbolos con los cuales se puede formar cualquier número son 0 y 1 (figura A1.2),
Con los mismos supuestos que planteamos para la pesada realizada en relación con la fig. A1.1, pesaremos (fig. A1.5) el mismo objeto cuyo peso en binario se quiere determinar, siendo que en base diez pesa 109 grs.
Veamos cuáles son las pesas en base dos. Si cada pesa en base diez en relación con la del tamaño anterior era diez veces más pesada, en binario lo será **dos veces**: juntando dos pesas de un valor se construye una pesa del tamaño siguiente. Si para fines didácticos simbolizamos en base diez el peso de cada tipo de pesa binario, se tendría la serie de valores: **1** gr; **2**d grs; **4**d grs; **8**d grs; **32**d grs; **64**d grs; **128**d grs; **256**d grs; **512**d grs., etc...
El símbolo "d" indica que se trata de base diez. El símbolo para representar 1 gr. es el mismo en ambas bases.
De cada uno de dichos tamaños existe ***una sola pesa*** (en base diez eran 9), siendo 1 el mayor símbolo binario:
- 1 pesa de **1** gramo
- 1 pesa dos veces mayor que la de 1 gr. (2d)
- 1 pesa cuatro veces mayor que la de 1 gr. (4d)
- 1 pesa ocho veces mayor que la de 1 gr. (8d)

y así de seguido, o sea que podemos tener más pesas, cada una múltiplo de dos (teóricamente infinitas), según sea la magnitud del peso de los objetos que se quiere pesar.

La pesa de cada valor se usa (**1**), o no se usa (**0**)

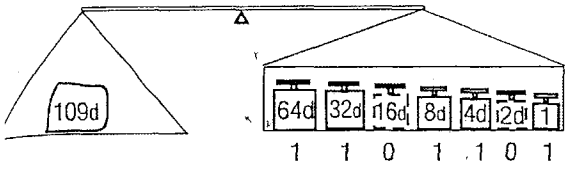


Figura A1.5

Pesaremos otra vez el objeto que en decimal pesaba 109 grs. La selección de las pesas adecuadas la simularemos haciendo cálculos simples en base diez, como se hizo en relación con las figuras A1.1 y A1.3.
Suponiendo (fig. A1.5) que se empieza adecuadamente a equilibrar la balanza colocando en el platillo derecho la pesa 64 veces más pesada que la de 1 gr, pensando en base diez se habrán equilibrado 64d grs, por lo que faltará equilibrar 109 – 64 = 45d grs.
Probando equilibrar agregando en el platillo derecho la pesa de tamaño menor siguiente, 32 veces mayor que la de 1gr., de los 45d que faltaba balancear, ahora restarán balancear 45d – 32d = 13 grs. Si se prueba colocar la pesa 16 veces mayor que la de 1gr, sumando en base diez resultaría (64 + 32 + 16)d = 112d > 109d, con lo cual el peso del platillo derecho superaría al del izquierdo, por lo que no puede colocarse la pesa 16 veces mayor que la de 1 gr. Esto se simboliza con un cero en relación con esa pesa 16 veces mayor.. Colocando la pesa 8 veces mayor que la de 1 gr. resultan balanceados (64 + 32 + 8)d = 104d grs., faltando ahora balancear (13 – 8)d = 5d grs. Agregando la pesa siguiente, 4 veces mayor que 1 gr. faltaría balancear (5 – 4)d = 1d gr., por lo que no se puede colocar (cero) la pesa 2 veces mayor que la de 1 gr., y sí debe agregarse la pesa de 1 gr.
En definitiva, en el platillo derecho se habrán colocado: **1** pesa binaria 64 veces más pesada que la de 1 gr.; **1** pesa binaria 32 veces más pesada que la de 1 gr.; **0** pesa binaria 16 veces más pesada que la de 1 gr.; **1** pesa binaria 8 veces más pesada que la de 1 gr.; **1** pesa binaria 4 veces más pesada que la de 1 gr.; **0** pesa binaria 2 veces más pesada que la de 1 gr.; **1** pesa binaria de 1 gr. Esto quedaría simbolizado binario (b) como **1101101b**.

100 10 1]d **64 32 16 8 4 2 1**]d
Entonces 1 0 9d = 1 1 0 1 1 0 1 b (sobre los símbolos en ambas bases se indican sus pesos en decimal)

El mismo objeto que en la fig. A1.1 fue fraccionado artificialmente en porciones indicadas como 109d; en la fig. A1.5 lo hemos fraccionado también en forma virtual en las porciones indicadas para simbolizar su peso en binario.
Asimismo, si pensamos en el conjunto de personas que en base diez fue dividido conforme indican los símbolos 109, dicha totalidad en base dos fue dividida (pensando el tamaño de los grupos en base diez), en un grupo de 64, un grupo de 32; entre los restantes no se pudo formar ningún grupo de 16, sí un grupo de 8 y otro de 4; entre los restantes no se pudo formar ningún grupo de 2, y si un grupo de una persona.

Nuevamente se verifica que un sistema numérico es una forma de fraccionar una totalidad en porciones, cuyos tamaños, a partir del valor uno, se van escalonando -a medida que se necesitan nuevos tamaños- de forma tal que cada nuevo tamaño que se necesite es el anterior multiplicado por la cantidad de símbolos de la base.

¿Qué son los bits y bytes ?

El numero binario 1101101 consta de 7 dígitos binarios. En inglés serían 7 **b**inary digi**t**s, o sea 7 **bits**.
O sea que **un bit** es un dígito binario, por lo que es un símbolo que puede valer 0 ó 1.
Un **byte** es un conjunto de 8 bits. El tamaño de las memorias se mide en bytes. En el interior de un procesador se opera con múltiplos pares de un byte: 2, 4, 8, 6 bytes (16, 32, 64, 128 bits)

¿Cómo se simbolizan los pesos en binario sin necesidad de estimarlos en base diez ?

Nuevamente por razones didácticas hemos estimado los valores de las pesas binarias en base diez.
Si bien se planteó que la balanza usaba pesas binarias, y que el peso simbolizado 1101101b se podía lograr -sin

conocer que es 109d- equilibrando la balanza con pesas binarias, *de hecho los cálculos que hemos realizado en base diez para llegar al* 1101101 *implican un pasaje de base diez a binario, método que seguiremos usando.*

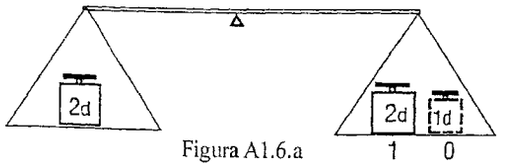


Figura A1.6.a 1 0

A fin de ratificar que cada sistema de numeración no surge del sistema decimal, sino que es independiente, como lo sugiere claramente la posibilidad de pesar usando pesas binarias, expresaremos en binario el valor de cada pesa de este sistema. Simplemente (figura A1.6.a) si ponemos en el plato izquierdo de la balanza una pesa binaria dos veces mayor que la de 1 gr. (2d grs.) para equilibrarla se utiliza (**1**) la pesa de dicho tamaño y no se usa (**0**) la pesa de 1 gr., o sea que esta pesa en binario se simboliza **10** (léase uno-cero, y no "diez").

Conforme con esto, en la fig. A1.2 luego del símbolo mayor **1** sigue el **10** (1 0ó $_{2\ 1}$ = 2d), así como en base diez después del 9 sigue el 10, y en base ocho al 7 le sigue 10.

Del mismo modo (fig. A1.6.b), si en el plato izquierdo se coloca un cuerpo cuatro veces el peso de 1 gr. (4d grs. en base diez), se equilibra usando (**1**) la pesa binaria cuatro veces mayor que la de 1 gr., sin usar (**0**) la pesa dos veces mayor que la de 1 gr., y sin usar (0) la pesa de 1 gr., por lo que en binario este peso se simboliza **100**b (léase uno-cero-cero, y no "cien"). Igualmente un objeto ocho veces mayor que 1 gr (8d grs.) se equilibra con la pesa binaria de ese tamaño, y ninguna de los tamaños subsiguientes, por lo que en binario este peso es **1000**ó, etc.

Por lo tanto, en binario las pesas se simbolizan 1, 10, 100, 1000 . . . (estos mismos símbolos representan los valores de las pesas que se usan en base diez, aunque en binario representan 1, 2, 4 y 8, respectivamente).

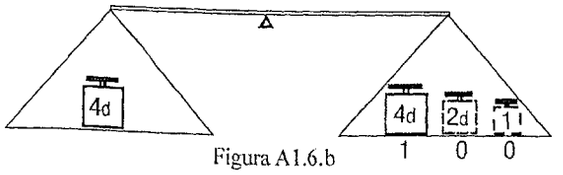


Figura A1.6.b 1 0 0

1000000 100000 10000 1000 100 10 1]b
1 1 0 1 1 0 1 b = (1x1000000 + 1x100000 + 0x10000 + +1x1000 + 1x100 + 0x10 + 1x1)b = 1101101b

Esta cuenta (que indica sumar: la pesa de 1000000, la de 100000, no la de 10000, la de 1000, la de 1000, la de 100, no la de 10 y sí la de 1 gr) realizada en binario daría, obviamente, 1101101b.

64 32 16 8 4 2 1
Los pesos de 1101101b en base diez eran 1 1 0 1 1 0 1 b = [1x64+1x32+0x16+1x8+1x4+0x2+1x1]d = 109d ; esto es, pensado en base diez se usaron la pesa de 64 más la de 32, más la de 8, más la de 4, más la de 1, suma que da 109d. Otra vez vemos que al valorar los pesos de una base en base diez, de hecho estamos pasando de esa base a base diez.

Ejercicios:

Con la balanza con pesas binarias pesar un objeto que en base diez pesa 100d. *Respuesta:* 1100100b
Idem otro que pesa 124d *Respuesta:* 1111100ó

¿Cuál son los números binarios naturales extremos representables en n bits expresados en decimal ?

Vamos a partir de n=4 bits, para luego generalizar para cualquier número **n** de bits.

El valor mínimo es NMIN = 0000b = 0d; y el máximo NMAX = 1111b = 15d = $16 - 1 = 2^4 - 1$

En general para **n** bits

NMIN = $\underbrace{00\ldots.00}_{\text{n bits}}$b = 0d; y el máximo NMAX = $\underbrace{11\ldots.11}_{\text{n bits}}$b = $(2^n - 1)$d (el número que sigue a NMAX menos uno)

Sistema en base dieciseis o hexadecimal ("hexa")

Si bien la representación de la información en el interior de un computador sólo puede ser simbolizada en binario, ello implica leer o escribir largas sucesiones de ceros y unos (16, 32, 64). La utilidad del hexadecimal reside en que, como se verá, por un lado resulta sencillo pasar de binario a hexa y viceversa; y por otro, en que para representar un mismo número, se requiere cuatro veces menos símbolos en hexa que en binario.

En hexa los dieciséis símbolos van del 0 a F (fig. A1.2), con los cuales se puede formar cualquier número. Mientras que en base diez para indicar del 10 al 15 se necesitan dos símbolos, en hexa se usa uno solo.

Con los mismos supuestos que planteamos para la pesada realizada en relación con la fig. A1.1, pesaremos (fig. A1.3) el mismo objeto cuyo peso en hexa supondremos desconocido, siendo que en base diez pesa 109 grs.

Veamos cuáles son las pesas en hexa. Si cada pesa en relación con la del tamaño anterior en base diez era diez veces más pesada, en hexa lo será **dieciséis veces**: juntando dieciséis pesas de un valor se construye una pesa del tamaño siguiente. Si para fines didácticos simbolizamos en base diez el peso de cada tipo de pesa hexadecimal se tendría la serie de valores: **1** gr; 1gr x16d = **16**d grs; 16d grs x16d = **256**d grs; 256d grs x16d = **4096**d grs., etc.

El símbolo "d" indica que se trata de base diez. El símbolo para representar 1 gr. es el mismo en ambas bases.

De cada uno de dichos tamaños existen un total de **F** *pesas hexadecimales* (en base diez eran 9), siendo F =15d el símbolo hexadecimal de mayor valor:

- F pesas de **1** gr.
- F pesas dieciséis veces mayor que la de 1 gr. (16d grs)
- F pesas docientoscincuenta y seis veces mayor que la de 1 gr. (256d grs) y dieciséis veces el tamaño anterior
- F pesas mil veinticuatro veces mayor que la de 1 gr. (1024d grs) y dieciséis veces el tamaño anterior.

y así de seguido, o sea que podemos tener más subconjuntos de F pesas que sean múltiplos de dieciséis (teóricamente infinitos), según sea la magnitud del peso de los objetos que se quiere pesar.

Otra vez pesaremos el objeto que en base diez pesaba 109 grs. La selección de las pesas adecuadas la simularemos haciendo cálculos simples en base diez, como se hizo en relación con las figuras A1.1 y A1.3.

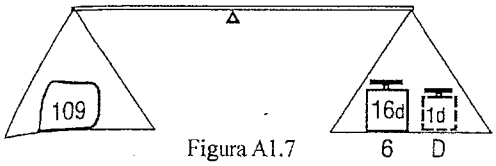


Figura A1.7 6 D

Suponiendo (fig. A1.7) que se empieza adecuadamente a equilibrar la balanza con las pesas dieciséis veces más pesadas que 1 gr., se podrán colocar hasta 6 pesas de este peso, con lo cual, calculando en base diez, habremos equilibrado 6x16 = 96d grs., faltando equilibrar 109 – 96 = 13d grs. El equilibrio de los platillos se logra agregando 13d = Dh pesas del tamaño menor siguiente, que es de 1 gr. Por consiguiente, el peso del objeto en hexa es **6D**. O sea: $109_d = 155_o = 1101101_b = 6Dh$

Asimismo, si pensamos en el conjunto de personas que en base diez fue dividido conforme indican los símbolos 109, dicha totalidad en base dieciséis fue dividida (pensando el tamaño de los grupos en base diez), en 6 grupos de 16, y 13 grupos de una persona, siendo que pueden existir hasta F = 15 grupos de cada tipo.

¿Cómo se simbolizan los pesos en hexadecimal sin necesidad de estimarlos en base diez ?

Al igual que cualquier otro sistema, el hexadecimal no surge del sistema decimal, sino que es independiente, como así lo sugiere la posibilidad de pesar el objeto en cuestión usando pesas hexadecimales, pudiéndose simbolizar en hexa el valor de las pesas o pesos de este sistema. Simplemente (fig. A1.8) si ponemos en el plato izquierdo de la balanza un peso dieciséis veces mayor que 1 gr. (16 grs. en base diez) para equilibrarlo hace falta una sola pesa dieciséis veces mayor que 1 gr, y ninguna (cero) pesa de 1 gr., o sea que esta pesa o peso en hexa se simboliza **10** (léase uno-cero, y no "diez").

Conforme con esto, en la fig. A1.2 luego del símbolo mayor F sigue el **10** ($1\,0_h = 16_d$), así como en base diez después del 9 sigue el 10, en octal después del 7 sigue 10, y en binario después del 1 sigue 10.

Del mismo modo si en el plato izquierdo se colocaría un peso docientos cincuenta y seis veces mayor que 1 gr se equilibra con una sola de peso de ese tamaño y ninguna de los tamaños menores, por lo que en hexa este peso se simboliza 100h (léase uno-cero-cero, y no "cien").

Por lo tanto, también en hexa las pesas se simbolizan 1, 10, 100, 1000 . . . (estos mismos símbolos representan los valores de las pesas que se usan en base diez, aunque en hexa representan 1, 16, 256, y 4096, respectivamente).

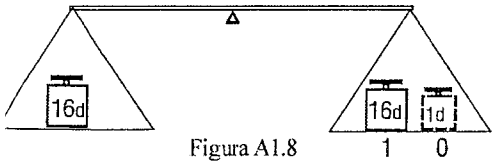


Figura A1.8 1 0

O sea 100% en hexa: $6\,D_h = 6x10 + Dx1]_h$

Esta cuenta (que indica 6 pesas de 100, más D pesas de 1) realizada en hexadecimal daría, obviamente, 6Dh.

Con los pesos de 6D h en base diez es: $6\,D_h = [6x16 + 13x1]_d = 109_d$; esto es, pensado en base diez se usaron una pesa de 64 más 5 de 8, más 5 de 1, suma que da 109_d.

Como ya se dijo, al valorar los pesos de una base en base diez en esencia se está pasando de dicha base a base diez.

Ejercicios:

Con la balanza con pesas hexadecimales pesar un objeto que en base diez pesa 112d. *Respuesta:* 70h

Idem otro que pesa 100d *Respuesta:* 64h

¿Cómo se halla fácilmente el siguiente de cada número en una base ?

El cuenta vueltas que indica los kms. recorridos por un auto consta de ruedas con los símbolos del 0 al 9. Cada rueda al cambiar de 9 a 0 obliga a la que está a su izquierda a avanzar una posición. La rueda de las unidades progresa una unidad merced a una acción exterior para que las otras puedan cambiar, si así debe ocurrir.

Suponiendo que el número que está frente al visor es 4588, si la rueda de las unidades avanza un símbolo, pasará de 8 a 9 sin afectar la rueda de las decenas, por lo que el número siguiente es 4589. Cuando las unidades vuelvan a aumentar uno, ahora pasarán de 9 a 0, lo que hará que las decenas también progresen uno, de 8 a 9, sin afectar las centenas. Por lo tanto, el número que sigue será 4590. Con las mismas consideraciones, si las unidades siguen aumentando uno, sucesivamente se tendrá: 4591, 4592, . . . 4599. Luego de éste las unidades pasan de 9 a 0, lo que hace que las decenas también pasen de 9 a 0, lo cual a su vez obliga que las centenas cambien de 5 a 6. Así se formará el 4600, y así de seguido.

Cuenta vueltas hexadecimales y binarios nos permitirán hallar fácilmente el número que sigue a otro dado.
En hexadecimal cada rueda tiene dieciséis símbolos (0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F), y cuando cambia de F a 0 hace cambiar la rueda que está a su izquierda. Salvo que hay 16 símbolos en cada rueda en vez de diez, todo es igual al cuenta vueltas anterior.
Asumiendo que el cuenta vueltas hexadecimal indica 3899, si la rueda de las unidades avanza un símbolo, pasará de 9 a A, sin hacer cambiar la rueda que está a su izquierda, por lo que el número hexadecimal siguiente es 389A. Luego, cada vez que las unidades aumentan en uno, sucesivamente se irán formando 389B, 389C, 389D, 389E, 389F. Con el siguiente cambio de la rueda de las unidades, ésta pasará de F a 0, con lo cual la rueda que está a su izquierda cambiará de 9 a A, sin afectar a la rueda que está a su izquierda. Por lo tanto, el número que sigue es 38A0, y así sucesivamente.
Suponiendo que el cuenta vueltas progrese hasta 3FFF, con el siguiente avance en uno de la rueda de las unidades ésta pasará de F a 0, forzando que la rueda que está a su izquierda también cambie de F a 0, lo que a su vez también hace que su rueda vecina izquierda pase de F a 0, lo que a su vez hará que la rueda que está a su izquierda cambie de 3 a 4. En definitiva de 3FFF se pasa a 4000.

Podemos imaginar un **cuenta vueltas binario** constituido por ruedas que sólo tienen dos símbolos: **0** y **1**, cada uno ocupando una mitad de cada rueda.. Cuando una rueda pasa de 1 a 0 obliga a la que está a su izquierda que gire media vuelta para que pase a 1 si estaba en 0, ó que pase a 0 si estaba en 1.
Suponiendo que este cuenta vueltas indique 1010, si la rueda extrema derecha de las unidades avanza uno pasará de 0 a 1, sin afectar a la rueda que está a su izquierda, por lo que el número binario que sigue será 1011. Cuando la rueda de las unidades vuelva a cambiar, esta vez de 1 a 0, hará que la rueda vecina izquierda que estaba en 1 pase a 0,. Esto a su vez obliga a que la rueda vecina izquierda que estaba en 0 cambie a uno, lo cual no afectará a la rueda vecina izquierda de la misma. Entonces a 1011 sigue 1100, etc.
En la fig. 1.4 puede verificarse con este método la generación de la sucesión de binarios del 0000 al 1111.

¿Cuántos bits se necesitan por cada dígito decimal a representar ?

El número decimal 109 de 3 dígitos, en binario es 1101101 de 7 bits. Si consideramos que 109 es un poco mayor que el 99 de dos dígitos, resultan aproximadamente 7/2 = 3,5 bits por cada dígito decimal.
512d = 1000000000b o sea 10bits para representar 3 dígitos decimales: 10/3 = 3,3 bits por dígito decimal.
Con 2 bits se forman $4 = 2^2$ combinaciones binarios (00 01 10 11). Con 3 bits se forman $8 = 2^3$ combinaciones binarios (000 001 010 011 100 101 110 111). En la figura 1.4 de la sección 1.2 aparecen las $16 = 2^4$ combinaciones binarios que se forman con 4 bits.

O sea, que el exponente de dos indica la cantidad de bits que se necesitan para formar un número de combinaciones que es potencia de dos, siendo que si el exponente aumenta en uno, el número de combinaciones se duplica.

Es importante recordar siempre que $2^{10} = 1024$, número cercano a $1000 = 10^3$ (1K)
El exponente diez de dos indica que con 10 bits pueden formarse 1024 números o combinaciones binarias distintas (de 0000000000 a 1111111111b = 1023d), número cercano a 1000d que es el número de números o combinaciones que en base diez pue en el platillo derecho den formarse con 3 dígitos decimales (de 000 a 999), siendo 3 el exponente de diez.

Por lo tanto, un número aproximadamente igual de combinaciones distintas se forman con 10 bits en binario, y con 3 dígitos en decimal. O sea unos 10 bits por cada 3 dígitos decimales, lo que da 10/3 = 3,33 bits por cada dígito decimal.

Ejercicio:
Cuántos bits se necesitan para formar 10^6 (1Mega) números distintos. *Respuesta* 6x3,3 3= 19,98 ~ 20 bits
Otra forma de hacerlo: dado que 2^{10}, se tiene $10^6 = 10^3 \times 10^3 \sim 2^{10} \times 2^{10} = 2^{20}$ El exponente indica que hacen falta 20 bits.

A1.3 CONVERSION ENTRE BASES

Conversión de una base cualquiera a base diez

Se trata de una metodología que de hecho hemos realizado anteriormente cuando evaluamos en base diez los valores de las pesas con las cuales formamos números en otras bases. Así, hemos realizado:

64 8 1
1 5 5 ó = [1x64 + 5x8 + 5x1]d = 109d

64 32 16 8 4 2 1 d
1 1 0 1 1 0 1 b = [1x64+1x32+0x16+1x8+1x4+0x2+1x1]d = 109d

16 1
6 D h = [6x16 + 13x1]d = 109d

Cuenta vueltas hexadecimales y binarios nos permitirán hallar fácilmente el número que sigue a otro dado.
En hexadecimal cada rueda tiene dieciséis símbolos (0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F), y cuando cambia de F a 0 hace cambiar la rueda que está a su izquierda. Salvo que hay 16 símbolos en cada rueda en vez de diez, todo es igual al cuenta vueltas anterior.
Asumiendo que el cuenta vueltas hexadecimal indica 3899, si la rueda de las unidades avanza un símbolo, pasará de 9 a A, sin hacer cambiar la rueda que está a su izquierda, por lo que el número hexadecimal siguiente es 389A. Luego, cada vez que las unidades aumentan en uno, sucesivamente se irán formando 389B, 389C, 389D, 389E, 389F. Con el siguiente cambio de la rueda de las unidades, ésta pasará de F a 0, con lo cual la rueda que está a su izquierda cambiará de 9 a A, sin afectar a la rueda que está a su izquierda. Por lo tanto, el número que sigue es 38A0, y así sucesivamente.
Suponiendo que el cuenta vueltas progrese hasta 3FFF, con el siguiente avance en uno de la rueda de las unidades ésta pasará de F a 0, forzando que la rueda que está a su izquierda también cambie de F a 0, lo que a su vez también hace que su rueda vecina izquierda pase de F a 0, lo que a su vez hará que la rueda que está a su izquierda cambie de 3 a 4. En definitiva de 3FFF se pasa a 4000.

Podemos imaginar un **cuenta vueltas binario** constituido por ruedas que sólo tienen dos símbolos: **0** y **1**, cada uno ocupando una mitad de cada rueda.. Cuando una rueda pasa de 1 a 0 obliga a la que está a su izquierda que gire media vuelta para que pase a 1 si estaba en 0, ó que pase a 0 si estaba en 1.
Suponiendo que este cuenta vueltas indique 1010, si la rueda extrema derecha de las unidades avanza uno pasará de 0 a 1, sin afectar a la rueda que está a su izquierda, por lo que el número binario que sigue será 1011. Cuando la rueda de las unidades vuelva a cambiar, esta vez de 1 a 0, hará que la rueda vecina izquierda que estaba en 1 pase a 0,. Esto a su vez obliga a que la rueda vecina izquierda que estaba en 0 cambie a uno, lo cual no afectará a la rueda vecina izquierda de la misma. Entonces a 1011 sigue 1100, etc.
En la fig. 1.4 puede verificarse con este método la generación de la sucesión de binarios del 0000 al 1111.

¿Cuántos bits se necesitan por cada dígito decimal a representar ?

El número decimal 109 de 3 dígitos, en binario es 1101101 de 7 bits. Si consideramos que 109 es un poco mayor que el 99 de dos dígitos, resultan aproximadamente 7/2 = 3,5 bits por cada dígito decimal.
512d = 1000000000b o sea 10bits para representar 3 dígitos decimales: 10/3 = 3,3 bits por dígito decimal.
Con 2 bits se forman $4 = 2^2$ combinaciones binarios (00 01 10 11). Con 3 bits se forman $8 = 2^3$ combinaciones binarios (000 001 010 011 100 101 110 111). En la figura 1.4 de la sección 1.2 aparecen las $16 = 2^4$ combinaciones binarios que se forman con 4 bits.

O sea, que el exponente de dos indica la cantidad de bits que se necesitan para formar un número de combinaciones que es potencia de dos, siendo que si el exponente aumenta en uno, el número de combinaciones se duplica.

Es importante recordar siempre que $2^{10} = 1024$, número cercano a $1000 = 10^3$ (1K)
El exponente diez de dos indica que con 10 bits pueden formarse 1024 números o combinaciones binarias distintas (de 0000000000 a 1111111111b = 1023d), número cercano a 1000d que es el número de números o combinaciones que en base diez pue en el platillo derecho den formarse con 3 dígitos decimales (de 000 a 999), siendo 3 el exponente de diez.

Por lo tanto, un número aproximadamente igual de combinaciones distintas se forman con 10 bits en binario, y con 3 dígitos en decimal. O sea unos 10 bits por cada 3 dígitos decimales, lo que da 10/3 = **3,33 bits** por cada dígito decimal.

Ejercicio:
Cuántos bits se necesitan para formar 10^6 (1Mega) números distintos. *Respuesta* 6x3,3 3= 19,98 ~ 20 bits
Otra forma de hacerlo: dado que 2^{10}, se tiene $10^6 = 10^3 \times 10^3 \sim 2^{10} \times 2^{10} = 2^{20}$ El exponente indica que hacen falta 20 bits.

A1.3 CONVERSION ENTRE BASES

Conversión de una base cualquiera a base diez

Se trata de una metodología que de hecho hemos realizado anteriormente cuando evaluamos en base diez los valores de las pesas con las cuales formamos números en otras bases. Así, hemos realizado:

64 8 1
1 5 5 ø = [1x64 + 5x8 + 5x1]d = 109d

64 32 16 8 4 2 1 d
1 1 0 1 1 0 1 b = [1x64+1x32+0x16+1x8+1x4+0x2+1x1]d = 109d

16 1
6 D h = [6x16 + 13x1]d = 109d

Regla:

1. Escribir sobre cada posición su peso en decimal
2. Sumar los productos del peso decimal de cada posición por el símbolo que aparece en ella (en hexa se debe pasar los símbolos A, B, C, D, E, F a base diez). El resultado de esta suma será el número decimal buscado.

Ejercicio: Convertir a decimal el número binario 10000 *Respuesta:* 16d
Convertir a decimal los números hexadecimales 109 y 100A *Respuesta:* 265d y 5006d
Convertir a decimal el número octal 1037 *Respuesta:* 543d

Conversión de base diez a otra base cualquiera por el "método de las pesas"

Como se puso de relieve anteriormente, el hecho de pesar con pesas de otras bases objetos cuyo peso es conocido en base diez (figuras. A1.3, A1.5, A1.7) permite determinar en octal, binario, etc. los símbolos que representan en estas bases el peso de dichos objetos.

O sea que el método de simular que se pesa un objeto, cuyo peso se conoce en base diez, con pesas de una cierta base cuya magnitud se estima en base diez **permite convertir un número en base diez a otra cualquiera.**

Regla para pasar un número decimal a binario: (no se requiere realizar dibujo alguno)

a. Dado el número a convertir, se parte de la pesa binaria que en base diez tiene un valor igual a dicho número, o que presenta el valor menor más próximo al mismo; y a partir de este valor se escriben en base diez los sucesivos valores decrecientes de las pesas binarias hasta el valor uno, siendo cada valor la mitad del anterior.

b. Se coloca un **uno** debajo del primer valor determinado en el paso anterior. A este valor decimal se le sumará el valor decimal de cada peso binario que sigue a la derecha *sin omitir ninguno* hasta el peso uno. Si al sumar el valor de la pesa que sigue a la última que se analizó el resultado alcanzado iguala o es menor que el número decimal a convertir, se colocará un uno debajo del valor de esa pesa; y si ese resultado supera dicho número se coloca un cero, para indicar que esa pesa no se usa para balancear.

c. Los unos y ceros así determinados de izquierda a derecha son los bits del número binario buscado.

EJEMPLO: Convertir el número 284d a binario

a. => **256 128 64 32 16 8 4 2 1 d**
b. => 1 0 0 0 1 1 1 0 0 b c. 284d = 100011100d

En el paso a. se comenzó con la pesa de valor 256, que es la menor en relación con 284, siendo que la de 512 lo supera. El paso b. comienza escribiendo un uno debajo de 256. Sumando el valor 128 que le sigue daría 256 + 128 = 384 > 284, por lo que se coloca 0 debajo de 128. Lo mismo ocurre si intentamos sumar al 256 ya sea 64 ó 32, por lo que también escribimos un cero debajo del 64 y del 32, para indicar que no se han usado estas pesas. Con el peso 16 resulta 256 + 16 = 272 < 284 (siendo que aún faltan equilibrar 284 – 272 = 12), por lo que escribimos un uno debajo del 16. Con el peso 8 resulta 272 + 8 = 280 < 284 por lo que también se escribe un uno bajo del 8. Restan equilibrar 284 – 280 = 4, lo cual se consigue con la pesa de ese valor, escribiéndose un uno debajo del 4. Dado que se ha equilibrado el 284 con las pesas indicadas con un uno, no se usarán las pesas de 2 y 1, colocándose un cero debajo de cada uno de esos valores.

Verificación: siempre es factible determinar si el resultado de una conversión está bien, realizando el pasaje inverso a base diez del número binario hallado, según la regla antes indicada:

256 128 64 32 16 8 4 2 1 d
1 0 0 0 1 1 1 0 0 b = (1x256 + 1x16 + 1x8 + 1x4)d = 284d Se verifica que la conversión fue bien hecha.

Ejercicio Convertir 100d a binario *Respuesta:* 1100100b
Con el presente método formar los números binarios del 0 al 15, y verificar su concordancia con la fig. 1.4.

Método para pasar de base diez a hexadecimal:

a. Dado el número a convertir, se parte de la pesa hexadecimal que en base diez tiene un valor igual a dicho número, o que presenta el valor menor más próximo al mismo; y a partir de este valor se escriben en base diez los sucesivos valores decrecientes de las pesas hexadecimales hasta el valor uno, siendo cada valor dieciséis veces menor que el anterior.

b. Repitiendo la metodología de las pesas desarrollada en relación con la figura A1.7, sistematizaremos la forma de hallar los dígitos hexadecimales a través del siguiente ejemplo representativo

EJEMPLO: Convertir el número 2574d a hexadecimal

a. => 256 16 1 d
b. A 0 E h

En el paso a. no se pudo usar ninguna pesa de valor 4096d, por lo que se comenzó con las de valor 256d que es la menor en relación con 2574d. Para el paso b., si usamos 10d = Ah pesas de 256d habremos equilibrado 10x256 = 2560 < 2574 (de haber puesto 11 = B pesas de 256 excedemos a 1574. Entonces debe escribirse **A** debajo de 256. Faltan equilibrar 2574 – - 2560 = 14 Si se probara equilibrar con pesas hexadecimales que valen 16 en base diez, no sería posible colocar ninguna (cero), pues si se pusiera una, se tendría 2560 + 16 = 2576 excediéndose el numero 2574, por lo que debe escribirse **0**

***Regla para pasar de hexadecimal a binario*:**

1. Separar los dígitos hexadecimales de modo de poder formar debajo de cada uno de ellos un cuarteto binario a determinar de pesos 8-4-2-1. Estos números pueden escribirse o ser imaginados mentalmente.
2. Convertir cada dígito hexadecimal de 0 a F a decimal (serán iguales del 0 al 9), y éste número a su vez en un cuarteto de pesos 8-4-2-1 por el método de las pesas ya visto.
3. El conjunto de cuartetos así formados constituirán el número binario buscado.

EJEMPLO: Convertir a binario el número hexadecimal A07

```
Paso 1      A      0      7 h
          8421   8421   8421 d
Paso 2    1010   0000   0111 b       O sea  A07h = 101000000111b
        A=10=8+2       7=4+2+1
```

Ejercicio: Convertir a binario 10h

Respuesta: 00010000b

A1.4 OPERACIONES ARITMÉTICAS CON NÚMEROS BINARIOS NATURALES

¿De qué forma la UAL suma dos números ?

En cualquier base numérica pueden definirse las distintas clases de números (naturales, negativos, imaginarios, reales, etc.), y todas las operaciones que empleamos en base diez. Estas presentan las mismas propiedades conocidas en base diez, y pueden aplicarse los mismos algoritmos que conocemos para realizarlas con números de varios dígitos.
Comenzaremos con la suma de binarios
Si bien pueden sumarse manualmente varios números binarios ordenados uno debajo del otro, interesa especialmente operar dos números binarios por vez, como lo hacen las unidades aritméticas de los microprocesadores y las calculadoras de bolsillo.
Para sumar en binario se debe tener presente que en la sucesión de los números naturales: 0, 1, 10, 11, . . . , si se suma cero a un número debe resultar el mismo, y si se suma uno debe obtenerse el siguiente.
Esto se verifica en las siguientes sumas elementales, que son las variantes que tienen lugar en la suma de dos números binarios:

```
  0     1     0     1     10
+ 0   + 0   + 1   + 1   +  1
  0     1     1    10     11
```

Efectuar 110110100 + 11010110

```
   1111  1
  110110100
+  11010110
 1010001010
  ihgfedcba
```

La suma se ha realizado, posición por posición, como se detalla:

a.	0 + 0 = 0	
b.	0 + 1 = 1	
c.	1 + 1 = 10 ⟶	se escribe el 0 y "me llevo 1" de acarreo (*"carry"*) a la posición siguiente
d.	1 + 0 + 0 = 1 + 0 = 1	
e.	1 + 1 = 10 ⟶	(ídem c.)
f.	1 + 1 + 0 = 10 + 0 = 10	(ídem c.)
g.	1 + 0 + 1 = 10	(ídem c.)
h.	1 + 1 + 1 = 10 + 1 = 11	se escribe 1 y "me llevo 1" a la posición siguiente.
i.	1 + 1 = 10	

¿De qué forma se realiza manualmente una resta de binarios ?

La tabla de restar binaria es sencilla:

0 − 0 = 0
1 − 0 = 1
1 − 1 = 0
0 − 1 = 1 y se "pide 1" a la siguiente; o sea se hace **10 − 1 = 1**

```
      0 0 1   0 1   0
  _ 1 0 1 1 0 0 1 0 0 1 0     (los dígitos 1 1 0 y 1 0 de las posiciones 3ª-5ª y 7ª-8ª aparecen tachados)
            1 1 1 0 0 1 0 1
  ---------------------------
    1 0 0 1 0 1 0 1 1 0 1
```

En la resta indicada[1], toda vez que se "pide 1" si la siguiente posición del minuendo vale **1**, éste pasará a ser **0**, dado que (1–1 = 0) como indica el renglón superior.

Si la siguiente es **0** pasará a ser **1** (10 – 1 = 1), debiéndose nuevamente "pedir 1" a la subsiguiente, que también pasará a ser 1 si es **0**, y así sucesivamente. Si hay ceros en el minuendo se transformarán en unos, hasta llegar a un **1** que pasará a ser **0**.

El procedimiento de "pedir prestado" no se emplea en los circuitos de un computador, por la complejidad y lentitud que ocasionaría. En su reemplazo se usa el método de **sumar al minuendo el complemento al módulo del sustraendo**, cuyos pasos se indican a continuación, y cuya justificación se trata en detalle en el "**Complemento para Enteros y Punto Flotante**" que está al final de esta Unidad.

¿Cómo efectúa la UAL una resta sin pedir prestado, mediante una suma?

Realizaremos la misma resta anterior sin pedir prestado, mediante **una sola** suma que hace la UAL

```
 10110010010          111   1  1
-00011100101           10110010010
                     + 11100011010
                     ------------
                               1
                     ------------
                     1 10010101101   (El resultado está en negrita recuàdrado)
```

Este uno se descarta ↵

Regla

1. El minuendo se sumará sin modificación.
2. Se invierten cada uno de los bits del sustraendo, y el número así formado es el segundo operando.
3. Se escribe un uno para ser sumado en la posición de las unidades.
4. Sumar los tres números indicados en 1, 2 y 3, y descartar el1 que está fuera del formato de los n bits que se restan. Los bits restantes a la derecha del bit descartado constituyen el resultado de la resta.

¿Cómo se multiplican y dividen manualmente números binarios naturales?

La tabla de multiplicar es muy sencilla, al igual que la operatoria manual (que no se usa en cálculos automáticos): se repite el multiplicando desplazado a la izquierda, conforme a la posición que ocupen los unos del multiplicador. Luego se realiza la suma con los sumandos así ordenados

0x0=0	
0x1=0	
1x0=0	
1x1=1	

```
   1 1 0 1 1
 x   1 0 1
 ---------
   1 1 0 1 1
 1 1 0 1 1
 ---------
 1 0 0 0 0 1 1 1
```

```
 1 1 1 0 1 1 | 1 1 0
-1 1 0       | -----
 0 0 1 0     | 1 0 0 1
-0 0 0
 -------
   0 1 0 1
 -   0 0 0
 -------
     1 0 1 1
   -   1 1 0
   -------
       1 0 1
```

La división se ha realizado con el método de las diferencias sucesivas, siendo que cada sustraendo se obtiene: multiplicando por **1** al divisor si este último es menor o igual que el resto parcial en cuestión, o por **0** si el mismo es mayor que dicho resto.

Es importante señalar que cada vez que se multiplica o divide un número entero binario **por la base $10_b = 2_d$ al igual que en base diez se agrega o se quita un cero**, respectivamente:

$(1011 \times 10)_b = 10110_b$ $\qquad$ $(10 \times 10)_b = 100_b$

$(10110 \times 10 \times 10 = 10110 \times 100)_b = 1011000_b$ $\qquad$ $(1110 : 10)_b = 111_b$

$(101011000\ 0 : 1000)_b = 1010110_b$

[1] Es conveniente comparar las similitudes de la resta efectuada con restas en base diez, como: $(80010010 - 5349809)_D$

En la resta indicada[1], toda vez que se "pide 1" si la siguiente posición del minuendo vale **1**, éste pasará a ser **0**, dado que (1–1 = 0) como indica el renglón superior.
Si la siguiente es **0** pasará a ser **1** (10 – 1 = 1), debiéndose nuevamente "pedir 1" a la subsiguiente, que también pasará a ser 1 si es **0**, y así sucesivamente. Si hay ceros en el minuendo se transformarán en unos, hasta llegar a un **1** que pasará a ser 0.

El procedimiento de "pedir prestado" no se emplea en los circuitos de un computador, por la complejidad y lentitud que ocasionaría. En su reemplazo se usa el método de **sumar al minuendo el complemento al módulo del sustraendo**, cuyos pasos se indican a continuación, y cuya justificación se trata en detalle en el "**Complemento para Enteros y Punto Flotante**" que está al final de esta Unidad.

¿Cómo efectúa la UAL una resta sin pedir prestado, mediante una suma?

Realizaremos la misma resta anterior sin pedir prestado, mediante una sola suma que hace la UAL

```
 10110010010          111   1  1
-00011100101           10110010010
                     + 11100011010
                     ------------1
                     1 10010101101   (El resultado está en negrita recuadrado)
```

Este uno se descarta ⌟

Regla

1. El minuendo se sumará sin modificación.
2. Se invierten cada uno de los bits del sustraendo, y el número así formado es el segundo operando.
3. Se escribe un uno para ser sumado en la posición de las unidades.
4. Sumar los tres números indicados en 1, 2 y 3, y descartar el1 que está fuera del formato de los n bits que se restan. Los bits restantes a la derecha del bit descartado constituyen el resultado de la resta.

¿Cómo se multiplican y dividen manualmente números binarios naturales?

La tabla de multiplicar es muy sencilla, al igual que la operatoria manual (que no se usa en cálculos automáticos): se repite el multiplicando desplazado a la izquierda, conforme a la posición que ocupen los unos del multiplicador. Luego se realiza la suma con los sumandos así ordenados

0x0=0
0x1=0
1x0=0
1x1=1

```
   11011
 x   101
 -------
   11011
11011
--------
10000111
```

```
 1110111 | 110
-110     | ---
  0010   | 1001
- 000
  ----
  0101
- 000
  ----
  1011
- 110
  ----
  101
```

La división se ha realizado con el método de las diferencias sucesivas, siendo que cada sustraendo se obtiene: multiplicando por **1** al divisor si este último es menor o igual que el resto parcial en cuestión, o por **0** si el mismo es mayor que dicho resto.

Es importante señalar que cada vez que se multiplica o divide un número entero binario **por la base** $10_b = 2_d$ **al igual que en base diez se agrega o se quita un cero**, respectivamente:

$(1011 \times \mathbf{10})_b = 10110_b$ $\qquad$ $(10 \times \mathbf{10})_b = 100_b$

$(10110 \times 10 \times 10 = 10110 \times \mathbf{100})_b = 1011\mathbf{000}_b$ $(1110 : \mathbf{10})_b = 111_b$

$(101011000\ 0 : \mathbf{1000})_b = 1010110_b$

[1] Es conveniente comparar las similitudes de la resta efectuada con restas en base diez, como: $(80010010 - 5349809)_D$

A1.5 CODIFICACION ASCII DE CARACTERES ALFANUMERICOS Y UNICODE

¿Qué es el código ASCII ?

El código ASCII (léase *"asqui"*) siglas de American Standard Code for Information Interchange es un código binario ampliamente usado para la transmisión de información, y para codificar los caracteres de un teclado, así como los que debe imprimir un impresora en modo texto o mostrar una pantalla. También es el código de los archivos de texto.
En la fig. A1.11 aparecen los caracteres alfabético-numéricos imprimibles, más los de teclas que ordenan movimientos del cursor que antes eran los del carro de una máquina de escribir (**SP** abreviatura de "space"; la tecla de retorno del carro y pasar a un nuevo renglón, **CR**, siglas de "carry return", que aparece en los teclados actuales como ↵ "Enter";la de "back space" = **BS**)[1], más otros relacionados con las teletipos (**STX** -Start of Text- **EOT** -End of Transmisión- etc)[1].
Suman $128 = 2^7$ caracteres y otros a codificar en binario por lo que bastarían 7 bits para formar igual número de combinaciones binarias distintas, aunque se usan 8 bits (el primer bit de todas las combinaciones del ASCII estándar es cero, pudiéndose usar para paridad u otros fines). Con 8 bits resultan $2^8 = 256$ combinaciones, 128 para ascii estándar y 128 para ascii extendido.
Por ejemplo, las mayúsculas de la A hasta la Z se codifican según una sucesión ordenada de números binarios (que permite realizar ordenaciones alfabéticas):

A	01000001 = $41_h = 65_d$;	**0**	00110000 = $30_h = 48_d$;	**SP**	00100000 = $20_h = 32_d$;
a	01100001 = $61_h = 97_d$;	**1**	00110001 = $31_h = 49_d$;		
B	01000010 = $42_h = 66_d$;	**2**	00110010 = $32_h = 50_d$;		
b	01100010 = $62_h = 98_d$;		....		
Z	01011010 = $5A_h = 90_d$	**9**	00111001 = $39_h = 57_d$		
z	01111010 = $7A_h = 90_d$				

Arriba aparecen combinaciones binarias del código ASCII cuando se pulsa la tecla con el carácter indicado. Así, cada vez que se pulsa **A** queda 01000001 en una celda de memoria. Para poder ordenar alfabéticamente las letras, la **B** es el número binario siguiente 01000010, y así sucesivamente se aumenta uno hasta la **Z** (01011010).
Al lado de cada combinación binaria aparecen su valores equivalentes en hexa y en decimal, usados en la tablas ASCII.
Una mayúscula se diferencia de su minúscula en un solo bit (el tercero desde la izquierda) lo cual se usa en ciertos programas para no diferenciar un nombre escrito con mayúsculas o minúsculas, dado que los 7 bits restantes permiten determinar la tecla de la letra tipeada sin importar si el carácter es el de arriba o el de debajo de dicha tecla.
Los dígitos del 0 al 9 empiezan con 0011, y el segundo cuarteto con pesos 8-4-2-1 indicados determina qué dígito es.

Si se tipea 109XP en memoria queda en ASCII como: 00110001 00110000 00111001 01010111 01010000
(1 0 9 X P)
Asimismo, si a la plaqueta de video le llegan en orden esas tres combinaciones binarias ASCII, en pantalla se verá 109XP

Cuando se tipea Alt 64 para que aparezca @, dado que 64d = 01000000b éste será el código ASCII de @.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	S C	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	SP	!	"	#	$	%	&	'	(	)	*	+	,	–	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Figura A1.11 **(TABLA DEL CODIGO ASCII)**

[1] Una tecla como **SHIFT** es una orden interna para el teclado a fin de que genere mayúsculas o el símbolo superior de los dos que aparecen en una tecla, por lo que no se codifica en ASCII

La tabla anterior es una forma compacta de mostrar la tabla con el código ASCII expresado en hexa, donde cada carácter tiene dos coordenadas que escritas consecutivamente conforman el equivalente hexa del código binario.
Así, para hallar el código ASCII de un carácter como la **A**, ésta tiene por coordenadas **4** (horizontal) y **1** (vertical), que forman 41_h = 01000001 Del mismo modo, **SP** tiene coordenadas **2** y **0**, que forman 20_h = 00100000;

ASCII extendido
Si se tipea Alt 164 para obtener **ñ**, dado que 164d = 10100100 b éste será el código ASCII de la ñ. El bit extremo izquierdo es uno. Esto caracteriza a las 128 combinaciones de cualquier ASCII extendido, que puede contener la ñ, las vocales con acento, y otros símbolos usados en otras lenguas latinas, según lo establezca la empresa que lo imponga.

EJERCICIO: visualizar con el Debug como quedan en memoria codificados en ASCII, los caracteres de Ana 3/6/80

Pasos a efectuar a partir del símbolo **C:\>** del DOS :
1. **C:\>** *COPY CON MARIO.COD*↵ (en itálica aparece lo que se debe escribir, y en negrita lo que escribe el DOS)
Ana 3/6/80 ^Z (^ Z se usa para cerrar el archivo, siendo que ^ se logra pulsando la tecla Ctrl)

1 ARCHIVO(s) COPIADO(s)

2. **C:\>** *DEBUG MARIO.COD*↵ [1] (Para poder ver en memoria mediante el Debug el archivo tipeado)
– (Guión titilante que indica que el Debug está esperando una orden)

3	*E 100*↵		(Orden de leer la memoria a partir de 0100, donde quedan los archivos)						
	309D: 0100	**41.**	**6C.**	**61.**	**20.**	**33.**	**2F.**	**36.**	**2F.**
	309D: 0108	**39**	**30.** ↵						

A partir de 41. se debe pulsar la barra espaciadora para obtener el valor siguiente (6C), y así de seguido, (como se hizo explicó al describir el uso del Debug) tantas veces como caracteres, espacios y "enter" se hayan pulsado en el teclado
Se verifica que los números hexadecimales 41, 6C, etc corresponden a los caracteres tipeados (Ana 3/6/80).
El lector debe realizar esta práctica tipeando otros caracteres distintos que Ana 3/6/80 y luego verificar que los códigos que se leen en memoria corresponden a la codificación ASCII según la tabla dada.

¿Qué es el Unicode ?

Hacia 1990 se buscó definir un sistema de codificación universal, que permitiese la incorporación de los caracteres (grafos) de todas las lenguas escritas del mundo, tanto actuales como pasadas, así como los símbolos utilizados en matemáticas y otros. El sistema debería ser además extensible, de forma que pudieran añadirse nuevos tipos en el futuro.
Por un lado el **UCS** ("Universal Character Set"), desarrollado por la Organización Internacional de Standarización (ISO); y por otro el denominado **Unicode** impulsado por empresas como Microsoft (integrante del Unicode Consortium), que había incorporado en sus sistemas Windows 9x un juego de caracteres adaptado a la mayoría de alfabetos occidentales. Posteriormente implementó Unicode en sus sistemas NT y sucesores. Aunque el desarrollo inicial de ambos sistemas fue independiente, luego convergieron, y el sistema **Unicode** se convirtió en un subconjunto del sistema **UCS**.

Unicode es una norma internacional de codificación de carácteres. La primeira versión (1991), usaba una codificación de 16 bits, por lo que podía codificar 65.536 caracteres. Actualmente se represénta con tres tipos de codificación, UTF-8, UTF-16 e UTF-32) según que use 8, 16 ó 32 bits para identificar cada carácter. Unicode 4.1, codifica cerca de un millón de caracteres que cubren los principales idiomas escritos del mundo. Los primeros 128 caracteres de Unicode corresponden a los caracteres del ASCII y tienen su mismo valor. Puesto que ASCII usa 7 bits por caracter es inadecuado para manejar texto multilingüe, Unicode adoptó un formato de 16 bits que extiende las ventajas del ASCII al texto multilingüe.
Unicode se ha convertido en la codificación dominante para el procesamiento de texto. Cubre idiomas de América, Europa, África, India, Asia, y los símbolos técnicos.
Unicode proporciona un número único para cada caracter, no importa cual sea la plataforma, el programa, o el idioma que se trate. Un número hexadecimal y un prefijo U, por ejemplo, **U+0041** representa la **A** en 16 bits. Unicode ha sido adoptado por Apple, el HP, la IBM, Microsoft, Oracle, SUN, Sybase, Unisys entre otros. Es usado en Java, ECMAScript (Javascript); XML. También por muchos sistemas operativos, por los browsers actuales, y muchos otros productos.
La aparición de Unicode, y la disponibilidad de las herramientas que lo apoyan se encuentran entre las tendencias globales recientes más significativas de la tecnología del software. Permite que los datos sean transportados a través de muchos sistemas distintos sin que sufran daños, y que un producto de software o un sitio Web pueda orientarse a múltiples plataformas, idiomas y países, sin necesidad de rediseñarlo. Una versión de un producto se puede utilizar por todo el mundo. No son necesarios lanzamientos separados para mercados regionales. Un texto en cualquier lengua se puede intercambiar por todo el mundo.

[1] Las actuales versiones de Windows™ por razones de seguridad pueden requerir alguna modificación menor de este comando

EJERCITACION

1. Escribir los sucesivos números hexadecimales del 1048h y 1070h.
2. Conforme al ejercicio anterior, indicar cuál es el siguiente de los siguientes números hexadecimales: FFF, 2ABF, 2B99, 1FF, ABCD, C0D0, A0F, 999.
3. ¿De cuántos bytes consta el número hexadecimal 003B, y qué número binario y decimal es ?
4. Dado el número 10 en base dos, y el número 10 en base dieciséis ¿qué números son en base diez ?
5. Generalizar la pregunta anterior: dado el número 10 en base X, ¿qué número es en base diez ?
6. Dado el número binario 1111111, hallar una forma rápida de pasarlo a decimal, sin tener que hallar el peso decimal de cada bit y luego sumar los pesos. Generalizar el procedimiento hallado.
7. Determinar en la expresión 2^n qué sucede con su valor, cada vez que **n** aumenta uno, y calcular el número de combinaciones binarias distintas que pueden formarse con n = 10 y n = 11. Tener presente la figura 1.4
8. Convertir a base doce el número decimal 140.
9. Convertir a hexadecimal el número $11\ 0000\ 0000\ 0001_b$ Luego convertir este número binario a octal.
10. Los números de las direcciones de una zona de memoria un computador van de 0000 a FFFF. Determinar a cuántas posiciones de memoria existen combinaciones.
11. Se tiene una memoria con 2^{20} posiciones, y se quiere identificar cada una con un número binario distinto.
    Expresar en binario la primera, segunda, anteúltima y última dirección de dicha memoria.
    Indicar en decimal cuántas posiciones son, y cuántos símbolos hexadecimales se necesitan para codificar cualquier posición. Indicar en hexa el valor de la posición cero y la última.
12. ¿Cuántos bits hacen falta para representar números decimales entre 0 y 999999_d ? ¿Cuántos dígitos hexadecimales?
13. Los registros de un 386, 486 y Pentium tienen 32 bits. ¿cuál es el mayor y menor número natural que se representa?
14. Dados los números naturales 180 y 40
    a) Representarlos en código ASCII
    b) Representarlos en binario y sumarlos en formato 8 y 16
    c) Representarlos en binarios y restarlos por el método del complemento a la base en formato 8 y 16.
15. Siendo P = 180, Q = 40 efectuar en formato 16 la operación P + P – Q. Verificar que el resultado sea correcto.
16. Comparar en base diez y dos, qué pasa cuando se multiplica por 10 y por 1000. Generalizar para la división.

RESPUESTAS A PARTE DE DE LOS EJERCICIOS

3. 2 bytes.; $003B_h = 0000000000111001_b = 59_d$
4. Dos y dieciséis
5. X
1. Conviene tomar el número siguiente $1000000 = 128_D$ y restarle uno
9. $3001_h = 30001_O$ 10. 65.536
11. 0000000000000000000 00000000000000000001 11111111111111111110 11111111111111111111
    1.048.576; 5 dígitos en hexa, de 00000 a FFFFF
12. 20 bits y 5 dígitos en hexa
13. $2^{32} = 2^{10} \times 2^{10} \times 2^{10} \times 2^2 = 1024 \times 1024 \times 1024 \times 4 > 4000.000.000$

EJERCICIO SISTEMATIZADOR DE CODIGOS

En un teclado se ha tipeado **3Z**, y la combinación binaria que resulta en memoria es interpretada por 3 programas creados con fines distintos que se indican, siendo que el significado que cada programa asigna a esa combinación debe aparecer en pantalla en caracteres alfabéticos y numéricos (alfa-numéricos), indicar en cada caso los pasos a seguir para que ello suceda.

1) Si dicha combinación es interpretada por el programa Debug que convierte de binario a hexa y viceversa.
2) Si 3Z es tipeado mientras se está escribiendo texto en el programa Word, y el mismo se visualiza en pantalla.
3) Si un programa que esta operando con números naturales (magnitudes) interpreta la combinación citada, independientemente de cómo fue generada.

Aclaración: como se detalla en la Unidad 2 de esta obra, el teclado no codifica en binario ASCII cada tecla tipeada. En cambio la plaqueta de video se encarga de convertir cada código ASCII que le llega en el carácter o acción correspondiente.

Para empezar, los dos caracteres tipeados (3Z), en memoria quedarán codificados en ASCII como 00110011 01011010_2 según se desprende de la tabla ASCII de la figura A1.11 (3 = 33 y A = 5A).

1) El Debug está preparado para "ver" –con vistas a que aparezca en hexa en pantalla- cada combinación binaria dividida en cuartetos con pesos 8-4-2-1, que deberá traducir en dígitos hexadecimales (sección A1.3):

```
8421 8421 8421 8421
0011 0011 0101 1010
  3    3    5    A
```

estos 4 símbolos son los que deben aparecer en la pantalla, para lo cual el Debug deberá generar en memoria sus correspondientes códigos ASCII, que deberán llegar luego a la plaqueta de video.
Ellos son: 000110011 (3) 000110011 (3) 00110101 (5) 01000001 (A) que en hexa son 33 33 35 41 (Tabla A1.11)

2) El Word trabaja directamente con los caracteres ASCII que han sido tipeados, por lo que al tipear 3Z simplemente sus códigos ASCII (00110011 y 01011010) pasan a la plaqueta de video, que los convertirá en pantalla en los caracteres 3Z. Igualmente, si dichos caracteres forman parte de un archivo del Word que se abrió, sus códigos ASCII pasan a dicha plaqueta, y así son visualizados en pantalla.

3) Dado que el programa opera con números naturales, cuando interpreta la combinación 00110011 01011010 -que ocupa dos posiciones sucesivas de memoria (una por byte)- para que se imprima en pantalla como un número decimal la interpretará con los pesos siguientes:

```
32783 16384 8192 4096 2048 1024 512 256 128 64 32 16 8 4 2 1
  0     0     1    1    0    0    1   1   0   1  0  1  1 0 1 0 b = 13146d
```

Para que estos 5 dígitos decimales aparezcan en pantalla, el programa deberá generar en memoria sus códigos ASCII que llegarán a la plaqueta de video: 00110001 (1) 00110011 (3) 00110001 (1) 00110100 (4) 00110110 (6).

EJERCICIO INTEGRADOR DE CONOCIMIENTOS

Un programador ha desarrollado para variables (datos) que son *magnitudes*, un programa en un cierto lenguaje "X" de alto nivel, cuyas sentencias[1] tipeó desde el teclado de un computador. Para una porción de dicho programa ha tipeado la sentencia que más abajo se indica. Se supone que luego de ella, se puede escribir el valor de las variables a operar.

Variable:
MAGNITUDES ↵ (este símbolo ↵ aparece para indicar que se pulsó "Enter")

R = P + Q – T ↵ (sentencia en alto nivel, en el lenguaje X)

P = 130 Q = 4103 T = 4 ↵
↑ ↑ (las flechas señalan que se hizo un espacio mediante la barra espaciadora)

2. Representar cómo quedan en memoria los caracteres tipeados, supuestos escritos a partir de la dirección 0100.
3. Si posteriormente el programador llama al programa compilador del lenguaje "X", indicar cómo el compilador deja traducida en memoria la sentencia R = P + Q – T en la secuencia de instrucciones en código de máquina, para un procesador de Intel o AMD. Para tal fin usar los códigos definidos en la figura 1.15, y escribirlos en memoria a partir de la dirección 03AC. Asignar a las variables (datos) R, P, Q y T las direcciones A23E, A240, A242, y A244, respectivamente. Para cada variable usar 2 bytes.
4. Una vez que todo el programa en el lenguaje "X" fue traducido a código de máquina, se ordena ejecutarlo.

[1] Designamos "**sentencia**" a cada orden (instrucción) que constituye cada "línea" escrita de un programa expresado en un lenguaje de alto nivel (Pascal, C, Basic, Cobol, Java, etc.), la cual si se tipea quedan codificados en ASCII los caracteres que la componen. Reservamos la palabra "**instrucción**" o sea "**código de máquina**" (a veces llamado "código" a secas) a las órdenes que puede ejecutar una UCP (microprocesador), siendo que cada sentencia debe ser traducida en una secuencia de estas instrucciones.

Indicar, suponiendo que se ejecute la secuencia de instrucciones traducida en el punto anterior, para las instrucciones que ordenan sumar o restar, cómo la UAL realiza estas operaciones con los datos correspondientes, y cómo queda el registro AX luego de que se ejecuta cada una de las instrucciones de dicha secuencia. También pasar a base diez cada resultado binario que está en AX, verificando que sea el valor esperado.

5. Representar luego de ejecutar la instrucción I_4 cómo queda en memoria el resultado asignado a la variable **R**, conforme ordena dicha instrucción.
6. Suponiendo que luego de la sentencia R = P + Q – T siga la sentencia **PRINT "R=" R** (o sea imprimir en decimal el valor de R hallado), el compilador la traducirá a varias instrucciones en código de máquina, una de las cuales llamará a una subrutina de impresión. Si ésta ordena que la impresora opere en modo texto, indicar cómo debe quedar codificada en memoria desde la dirección 7100, la información a enviar a la impresora, para que imprima en decimal el valor de R.

Este ejercicio integra de forma concreta, los 4 pasos de un proceso de datos: entrada, memorización, proceso y salida.

1. Como se estableció en A1.5, al tratar el código ASCII, en una operación de entrada desde el teclado, los sucesivos caracteres tipeados (sean de un texto para un procesador de texto, o de un programa en alto nivel) quedan almacenados en binario, en posiciones sucesivas de memoria (figura A1.15) en este caso desde la 0100, codificados en ASCII. Para más claridad, al lado de cada casillero aparece el carácter codificado, y su código ASCII en hexa, según la tabla (fig. A1.11).

2. Cualquier programa en alto nivel se encuentra codificado en ASCII, al igual que una carta, o cualquier texto, es una sucesión de caracteres aislados, codificados individualmente en combinaciones de 8 bits que los representan.

*Si bien el código ASCII es un código binario, la UCP **no** puede ejecutar ningún programa en lenguaje de alto nivel, cualquiera sea este lenguaje; del mismo modo que la UCP no puede ejecutar una carta o una nota que hayamos escrito* y que haya quedado en memoria.

Un programa en alto nivel es un texto codificado en ASCII, compuesto por órdenes (sentencias), que debe ser escrito siguiendo estrictamente una sintaxis especificada, para que pueda ser reconocida como un programa por el programa traductor (**compilador**). El se encarga de *traducir cada sentencia en una secuencia de instrucciones de máquina* apropiadas al microprocesador (UCP) que lo va a ejecutar.

Del archivo del disco rígido donde se encuentra el compilador de un lenguaje X, éste pasa a memoria. Al ordenar compilar, , los datos a procesar son los caracteres codificados en binario ASCII, que representan el programa en alto nivel para el programa compilador. Luego de ser ejecutado (por la UCP), el programa compilador dejará en memoria, traducido a código de máquina (en este caso para el microprocesador de Intel), el programa que originariamente estaba en alto nivel. Esto es, el compilador es un programa cuyo "input" (datos) es un programa en alto nivel codificado en ASCII (unos y ceros), y cuyo "output" (resultado) es el programa que fue su "input", pero en código de máquina (unos y ceros), para que pueda ser ejecutado por un microprocesador específico.

Un compilador para un lenguaje de alto nivel "X" sólo traduce programas escritos en este lenguaje, para que **sólo** puedan ser ejecutados en microprocesadores de un cierto tipo. Por ejemplo, un compilador que traduce de un lenguaje "X" para procesadores de Intel o AMD, los programas que genera en código de máquina no pueden ser ejecutados como ser en procesadores HP, o PowerPC, o SPARC, u otros, por tener cada procesador sus propios códigos binarios de instrucciones.

Una misma orden de sumar se codifica distinto en procesadores de distinto fabricante, y aún en modelos no compatibles de un mismo fabricante de microprocesadores.

El compilador para un lenguaje "X" es un programa "inteligente" que recorre repetidamente la zona de memoria donde quedó codificado en ASCII el programa en lenguaje "X". Así va identificando cada sentencia y los datos (variables) que ella ordena operar, a fin de traducirlos en *instrucciones (códigos) de máquina que el microprocesador pueda ejecutar, y códigos de datos que la UAL pueda operar.*

2a. Con estos fines (figura A1.15) **primero** debe determinar el tipo de variable (en este caso magnitudes[1], o sea números naturales) que las sentencias ordenan procesar, para llevar a cabo las traducciones a instrucciones para procesar magnitudes, y códigos de datos correspondientes a éstas. Entonces, el compilador **antes** de traducir leerá en memoria que en ASCII está escrito "MAGNITUDES", a fin de poner en juego las subrutinas de traducción para este tipo de variables.

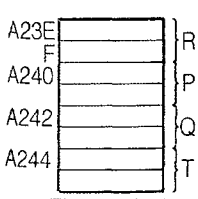


Figura A1.14

2b. ASIGNACIONES DE DIRECCIONES A LAS VARIABLES: como se estableció en relación con la figs. 1.3 y 1.15 **cada instrucción debe indicar la dirección del dato** (valor de la variable) que ordena operar. Es por ello que antes de traducir cada sentencia con sus variables en instrucciones para un procesador, el compilador debe asignar la dirección que tendrá cada variable, que será también la dirección del dato cuando se les da un valor a cada variable antes de que se ejecute el programa. Esto se hace en 2.d.

El compilador reservará (conforme al enunciado del ejercicio) dos bytes de memoria para las variables R, P, Q y T, en direcciones de memoria **arbitrarias** que en este ejemplo van de A23E hasta A245 (fig. A1.14). Esto es, **para el compilador la variable R será la dirección A23E** y la variable P será la dirección A240, etc., *bastando la primer dirección de una variable para identificarla*, dado que se sabe que ocupan 2 celdas todas las variables tipo "Magnitudes".

O sea que el compilador en primer término sólo identifica cada variable por su primer dirección, sin que interese para formar las instrucciones (paso **2c**) el valor concreto de la variable, pues **en estas instrucciones se indica dónde está la variable, a fin de poder localizar en memoria este valor** (por lo que no es imprescindible asignar direcciones consecutivas como se hizo). **No interesa** en la traducción el valor concreto que se le asigna a esa variable antes de la ejecución del programa, como se hace en 2d, cuando se escribe dicho valor concreto en las posiciones reservadas para cada variable.

2c. CODIFICACIÓN DE LAS INSTRUCCIONES RESULTANTES DE LA TRADUCCION: Una vez asignada a cada variable una dirección de localización, el compilador traducirá la sentencia R = P + Q – T en instrucciones cuyos códigos de

[1] Más adelante se realizan ejercicios similares al dado, pero para variables que son del tipo "Integers" , "Reales" (punto flotante)

máquina los tomaremos de la fig. 1.15. Recordar en relación con la fig. 1.2 que todo sucede como si tuviéramos que pensar los pasos para hacer esa suma algebraica escrita en un papel con una calculadora de bolsillo. Para ello primero (I_1) entramos P al visor (o sea AX). Luego (I_2) entramos Q y lo sumamos a P, para obtener P+Q en el visor. A este valor le restamos T, resultando P + Q – T en el visor (I_3), valor que se guarda en la memoria de la calculadora mediante la tecla correspondiente (I_4). Como en la descripción relacionada con la fig. 1.15, sin la cual esta traducción no puede entenderse, estos 4 pasos en un procesador se realizan mediante las instrucciones **I1** a **I4**. Sus códigos de máquina se indican en la *zona de instrucciones* de la fig. A1.15 desde la dirección 03AC tomada **arbitrariamente** (pues bastaría hacer IP = 03AC antes de ejecutar las instrucciones para que se localice cada una de ellas). Tanto en las suma algebraica R = P + Q – T, como en la R = P + P – Q traducida en la fig. 1.15, la secuencia de instrucciones **I1** a **I4** ordena una asignación a AX (cod-op A1), seguida de una suma (cod-op 0305), seguida de una resta (cod-op 2B06), seguida de otra asignación (cod-op A3). Debajo de cada cod-op está la dirección de la variable (asignada en 2b) que se ordena operar (no el valor de ésta), que obviamente difieren en un caso y en otro. Recordar para las codificaciones que para Intel™ una dirección o dato xxyy se escribe yyxx.

En definitiva, el compilador reserva una zona para variables tipo magnitudes, y en otra "zona de programa" deja éste codificado en bajo nivel (instrucciones en código de máquina), que puede ser ejecutado por la UCP para la que fue traducido.

Hasta acá se describió el proceso de datos que es una traducción[1]. Ahora se supone que se ejecutará la secuencia traducida.

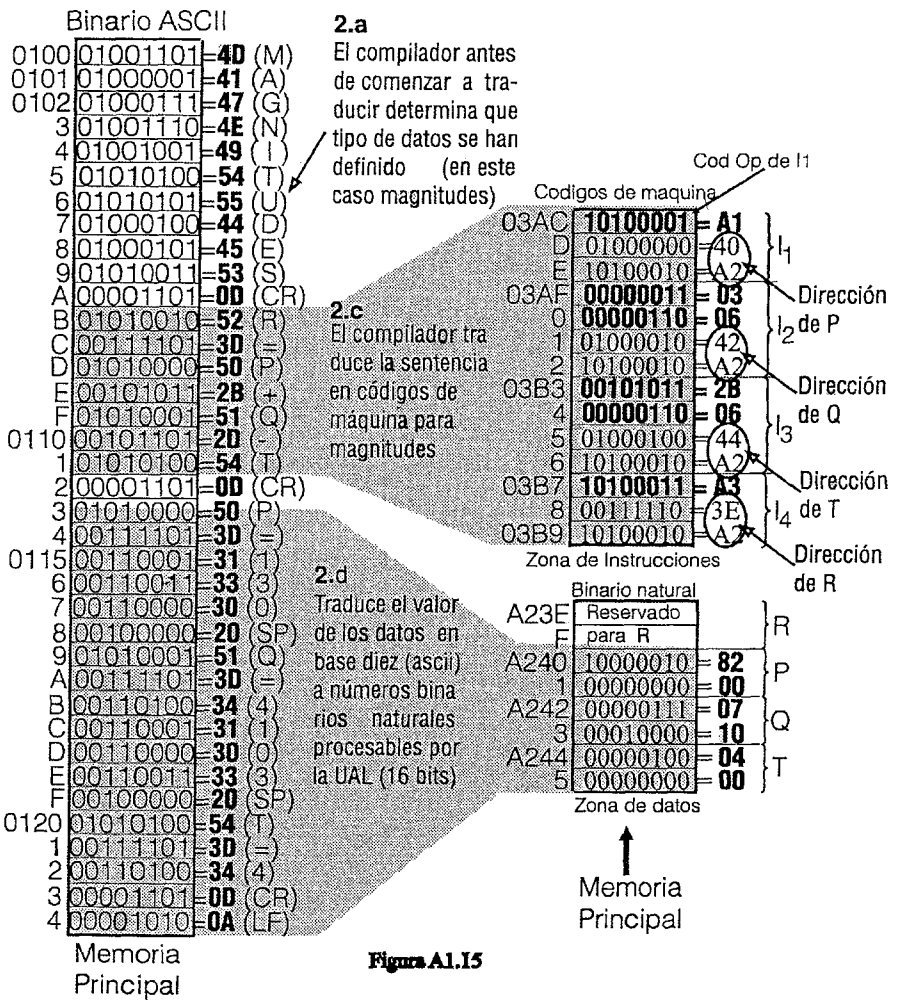


Figura A1.15

2d. ASIGNACIÓN DEL VALOR DE LOS DATOS ANTES DE LA EJECUCIÓN DE LAS INSTRUCCIONES:

Cada vez que se ejecuta la secuencia I1 a I4 las variables pueden tener un valor distinto (de ahí su denominación).

El valor de cada variable (dato) debe estar asignado antes de ejecutar las instrucciones en las posiciones reservadas para cada variable (zona de datos), antes de que se ejecuten las instrucciones, como se hace a continuación.

Una subrutina traductora preparada para "MAGNITUDES") traducirá cada valor de variable decimal -escrita en ASCII como dígitos separados según aparecen en la columna de caracteres tipeados- en el número binario natural de 16 bits equivalente, **pues la UAL sólo suma binarios naturales** y no combinaciones binarias ASCII, que además empiezan arbitrariamente con 0011. O sea, que **esta traducción es para que la UAL pueda operar los datos a procesar.**

Así, el valor decimal 130 de la variable **P** que en memoria está en ASCII como los dígitos 00110001 00110010 00110000 en las posiciones 0115, 0116 y 0117 (fig A1.15) quedará traducida como el valor binario natural $000000010000010_b = 130_d$ que ocupará las posiciones A240 y A241 asignadas a la variable **P**.

El valor decimal 4103 de **Q**, que el traductor encuentra en binario ASCII en las posiciones 011B a 011E (4 bytes) las traducirá como el binario natural $0001000000000111_b = 4103_d$, que ocupará las posiciones A242 y A243.

Del mismo modo, el valor 4 de **T** que al compilador llega como 00110100 en ASCII, ocupando un byte, pasará a ser $0000000000000100_b = 4_d$ que el compilador escribirá en las direcciones A244 y A245. Para la variable **R** (cuyo valor recién se conocerá cuando se ejecute el programa que se está traduciendo) *el programa compilador (que en este paso es el que se ejecuta)*, le reserva la posición A23E y la siguiente (como supone el enunciado). De esta forma cada variable, *cuyo valor al ser tipeado ocupa tantos bytes de memoria como dígitos decimales tenga*, al ser codificada como magnitud binaria ocupará 2 bytes, siendo que el máximo número que se puede representar en 16 bits es el $65535_d = 1111111111111111_b$.

---

[1] Debe consignarse que las direcciones que asigna un compilador no son las definitivas, sino que el programa cargador reubica las zonas de instrucciones y datos conforme lo determina el sistema operativo, administrador de la memoria.

3. Cuando se ordene ejecutar el programa que el compilador dejó en código de máquina en memoria y la UCP ejecute las instrucciones I_1 a I_4, luego de cada una de ellas, el registro AX quedará como se indica a continuación. La UAL sólo operará en las instrucciones I_2 e I_3, que ordenan sumar y restar, respectivamente, indicándose para las mismas la operación de la UAL.

Después de ejecutar I_1:
AX = 0000000010000010 que es el valor de **P** (130d = 0182h; éste y otros valores en hexa sirven para el Debug).

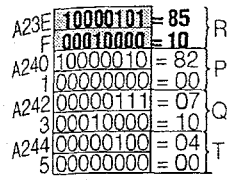


Figura A1.17

Después de ejecutar I_2:
AX = 0001000010001001 que es P + Q (4233 en decimal).
Este valor es el resultado de sumar **en la UAL** ⟶

```
  0000000010000010  (P)
+ 0001000000000111  (Q)
  0001000010001001 = 1089h  (P + Q)
```

Después de ejecutar I_3:
AX = 0001000010000101 que es P + Q − T = R (4229d = 1085h).

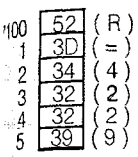


Figura A1.16

Este valor es el resultado de sumar **en la UAL** ⟶

```
  0001000010001001  (P + Q)
+ 1111111111111011  (bits de T invertidos)
  ----------------
  0001000010000101  (P + Q) − T = R
```

Después de ejecutar I_4 el valor de AX no cambiará, pues I_4 ordena que una copia de AX pase a memoria
AX = 0001000010000101 = 1085h

4. La ejecución de I_4 ordena que en la dirección A23E y en la siguiente (asignadas a R) copiar el valor de AX (8510 en hexa), por lo que luego de ejecutarse I_4 en memoria se tendrá para R los valores 10 y 85 indicados en la figura A1.17

5. En modo texto, la subrutina de impresión dejará en posiciones sucesivas de memoria los caracteres a imprimir, cada uno en código ASCII. Previamente dicha subrutina interpretará la orden de impresión (en este caso R = valor de R), siendo que el valor de R que está en A23E y A23F deberá pasarlo a dígitos decimales codificados en ASCII. Esto es, determinará que el valor de R, que es 0001000010000101 es el 4229, el cual en ASCII resultará 00110100 00110010 00110010 00111001, como aparece en memoria en la figura A1.16 Antes de estos códigos aparecen los códigos de **R** y el de igual (=), como se indica.

EJERCICIO COMPLEMENTARIO:

Usando el Debug como se explicó en la sección 1.6, escribir en memoria los datos e instrucciones como quedaron en la fig, A1.15. Luego ejecutar cada instrucción y verificar que los resultados concuerden con las respuestas dadas para 3 y 4.

Dado que R no se conoce, se comienza por escribir el valor de P en la dirección A240 , seguido por los valores de Q y T
–E A240 (Comando para examinar memoria y escribir en ella si se desea, siendo que xx indica el contenido "basura" pre-existente)
309D:A240 xx.*82* xx.*00* xx.*07* xx.*10* xx.*04* xx.*00* ↵ (0082 se escribe 8200, etc.)

Para corroborar que los valores recién escritos son los nuevos contenidos de las posiciones modificadas, otra vez se examina con E
–E A240 ↵ (Examinar memoria para verificar si la escritura anterior fue correcta)
309D:A240 **82.** **00.** **07.** **10.** **04.** **00.** ↵ (la escritura fue correcta)
– (A240) (A241) (A242) (A243) (A244) (A245) Entre paréntesis las direcciones

Con el mismo procedimiento se escriben los códigos de máquina (cod-op + dir del dato) de las instrucciones a partir de 03AC
–E 03AC (Examinar memoria y escribir en ella)
309D:03AC xx.*A1* xx.*40* xx.*A2* xx.*03* xx.*06* xx.*42* xx.*A2* xx.*2B* (A240 se escribe 40A2, etc.)
309D:03B4 xx.*06* xx.*44* xx.*A2* xx.*A3* xx.*3E* xx.*A2* ↵ (los cod-op no se invierten: 0306 sigue igual, etc)

–E 03AC ↵ (Examinar memoria para verificar si la escritura anterior fue correcta)
309D:03AC **A1.** **40.** **A2.** **03.** **06.** **42.** **A2.** **2B.**
309D:03B4 **06.** **44.** **A2.** **A3.** **3E** **42** ↵
– (03B4) (03B5) (03B6) (03B7) (03B8) (03B9) Entre paréntesis algunas direcciones de los valores

–R IP ↵ (comando al Debug para examinar el valor del Registro IP y cambiarlo si se desea)
IP 0100 (el Debug informa que actualmente el IP contiene 0100)
: *03AC* ↵ (al lado de los dos puntos que deja el Debug escribimos 03AC, nuevo valor que debe tener IP)
–
–R ↵ (Antes de ejecutar se deben examinan registros, siendo que el Debug muestra la información siguiente)
AX=0000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
ES=309D SS=309D CS=309D **IP=03AC** NV UP EI PL NZ NA PE NC
309D:03B4 **A140A2** (en negrita se verifica que, como debe ser, el IP está en 03AC, y que la próxima instr. a ejecutar es A140A2)
–
–T ↵ (Orden para ejecutar una instrucción I_1)
AX=0082 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
ES=309D SS=309D CS=309D **IP=03AF** NV UP EI PL NZ NA PE NC
309D:03AF **030642A2** (próxima instrucción I_2 a ejecutar)

: Constatamos que I_1 se ha ejecutado correctamente, pues se ha cumplido la orden que portaba su código: escribir en AX una copia del contenido de la posición A240, que es 0082. También ha cambiado automáticamente IP a 03AF, para apuntar la dirección de I_2, como habíamos previsto al hablar de IP. Asimismo vemos que el código 030642A2 de I_2 es el correcto, por lo que podemos ejecutar I_2

–T ↵ (Ejecución de la instrucción I_2)
AX=1089 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
ES=309D SS=309D CS=309D **IP=03B3** NV UP EI PL NZ NA PE NC
309D:03B3 **2B0644A2** (próxima instrucción I_3 a ejecutar)

– Se ha realizado lo que ordenaba I_2: sumar al valor 0082 de AX el contenido de la dirección A244 (que es 1007), y el resultado (1089) escribirlo en lugar de 0082. También se verifica que 2B0644A2 es el código de I_3, instrucción de resta que pasaremos a ejecutar:

–T ↵ (Ejecución de la instrucción I_3)
AX=1085 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
ES=309D SS=309D CS=309D **IP=03B7** NV UP EI PL NZ NA PE NC
309D:03B7 **A346A2** (próxima instrucción a I_4 ejecutar)

_ I_3 ordenaba restar a AX el contenido de A246, que es 0004h, o sea que la UAL ha efectuado 1089h – 0004h = 1085h, como aparece en AX.

–T ↵ (Ejecución de la instrucción I_4)
AX=1085 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
ES=309D SS=309D CS=309D **IP=03BA** NV UP EI PL NZ NA PE NC
309D:03BA **XXXXXX** (no interesa cuál sea la instrucción, pues no se ejecutará.

Es necesario verificar que en las direcciones A23E y A23F se escribió el resultado que está en **AX**. Hacemos:
–E E23E ↵ (Examinar memoria)
309D:**E23E** **85.** **10.** ↵
_ con lo cual constatamos que efectivamente se cumplió lo que ordena I_4

CUESTIONARIO.

1. ¿Qué debe llegar a memoria antes de ejecutar un programa y cuál debe ser el valor del **IP** ?
2. ¿Por qué las instrucciones de cada secuencia deben estar en posiciones consecutivas en memoria ?
3. ¿Por qué las variables no deben necesariamente estar una debajo de otra ?
4. ¿Por qué no hay problemas si las instrucciones y los datos comparten una misma memoria ?
5. Durante la ejecución de un programa, ¿cómo se localizan las instrucciones si ocupan distinta cantidad de celdas en memoria ? y ¿cómo se localiza el dato que cada instrucción opera ?
6. ¿Hay algún problema si la dirección presente en una instrucción coincide con su cód-op ó con el cod-op de otra instrucción ?
7. En las instrucciones tratadas, ¿quién indica que el resultado debe ir a AX ?

Más adelante se trata la representación de los números **enteros** y **reales**, y al final de estos temas también se dan ejercicios integradores como el recién desarrollado, también para ser verificados mediante el Debug.

Apéndice 2 de la Unidad 1

SISTEMAS OPERATIVOS: su lugar y funciones como parte del software

SOFTWARE DAÑINO: VIRUS

Clasificación del software

Podemos clasificar el software en tres categorías: **software de aplicación**, **software de base** y **software "dañino"** (virus).

SOFTWARE DE APLICACIONES

El **software de aplicaciones** son los programas que desarrolla o adquiere un usuario para que un sistema de computación realice las tareas que requiere.

SOFTWARE DEL SISTEMA

Junto con un computador se vende o entrega el denominado "software **del sistema**" (o parte del mismo) sin el cual su manejo sería bastante complicado para la realización de las tareas requeridas, y su programación estaría a cargo de especialistas en el hardware, amen de resultar muy lenta y engorrosa. También *por seguridad* no conviene en muchos sistemas que cualquier usuario acceda a discos o destruya información de otros.
Este software se compone, pues, de *programas* para llevar a cabo funciones del sistema *estrechamente relacionadas con el hardware* de un sistema de computación (operaciones de E/S, supervisión de multitarea, traducción de lenguajes, manejo de archivos, etc.).
La función del software del sistema es controlar y dirigir la operación de una computadora, de modo que al usuario le parezca estar frente a una potente máquina **"virtual"**, fácil de operar y programar, con la que se puede "dialogar". Así no tiene que vérselas con la máquina "real", electrónica.
Ésta en esencia sólo realiza un limitado número de operaciones elementales a gran velocidad, pero "sobre ella" el software del sistema *simula* otra máquina virtual, que ante el usuario aparenta ser cualitativamente superior.
Vale decir que *los programas del sistema sirven para que un computador pueda usarse para resolver los problemas de aplicaciones de los usuarios, y en general han sido desarrollados para un determinado tipo de procesador.*

Cuánto de este software necesita un computador depende de su aplicación. Si está *dedicado* siempre a una misma tarea, ejecutando sólo un programa, lo más probable que no hagan falta programas del sistema. En cambio si es usado para múltiples usos, o sea para *propósitos generales,* es indispensable contar con una jerarquía de tales programas.

El *software del sistema* puede clasificarse como sigue:

- el **"software de base"**
- el **"software de control de comunicaciones"**
- el **"software de administración de base de datos"**

El *software de base* (**sistema operativo + utilitarios**) es el principal encargado de transformar la máquina "desnuda" en otra virtual, con facilidades y potencialidades propias.
Fue el primero que se desarrolló para ayudar al usuario en el desarrollo y ejecución automática de sus programas de software de aplicaciones; así como para controlar dicha ejecución, y salvar errores que puedan subsanarse durante la misma (como los de lectura de disco y otros). La experiencia fue indicando que existe un

conjunto de procesos, como los de E/S, la traducción de un lenguaje de programación de alto nivel (Cobol, Fortran, Basic, Pascal, C, ...) a instrucciones de máquina, ciertos cálculos rutinarios, y otros, que independientemente del proceso de datos que realice un computador, siempre aparecen en alguna etapa del mismo.

Los fabricantes de computadoras desde un principio comenzaron a proveer los programas y subrutinas estándares para poder realizarlos, a la vez que tornaban más automático y fácil el manejo de las máquinas, merced a otros programas que también vendían. Su desarrollo implica muchas horas-hombre de trabajo, que un usuario común no puede concretar. Aparecieron así los primeros sistemas operativos (SO), como también se indica en el Apéndice 3.

Un **sistema operativo** puede definirse como un conjunto de *programas* que controlan la operación automática de un sistema de computación, con dos funciones complementarias

I. Para que sea una máquina virtual *fácil de operar* y programar.
II. Para administrar los recursos de dicho sistema, a fin de optimizar su funcionamiento, detectar errores e intentar salvarlos

Según el SO que se trate, se da distinta importancia a estas funciones.
Con el fin de facilitar la operación de un computador, un SO decodifica un conjunto de *comandos* que el usuario ordena por teclado, los cuales conforman un "lenguaje de control de trabajos". En el presente, a través de programas interactivos como el Windows, estos comandos se ordenan clickeando un botón del mouse sobre la porción de pantalla que representa el comando en cuestión.

Un SO puede dividirse en módulos que administran los **cuatro recursos** de un sistema de cómputo: UCP, MP, PERIFÉRICOS Y ARCHIVOS.

Así se tiene:

- Programas para determinar cuál será el próximo programa que ejecutará la UCP, y ordenar su ejecución sin intervención del operador.
- Programas para establecer los lugares disponibles de la MP donde se almacenarán programas a ejecutar y sus datos.
- Programas para procesos de E/S de datos entre MP y periféricos. (Para leer/escribir un sector de un disco, para leer/escribir un port de interfaz, borrar pantalla, etc.)[1]
- Programas para manejar archivos en discos y cintas.
  (creación, borrado, apertura, cierre, escritura, lectura, etc de archivos).

En **multiprogramación** *("multitasking"- multitarea)* varios programas presentes en MP dan lugar a procesos que se disputan los 4 recursos citados. El SO optimiza el funcionamiento del conjunto, oficiando de "árbitro", posibilitando que al mismo tiempo, mientras la UCP ejecuta uno de los programas de usuario, se realicen también procesos de E/S correspondientes a otros programas de usuario, almacenados en MP.
Los programas de un SO están constituidos básicamente por las mismas instrucciones que los programas de usuario, sólo que organizadas para cumplir distintos propósitos (como los arriba indicados). Asimismo, *estos programas* –como cualquier otro– *son ejecutados por la UC de la forma vista.*
Por ejemplo, no existe una instrucción "especial" del SO, que ordene controlar si hay o no papel en una impresora, sino una *secuencia* de instrucciones *simples* que combinadas adecuadamente cumplen dicho cometido, formando parte de una subrutina del SO.
Los SO pueden clasificarse como sigue:

- **SO monotarea**: sólo pueden controlar la ejecución de una tarea del usuario por vez. Simplemente cargan y ubican en MP la aplicación en curso, posibilitando que pueda usar los recursos del sistema. Cuando aparece un comando tipo EXIT el SO da por finalizada la aplicación, y se encarga de encadenar la siguiente. Ejemplo el D.O.S.

- **SO multitarea:** permite la multiprogramación, cargando y ubicando en MP diversas aplicaciones, pro-porcionando a cada aplicación la posibilidad de usar los recursos existentes. Controla que la UCP ejecute sucesivamente porciones de cada una, con la suficiente rapidez para dar la impresión que cada tarea o usuario tiene todo el sistema a su disposición. Ejemplos: los SO "UNIX" y "OS/2" y "WINDOWS NT"

Existen dos clases de SO multitarea:

[1] Las rutinas de manejo de cada periférico, que contemplan temporizaciones, sincronizaciones, detalles circuitales del mismo, se conocen como el **driver** del periférico en cuestión.

a. **SO de "tiempo compartido"** *("time sharing")* que asignan a cada aplicación el tiempo que periódicamente, en una "ronda" de procesos, tiene asignado para que la UCP ejecute parte del mismo.

b. **SO de "tiempo real"**: cada tarea se ejecuta durante un tiempo que depende de determinados eventos –como ser la realización de una E/S– que ocasionan la *conmutación* de una tarea a otra, volviendo más tarde a completarse el proceso de una actividad inconclusa.

Los **programas utilitarios** agilizan ciertos procesos que requiere el uso normal de un computador. Así entre otros tenemos:

- **Programas traductores (compiladores** e intérpretes) a código de máquina de programas escritos en lenguajes de alto nivel (Fortran, Cobol, Basic, etc).
- **Programas editores de enlace ("link-editores")** que acoplan subrutinas a programas compilados, de modo que puedan ejecutarse como un todo.
- **Programas editores de texto** o de programas en curso de desarrollo.
- **Programas cargadores**, que pasan programas de discos a MP.
- **Programas de servicio**, como los que pasan datos de disco a cinta.
- **Programas** para manejar "hojas electrónicas" y "menús".
- **Programas para depurar errores** en otros programas en desarrollo.

Se trata de programas que dan "apoyo", en el sentido que preparan la ejecución de otros programas, y ayudan al usuario a desarrollar nuevos programas. Estos programas son coordinados por el SO. Mediante un "lenguaje de comandos" el usuario especifica al SO qué desea realizar, para que sea éste el que tome el control del proceso a partir de dicha orden, quedando usuario libre de esa responsabilidad.

Con el advenimiento del teleprocesamiento en gran escala, con computadores conectados a redes, se fueron desarrollando **"software de control de comunicaciones"**, encargado de la gestión y manejo de las comunicaciones a distancia. Gracias al mismo, cuestiones tales como los protocolos para establecer y concluir una comunicación entre computadores y/o terminales, la verificación de errores en la transmisión de datos, y el pedido de retransmisión en caso de error, son "transparentes" para el usuario, que así no debe desarrollar un software para comunicaciones a distancia. En general este software se activa a través del SO, ante un requerimiento de un programa de usuario de hacer una entrada o salida desde o hacia una terminal o computadora remota.

El concepto de **"base de datos"** o **"banco de datos"** se refiere a una colección de datos sobre un mismo tema, interrelacionados, almacenados con un mínimo de redundancia, tales que los mismos puedan ser usados por tantas aplicaciones como sea posible.
En una primer etapa, las denominadas bases de datos eran varios archivos relacionados, diseñados para una aplicación determinada, o para un grupo de aplicaciones muy semejantes, a las que un programa accedía a través del módulo administrador del SO.
Así el archivo del departamento de compras de una empresa servía en menor grado para producción, pero no era muy útil para otras gerencias. A medida que las bases de datos se emplearon para múltiples aplicaciones fue necesario asegurar que su crecimiento, al varias la estructura general de los datos, no requiera modificar los programas de aplicación que la utilizaban, y viceversa.
Por otra parte, surgieron nuevas exigencias para las bases de datos: deberían permitir que los usuarios utilicen los datos de una manera que no fuera siempre la prevista por los diseñadores, para poder responder a distinto tipos de consultas. No es lo mismo pedir un listado del personal de una empresa por apellido, edad, cargo y sueldo, que solicitar a una base qué empleados ganan más de X pesos, su nombre y edad. Esto último supone un programa "inteligente", con capacidad para deducción lógica y para dar respuestas a consultas de alto grado de abstracción.
Fue necesario desarrollar programas complejos para crear y mantener bases de datos, que relacionaran entre sí los distintos archivos de una base para que ella se comporte frente al usuario como si virtualmente tuviera cierta "inteligencia". Apareció así el **"software de administración o manejo de base de datos"** (sus siglas en inglés son **DBMS**).

Diferenciación en MP de software y datos:

Los datos a procesar y los resultados son registrados en MP y registros de la misma manera que los programas mediante estados eléctricos de los circuitos. Si bien resulta claro que los datos no forman parte del hardware, en cada aplicación es menester distinguir cuáles son los datos y cuál es el software requerido para procesarlos.

En general, los datos se almacenan en zonas de MP separadas de la zona donde está el programa que indica cómo procesarlos y ubicarlos.

En cada procesamiento los datos a tratar no forman parte del software que describe su tratamiento.

Es factible que un programa oficie de datos para otro programa. Por ejemplo un programa que está en un lenguaje que la UCP no puede ejecutar directamente (Cobol, C, Fortran, Basic, Pascal, ...) debe traducirse a instrucciones en código de máquina, mediante un programa traductor correspondiente, el compilador.

SOFTWARE DAÑINO ("VIRUS")

El **software "dañino"** está constituido por los "**virus**", que son *programas.* Un programa "virus" consta de dos partes.

Una de ellas al ser ejecutada por la UCP, sirve para la autocopia del programa, con el fin de autoreproducirse tantas veces como pueda, para pasar de un computador a otro. La otra parte, al ser ejecutada produce los daños para el que ha sido concebido el virus.

El "contagio" de un virus consiste en pasar *subrepticiamente* a formar parte de otro programa o de un programa en disco (archivo "ejecutable", tipo .EXE, .COM, .SYS, .BIN, .BAT, los programas que están en los sectores de booteo de un disco y otros). Luego tiene lugar una fase de "incubación" o propagación, durante la cual sólo se ejecuta la porción que autoreproduce el programa virus. Cuando el virus se autocopió un cierto número de veces, o en una fecha prefijada, o cuando el usuario realiza algo especificado, se ejecuta la porción que provoca el daño objeto del virus.

Por ejemplo una forma típica de entrar un virus es cuando un disquete con un "game" infectado (programa ejecutable conteniendo un programa virus) es copiado de un disquete al disco rígido. Cuando se quiere usar el "game" se ejecuta éste y el programa virus, que así se auto reproduce, y copias del mismo se instalan en otros programas del usuario. Asimismo cualquier copia del "game" que se pase a un amigo propagará el virus, y aunque luego el "game" se borre del disco, ya copias del mismo infectaron otros programas. Otros virus pasan a memoria cada vez que un archivo infectado se ejecuta, y residen en memoria esperando que se ejecute otro archivo, para infectarlo. Otra forma de contagio es encender el computador con un disquete en cuyo sector de booteo se ha introducido un programa virus. Puesto que cuando un computador arranca primero intenta hacerlo desde el programa de booteo del disquete, una copia del programa virus pasará luego al sector de booteo del disco rígido. De esta forma, cada vez que luego se encienda el equipo, se generará en memoria una copia del programa virus. Existen virus que suprimen la protección contra escritura de archivos y que restauran la última fecha de actualización de un archivo para que no se detecte que lo han cambiado. Asimismo pueden detectar las llamadas que se hacen a un archivo, de modo de simular la información que se obtendría si el archivo no estuviera infectado. Los virus "polimórficos" encriptan los códigos de las instrucciones que los componen de una manera distinta en los archivos que infectan. Con este auto encriptamiento evitan ser detectados por los programas "anti-virus"

El daño de un virus puede tener como objetivo el borrado de información (archivos o discos completos), la alteración de tablas o parámetros que utiliza el SO (como la FAT –tabla de localización de archivos–), o directamente la modificación archivos con programas del SO. Existen virus que suplantan al SO en funciones claves, como la lectura/escritura del disco, asignación de memoria, obtención de información acerca de la configuración del sistema, etc.

También pueden destruir hardware, como ser obligar al cabezal del disco rígido que salte continuamente entre dos posiciones extremas (lo cual a su vez produce la pérdida de la información contenida en el disco), o intensificar la acción del haz de rayos catódicos, para producir un daño irreparable en la pantalla del monitor.

Apéndice 3 de la Unidad 1

HISTORIA DE LA COMPUTACION

CONTENIDOS DE ESTE APENDICE

Este apéndice no debe verse meramente como una "historia", sin más, de la evolución técnica del procesamiento de datos. Ella es más bien un pretexto para recopilar todos aquellos aportes teóricos y tecnológicos que están presentes hoy día en las modernas computadoras.

La gran mayoría de ellos no fueron pensados para las computadoras, pero son muchos los conceptos de la Lógica, y las ideas subyacentes en ciertos dispositivos mecánicos de cálculo, y otros, anteriores a las computadoras, que fueron incorporados a éstas.

Sólo ha variado su forma de implementación, que pasó a ser electrónica, de modo de lograr mayor velocidad y confiabilidad operativa, acorde a una sociedad industrial que devora información en sus más variadas formas.

Al tratar las generaciones de computadoras se establece –quizás por primera vez– un paralelo entre los avances del hardware y software.

Por otra parte, una función complementaria que intenta cumplir esa sección, es familiarizar al lector con términos usuales en el ámbito de la informática, tales como Sistemas Operativos, multiprogramación, memoria virtual, etc., dando una primer idea sobre los mismos, como un diccionario..

Finaliza esta historia con una breve referencia al proyecto japonés de "Quinta generación de computadoras", en relación con la inteligencia artificial.

INTRODUCCIÓN

La palabra **computar** proviene del latín. Significa genéricamente *contar*, *calcular*, aunque hoy día está asociada a una máquina determinada.

Los procesos de datos, y en particular la computación, en el sentido genérico citado, son tan antiguos como el hombre.

Antes de iniciar la cronología de los acontecimientos más significativos ocurridos en este área, trataremos dos formas ancestrales de procesamiento: el uso de los dedos y del ábaco para contar y calcular.

De las primeras etapas no han quedado manifestaciones visibles; son difíciles de precisar, pero están apareciendo vestigios indirectos de ellas, cada vez correspondientes a épocas más remotas .

Podemos decir que la primer herramienta para computar o primera computadora, fueron los dedos de las manos del hombre. Con ellos simbolizó otros conjuntos de entes o sucesos numéricamente equivalentes. Asimismo, manteniendo los dedos en determinada posición, pudo memorizar y comunicar números. Puesto que la suma y la resta implican contar, aumentando o reduciendo el número de dedos estirados, fue posible sumar y restar. Hoy en Africa, una persona cuenta con sus dedos los grupos de de diez formados con los dedos de otra otra persona, ampliando el rango de una cuenta.

Todas esta técnicas manuales usadas para representa números con los dedos, son *posicionales*, como también lo es nuestro sistema decimal. No es casual que "dígito" –palabra latina que significa dedo, se use también para indicare cada una de las posiciones de un números, y se hable de computadoras "digitales".

La etapa siguiente de la computación pudo haber empezado con la representación simbólica de números mediante montoncitos de diez guijarros o piedritas. Del uso de estas últimas para dicho fin, ha derivado la palabra "**cálculo**", siendo que en latín piedras es *calculus* .

De gran trascendencia fue la posterior invención del **ábaco**, que fue el segundo computador digital. Primitivamente se realizaba practicando varios surcos paralelos iguales sobre la superficie de la tierra o arena, esta última a veces contenida en una bandeja. Por cada objeto contado se coloca una piedra en el surco que esté más a la derecha (unidades). Cada vez que en éste había diez piedras, se quitan las mismas y se reemplazan por una sola piedra colocada en el surco siguiente de la izquierda (decenas), obteniéndose así el equivalente a diez dedos del hombre. Cuando se tenía diez piedras en el surco de las decenas, también se reemplazaban por una sola piedra colocada en surco siguiente de la izquierda (centenas).

Se lograba así un equivalente a los dedos de diez hombres, sólo con una veintena de piedras.

De esta forma, *con unas pocas piedras se podían contar conjuntos tan grandes de elementos* como fuese necesario. Bastaba con agregar nuevos surcos a la izquierda, y seguir con la misma mecánica de conteo descripta

Asimismo, se podía interrumpir transitoriamente una cuenta, dado que el resultado alcanzado quedaba registrado por las piedras contenidas en cada ranura.

Posteriormente el ábaco fue motivo de varias mejoras que le dieron más velocidad de cómputo y portabilidad. Los surcos fueron reemplazados por finas varillas paralelas de madera paralelas, sujetas a una base, sobre las cuales se podían ensartar las "cuentas". La figura A3.1 ilustra un ábaco americano precolombino.

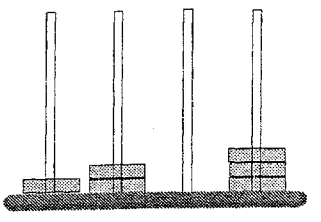


Figura A3.1

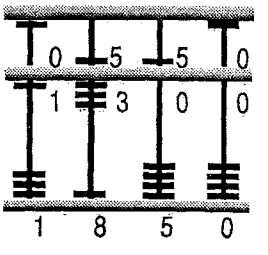


Figura A3.2

En Oriente encerraron sobre varillas de madera 9 (ó diez) cuentas o bolillas deslizables hacia arriba o hacia abajo, resultando así mas veloz. En el presente el ábaco japonés en lugar de las nueve cuentas por hilera tiene dos "campos" (figura A3.2) que se suman. Uno con cuatro cuentas de valor unitario,

y otro con una cuenta que vale por 5 al ser movida hacia la divisoria. En esta se suman ambos campos.
Por sumas repetidas se multiplica, y multiplicaciones repetidas permiten hallar potencias. Operando en forma inversa, es posible restar, dividir y obtener raíces cuadradas.

Durante milenios el ábaco fue la calculadora "de bolsillo" de las más variadas culturas, incluida la romana, hasta el advenimiento de la notación numérica "arábica", que permitió operar como lo hacía el ábaco, pero con lápiz y papel..
El ábaco sigue siendo muy usado hoy día en Oriente, donde es manejado a una velocidad prodigiosa. Debemos mencionar el torneo realizado en 1946 entre un operador yankee, experto en calculadoras eléctricas de escritorio, y otro japonés experto en ábaco, quien ganó 4 a 1 en cálculos complejos, que incluían las cuatro operaciones fundamentales.
Cuando se generalizó en Europa la notación simbólica que imita su operación el ábaco fue sustituido.

CRONOLOGIA DE PROCESOS DE DATOS MANUALES, MECANICOS Y APORTES TEORICOS

Desde su aparición, la subsistencia de los seres vivientes está ligada a su capacidad de procesar estímulos para responder a las variaciones del ambiente, actividad que comparte el hombre a nivel instintivo, como se planteó en la sección 1.1
La transmisión de información a nivel genético también es esencial para cada especie.

1.000.000 AC? Cuando emerge el hombre como tal, como animal simbólico, permanentemente necesita procesar símbolos que también le permiten remitir a entes y sucesos no presentes físicamente. Esto tiene lugar con el lenguaje hablado y la mente, lo cual implica procesar símbolos, para combinarlos; recordarlos y comunicarlos, amen de captarlos. Por lo tanto, el proceso de símbolos es tan ancestral como el hombre, siendo que también está presente en las especies vivientes.

300.000 AC ? Además del cerebro se usan otros medios para registrar símbolos, Mediante los dedos, marcas en maderas y cuerdas, conjuntos de objetos pequeños (piedras, guijarros), el hombre representa otros conjuntos de entes numéricamente equivalentes, que necesitó contar. Al contar hasta un cierto número de elementos se hace una marca o signo que representa a todos ellos

30.000 AC: De esta época del Cro-Magnon se conservan pinturas rupestres para registrar entes y sucesos en general relacionados. Hay indicios de representación de calendarios.

10.000 AC: En Medio Oriente se emplean medallones de arcilla con sistemas de numeración decimal y hexadecimal.
Se supone la existencia de ábacos.

3000 AC: Los egipcios tenían un sistema de numeración jeroglífico. Grupos de 1, 10, 100, ... eran indicados por un símbolo distinto y agrupados se sumaban, idea que usaron los romanos.
Usaban números fraccionarios y ecuaciones. En Babilonia nació el sistema sexadecimal que hoy empleamos para medir tiempo y ángulos. No se sabe si usaba cero, pero ya un mismo signo podía tener varios significados, (sistema tipo posicional). Allí se han encontrado tablillas de arcilla, donde se indican paso a paso algoritmos para cálculos complejos con ecuaciones y fórmulas. Su notación para los fraccionarios les permitió desarrollar aproximaciones sucesivas para calcular raíces cuadradas, que siglos más tarde adjudicaron a Newton. Un producto a,b se hacía despejando en $(a+b)^2 = a^2+2a.b+b^2$ usando tablas de cuadrados a fin de facilitar cálculos. Para dividir multiplicaban por la inversa: a:b = = a. 1/b . Resolvían sistemas de 2 ecuaciones con 2 incógnitas y ecuaciones cuadráticas. Usaron el cambio de variable $t = ax$ y calculaban la suma de progresiones aritméticas y geométricas.[2]
Los chinos usaban un sistema decimal con jeroglíficos. Un ábaco primitivo con palillos de dos colores permitía operar con números positivos y negativos. Asimismo resolvían sistemas de ecuaciones lineales (con matrices) por un método semejante al de Gauss.
En la India 800 AC parece surgir el sistema decimal posicional y más tarde matemáticos hindúes como Aryabhata y Brahmagupta hicieron importantes contribuciones a la matemática.
Grecia fue cuna de Tales de Mileto, Pitágoras y de Euclides de Alejandría

350 AC
El "Organon" de ARISTOTELES compendia sus obras sobre Lógica, que entonces se llamaba "analítica". Esto es, el análisis de lo que consideraba la estructura común a todo razonamiento: el silogismo y sus variedades formales, independientes de los significados en juego. Es un estudio del pensamiento y su coherencia, con vistas a alcanzar la verdad.
ARQUIMEDES desarrolló un método de cálculo infinitesimal para determinar volúmenes de cuerpos con formas curvas, anticipándose notablemente a los desarrollos del siglo XVII.

Siglo I (Era cristiana)
Los matemáticos hindúes emplean un sistema posicional decimal, que incluye el número cero como símbolo equivalente a un alambre del ábaco en el que no se uso ninguna cuenta. Ahora se podrá escribir un número por grande o pequeño que fuera, combinando diez símbolos básicos. Esta nueva simbología permite fecundos desarrollos matemáticos, en parte trabados por las representaciones numéricas anteriores.
Los hindúes predominan en matemática hasta el siglo XII.
En América precolombina, los mayas usaban el ábaco, y varios sistemas posicionales que incluían el número cero.

Siglo VII
El matemático indio Brahmagupta, enuncia una serie de procedimientos para el cálculo numérico decimal.

Siglo IX
ALKIUORITMI, matemático irani, basándose en Brahmagupta, enuncia los procedimientos para realizar, paso a paso, las operaciones aritméticas básicas, que usamos actualmente para realizarlas con papel y lápiz. De su nombre deviene la palabra "*algoritmo*" o algoritmia. Su libro "Algebrwel mucabala" dió nacimiento a la palabra "álgebra".

Siglo XII
La notación numérica arábica es introducida en Italia, junto con sus métodos de cálculo para las operaciones mercantiles.

Siglo XIII
El monje Raimundo Lulio crea la "máquina de la verdad" formada por tres discos que permitían inferir la verdad o falsedad de enunciados conformados con combinaciones de letras de los discos, que simbolizan atributos, virtudes, sujetos lógicos y preguntas. Así, la letra B representa Dios, la bonada, y la pregunta si algo existe.

Siglo XV
Se introduce la coma decimal para separar la parte entera de la fraccionaria .

[2] Ver al respecto "Siria Histórica, Cuna de la Ciencia" de Samira Abdel Masih. Editorial- Universidad Abierta Interamericana-2005

1614

Juan NEPER, escocés, concibe la representación de cualquier número N como 10^X, siendo x lo que se dio en llamar el "logaritmo" en base 10 de N ($x = \log_{10} N$). Aparecen las tablas de logaritmos que dan X para cada N, y permiten transformar cualquier producto o división en una suma o resta de logaritmos, respectivamente. Neper también construye unas tablillas de madera grabadas, que llevaban su nombre, que permiten multiplicar rápidamente dos números Estas se usaban en Oriente desde mucho tiempo atrás.

Estas ideas dieron nacimiento a las **reglas de cálculo**, utilizadas hasta 1970 como *calculadoras portátiles.*

1623

SCHICKARD, alemán, construye para Kepler, un reloj de cálculo, basado en ruedas contadoras, para hacer sumas y restas de hasta seis dígitos

1642

Calculadora mecánica para sumar y restar

Blaise PASCAL, francés, inventa una pequeña *calculadora mecánica*, con engranajes, capaz de sumar números y totalizar suresultado Se llegaron a construir 50 de estas máquinas (figura A1.3).

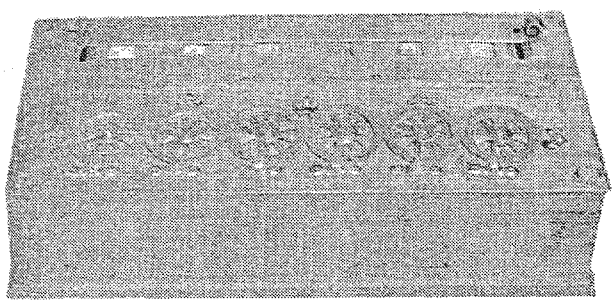


Figura A1.3

Reemplazaba el cálculo con pluma y papel, o con ábaco, *sin requerir un operador experto*. Era más lenta que el ábaco, pero más cómoda, dado que mecaniza las acciones de mover una a una las cuentas de éste, de cambiar 10 cuentas de una posición por una de la siguiente (arrastre o acarreo), y daba directamente el resultado mediante un número decimal.

Consiste en dos contadores-registros con engranajes superpuestos. El de más arriba –con ventanillas– oficiaba de acumulador de resultados (como el visor de una calculadora de bolsillo actual). El registro inferior servía para entrar los números a sumar mediante el posicionamiento de sus diales decimales externos (equivalentes a un teclado). Estos eran ruedas independientes entre sí. Cada uno al ser girado hacia la izquierda, provocaba que la rueda del acumulador que estaba sobre ella girara igual número de posiciones. Girada a la derecha no producía efecto mecánico alguno.

Las ruedas del acumulador estaban relacionadas mecánicamente como las del cuenta-vueltas de un pasacasetes: cuando una rueda pasaba por la ventanilla de 9 a 0, hacía girar una posición a la rueda vecina izquierda. Se sumaba así automáticamente uno a esta posición.

Para sumar dos números **A** + **B**,. primero se giraban a la izquierda las ruedas-diales hasta lograr que todas las ruedas del acumulador indiquen cero en las ventanillas. Luego, los diales se giraban a la derecha hasta quedar también en cero.

A continuación, girando cada dial hacia la izquierda, se formaban las unidades, decenas, etc., del número **A**, por lo que simultáneamente se movían igualmente las ruedas del acumulador, formándose en el mismo también el número **A** (sin que se produzca ningún arrastre), que aparecerá en las ventanillas.

Nuevamente se giraban a la derecha los diales hasta quedar indicando cero. Este sentido de giro no afectaba las ruedas del acumulador, que seguían guardando el valor de **A**

Girando otra vez a la izquierda los diales, se formaba ahora las unidades , decenas , etc . , del número **B**.

Cada dial hacía mover igualmente su correspondiente rueda del acumulador, a partir de la posición que había quedado cuando se registró el número **A**, pudiéndose producir arrastres hacia otras ruedas. De esta forma al número **A** del acumulador se le suma **B**, y el resultado acumulado **A+B**, aparecerá en las ventanillas. Si después de este

Si después de este resultado se quería sumar otro número C, se procedía de la misma forma como se hizo con **B**.

La máquina hacia la resta por el método del complemento a la base (diez) del minuendo. Por ejemplo, para efectuar 710 – 84, se realizaba 710 + 916 = **1626**, siendo 916 = 1000 – 84.

El resultado eran los dígitos (subrayados), que se formaban sin considerar el último uno de la izquierda, o sea 626.

Varias ideas de Pascal siguen vigentes hasta hoy: la necesidad de un registro acumulador, y el uso de un mismo dispositivo para sumar y restar. Esto último sumando al minuendo el complemento a la base del sustraendo

1671

Primer calculador mecánico con 4 operaciones

Godtfried W. LEIBNITZ, alemán, le incorpora a la calculadora de Pascal la multiplicación y la división. Para ello, cada dígito presentaba un engranaje tambor con 9 dientes de longitud escalonada (figura A1.4), vinculado a una rueda del acumulador. Esta podía engranar, según su posición, seleccionable, de 0 a 9 dientes del tambor citado.

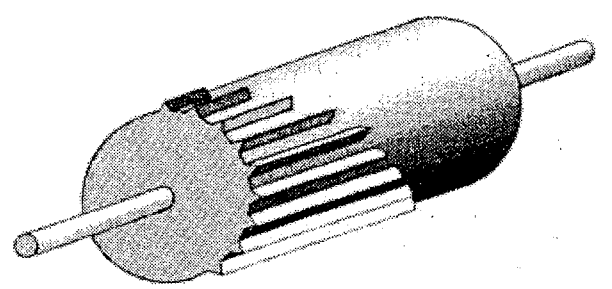


Figura A1-4

De este modo, existían dos contadores: uno que ejecutaba de 1 a 9 veces la suma de un número consigo mismo, y otro que controlaba cuantas veces debía repetirse dicha suma. *Tanto la máquina de Pascal, como la de Leibnitz, se basan en la acción básica de contar*, en este caso dientes de engranajes.

A partir de entonces se hicieron sucesivos intentos para fabricar comercialmente calculadoras mecánicas de mesa.

La tecnología requerida recién pudo concretarse efectivamente hacia 1810, cuando Charles Thomas, alsaciano, fabricó la primer máquina comercializable, producida en serie.

En el presente, aún se usan calculadoras mecánicas, con teclado, y mecanismo para multiplicar impulsado por motor, en reemplazo del accionado manualmente.

Leibnitz en 1703 estudió el sistema binario de numeración.

y otro con una cuenta que vale por 5 al ser movida hacia la divisoria. En esta se suman ambos campos.
Por sumas repetidas se multiplica, y multiplicaciones repetidas permiten hallar potencias. Operando en forma inversa, es posible restar, dividir y obtener raíces cuadradas.

Durante milenios el ábaco fue la calculadora "de bolsillo" de las más variadas culturas, incluida la romana, hasta el advenimiento de la notación numérica "arábica", que permitió operar como lo hacía el ábaco, pero con lápiz y papel..
El ábaco sigue siendo muy usado hoy día en Oriente, donde es manejado a una velocidad prodigiosa. Debemos mencionar el torneo realizado en 1946 entre un operador yankee, experto en calculadoras eléctricas de escritorio, y otro japonés experto en ábaco, quien ganó 4 a 1 en cálculos complejos, que incluían las cuatro operaciones fundamentales.
Cuando se generalizó en Europa la notación simbólica que imita su operación el ábaco fue sustituido.

CRONOLOGIA DE PROCESOS DE DATOS MANUALES, MECANICOS Y APORTES TEORICOS

Desde su aparición, la subsistencia de los seres vivientes está ligada a su capacidad de procesar estímulos para responder a las variaciones del ambiente, actividad que comparte el hombre a nivel instintivo, como se planteó en la sección 1.1
La transmisión de información a nivel genético también es esencial para cada especie.

1.000.000 AC? Cuando emerge el hombre como tal, como animal simbólico, permanentemente necesita procesar símbolos que también le permiten remitir a entes y sucesos no presentes físicamente. Esto tiene lugar con el lenguaje hablado y la mente, lo cual implica procesar símbolos, para combinarlos; recordarlos y comunicarlos, amen de captarlos. Por lo tanto, el proceso de símbolos es tan ancestral como el hombre, siendo que también está presente en las especies vivientes.

300.000 AC ? Además del cerebro se usan otros medios para registrar símbolos, Mediante los dedos, marcas en maderas y cuerdas, conjuntos de objetos pequeños (piedras, guijarros), el hombre representa otros conjuntos de entes numéricamente equivalentes, que necesitó contar. Al contar hasta un cierto número de elementos se hace una marca o signo que representa a todos ellos

30.000 AC: De esta época del Cro-Magnon se conservan pinturas rupestres para registrar entes y sucesos en general relacionados. Hay indicios de representación de calendarios.

10.000 AC: En Medio Oriente se emplean medallones de arcilla con sistemas de numeración decimal y hexadecimal.
Se supone la existencia de ábacos.

3000 AC: Los egipcios tenían un sistema de numeración jeroglífico. Grupos de 1, 10, 100, ... eran indicados por un símbolo distinto y agrupados se sumaban, idea que usaron los romanos.
Usaban números fraccionarios y ecuaciones. En Babilonia nació el sistema sexadecimal que hoy empleamos para medir tiempo y ángulos. No se sabe si usaba cero, pero ya un mismo signo podía tener varios significados, (sistema tipo posicional). Allí se han encontrado tablillas de arcilla, donde se indican paso a paso algoritmos para cálculos complejos con ecuaciones y fórmulas. Su notación para los fraccionarios les permitió desarrollar aproximaciones sucesivas para calcular raíces cuadradas, que siglos más tarde adjudicaron a Newton. Un producto a,b se hacía despejando en $(a+b)^2 = a^2+2a.b+b^2$ usando tablas de cuadrados a fin de facilitar cálculos. Para dividir multiplicaban por la inversa: a:b = = a. 1/b . Resolvían sistemas de 2 ecuaciones con 2 incógnitas y ecuaciones cuadráticas. Usaron el cambio de variable $t = ax$ y calculaban la suma de progresiones aritméticas y geométricas.[2]
Los chinos usaban un sistema decimal con jeroglíficos. Un ábaco primitivo con palillos de dos colores permitía operar con números positivos y negativos. Asimismo resolvían sistemas de ecuaciones lineales (con matrices) por un método semejante al de Gauss.
En la India 800 AC parece surgir el sistema decimal posicional y más tarde matemáticos hindúes como Aryabhata y Brahmagupta hicieron importantes contribuciones a la matemática.
Grecia fue cuna de Tales de Mileto, Pitágoras y de Euclides de Alejandría

350 AC
El "Organon" de ARISTOTELES compendia sus obras sobre Lógica, que entonces se llamaba "analítica". Esto es, el análisis de lo que consideraba la estructura común a todo razonamiento: el silogismo y sus variedades formales, independientes de los significados en juego. Es un estudio del pensamiento y su coherencia, con vistas a alcanzar la verdad.
ARQUIMEDES desarrolló un método de cálculo infinitesimal para determinar volúmenes de cuerpos con formas curvas, anticipándose notablemente a los desarrollos del siglo XVII.

Siglo I (Era cristiana)
Los matemáticos hindúes emplean un sistema posicional decimal, que incluye el número cero como símbolo equivalente a un alambre del ábaco en el que no se uso ninguna cuenta. Ahora se podrá escribir un número por grande o pequeño que fuera, combinando diez símbolos básicos. Esta nueva simbología permite fecundos desarrollos matemáticos, en parte trabados por las representaciones numéricas anteriores.
Los hindúes predominan en matemática hasta el siglo XII.
En América precolombina, los mayas usaban el ábaco, y varios sistemas posicionales que incluían el número cero.

Siglo VII
El matemático indio Brahmagupta, enuncia una serie de procedimientos para el cálculo numérico decimal.

Siglo IX
ALKJUORITMI, matemático irani, basándose en Brahmagupta, enuncia los procedimientos para realizar, paso a paso, las operaciones aritméticas básicas, que usamos actualmente para realizarlas con papel y lápiz. De su nombre deviene la palabra "*algoritmo*" o algoritmia. Su libro "Algebrwel mucabala" dió nacimiento a la palabra "álgebra".

Siglo XII
La notación numérica arábica es introducida en Italia, junto con sus métodos de cálculo para las operaciones mercantiles.

Siglo XIII
El monje Raimundo Lulio crea la "máquina de la verdad" formada por tres discos que permitían inferir la verdad o falsedad de enunciados conformados con combinaciones de letras de los discos, que simbolizan atributos, virtudes, sujetos lógicos y preguntas. Así, la letra B representa Dios, la bonada, y la pregunta si algo existe.

Siglo XV
Se introduce la coma decimal para separar la parte entera de la fraccionaria .

[2] Ver al respecto "Siria Histórica, Cuna de la Ciencia" de Samira Abdel Masih. Editorial- Universidad Abierta Interamericana-2005

1614

Juan NEPER, escocés, concibe la representación de cualquier número N como 10^X, siendo x lo que se dio en llamar el "logaritmo" en base 10 de N ($x = \log_{10} N$). Aparecen las tablas de logaritmos que dan X para cada N, y permiten transformar cualquier producto o división en una suma o resta de logaritmos, respectivamente. Neper también construye unas tablillas de madera grabadas, que llevaban su nombre, que permiten multiplicar rápidamente dos números Estas se usaban en Oriente desde mucho tiempo atrás.

Estas ideas dieron nacimiento a las **reglas de cálculo**, utilizadas hasta 1970 como *calculadoras portátiles.*

1623

SCHICKARD, alemán, construye para Kepler, un reloj de cálculo, basado en ruedas contadoras, para hacer sumas y restas de hasta seis dígitos

1642

Calculadora mecánica para sumar y restar

Blaise PASCAL, francés, inventa una pequeña *calculadora mecánica*, con engranajes, capaz de sumar números y totalizar suresultado Se llegaron a construir 50 de estas máquinas (figura A1.3).

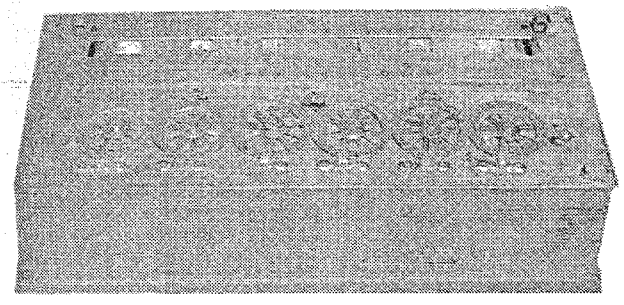


Figura A1.3

Reemplazaba el cálculo con pluma y papel, o con ábaco, *sin requerir un operador experto*. Era más lenta que el ábaco, pero más cómoda, dado que mecaniza las acciones de mover una a una las cuentas de éste, de cambiar 10 cuentas de una posición por una de la siguiente (arrastre o acarreo), y daba directamente el resultado mediante un número decimal.

Consiste en dos contadores-registros con engranajes superpuestos. El de más arriba –con ventanillas– oficiaba de acumulador de resultados (como el visor de una calculadora de bolsillo actual). El registro inferior servía para entrar los números a sumar mediante el posicionamiento de sus diales decimales externos (equivalentes a un teclado). Estos eran ruedas independientes entre sí. Cada uno al ser girado hacia la izquierda, provocaba que la rueda del acumulador que estaba sobre ella girara igual número de posiciones. Girada a la derecha no producía efecto mecánico alguno.

Las ruedas del acumulador estaban relacionadas mecánicamente como las del cuenta-vueltas de un pasacasetes: cuando una rueda pasaba por la ventanilla de 9 a 0, hacía girar una posición a la rueda vecina izquierda. Se sumaba así automáticamente uno a esta posición.

Para sumar dos números **A** + **B**,. primero se giraban a la izquierda las ruedas-diales hasta lograr que todas las ruedas del acumulador indiquen cero en las ventanillas. Luego, los diales se giraban a la derecha hasta quedar también en cero.

A continuación, girando cada dial hacia la izquierda, se formaban las unidades, decenas, etc., del número **A**, por lo que simultáneamente se movían igualmente las ruedas del acumulador, formándose en el mismo también el número **A** (sin que se produzca ningún arrastre), que aparecerá en las ventanillas.

Nuevamente se giraban a la derecha los diales hasta quedar indicando cero. Este sentido de giro no afectaba las ruedas del acumulador, que seguían guardando el valor de **A**

Girando otra vez a la izquierda los diales, se formaba ahora las unidades , decenas , etc . , del número **B**.

Cada dial hacía mover igualmente su correspondiente rueda del acumulador, a partir de la posici6n que había quedado cuando se registr6 el número **A**, pudiéndose producir arrastres hacia otras ruedas. De esta forma al número **A** del acumulador se le suma **B**, y el resultado acumulado **A+B**, aparecerá en las ventanillas. Si después de este

Si después de este resultado se quería sumar otro número C. se procedía de la misma forma como se hizo con **B**.

La máquina hacia la resta por el método del complemento a la base (diez) del minuendo. Por ejemplo, para efectuar 710 – 84, se realizaba 710 + 916 = **1626**, siendo 916 = 1000 – 84.

El resultado eran los dígitos (subrayados), que se formaban sin considerar el último uno de la izquierda, o sea 626.

Varias ideas de Pascal siguen vigentes hasta hoy: la necesidad de un registro acumulador, y el uso de un mismo dispositivo para sumar y restar. Esto último sumando al minuendo el complemento a la base del sustraendo

1671

Primer calculador mecánico con 4 operaciones

Godtfried W. LEIBNITZ, alemán, le incorpora a la calculadora de Pascal la multiplicación y la división. Para ello, cada dígito presentaba un engranaje tambor con 9 dientes de longitud escalonada (figura A1.4), vinculado a una rueda del acumulador. Esta podía engranar, según su posición, seleccionable, de 0 a 9 dientes del tambor citado.

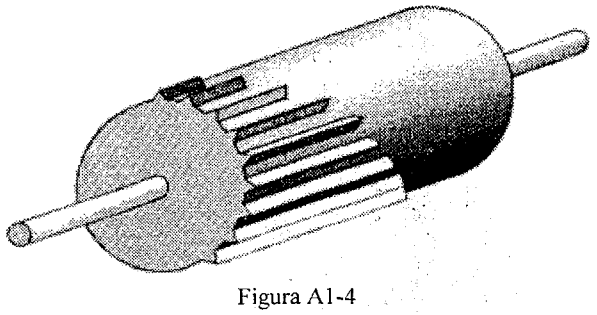


Figura A1-4

De este modo, existían dos contadores: uno que ejecutaba de 1 a 9 veces la suma de un número consigo mismo, y otro que controlaba cuantas veces debía repetirse dicha suma. *Tanto la máquina de Pascal, como la de Leibnitz, se basan en la acción básica de contar*, en este caso dientes de engranajes.

A partir de entonces se hicieron sucesivos intentos para fabricar comercialmente calculadoras mecánicas de mesa.

La tecnología requerida recién pudo concretarse efectivamente hacia 1810, cuando Charles Thomas, alsaciano, fabricó la primer máquina comercializable, producida en serie.

En el presente, aún se usan calculadoras mecánicas, con teclado, y mecanismo para multiplicar impulsado por motor. en reemplazo del accionado manualmente.

Leibnitz en 1703 estudió el sistema binario de numeración.

1808

Joseph JACQUARD, francés, perfecciona el uso de cartones perforados para cambiar a voluntad el dibujo que teje un telar automático. Los ganchos que asían las fibras textiles sólo podrán pasar por los agujeros previstos. Se tenía así, una máquina programable de control de procesos.

1832

Proyecto de computador mecánico

Charles BABBAGE, inglés, fue el primero en intentar construir una máquina de calcular automática, capaz de realizar cálculos extensos sin la constante acción del operador, que cada vez debe darle los datos y la operación a realizar.

Hacia 1823 Babbage construye su "*Maquina de diferencias*", que calcula mecánicamente valores numéricos sucesivos de polinomios de 2° grado, con 8 cifras decimales, para construir tablas. Este cálculo se basa en el algoritmo de las diferencias finitas, y permite hallar mediante sucesivas sumas valores numéricos de polinomios, para valores sucesivos de X.

Puesto que un gran número de funciones se pueden aproximar con suficiente exactitud mediante polinomios, resulta así sencillo hallar sus valores numéricos para cada valor de X.

La máquina (figura A1.5) constaba de registros contadores mecánicos, constituidos por ruedas. Cada una almacenaba un dígito de un número decimal. Los registros vinculados realizaban sumas con un mecanismo similar al que usó Pascal.

Para hallar un resultado, valores iniciales conocidos eran puestos en los registros. Cada resultado era mostrado en una chapa de cobre que era perforada por agujas de acero.

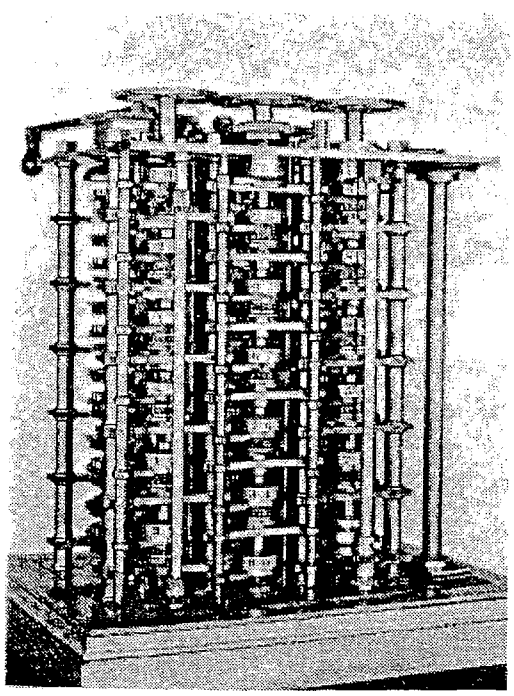


Figura A1.5

En 1822 Babbage obtiene fondos oficiales para construir otra máquina de diferencias más grande, capaz de calcular tablas de valores para polinomios de sexto grado, con 8 dígitos decimales, que nunca terminó.

Hacia **1832** concibe su *Máquina Analítica*, una computadora mecánica programable y automática, para realizar cualquier tipo de cálculo. No pudo concretarla, debido a que la precisión mecánica que requerían las piezas no era factible de obtener entonces, y en ella trabajó hasta su muerte en 1871.

La descripción que quedó de esta máquina coincide en gran medida con la arquitectura de los prototipos de las primeras computadoras electrónicas, y Aitken la utilizó.

Debería operar con números decimales de 50 dígitos, con una memoria capaz de guardar 1000 de estos números, constituida por contadores mecánicos. Cada suma insumiría un segundo, gracias a un sumador con transporte anticipado.

Mediante dos juegos de tarjetas perforadas –tipo Jacquard– se lleva a cabo la ejecución de cada programa. Una tarjeta de un juego ordenaría a la U aritmética cuál de las 4 operaciones se realizará. Otra tarjeta del segundo juego, determinaba en qué registro contador de la memoria se encontraba un dato a operar

La U. aritmética operaba sobre dos números registrados en dos contadores indicados por las tarjetas citadas, y el resultado era registrado en uno de los contadores que forman la memoria

Aunque en un principio, la máquina no trabajaba con el programa almacenado en memoria, podía saltar de una secuencia de instrucciones a otra, en función de un resultado alcanzado. Esto se haría desplazando los juegos de tarjetas hacia delante o atrás una cierta distancia. Los resultados aparecerían perforando chapas de cobre.

Lady LOVELACE, colaboradora de Babbage, *fue la primer persona que desarrolló programas y creó un lenguaje simbólico de máquina.* Escribió: "La máquina puede efectuar un exámen para determinar si ha tenido lugar un evento entre varios posibles, y seguir después el camino que convenga" ..."puede hacer aquello que sepamos ordenarle que haga".

1854

George BOOLE, inglés, publica su obra "Las leyes del pensamiento", en la cual utiliza símbolos matemáticos como variables lógicas, para representar enunciados de la lógica deductiva, que podían ser verdaderos o falsos.

Asimismo, presenta reglas mecánicas para determinar la falsedad o verdad de enunciados compuestos, mediante la manipulación de dichos símbolos, usando solo tres operaciones lógicas básicas: negación, conjunción y disyunción.

1890

Herman HOLLERITH, en EEUU incorpora la tarjeta perforada al proceso *de datos*, para poder llevar a término el censo poblacional, dado que de efectuarse en forma totalmente manual se terminaría después de 1900, cuando se empezaba el próximo censo.

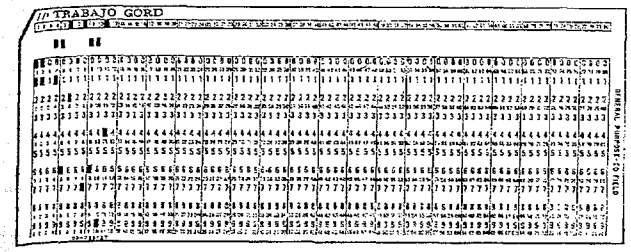
Se dio cuenta que muchas preguntas podían contestarse por Si ó No, perforando o no un lugar de la tarjeta. Si hacia falta codificar números, para la edad y otras, estableció la forma de hacerlo mediante perforaciones en posiciones prefijadas.

También inventó una máquina "tabuladora", para leer, clasificar y distintos rubros del censo. Detectaba la presencia de perforaciones en cada tarjeta mediante escobillas que generaban (o no). corrientes eléctricas, que excitaban a máquinas electromecánicas a relés.

Después que las tarjetas fueron perforadas por los censistas, eran primero procesadas por una máquina clasificadora, que las disponía en cierta secuencia en casilleros numerados. Pasaban luego a una intercaladora, que podía fusionar en uno solo, dos mazos de tarjetas, o bien comparar la correlación entre dos mazos, sin combinarlos.

Una vez dispuestas en el orden requerido, un calculador realizaba operaciones aritméticas, y proporcionaba resultados. perforando tarjetas.

Por último la "tabuladora" sumarizaba los datos del calculador en otras tarjetas perforadas.
De esta forma, el tiempo de censado duró tres años
En 1896, Hollerith fundó una compañía de venta de tarjetas y máquinas, para aplicaciones administrativas e industriales, que más tarde se fusionó con IBM.
El proceso de datos con tarjeta perforada comenzaba a aplicarse en las grandes corporaciones, mucho antes que se inventaran las computadoras.



1907
La **válvula electrónica tríodo de vacío** es inventada por Lee DE FOREST

PRECURSORES Y PROTOTIPOS DE CALCULADORAS AUTOMATICAS

1925
Vanevar BUSCH, ingeniero norteamericano, construye una máquina capaz de resolver ecuaciones diferenciales, con relés (contactos mecánicos accionados eléctricamente)

1936
Konrad ZUSE, alemán, siendo estudiante de ingeniería inicia la construcción de la calculadora digital automática Z1, para cálculos corrientes en estadística.
Alan TURING investiga en Inglaterra aspectos teóricos de la lógica, en lo referente a la relación existente entre una descripción simbólica de un proceso real, y la posibilidad de que un autómata pueda seguir tal descripción con la suficiente exactitud como para reproducirlo.
En el área de proceso de datos, implica si el problema asociado con dicha descripción tiene o no solución efectiva mediante un programa ejecutable por un computador
Para dilucidar esto, propuso un modelo matemático de autómata general –hoy conocido como "máquina de Turing"– asimilable a una computadora. Si con tal modelo se puede representar un problema, éste tiene solución mediante un algoritmo (programa).

1937
Howard AITKEN de la U. de Harvard (EEUU), con apoyo de IBM empieza a desarrollar la calculadora MARK I con relés, basada en el modelo de Babbage.

1938
Claude SHANNON, norteamericano, propone tratar la lógica deductiva en la forma conocida como álgebra de Boole, mediante el sistema binario.

1939
G. ATANASOFF ayudado por C. Berry, construye en Iowa (EEUU) un prototipo de calculadora digital electrónica con válvulas electrónicas, para aplicaciones limitadas. Tenía una memoria basada en capacitores, como las actuales DRAM.
Es visitado por Mauchly, futuro creador de la ENIAC.
George STIBITZ, de la Bell Telephone de EEUU, construye la Model T, calculadora con relés, no programable, para cálculos complejos, con datos numéricos entrados por el teclado de una teletipo, que también los puede transmitir a distancia.

1941
Calculadora automática programable, de uso general, Z3, de K. ZUSE (Alemania)
Realizada con 600 relés electromecánicos. El programa se almacenaba en una película de cine con perforaciones, o sea, *era externo* al sistema de relés procesador de datos.
El movimiento mecánico de arrastre de la misma hacía entrar la instrucción a ejecutar. La entrada y salida de datos se efectuaba mediante cinta de papel perforada.

1942
Calculadora electrónica automática ABC (Atanasoff-Barry-Computer), en Iowa (EEUU), realizada con válvulas electrónicas. No era de uso general. Fue pensada para resolver sistemas de ecuaciones lineales.
Para esa misma época se propone el uso de discos y tambores magnéticos para almacenar grandes cantidades de información.

1943
En Inglaterra se termina de construir, secretamente durante la guerra, la computadora electrónica COLOSSUS, bajo la dirección de H. NEWMAN Constaba de 2000 válvulas. Se fabricaron 10 equipos. Descifraba códigos de guerra alemanes.
Bajo la dirección de John MAUCHLY y Prosper ECKERT, de la Universidad de Pennsylvania (EEUU) se comienza a construir la **ENIAC**

1944
ZUSE construye en Alemania una máquina que puede considerarse como primera con *programa almacenado internamente*
Comienza a operar en EEUU la "**MARK I**" o ASCC (Automatic Sequence Control Calculator), *primer computador de uso general.* Primero se usó para confeccionar tablas matemáticas, y luego para resolver sistemas de ecuaciones algebraicas y diferenciales. Operaba internamente en el *sistema decimal*, con números enteros de hasta 23 dígitos y signo. Cada número se podía almacenar en registros aritméticos de 24 ruedas decimales. Existían 72 de estos registros. Cada uno se podía sumar o restar con el contenido de otro cualquiera que luego guardaba el resultado. Las ruedas, en forma independiente, podían girar si se acoplaban o no al eje, que giraba continuamente. Mediante 10 escobillas se obtenía eléctricamente su posición. En una suma, las ruedas de un registro giraban según el número registrado en el otro, que así se suma al número contenido en el primero, como en la calculadora de Pascal. Datos del tipo constantes, se podían entrar manualmente en registros con llaves selectoras decimales. Las instrucciones de un programa estaban en forma binaria en una cinta de papel perforado, materializadas por la presencia o no de agujeros en la misma (hasta 24 por fila). Así, una fila podrá indicar la operación a realizar o en que registros estaban los operandos. Esta información almacenada en la cinta, que oficiaba de memoria externa, era tomada a razón de una fila por vez, para establecer qué hacer a cada paso. En función del valor alcanzado por una variable, se podía dejar de lado una cinta para seguir con otra secuencia de instrucciones de otra cinta. Esto era equivalente a la existencia de instrucciones de salto condicionado, requeridas en cualquier computadora automática Los datos se introducían en los

registros mencionados mediante tarjetas perforadas de IBM.
Los resultados podían salir por tarjeta perforada o ser escritos en papel, por máquinas de escribir comandadas.
Podía trabajarse con doble precisión, o sea duplicar, si fuera preciso, el número de dígitos de trabajo.
Componían el equipo unas 250.000 piezas y 800.000 mts de cable. Realizaba una 3 sumas por segundo.
Fue usada día y noche durante 15 años. Para muchos es la primer computadora de uso general con tecnología de procesamiento electromecánico (relés y mecanismos)

El matemático húngaro John VON NEUMANN, junto con BURK y GOLDSTINE, estudiando en EEUU las limitaciones de la calculadora automática Eniac, plantearon la necesidad de que estas máquinas posean una memoria interna (o principal), para registrar, además de los datos numéricos involucrados en el cálculo, las instrucciones del programa correspondiente, antes que este sea ejecutado. O sea que éstas también sean números.
Además planteó la conveniencia de usar numeración binaria en el interior de las máquinas
Hasta entonces en los registros internos de las calculadores solo se memorizaban los datos numéricos y los resultados que se iban obteniendo. El programa se guardaba externamente en una cinta movida por un motor. Las instrucciones pasaban de una a una a la unidad de proceso, con una velocidad máxima propia del mecanismo, que es muy lenta respecto de la velocidad de proceso electrónico de la UCP.
La existencia de una memoria interna, que proveyera rápidamente las instrucciones y los datos posibilitaría una mayor velocidad de procesamiento. Asimismo, será rápido pasar de una secuencia de instrucciones a otra, que esté en la misma memoria, para lo cual deberían existir instrucciones de salto que así lo ordenasen.
Otra ventaja sería el uso de una misma forma de codificación (la binaria) para las instrucciones y los datos.
Todo esto posibilitaría un mejor aprovechamiento y flexibilidad en el uso de los equipos, y el concepto de software adquiriría su verdadera dimensión

Desde entonces se habla del MODELO o ARQUITECTURA VON NEUMANN para hacer mención al esquema de la figura 1.@ que se conserva en esencia hasta el presente, en el cual se ejecuta una instrucción por vez.

Las nuevas arquitecturas posibilitan la ejecución de varias instrucciones simultáneamente

1946

Funciona la **ENIAC** (Electrical Numerical Integration Automatic Computer) proyectada por Eckart y Mauchly para generar tablas de balística. Constaba de 18.000 válvulas y 1.500 relés. Disipaba 150 Kvatios Pesaba 30 toneladas.
Internamente operaba en el sistema decimal, realizando 5000 sumas/seg (unas 1000 veces mas rápido que la Mark I). Esencialmente, las ruedas decimales de la Mark I fueron reemplazadas por válvulas contadoras electrónicas (décadas) con arrastre decimal electrónico. *Para cada aplicación había que modificar el interconexionado de un conjunto de cables, configurando una especie de ROM alterable. Esta preparación insumía de media hora a un día de trabajo.*
Por lo tanto, no presenta memoria interna para instrucciones. Existían 20 registros internos en la UCP, y podía realizar salto de secuencia condicionales. La ENIAC operó durante 10 años.

1947

F. Williams desarrolla en Inglaterra la denominada **memoria Williams** interna, de acceso directo (RAM) basada en un tubo de rayos catódicos que almacenaba electrostáticamente en la pantalla información binaria, el ser impactada en puntos por el haz electrónico móvil. La misma podía ser luego sensada rápidamente en una placa de un capacitor ligado a la pantalla.

1948

BARDEN Y BRATTAIN descubren el **efecto transistor** en EEUU.

PROTOTIPOS DE COMPUTADORES ELECTRONICOS CON ARQUITECTURA VON NEUMANN:

1949

En la U. de Cambridge opera la EDSAC (Electronic Delay Storage Automatic Calculator), proyectada por. WILLLIAMS. *Primer máquina tipo Von Neumann* con programas intercambiables almacenados en memoria interna.
Esta era una línea de retardo de mercurio, con información recirculante). La memoria auxiliar era un cilindro magnético. Poseía 3000 válvulas electrónicas.

1951

En el MIT (EEUU) se desarrollan la Wirlwind I (20.000 sumas/seg) de respuesta rápida en tiempo real, para control de tráfico aéreo y procesos fabriles, y las memorias internas RAM de núcleos de ferrite ("core") .

1952

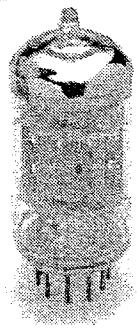
Comienza a funcionar la computadora **IAS** (Institute for Advanced Studies), creada por Eckard y Mauchly.
La memoria la formaban 40 tubos electrostáticos Williams.

LAS GENERACIONES DE COMPUTADORAS

La división en "generaciones" se basa *en la tecnología circuital empleada*, siendo que los computadores de una generación presentan *mayor velocidad, mayor capacidad de memoria y menor tamaño* que los de la anterior.
En las generaciones primera a cuarta, predomina la construcción de computadores según el modelo de Von Neumann, que ha merecido sucesivas mejoras en velocidad. El denominado "Proyecto de 5ta generación" plantea el desarrollo de computadores con procesadores en paralelo (Arquitecturas designadas "no Von Neumann")

PRIMERA GENERACION DE COMPUTADORAS (1952-1958)

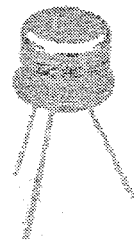
Fabricación de computadores en serie, con tecnología de **válvulas electrónicas** (figura derecha)
Ejecutaban algunas miles de instrucciones por segundo. Hasta la Pegasus (1958) la UCP sólo tenía un solo registro, designado Acumulador.
Almacenaban de 10.000 a 20.000 bytes en su memoria. Eran muy voluminosas.
Medios de entrada/salida: tarjetas perforadas, cintas de papel perforadas.
Memoria principal con tiempo de acceso muy grande en comparación con los tiempos de procesamiento internos de la UCP. Memoria secundaria: cinta magnética.



	HARDWARE PARTICULAR	SOFTWARE GENERAL
1952	**UNIVAC I** (Universal Automatic Computer), creada por Eckart y Mauchly, socios en una empresa comprada luego por Remington Rand. Operó hasta 1963 en la Oficina de Censos, Trabajaba con 12 dígitos decimales, codificados en binario. Presentaba memoria Williams y unidades de cinta magnética. Mecanismo de interrupción. Esta computadora "universal" fue pensada para aplicaciones de estadística y comerciales.	Se proveen, listos para usar, programas "cargadores", desarrollados para ubicar en la memoria principal otros programas a ejecutar . Programación en lenguaje simbólico de máquina (**Assembler**), que permite expresar los códigos binarios de las instrucciones de máquina mediante símbolos de nuestro alfabeto, para facilidad del programador Requiere un programa traductor, que provee el fabricante, para transformarlo en códigos que la máquina entiende. Univac desarrolla los primeros lenguajes de programación mas próximos al lenguaje humano, o de alto nivel, que requieren de un programa traductor para llevarlos a código de máquina . Ellos fueron **'Mathmathic'** y **'Flowmatic'** precursores del Algol y Cobol
1953	**701/IBM**, con un repertorio de 24 instrucciones. Memoria Willliams de 2K, de 36 bits por celda. Para usos científicos.	
1955	**704/IBM**, primer máquina electrónica con representación de números en punto flotante. Registros de la UCP para direccionamiento indirecto e indexacion de datos en memoria. Memoria principal de 4K de ferrites. y circuitos para operaciones en punto flotante. Repertorio de 40 instrucciones. 2,5 veces mas rápida que la 701. **UNIVAC 1103**, para usos científicos. Representación en punto flotante. Primer máquina con mecanismo de interrupciones de programas en ejecución, para pasar a ejecutar una rutina cuando ocurre cierto evento, y luego retomar el programa interrumpido. La UCP tenía un registro.	
1956	**PEGASUS:** primera en tener 8 registros en la UCP.	
1957	**BURROUGHS** desarrolla una computadora totalmente transistorizada para la Fuerza Aérea de EEUU.	
1958	**709/IBM** introduce los *canales* procesadores de las operaciones de entrada/salida que realizan los periféricos bajo su control. La UCP se libera de dicha tarea, para ejecutar distintos programas. Se consigue simultaneidad de operaciones de E/S de datos, correspondientes a distintos programas. La UCP delega a los canales la gestión de los periféricos según instrucciones preparados para ellos en la memoria principal. Esta innovación, junto con las interrupciones, son modificaciones del esquema básico de Von Neumann, en el cual las E/S se hacían a través del acumulador de la UCP. Permiten un mejor aprovechamiento y automatismo de las máquinas, especialmente en multiprogramación. Ultima máquina de IBM con válvulas. **PILOT**: multiprocesador financiado por el National Bureau of Standards. Varias UCP comparten la memoria principal. Cada una realiza una tarea distinta: aritmética de E/S, y todas están supervisadas por un control maestro	Entre 1954/57 Backus desarrolla el lenguaje de programación **FORTRAN** (Formula Translation), cuyo programa traductor será empleado en la 709/IBM. Primaba el procesamiento de datos en lotes (batch), con mucho trabajo de preparación y recolección de datos fuera de línea, ("off line"). Esto es, en dispositivos fuera del control de la UCP, tales como las perfo-verificadoras de tarjetas. Lotes de tarjetas perforadas, recolectadas en distintas fuentes eran llevadas al centro de cómputo, y luego agrupadas para ser procesadas secuencialmente por los programas correspondientes (Proceso "batch") . Un proceso comenzaba sólo cuando el anterior terminó. Lenguaje **LISP** (J. Mc Carthy) para inteligencia artificial. Lenguaje **ALGOL** (Algorithmic Lenguaje) Se proveían programas para procesos "batch" (en lotes) En proceso de archivos, los lotes de tarjetas pasan a una cinta magnética, que luego será procesada, para generar una nueva cinta actualizada. Aparecen, junto con los discos magnéticos, las operaciones en línea (on line), en las cuales los datos se procesan sin demoras, desde periféricos ligados a la UCP Los fabricantes generan programas para manejo de discos y demás periféricos en uso. Estos programas y otros serán la base de los futuros **Sistemas Operativos** (SO)

SEGUNDA GENERACION DE COMPUTADORAS (1959-1964)

Hacia 1960 la primera generación. con válvulas electrónicas resulta obsoleta.
Se impone el transistor (figura de la derecha), más confiable, de menor tamaño, menor disipación de calor y más rápido que la válvula para cambiar de estado. Así se alcanzan velocidades de procesamiento de centenares de miles de instrucciones por segundo.
También aumenta la velocidad de acceso a la memoria principal, que en todas las máquinas pasa a ser de núcleos de ferrite
Crece el repertorio de instrucciones de máquina
Dispositivos de E/S: tarjetas perforadas, tinta de papel perforado teletipos, impresoras, cintas magnéticas de alta velocidad.
Memoria auxiliar: cintas y discos magnéticos.
Se generaliza el uso de los canales y aparecen elaborados mecanismos de manejo de interrupciones.
Aparecen los sistemas operativos, el tiempo compartido ("time sharing y se generalizan los lenguajes de programación de alto nivel.
Las maquinas disminuyen de tamaño y costo. Se instalan equipos por valor de 4.000 millones de dólares.
Los principales avances y exponentes se indican a continuación, siendo que el esfuerzo estuvo centrado en lograr máquinas más veloces y bajar costos. El desarrollo del software estuvo relegado, a pesar de su importancia (salvo en la B5000 de Burroughs)

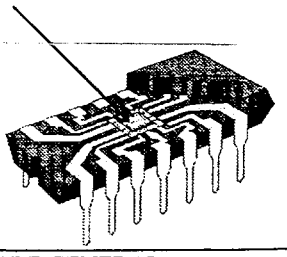


	HARDWARE PARTICULAR	SOFTWARE GENERAL
1959	**RCA 501** y **NCR-GE 304**, con UCP conteniendo 8000 diodos y 4000 transistores. Hacían 20.000 sumas/egundo . **TEXAS INSTR.** *patenta el circuito integrado ("chip"), en una pastilla plástica que será la tecnología de la tercer generación de computadoras*	Programa traductor (compilador) para lenguaje de programación **COBOL** (Common Business Oriented Lenguage).
1960	**Serie 7000/IBM**, con memoria de ferrites de 32.000 celdas de 32 bits, y 25 veces más rápida que la 701. 185 tipos de instrucción y 7 registros en la UCP. La 7090 era como la 709 (tipo ENIAC) pero transistorizada, para usos científicos. **PDP/1** de DEC (Digital Equipment Corp): primer intento de K. Olsen de realizar una MINICOMPUTADORA de menor tamaño y costo (U$S 120.000) que las fabricadas hasta entonces.	Aparecen los "paquetes de software", provistos por los fabricantes de computadoras. Se desarrollan sistemas operativos que residían permanentemente en memoria, para manejar las operaciones de E/S, limitar los tiempos de ejecución de programas, y otras tareas. Se los conoció como "Sistemas Ejecutivos". Surge el lenguaje APL.
1961	**1620** y **1401** de IBM: máquinas más pequeñas, organizadas hacia el carácter, y para números decimales de longitud variable, orientada hacia pequeñas empresas.	Software para simular la memoria virtual mediante los discos.
1962	**ATLAS** (Universidad de Manchester-Inglaterra), con memoria "virtual", para simular una memoria principal mas grande que la físicamente real. **D-825** de Burroughs: multiprocesador con hasta 4 UCP conectados a 16 módulos de MP conmutables, para aplicaciones militares	Lenguaje PL/I , para usuarios de IBM. Sistema de tiempo compartido (time sharing) desarrollado en el MIT. Los usuarios interactúan a través de terminales con teclado en forma rotativa con una computadora central Cada uno lo hace durante una fracción de segundo, pero se le parece que él solo usa la máquina.
1963	**B5000** de Burroughs, máquina organizada en pilas alejada del modelo de Von Neumann, Pensada para un eficaz procesamiento del lenguaje de alto nivel Algol 60. O sea el hardware en función del software. Memoria virtual por segmentación.	
1964	**7094** de IBM con solapamiento de funciones en la ejecución de cada instrucción. 50 veces más rápida que la 701. Presentaba solapamiento para la ejecución de las instrucciones.	

TERCERA GENERACIÓN DE COMPUTADORAS (1964-1972?)

El desarrollo de los circuitos integrados en pequeña y mediana (figura derecha) escala de integración, y de plaquetas impresas con caminos de cobre para soportarlos, permitieron equipos más compactos, confiables y económicos. En éstos predomina el uso del disco rígido.
Hacia 1975 el total de equipos de computación instalados asciende a U$S 24000 millones Se generaliza el uso de las computadoras para los más diversos tipos de actividades. El sistema 360 de IBM (y sus sistemas operativos), es el equipamiento representativo de esta generación, en la cual aparecen las *minicomputadoras*.

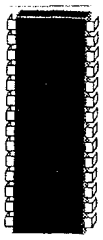
CHIP



	HARDWARE PARTICULAR	SOFTWARE GENERAL
1964	**System/360 de IBM**: (de G. Amdhal) computadora de uso universal (científico- comercial): para reemplazar tanto a una 7000 como a una 1401, a las cuales podía simular. Máquina *microprogramada*: la secuencia de pasos necesaria para ejecutar una instrucción está almacenada en una ROM, en la UC. Con memoria de ferrite, de un millón de celdas de 32 bits, ampliable a 16 millones (MB). Gran diversidad de canales y periféricos. La UCP presenta 16 registros de uso general. Operaciones en binario, decimal codificado en binario y binario punto flotante (simple y doble precisión). Hacía 10^6 sumas por segundo, con números de 32 bits. Sistema de *prioridades para interrupción* de programas. Protección de zonas de MP reservadas, y elaborado registro de estado de programa, permiten una *multiprogramación compleja*. Luego se agregan la memoria virtual y la antememoria "*cache*" (memoria rápida ubicada entre la MP y la UCP). **6600** de CDC: para aplicaciones nucleares. Inicia una serie de *supercomputadoras*. Con unidades "pipe line" para efectuar varias operaciones aritméticas en paralelo. Podía ir ejecutando 10 instrucciones juntas. Era 10 veces más rápida que la IBM 7090. Creada por Cray, fue un *prototipo de los procesadores RISC*, por su arquitectura sencilla, y la limitada cantidad de instrucciones.	Los sistemas operativos de IBM apuntaban a cubrir las necesidades de un amplio espectro de usuarios, siendo tomados como modelos por otras empresas. Estos eran el CP67/CM6 para máquinas grandes, los OS/MFT y OS/MVT para medianas y grandes, y el DOS/360 para las más chicas. En general permitían la multiprogramación, el tiempo compartido y la memoria virtual. En materia de lenguajes se universaliza el uso de FORTRAN y COBOL Se comienza a poner énfasis en lenguajes modulares, fáciles de corregir. Se desarrolla en 1965 el BASIC (J.Kemeny y T.Kurtz) para cursos de iniciación a la computación.
1965	**PDP/8** de DEC: *inaugura la era de las minicomputadoras*. La parte central podía entrar en un cajón de manzanas. Costaba 18.000 dólares y realizaba 300.000 sumas por segundo. **Tecnología MOS** con transistores de metal-oxido-semiconductor, que permiten una alta densidad de integración de transistores en un microcircuito. Creada por RCA. *Permitió el advenimiento de los microprocesadores y las memorias de gran capacidad* **RED ARPA** en EEUU: *primera red de minicomputadoras distribuidas*, para transmisión de paquetes de datos.	Los procesos "batch" comienzan a dejar lugar a las operaciones "on line", desde terminales con teclado y pantalla. Esta interactividad hombre-máquina. permite desarrollar más eficazmente nuevos programas, realizar cómputos, y manejar archivos en forma más directa y descentralizada. Un ejemplo son las termina-les usadas en los bancos. Se acelera el ocaso de las tarjetas.
1966	**PDP/11** de DEC: serie de minicomputadoras para uso general.	
1970	**Primer Microprocesador:** el **4004** de INTEL, que operaba de a 4 bits por vez, En dos pastillas con microcircuitos integrados contenía la UCP. En total tenía 2.300 transistores. Ese mismo año aparece el **8008** para 8 bits. Fueron pensados para aplicaciones de control electrónico, no para computación.	Hacia 1970 nace el lenguaje PASCAL, creado por el suizo N. Wirth, para programación estructurada En el MIT se termina el sistema operativo **Multics**, empezado en 1965. Sirvió de base al UNIX
1971	**370** de IBM con memoria virtual. **ILLIAC IV**: desarrollo poco exitoso de un supercomputador con 4 UC. Cada una opera simultáneamente sobre 64 conjuntos de datos organizados (matrices, vectores etc.). Las 4 UC pensadas para hacer 4 cálculos distintos al unísono, resultando 4x64 = 264 cálculos en paralelo. Varias ideas fueron usadas posteriormente. **STÁR 100** de CDC (Compañía de grandes máquinas) Con el mismo fin que la anterior usa el procesamiento *pipe-line* . Como en una fábrica de autos el hardware de la UCP conforma una cadena de unidades (U) dedicadas a una acción especifica: U. de obtención de la Instrucción → U. de interpretación de la misma → U. de localización de datos en MP → U. de cálculo A medida que avanza la ejecución de una instrucción, la U de obtención extrae la siguiente de MP, y así sucesivamente.	Aparece el SO **UNIX**, de los laboratorios Bell. Es eficaz, económico y más sencillo que los SO de IBM. Adaptable a las más diversas arquitecturas, también pasará a. ser el modelo para las más pequeñas Bill Gates que participó de este proyecto, luego aplicó aspectos del mismo al sistema operativo DOS de Microsoft. Lenguaje PROLOG (Calmeraver y Roussel) para inteligencia artificial

Cuarta Generación de Computadoras (1972 – . . ?. .)

El desarrollo de chips en muy grande escala de integración (Very Large Scale Integration-VLSI), con millones de transistores (Pentium tiene 3.200.000) ha permitido el advenimiento de *microprocesadores* que hoy día superan en velocidad al 360 de IBM, referente de la tercer generación.
Ellos permiten fabricar microcomputadoras personales baratas, que han invadido todos los ámbitos, con una gran variedad de periféricos fabricados para las mismas.
Por otra parte, las computadoras personales, cada vez en mayor grado, se comunican entre sí a través de módems y redes globales.
No hay acuerdo general acerca del año establecido como de comienzo de esta generación, ni tampoco cuando termina, o sea si estamos en la cuarta o quinta generación.



1972	Se comercializan las **primeras microcomputadoras** **STAR-100** de CDC, primer supercomputador *vectorial*, que opera sobre datos que son vectores(matrices lineales de números) Una instrucción vectorial equivale a un loop de instrucciones.	Comienzan a imponerse la **programación estructurada**
1974	**8080** de Intel. en un solo chip. Primer microprocesador de uso general. Procesa internamente datos de 8 bits. Lee o escribe de a 8 bits en cada acceso a memoria. Presenta 6000 transistores. **6800** de Motorola. **Primer mouse** e **impresora láser** en el computador **ALTO** de XEROX	Cuando Intel creó el 8080 suministró un disquete con el programa **CP/M** (Control Program/Microprocessor). Este administraba la disquetera, permitía programar en alto nivel, ejecutar programas y dar órdenes.
1975	Aparecen los primeros modelos de la microcomputadora hogareña **Commodore.**. Aoarece el microprocesador **Z80** de Zilog, y el **6502**	Fue creado por Digital Research y era un sistema operativo básico. También se usó en máquinas con el procesador Z80.
1976	**APPLE I y Commodore PET**, microcomputadores personales **CRAY I** supercomputador más rápido de la época.	
1977	**Apple II** **TI99/4A** microcomputador hogareño de Texas Instruments, basado en su microprocesador TMS 9900. RAM de 16KB ampliable a 48 KB **TRS-1** microcomputador hogareño de la Tandy Radio Shack Corp. con el microprocesador Z80 **PET 2001** de Commodore, microcomputador hogareño	
1978	**8086** de Intel. Procesa internamente datos de 16 bits, que lee o escribe 16 bits en cada acceso a memoria. Presenta 29.000 transistores. Utiliza "pipe line" para ejecutar instrucciones. El chip **8087** es el coprocesador matemático opcional para el 8086, para operar en punto flotante (números con coma y otros) Frecuencia reloj máxima: 10 MHz **VAX-11/780** (Virtual Adress eXtension of the PDP**11**) de DEC. en VLSI. Minicomputador en gran medida compatible con los PDP11 de 1 MIPS. **CBM 3032** (Commodore Business Machine). Con teclado, video, doble unidad de disquetes e impresora con cabezal de agujas. Microprocesador 6502. RAM de 32 KB	Microsoft de Bill Gates, elabora la implantación del lenguaje BASIC para la mayoría de las microcomputadoras existentes
1979	**8088** de Intel, que procesa internamente datos de 16 bits, que lee o escribe 8 bits en cada acceso a memoria. El chip **8087** es el coprocesador matemático opcional para el 8088. Frecuencia reloj máxima: 8 MHz **Primer procesador RISC**: el proyecto IBM 801, de J.Cocke	
1980	1980 IBM *elige para sus IBM PC el 8088 de Intel,* en los modelos PC XT y PCjr, con hasta 1 MB de memoria Macintosh elige para sus microcomputadores el microprocesador 68000 de Motorola **RISC I** de Berkeley, (de D. Patterson) y el **MIPS** de la U. de Stanford. (de J.Hennessy) Hasta 5 veces más rápidas que otras computadoras de. igual tecnología	
1981	IBM lanza la **PC XT** con el 8088, 4,77 MHz, 64 KB de memoria, disquetera de 5 ¼, monitor e interfaz para cintas en casetes. **CYBER-205** de CDC máquina vectorial **APPLE II Plus**, computador personal basado en el microprocesador 6502, con 48 KB de RAM. Puede programarse en Basic, Cobol, Fortran. El APPLE III presentaba una unidad de disquete de 5 ¼ . RAM de 96 KB, ampliable a 256 KB **VIC-20** de Commodore, primer computadora color. Costaba U$S 300. Se llegaron a fabricar 9000 unidades por día..	Bill Gates termina el sistema operativo **DOS 1.0** tomando como base el CP/M y el UNIX.
1982	**80286** de Intel, que procesa internamente datos de 16 bits, que lee o escribe 16 bits en memoria. Permite una memoria real de hasta 16 MB. Puede operar en multiprocesamiento, con memoria virtual, si se usa un sistema operativo adecuado. El chip **80387** es el coprocesador matemático opcional	Versión 1.1 del MS-DOS

	para el 80286. Frecuencia reloj máxima: 12,5 MHz. Contenía 134.000 transistores	
	~~Commodore 64~~, computador personal basado en el microprocesador 6510, con 20 KB de ROM, y 64 KB de RAM	
1983	**ATARI 800** microcomputador con microprocesador 6502. RAM de 16KB	
	CRAY X-MP supercomputadora más rápida que la CRAY 1	Aparece la interfaz gráfica **Windows 1.0** de
1984	Aparece la **Macintosh** de Apple	Microsoft, y MS-DOS 2.0
	Aparece la **PC AT, PS/1** y **PS/2**de IBM usando el 80286	Lenguajes: C++ Pascal AFNOR, ADA83
	370-XA de IBM que amplía a 32 las líneas de direccionamiento de memoria que en el modelo 370 eran 24.	Versiones, 3.0 y 3.1 de MS-DOS para PC AT Lenguajes: Prolog III, Concurrent C.
1985	**Micro VAX 32** con un microprocesador	
	80386DX de Intel, con 275000 transistores, que procesa internamente datos de 32 bits, que lee o escribe 32 bits en memoria. Permite una memoria real de hasta 4 GB. Puede operar en multiprocesamiento, con memoria virtual, si se usa un sistema operativo adecuado. El chip **80387** es el coprocesador matemático opcional para el 80386. Frecuencia reloj máxima: 40 MHz.	SO: Amiga OS para Commodore Amiga, MIPS OS para procesadores MIPS Lenguajes: Cobol 85, Object Pascal
	Amiga 1000 de Commodore, primer computador multimedia. Podía exhibir 4096 colores. Presentaba video acelerado y síntesis de voz. La pantalla podía mostrar múltiples pantallas de distinta resolución. Su sistema operativo propio era multitasking. La RAM era de 256 KB.	
	Primeros **RISC** fabricados por H. Packard para sus minicomputadoras.	Versión 3.2 del MS-DOS
1986	**3090/VF** de IBM: es la 370 convertida en un procesador vectorial	AIX de IBM y HP-UX basados en el UNIX
	Caché interno en microprocesadores: en el **SUN3/260** y en el Motorola 68020	Lenguajes: Self y Eifel
	Primeros **multiprocesadores** (varias UCP trabajando en paralelo) producidos comercialmente: **ENCORE MULTIMAX** y el **SYMMETRY**	
	IBM RT PC primer modelo comercial **RISC** de IBM	
	Mini-supercomputadora C-1 de Convex Máquina vectorial diez veces más barata (U$S 500.000) que una supercomputadora grande.	
1987	Estaciones de trabajo **SUN** basadas en el procesador RISC **SPARC** (Scalable Processor ARChitecture).	Lenguajes: Perl 1.000, Haskell 1.0, ADA ISO MS-DOS 3.3 para la PS/2 de IBM
1988	**30386SX** de Intel, que procesa internamente datos de 32 bits, que lee o escribe 16 bits en memoria. Permite una memoria real de hasta 16 MB. Puede operar en multiprocesamiento, con memoria virtual, si se usa un sistema operativo adecuado. Aunque es menos rápido que el 385DX, convenía, por ser más barato, a usuarios, siendo que los chips de memoria disponibles masivamente eran para leer o escribir 16bits por vez. El chip **80387SX** es el coprocesador matemático opcional para el 80386SX. Frecuencia reloj máxima: 33 MHz	IBM y Microsoft lanzan el OS/2, sistema operativo multitarea.
	Los siguientes microprocesadores **RISC** se empleaban en distintos equipos: **SPECTRUM** H. Packard, **CLIPER** de Fairchild, **CRISP** de AT&T, **MIPS** de MIPS Co., **PYRAMID** de Pyramid, y otros. Todos ellos ejecutaban unas 8 a 10 millones de instrucciones por segundo (MIPS), con frecuencias entre 15 y 30 MHz.	Versión 4.0 del MS-DOS para discos mayores que 32 MB. AUX (Apple UNIX)
	88000, procesador RISC de Motorola	Aparecen el lenguaje A y el Tcl/Tk
	80486DX, que procesa internamente datos de 32 bits, que lee o escribe 32 bits en memoria. Permite una memoria real de hasta 4 GB. Puede operar en multiprocesamiento, con memoria virtual, si se usa un sistema operativo adecuado. Contiene dentro del chip una memoria caché de 8 KB, y un coprocesador matemático. Consta de 1.200.000 transistores	
	A 50 MHz procesa hasta 50 MIPS.	
1989	**i860**, procesador RISC de Intel. Con un "pipe line" más elaborado que los	Lenguajes; Borland Pascal Object, Eiffel 2,
1990	anteriores 80x86, acorta el tiempo de ejecución de las instrucciones.	Perl 3.000
	CRAY Y-MP más rápido que el CRAY X-MP (ciclo de reloj 6 nseg.)	
1990	**RS/6000** RISC sistema RISC de IBM, para estaciones de trabajo.	
	Supercomputadores vectoriales NEC SX/2, Fujitsu VP200 y el Hitachi S829	
	Procesadores RISC: **Alpha** de DEC, y **R4400** de Silicon Graphic	
1991	**80486SX** de Intel, similar al 80486DX, pero sin procesador matemático interno. El mismo es externo y opcional:: el **80487SX**	Lenguajes: Pythón derivado del C y del ABC
	Acuerdo entre IBM, Motorola y Apple para implantar un nuevo estándar en hardware y software de PCs, tomando como base el microprocesador **PowerPC** (Performance Optimized With Enhanced RISC)	Nace el **LINUX** basado en el UNIX
1992	**80486DX2** de Intel, similar al 80486DX, pero con una frecuencia interna de reloj máxima: 66 MHz (externamente se comunica a 33 MHz)	Sale la versión Windows 3.1 y DOS 5.0 Solaris de SUN

1993	**Pentium** (80586) que procesa internamente datos de 32 bits, y que lee o escribe 64 bits en memoria. Permite una memoria real de hasta 4 GB. Puede operar en multiprocesamiento, con memoria virtual, si se usa un sistema operativo adecuado. Contiene dentro del chip una memoria caché de 16 KB, y un coprocesador matemático. En una pulgada cuadrada tiene cerca de 3.200.000 transistores. Frecuencia reloj máxima: 100 MHz	Aparece el sistema operativo Window NT y la versión DOS 6.0
	Se trata de un procesador "*superescalar*" que puede terminar de ejecutar dos instrucciones simultáneamente si son simples y cumplen otras condiciones. Para este tipo de instrucciones opera como un RISC. A 66 MHz opera 100 MIPS. En punto flotante es 4 veces más rápido que el 486	Lenguajes: M derivado del MUMPS
1994	**Power PC 601** y **603** para PCs de escritorio y notebooks.	El PERL 5.000
	Paragon de Intel, *multicomputador* que permite conectar hasta 4000 procesadores RISC i8609 en paralelo, con los que llega a 300 GigaFLOPS	
1995	**Arquitectura P6** de Intel, adosado con otro chip para caché externo de 512 KB, con lo cual tiene 31 millones de transistores. Opera interna y externamente con igual número de bits que el Pentium. También tiene dos cachés internos de 8 KB cada uno. Presenta 3 pipe lines, pudiendo así terminar 3 instrucciones simples juntas. Convierte las instrucciones 80x86 en instrucciones simples, tipo RISC. Es 50% más rápido que el Pentium si opera con OS/2 o el Windows NT. Puede conectarse directamente a otros 3 procesadores P6 para multiprocesamiento.	Sale la versión 6.2 del MS DOS Aparece el sistema operativo Windows 95 y el Tru Unix de 64 bits. IBM lanza el OS/2 Warp Windows NT 4.0 Lenguajes: Java 1, Ada 95, Delphi, Eiffel 3
	Pentium Pro (200 Mhz) con arquitectura P6. Con 5,5 y 15 millones de transistores en el procesador y en el caché L2, respectivamente. Para Windows NT y otros sistemas de 32 bits usados en redes.	
1996	**PowerPC 620** de 64 bits, para estaciones de trabajo y servidores de redes de alta velocidad. Utiliza 7 millones de transistores. Muy buena performance en punto flotante, superior a la de enteros.	Lenguajes: PostScript level 3, Eiffel 4 , Haskell 1.3
	K5 procesador RISC de AMD, competidor de Intel. Compatible con los 80x86. Insertable en lugar de un Pentium. Convierte instrucciones CISC en RISC.	Wundows NT 4.0
	Pentium MMX con nuevas instrucciones para multimedia y comunicaciones. De menor rendimiento que el Pentium Pro para sistemas operativos de 32 bits.	
1997	**K6** de AMD (233 Mhz) con núcleo RISC. Agrega instrucciones multimedia.	Lenguajes Fortran 95 ISO, Prolog IV,
	Pentium II (Klamath) 330 Mhz. Conjuga Pentium Pro con MMX. Proc. **MIPS**	OO Cobol, Modula 2 ISO, Alpha 0
1998	**Pentium II Xeon** con caché de mayor capacidad.	Windows 98 Lenguajes C++ , Visual Basic
1999	**Celerón**: Es un Pentium II sin caché L2 adosado. Sustituto del Pentium MMX como micro-procesador barato.	Java 2, Eiffel 4.2, Perl 5.005
	Pentium III (500 Mhz a 1 Ghz) 9,5 millones de transistores. Incorpora nuevas instrucciones SIMD para multimedia y red (Streaming SIMD Extension-SSE).	Mac OS versión 8 Lenguajes M ISO, Delphi 5, Ruby 1.3.2,
	K6-2 de AMD, 9,3 milll, de transistores. 550 Mhz. Cache interno (L1) de 64K	Haskell 98,
	Power PC G4 de Motorota, totalmente RISC, de 500 MHZ.	
2000	**Pentium 4** (1,4 a 1,6 Mhz) con versiones Celerón y Xeón.	Windows 2000 Java Script, C#, Perl 5.6.0
2001	**Pentium 4 Xeon** (1,4 1,5 1,7 Mhz) .	Windows XP Lenguajes: Cobol 2002
2002	**Athlon** de AMD 4ta generación (2 Ghz)	Windows CE para dispositivos móviles (celulares, palm, etc.)
	Itanium de Intel de 64 bits. Duron de AMD para competir con el Celerón.	
	Itanium 2 –IA64 (1 Ghz) de Intel-HP. Reemplaza al Xeon en servidores.	Microsoft Net: para comunicaciones en red.
	De 210 a 410 millones de transistores	Lenguajes: Fortran 2000, Delphi 7, Perk 5,8
2003	**Athlon** (5ta generación) AMD lanza los procesadores **Opteron** para servidores para 32 y 64 bits. 4 GB de RAM. Para multiprocesamiento.	Windows XP 64 bits para plataformas
	Pentium 4 (2,6-3 Ghz) para Hyper Threading (HT).	Lenguajes: Java 2 v1.5, C# ISO, Ruby 1.8
	Pentium 4E (3,4 Ghz) "Prescott" (HT). 7500 a 11000 MIPS.	PHP 4.3.5
2004	**AMD 64** 8va. generación de procesadores x86 con ahorro de energía.	
	Pentium 4 EE (Extreme Edition) con 2 MB de caché L3.	
2005	**Pentium M** bajo consumo, para notebooks, 140 millones de transistores	Lenguajes: Ada 2006, Pythón 2.4.1, C# 3.0
	Pentium D: 2 procesadores en un chip. Cada uno con 2 MB de caché L2. Puede operar con 32+32=64 bits.	PHP 4.4.1, Perl 5.8.7, Ruby 1.8.3
	Opteron Dual Core 2.6 GHZ	
2006	**Athlon 64 X2 Dual Core** 154 a 233 millones de transist. Controlador de memoria incluido Bus Hyper Transport (8 GB/seg-max)	
	Pentium 4 EE Dual Core Tecnología .065 micras 3,46 Ghz.	

PROYECTO "QUINTA GENERACIÓN"

En 1982, en Japón se anunció el comienzo del proyecto de "Quinta Generación de Computadoras", a desarrollar en un lapso de diez años, con el fin de crear una arquitectura basada en una *máquina paralela de inferencias lógicas* y el software correspondiente para la misma.
Se planteó un nuevo modelo de computador, que pretendía innovar en gran medida las técnicas informáticas (como lo hicieron el microprocesador y el microcomputador) en especial en los siguientes aspectos:

Velocidad de procesamiento: se busca procesar más datos e instrucciones por unidad de tiempo.
Para tal fin se requiere circuitos electrónicos integrados más rápidos que los actuales, y una arquitectura tipo *paralelo*.
Esto último supone varios microprocesadores realizando varias tareas simultáneamente, coordinados por otro, que en cada oportunidad puede ser cualesquiera del conjunto.
De este modo se supera la clásica forma de operación secuencial.

Operaciones con lógica simbólica: en el presente, en las computadoras, mediante estructuras numéricas, se representan también letras y caracteres, resultando en general programas y datos excesivamente largos. Se trata ahora de poder lograr programas acordes al tipo de datos a procesar. Este aspecto se relaciona con el tema siguiente.

Aplicaciones orientadas hacia la Inteligencia Artificial: esto es, computadoras con características apropiadas para esta disciplina, que se viene desarrollando desde la década del 60.
Para poder entender los objetivos y resultados de este proyecto, es necesario desarrollar las nociones que siguen.

La **Inteligencia Artificial (IA)** supone una nueva forma de resolver problemas, desarrollando sistemas informáticos "inteligentes", que presentan ciertas características propias del hombre, en el manejo de los símbolos, en áreas tales como" entendimiento de lenguajes, aprendizajes, resolución de problemas, consultas, simulación y otros.
La IA supone la existencia de computadoras, y para el manejo efectivo de los símbolos se nutre de otras disciplinas, como la Lógica formal, la Sicología del Conocimiento y la Lingüística.
Suele dividirse la IA en tres campos de aplicación:

Sistemas expertos: para brindar información relevante sobre; ciertos temas, realizar diagnósticos, pronósticos, simular situaciones reales, etc., sobre la base de un almacenamiento de conocimientos previos, introducidos por un experto en el tema.
Robótica: para aplicaciones industriales, y con todo lo relacionado con la visión y manipulación de objetos.
Procesadores de lenguajes humanos: para facilitar la comunicación de las computadoras con los hombres.

Los Sistemas Expertos (SE) se refieren, en primera instancia, a computadoras programadas para brindar información relevante sobre un tema determinado, emulando a un experto en el mismo.
Un SE permite superar las limitaciones de la programación tradicional para resolver problemas de alta complejidad, relacionados con el suministro de información precisa, clara y resumida en respuesta a preguntas que formulan usuarios, a fin de tomar decisiones.

Estos, interactuando con el computador, podrán conocer las mejores opciones o diagnósticos en determinados temas de interés. En los sistemas de enseñanza inteligente se podrá recibir instrucción y ser examinado por la computadora.
Un factor que ha contribuido al desarrollo de los SE es la necesidad de las grandes empresas de simular situaciones reales por computadoras.

Al comenzar una partida de ajedrez, o durante su desarrollo, cada jugador no prueba todas las jugadas posibles. Su experiencia, lo lleva a dejar de lado las que tienen poca probabilidad de ser apropiadas, dedicando su tiempo a analizar un reducido subconjunto de jugadas posibles
En general, el método *heurístico* se aplica a ciertos problemas, y supone una serie de tanteos, para indagar si se está o no por un camino correcto. En cambio el método algorítmico garantiza que producirá un resultado buscado, al aplicar un procedimiento totalmente especificado. Del mismo modo, los programas usados en IA, en general, incorporan un conocimiento que permite descartar soluciones altamente improbables de ser adecuadas, y generan aquellas que pueden serlo.
O sea que se basan en reglas heurísticas, que guían a los programas según una línea de razonamiento.
Dicha incorporación de conocimiento se puede llevar a cabo al generar el sistema, merced a un trabajo conjunto del programador del SE y de un experto en el tema, que permitirá sistematizar el conocimiento existente sobre el mismo en un conjunto de reglas. Por ejemplo, un supuesto sistema de diagnósticos mecánicos, deducirá en función de los datos sensados, que un automóvil tiene problemas de encendido, sin perder tiempo en determinar antes qüe no es la bomba de nafta, o la de agua, etc.

La estructura del programa de un SE permite:

- Obtener a partir de una gran masa de información almacenada sobre el tema, respuestas a preguntas no previstas por los programadores
- Aprender a inferir en forma limitada a partir de la información existente. Ello permite:

- Actualizar la información almacenada,
- Usar la nueva información para realizar inferencias reque
. requeridas.

Un SE puede aprender a adquirir nueva información ya sea:

a. Infiriendo información de la existente
b. Obteniéndola del usuario, si éste advierte que la misma influirá en el resultado buscado,

Básicamente, un SE consta de dos porciones:

1. Una "*base de conocimiento*", que almacena toda la información disponible
2. Un denominado "*motor de inferencia*" , que es un programa encargado de encauzar el SE, de forma que responda coherentemente a las preguntas formuladas . Para tal fin, debe seleccionar las reglas de inferencia necesarias para la respuesta adecuada, entre todas las reglas que están almacenadas en la base de conocimiento.

Si bien el desarrollo de la IA recién empieza, sus aplicaciones son bastante restringidas. En el presente se observan en los SE limitaciones importantes:
No pueden inferir a partir de postulados o teorías, –tampoco pueden razonar por analogía– ni obtener información com-

pletamente nueva respecto de la existente. Su capacidad de aprendizaje es muy elemental.

Lo anterior trae aparejado:

- Carencia de sentido común.
- Capacidad de inferencia *degradable* al extremo, si el problema o pregunta no se ajusta a los fines para los cuales el SE fue creado.

Debe notarse que ya, desde el advenimiento de las computadoras, surgieron falsas expectativas acerca de lo que realmente son capaces de hacer ellas por sí mismas suplantando al hombre, o en relación a una supuesta mejor calidad de vida humana merced a estas máquinas.

En el terreno de los procesadores de lenguajes naturales (PLN) se busca que la comunicación con una computadora esté al alcance de todos, sin requerir especialistas para manejarlas, o cursos de apoyo. Se ha traspasado a la máquina, cada vez en mayor grado, la tarea de facilitar esa comunicación, por medio de programas complejos. Un PLN puede formar parte auxiliar de un SE de consulta de información. Su función es analizar la estructura gramatical y el significado de las preguntas que un usuario plantea, para luego generarlas en el lenguaje artificial propio del sistema en cuestión.

Asimismo convertirá las respuestas en este lenguaje al lenguaje natural correspondiente. Debido a la ambigüedad de los lenguajes humanos, a sus múltiples estructuras y a las dificultades para formalizar toda su riqueza expresiva en forma lógica, los PLN no han podido aún desarrollarse en la medida de las necesidades.

Por otra parte, los PLN requieren grandes computadoras para ser realizados. Algunas de las cosas que pueden hacer los PLN son la corrección de errores lexicográficos cometidos por el usuario y la sustitución de una palabra formulada antes, por otras estructuras gramaticales. Si se preguntó:

¿Qué acciones están por debajo de su valor nominal?

puede luego preguntarse:

¿Cuáles de ellas valen menos de $50

El proyecto "Quinta Generación" incluía en el área PLN:

- Un traductor automático multilingüe, capaz de manejar 100.000 palabras.
- Un sistema capaz de hablar y entender un lenguaje natural con unas 10.000 palabras de vocabulario.
- Un módulo auxiliar de comunicación., para lenguaje natural, de 5000 palabras, en un sistema de consulta sobre varios temas con 10.000 reglas de inferencia.

BALANCE DEL PROYECTO QUINTA GENERACIÓN

Del artículo de la revista BYTE Argentina, de J.C. Sabbione, de julio de 1993: "El Proyecto de Quinta Generación, ¿éxito o fracaso?" resultan las conclusiones que siguen.

Hacia 1985 se desarrolló un modelo de lenguaje lógico para inferencia en paralelo, denominado GHC. A partir de él se elaboró –como base para la segunda mitad del proyecto– el lenguaje **KL1**, de suficiente capacidad expresiva para describir el procesamiento paralelo de la información. Si bien KL1 es un lenguaje de muy alto nivel, es también un lenguaje de máquina a ser ejecutado por los prototipos de procesadores desarrollados durante el proyecto.

Al comenzar el proyecto se construyó la máquina **PSI** (Personal Sequential Inference Machine). El modelo PSI-III –al final del proyecto– fue capaz de realizar cerca de 1,5 millones de inferencias lógicas por segundo (LIPS).

En 1988 se construyó un sistema que utilizaba múltiples PSI en paralelo, base para desarrollos en software.

Hacia el final del proyecto se construyeron varios prototipos de **PIM** (Parallel Inference Machine), o máquinas KL1.

En 1992 se desarrolló para el PIM un modelo judicial, que generó correctamente un juicio sobre un caso policial ingresado, luego de haber analizado más de doscientas leyes, y de haber comparado el caso con 115 casos reales juzgados.

Los investigadores participantes sostienen que se alcanzaron buena parte de los objetivos, y que la tecnología por ellos creada tiene una inercia natural de entre diez a veinte años antes de ser aplicada, como sucedió con UNIX, C y RISC.

También afirman que se crearon falsas expectativas, como que en diez años se resolverían los problemas más difíciles que plantea la IA, o que se iba a crear un traductor con las mismas capacidades de un ser humano. Asimismo no se desarrollaron sistemas para el reconocimiento de patrones, o para traducir.

Como resultado de este proyecto, Japón se plantea el "*Proyecto de Computación del Mundo Real*" en base a redes neuronales, también de diez años de duración.

E. Shapiro y D. Warren en comentarios aparecidos en la revista Communications of The ACM (marzo 1993), plantean que la computación paralela requiere un mercado que necesite grandes cálculos numéricos regulares, que hoy son suficiente-mente simples como para ser planteados mediante lenguajes convencionales. Todavía no existe una demanda de computadoras paralelas para realizar una clase más amplia de problemas, a nivel de requerimientos industriales.

En relación con la IA, manifiestan que el "boom" que provocó en los años 80 ha declinado, reduciéndose el número de investigadores en este área, y que los que siguen en ella no requieren un poder computacional mayor. Tampoco hoy lo necesitan las aplicaciones basadas en SE.

Sostienen que los investigadores de la "Quinta Generación" no han logrado determinar demasiadas aplicaciones que pongan de relieve el poder computacional de las máquinas construidas.

Quedaron en el camino falsas expectativas acerca de concebir un computador inteligente, capaz de razonar como el hombre; o que podría recibir instrucciones en nuestra lengua.

ACERCA DE LA TECNOLOGIA

El modelo instrumental concibe la tecnología a partir del aparato técnico tomado aisladamente. De esta manera, los criterios de análisis que sobresalen son los de *medios, usos, fines.* Un lavarropas es un medio para un fin. Su comportamiento se iguala al de cualquier instrumento. El modelo instrumental generaliza el ejemplo del lavarropas a la totalidad del sistema de la tecnología: aunque más complejo, el sistema tecnológico seguiría siendo un medio para un fin.

Por supuesto, este modelo considera obvio que los fines a los que sirve la tecnología son determinados por la libre decisión humana: el hombre puede emplear la técnica para fines bélicos o pacíficos. En cualquier caso, se supone que el hombre controla los medios técnicos en función de sus objetivos. Este modelo, por lo tanto, considera la tecnología como un medio *neutral.*

La general difusión de este modelo obedece al hecho de que responde a la experiencia inmediata que el hombre corriente posee en su relación con los artefactos que lo rodean: él decide si va a encender el televisor o no, si usará una agenda electrónica o no.

Sin embargo, es erróneo transferir esta experiencia individual y limitada del especio personal a la totalidad del sistema tecnológico. Ocurre que la tecnología, en tanto *sistema*, presenta características e impone conductas que superan ampliamente el modo de comportamiento del hombre con el artefacto aislado.

En primer lugar, el sistema tecnológico no es un simple conjunto más complejo de instrumentos: presenta el aspecto de una *red,* en la cual cada nudo supone y promueve a los otros.

Un avión no es un simple aparato que nos transporte de un lugar a otro: implica un sistema de aeropuertos, equipamiento industrial, talleres mecánicos, oficina de controles, además de un *staff* de personal especializado y entrenado, lo que supone a su vez una instancia educativa y una estructura tecnocientífica de investigación, planificación y diseño. A su vez, en la trama de esta red, los mismos aparatos generan las condiciones que los hacen indispensables.

De este modo, cada instrumento supone la totalidad del sistema tecnológico. Por otra parte, este sistema no es una entidad que opere en el vacío. Integra a la sociedad tecnológica en su conjunto: implica una cultura para la cual los aparatos, su posesión y su uso son considerados valiosos en función de un determinado estilo de vida, un criterio de felicidad y un concepto de progreso. Es por eso que no es posible transferir los instrumentos de la sociedad tecnológica a otras culturas sin producir al mismo tiempo modificaciones profundas en el sistema de referencias de la cultura receptora.

Toda transferencia de tecnología comporta al mismo tiempo transferencia cultural. En este sentido, cada instrumento posee el carácter de un *holograma*: esto es, un tipo de organización en el que cada elemento contiene la presencia de la estructura en su totalidad. En un holograma, "el todo está en la parte, que a su vez está en el todo", y de tal modo que una parte es apta para regenerar el todo.

El carácter de la tecnología, como vemos, supera en mucho la mera instrumentalidad. El comportamiento del hombre frente a cualquier aparato rebasa en mucho su simple uso. Basta observar la actitud del hombre común frente al automóvil: en su posesión se proyectan valores de prestigio, poder, potencia. Se le atribuyen significados que van mucho más allá de su mera utilidad.

Podría argumentarse que esta proyección de valores es ajena a la tecnología en sí, y que nada tiene que ver con la eficacia del aparato. Sin embargo, la sociedad tecnológica no podría funcionar sin esa proyección emocional-valorativa. Es lo que sabe muy bien la publicidad, al recurrir más frecuentemente a este tipo de proyección que a la utilidad específica del aparato. En efecto, el apelar a esta fetichización del aparato no es un mero recurso publicitario: la publicidad no alcanzaría efectividad si la identificación *automóvil-prestigio-poder-felicidad* no estuviera ya profundamente internalizada en los consumidores. La publicidad reproduce e incrementa las connotaciones del imaginario social de la civilización tecnológica.

[. . .]

En efecto, el sistema tecnológico no es un mero medio: impone al hombre transformaciones profundas en su conducta, criterios de valor y pautas de interacción humana. Pensemos solamente cómo la televisión ha revolucionado los estilos informativos, el criterio de esparcimiento, el arte, los hábitos de vida. El analista Langdon Winner reflexionaba sobre el hecho señalado por las encuestas relativo a que la población norteamericana pasa aproximadamente siete horas diarias (un tercio de su ciclo vital) mirando televisión. Los argumentos de quienes adhieren al carácter meramente instrumental de la tecnología hacen referencia a la voluntad del hombre, para recordar que "siempre se puede apagar el televisor". Esto es trivialmente cierto. Pero teniendo en cuenta el rol de la televisión en nuestra cultura, debemos acordar con Winner cuando observa que la televisión (no el televisor) es un fenómeno que no puede "apagarse" de ningún modo.

(Tomado de: Regnasco, M. J., *Crítica de la razón expansiva – Radiografía de la sociedad tecnológica,* Buenos Aires, Biblos, 1995)

A medida que el volumen de información aumenta a ritmo vertiginoso, se obstaculiza el proceso que convierte la información en conocimiento.

Con el poder de procesamiento de las computadoras, pareciera que podemos manejar cualquier volumen de información sin ninguna dificultad. Pero esta situación encierra un problema: la selección, clasificación y sistematización de la información exige un *marco teórico* que no está al mismo nivel que los datos.

Se trata de los presupuestos básicos, de las valoraciones subyacentes y de los marcos de contextualización que están en la base del pensamiento racional y simbólico.

[. . .]

Es necesario tener en cuenta que las ideas que gobiernan los datos no son información. Son compromisos filosóficos, convicciones profundas. Como observa Th. Roszak, el principio "Todos los hombres son iguales ante la ley", no es el resultado de una inducción empírica. Es una convicción profunda, se refiere al valor esencial de las personas, remite a reflexiones filosóficas, éticas, y no a datos, cifras o estadísticas.

(Tomado de : Regnasco, M.J., (compliladora) *Para comprender la problemática del mundo actual,* Buenos Aires, Ed. Imago Mundi, 2006)

Apéndice 4 de la Unidad 1

MODELO CIRCUITAL DIDÁCTICO DE UCP

Modelo circuital conceptual que no requiere conocimientos previos de circuitos lógicos, con camino de datos de concepción RISC

El presente modelo apunta a ser una aproximación conceptual del funcionamiento interno de una UCP sin que el lector tenga un conocimiento previo del funcionamiento de los circuitos lógicos que se describen en la obra "De la Compuerta al Computador".

Teniendo presentes los esquemas las figuras 1.31 y 1.32 con múltiples caminos entre registros se desarrollarán los mismos para comprender más en detalle como funciona un procesador en la ejecución de las instrucciones, sobre la base de un camino de datos ("data path") de 3 buses, como tienen los actuales procesadores.

A los fines didácticos, a partir de la fig. A4.6 cada nueva figura incorpora (en trazo grisado) las descripciones de pasos en la ejecución de un total de 3 instrucciones llevados a cabo en las figuras anteriores, siendo que las líneas en trazo negro ilustran acerca del paso elemental que se está llevando a cabo. De esta manera, la fig. A4.14, última de la serie, permite integrar las anteriores, y que el lector pueda tener una idea más general acerca del interior de una UCP sencilla, y cómo se repite la generación de las señales de control.

Descripción general

Los bloques que aparecen bien delimitados en las figuras A4.6 hasta A4.14, que intervienen en la descripción, se comunican entre sí mediante caminos internos, es este caso de 8 ó 4 líneas conductoras, por las que pueden transitar simultáneamente hasta 8 ó 4 bits (como en una autopista de 8 carriles por la que circularían 8 ó 4 autos a igual velocidad como se detalla en la fig. 1.52). Este conjunto de líneas o "bus" se representa mediante una única línea gruesa, con la indicación del número 8 ó el 4 según sea.

Las líneas de un solo conductor forman parte de la **UC** (no delimitada como bloque para no complicar los dibujos). La **UC** por un lado *entran pulsos eléctricos* de igual duración de un **Secuenciador** (realizado en forma didáctica mediante una llave rotatoria que permanece igual tiempo en cada posición); y por otro lado la **UC** *recibe la orden que porta cada instrucción* mediante cada una de las líneas que salen del **Decodificador.**

La **UC** realiza sus funciones de control mediante líneas de un solo conductor denominadas **líneas de control**, la mayoría de las cuales *determina si por el camino (bus) que cada una de ellas controla va a pasar o no información* (Esta alternativa se representa por un contacto o llave designado **L**, que puede estar cerrado o abierto, respectivamente, el cual es comandado por una de dichas líneas de control a él asignada en forma fija. Puesto que cada camino consta de 4 u 8 líneas, cada contacto dibujado representa 4 u 8 contactos que se deben cerrar o abrir al unísono. Otras líneas de control de la **UC** gobiernan la operación que debe hacer la **UAL**, y la línea L/ (que sale al exterior de la UCP) ordena si la memoria será leída o escrita. Otra línea E/ ordena escritura

Las líneas de control cierran transitoriamente o mantienen abiertos los contactos **L** según el estado de otras líneas de un solo conductor del interior de la **UC** que las gobiernan. Estas se designan con indicación del tiempo durante el cual están activadas y del código de la instrucción del cual depende su aparición, como ser "3v.(volts) durante T_1" ó "3v para 0001 y T_3", etc.

Dado que una línea de control puede cerrar el conjunto (4 u 8) de contactos **L** que comanda durante la ejecución de diferentes instrucciones o en distintos pasos de las mismas (en tal instrucción o en tal otra, o en tal otra o en tal paso de una instrucción o en tal otro de la misma), estas distintas alternativas que se pueden enunciar verbalmente, se representan mediante una compuerta OR, que se simboliza como una "media luna". Por lo tanto las líneas de control que salen de la **UC** *en general son salidas de compuertas OR* (fig. A4.14)

Los bloques delimitados en las figuras A4.6 hasta A4.14 son:

1. La **UAL** que recibe los dos datos a operar mediante los buses **X** e **Y** de 8 líneas que entran a ella, siendo que el resultado que genera en sus salidas van al bus de 8 líneas designado **W**.

2. Los **registros** (**R**, **A**, **RDI**, **IP**), que guardan información. Ellos pueden tomar o no información del bus **W** según el estado cerrados/abiertos de los contactos **L** que los vinculan con el bus **W**, para luego retenerla. Asimismo **R** y **A**, pueden enviar o no una copia de su contenido al bus **X** conforme el estado de los **L** que los ligan con el bus **X**. Al registro **IP** sus **L** lo comunican con el bus **Y**. El registro **RDI** (Registro de Direcciones) tiene sus 4 salidas conectadas directamente al bus de direcciones, por donde se envía cada dirección a la memoria. Sólo **RDI** e **IP** tienen 4 salidas; los restantes tienen 8.
3. La **memoria**, que en este esquema consta de 16 celdas para 8 bits cada una, numeradas con direcciones de 0000 a 1111. Ya sea para leerla o escribirla, al registro **RDI** le debe llegar la dirección de la celda a la que se quiere acceder, la cual a través del bus de direcciones que comunica **RDI** con memoria llega al decodificador de ésta que permite localizar la celda direccionada de la forma vista en la figura 1xxx. Cuando la memoria es direccionada con orden de lectura (mediante la línea L/E) puede pasar una copia de los 8 bits contenidos en una celda de la misma al registro **RDA** (Registro de Datos) vinculado a ella. Desde éste una copia de dicha información puede pasar al bus **X** (si los contactos **LX** está cerrado). Si se direcciona la memoria para escribir una celda, a **RDA** deben llegar los 8 bits a escribir desde el registro **A** por el bus **Y**, para lo cual los contactos **LY** deben estar cerrados. Luego la línea L/E (lectura./escritura) debe dar la orden de escritura, para que una copia de **RDA** pase a la celda direccionada..
4. El registro de instrucción, **RI**, que puede tomar directamente su contenido (una instrucción del programa) de **RDA** la cual a su vez provino de memoria, cuando los contactos L_{RI} están cerrados.
5. El **decodificador** cuyo accionar se verá luego, siendo que cada una de sus líneas de salida (una distinta por cada instrucción) determina los pasos para ejecutar la instrucción que activó la línea que le corresponde.
6. Un **secuenciador** o generador de secuencias de pulsos de igual duración temporal.
   Como se ha planteado (Sección 1.7), la ejecución de una instrucción se lleva a cabo mediante una secuencia de pasos que progresan con cada nuevo pulso originado por dicho generador.
   A los fines didácticos supondremos (fig. A4.1) una llave rotativa selectora de varias posiciones, capaz de comunicar los 3 volts (3v) de una batería conectada a su contacto central a la línea conductora que sale del punto donde la llave hace contacto. Si la llave permanece por ejemplo T = 1 segundo en cada posición, mientras queda ese tiempo en la posición o contacto 1, la línea unida al mismo (designada "3v durante T1") estará en toda su longitud con 3v. (fig. A4.2). También se dice que sobre esa línea tiene lugar un pulso de tensión de 3v designado T1, de un segundo de duración.
   Si luego que termina el primer segundo, instantáneamente la llave pasa al contacto 2 permaneciendo otro segundo (designado T2 en relación con el contacto 2) en esta posición, los 3v de la batería se comunicarán a la línea "3v durante T2", naciendo un pulso T2 al mismo tiempo que "muere" T1. Del mismo modo, cuando la llave pasa al contacto 3, nace T3 y muere T2 como se aprecia en la fig. A4.2
   Estos pulsos serán usados por la **UC** *para generar y determinar la duración de los pulsos de control que proveen las líneas de control, mediante los cuales cierra llaves ó indica órdenes para la UAL.*

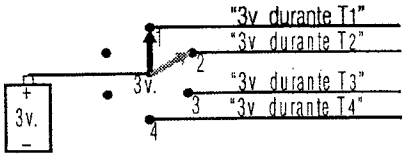


Figura A4.1

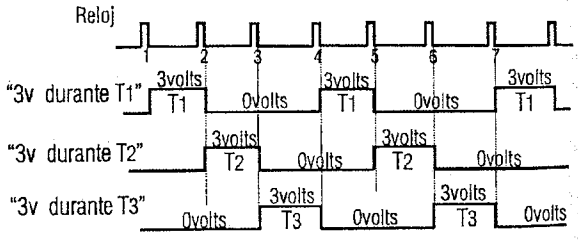


Figura A4.2

Puede considerarse a su vez que otros "pulsos reloj" más angostos que caen a 0 volts cada segundo, en los instantes de tiempo 1, 2, 3, 4, 5 ... generados por un cristal piezo-eléctrico, son los que hacen rotar la llave. *Esta vuelve siempre al contacto* 1, para generar el pulso T1, pudiendo hacerlo desde el contacto 3, 4 ó 5, según se necesite.

Para un computador supondremos -según el ejemplo de la figura 1.29- que entre la aparición de un pulso reloj y el siguiente transcurre un tiempo de tan solo 20 nanosegundos (1 nseg. es una milésima de millonésima de segundo), que sería también la duración de los pulsos T1, T2 ,T3 .... generados hipotéticamente por la llave rotatoria temporizadora descripta. Esta frecuencia de aparición equivale a 50 millones de pulsos reloj generados por segundo, o sea 50 megahertz (MHZ).

La **UAL** en relación con sus entradas **X** e **Y**, y salida **W**, en general puede hacer sumas o restas: $X \pm Y = W$. Los valores X e Y son la información que viaja por los buses **X** e **Y**, pudiendo ser provista por dos registros, como ser **A** y **RDA**. El resultado (que siempre pasa al bus **W**) puede ser asignado a un registro, como el **R**.

La operación que efectuará la **UAL** depende en este esquema de cuál de sus líneas de operación (X+Y, 0+Y, etc.) esté activada por la línea de control de la **UC** que la activa, mediante un pulso de duración T1, T2 ,T3 ....
Si la **UC** ordena hacer W = X + 0, y también ordena cerrar los contactos $L_{RDA,X}$ y $L_{Z,A}$ (que respectivamente permiten pasar información de **RDA** al bus **X**, y del bus **W** al registro **A**) resultará que una copia del registro **RDA** pasará, sin modificación alguna, al registro **A**. *Así pasa una copia del contenido de un registro a otro.*
En caso de que la **UC** ordenara hacer W = 0 + Y y cerrar los contactos que correspondan, mediante un pulso de duración T1, T2 ,T3 ...., es factible transferir una copia del **IP** al **RDI**, o del **RI** al **RDI**, según se necesite.
De esta forma, los múltiples caminos entre registros de las figuras 1.31 y 1.32 han sido reemplazados por los tres buses internos **X**, **Y**, **W**, que a través de la **UAL** permiten por ejemplo transferencias entre registros, o sumar los contenidos de dos registros y el resultado guardarlo en un tercero.

Las líneas "3v durante T1", "3v durante T2" .... que salen de la llave rotativa (fig. A4.1), *cuando están en 3v. se emplean mayormente para indicar intervalos de tiempo* (indicados T1, T2 ...) *en que deben permanecer cerradas* las **L** que ellas comandan, a fin de habilitar los caminos (buses) que dichas llaves gobiernan.
Cuando dichas líneas dejan de estar en 3v las llaves que dichas líneas comandan quedan abiertas, deshabilitando dichos caminos para que no puedan realizarse movimientos por ellos.
En la fig. A4.3a (que es parte de la fig. A4.6) se quiere simbolizar, que la línea "3v durante T1" comanda en ese tiempo T1 en que está en 3v. el cierre de las llaves $L_{IP,Y}$ (la flecha al final de la línea que llega hasta la llave quiere indicar una fuerza que produce el cierre), para que una copia del contenido del registro **IP** pase al bus **Y**. Al mismo tiempo dicha línea cierra las llaves L_{RDI} para que el registro **RDI** pueda tomar la información 0000.

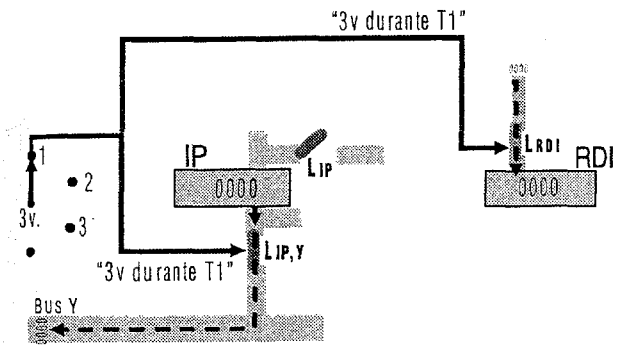


Figura A4.3a

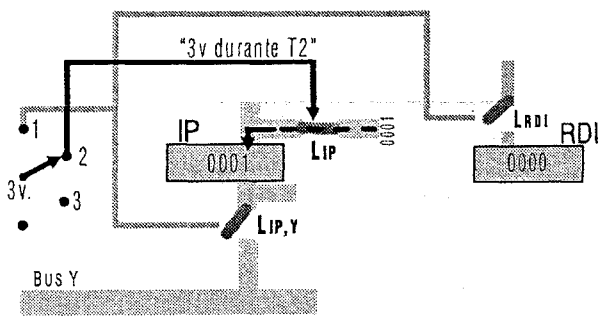


Figura A4.3b

La fig. A4.3b (parte de la A4.7) simboliza que al pasar la llave rotativa a la posición 2 deja de estar en 3v la línea "3v durante T1" por lo que aparece en grisado, con lo cual se abren las llaves $L_{IP,Y}$ (que comunicaban **IP** con el bus **Y**) y las L_{RDI}, a la par que la línea "3v durante T2" cierra durante T2 las **LIP**, para que 0001 pase al **IP**.

Compuertas AND y OR

Es factible condicionar el cierre de llaves **L** a que dos (o más) líneas estén simultáneamente en 3v un tiempo T. Para ello dichas líneas deben ser entradas de una compuerta AND, cuya línea de salida comanda el cierre de llaves cuando permanece en 3v, y las abre mientras no se de dicha simultaneidad. Por ejemplo (fig. A4.4 que es parte de la fig A4.8), la línea de salida AND "3v para 0001 y T3" comanda el cierre/apertura de las llaves $L_{IR,X}$. Esta línea durante el tiempo T3 permanecerá en 3v (cerrando las $L_{IR,X}$) si están en 3v la línea "3v durante T3" y también la línea "3v para 0001" (esta última está en 3v mientras se ejecuta la instrucción de código 0001, durante T3 y T4, como se verá). Puesto que sólo durante el tiempo T3 ambas líneas están en 3v, la salida de la AND "3v para 0001 y T3" *sólo estará en 3v durante T3*, y *sólo para la instruc-ción de código 0001.*

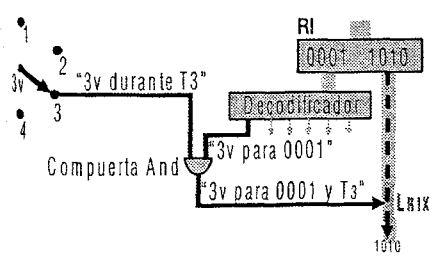


Figura A4.4

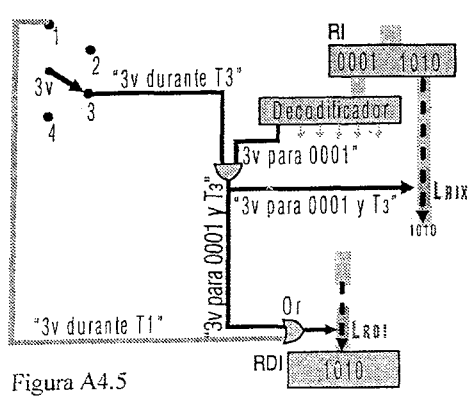


Figura A4.5

De manera general, una **compuerta** circuital genera el estado eléctrico (0v ó 3v) de su línea de salida en función del estado eléctrico (0v ó 3v) de dos o más líneas que son sus entradas.
En definitiva el cable de salida "3v para 0001 y T3" acompañará el estado de la línea "3v durante T3", o sea que sobre esa línea tendrá lugar un pulso de 3v que durará un tiempo T3. Es como si el pulso hubiese pasado de la líneas de entrada "3v durante T3" a la de salida. De esta forma, de todo el tiempo en que "3v para 0001" está en 3v, con la línea "3v durante T3" se determina el tiempo T3 a fin de que las $L_{IR,X}$ estén cerradas sólo durante T3.

Si para dos instrucciones distintas o pasos diferentes de una misma instrucción (durante la concreción de uno o del otro) se necesita cerrar las mismas llaves, se debe recurrir a una compuerta **OR** simbolizada como una media luna. Así (fig. A4.5 que es parte de A4.8), las llaves L_{RDI} se deben cerrar durante T1 por que está en 3v la línea "3v para T1 (fig, A4.3.a) **o** bien en T3 por estar en 3v la línea "3v para 0001 y T3" (fig. A4.4).
A su vez como se vio, la salida AND "3v para 0001 y T3" para estar en 3v requiere la conjunción de dos líneas en 3v. En la figura A4.13 la OR se amplió a 4 entradas, pues como se verá, las L_{RDI} se deben cerrar durante T3 en cada una de las 3 instrucciones distintas ejecutadas, y además siempre durante T1 en el primer paso de cualquier instrucción, cuando se la solicita a memoria (en una **o** en otra **o** en otra **o** ....).
En una OR basta que una entrada esté en 3v. un tiempo T para que su línea de salida también lo esté ese mismo tiempo. Mientras ninguna de sus entradas esté en 3v. su salida tampoco lo estará.

Ejecución de instrucciones

Efectuaremos en el modelo de procesador la operación R = P + Q, teniendo presente sus analogías con una calculadora de bolsillo (fig. 1.2), y siendo: P = 5 = 00000101 Q = 4 = 00000100.
En la fig. A4.6 y siguientes, las instrucciones tienen 8 bits: los 4 primeros para indicar la operación ordenada (**código de operación**), y los 4 últimos para indicar en que dirección leer o escribir un dato en memoria.
Las 3 instrucciones que usaremos se ha supuesto que ocupan las 3 direcciones sucesivas 0000, 0001 y 0010 de la memoria (fig. A4.6). Estas, al igual que los datos P y Q, en la práctica llegan a memoria provenientes de un disco, o desde el teclado.
La instrucción **I1** (código de operación 0001) ordena llevar al registro **A** una copia del contenido de la celda de memoria de dirección 1010, donde se encuentra el valor de P (código completo: 0001 1010). Esto equivale a llevar el valor de P al visor de una calculadora de bolsillo, como primer paso para hacer P + Q.
La instrucción **I2** (código de operación 0011) ordena sumar al contenido del registro **A** una copia del contenido de la celda de dirección 1100 (donde se ha supuesto que está el valor de Q), y el resultado de la suma (en este caso P + Q) asignarlo al registro **A**, perdiendo éste su valor anterior. En cierto modo es como en una calculadora entrar el valor de Q y apretar la tecla + , de manera que cuando se pulsa la tecla = el resultado de la suma aparecerá en el visor en lugar del número anterior. Código completo: 0011 1100
La instrucción **I3** (código de operación 0111) indica llevar a la posición de memoria 1110 una copia del registro A. Se puede equiparar en cierta medida a la acción de pulsar en una calculadora la tecla $M.^{+}$ para que una copia de lo que está en el visor se guarde en la memoria de la calculadora. Código completo: 0111 1110.
Las acciones que ordenan I1, I2 e I3 están escritas en memoria, constituyendo un corto programa, para que se lleven a cabo cuando la UCP las ejecute, como se describe a continuación.

¿ Cómo se ejecuta un programa constituido por I_1 , I_2 e I_3 en el modelo de la figura A4.6 y siguientes ?

Antes de ejecutar un programa, es necesario que en registro **IP** esté la dirección de su primer instrucción (que en este caso es 0000), la cual en la práctica es determinada por el sistema operativo..
Como se detalla en la figura 1.27, la ejecución de una instrucción comprende en general los siguientes pasos, que progresan en sincronismo con los pulsos reloj:

1. Obtener la instrucción mediante una lectura de la memoria, y llevar una copia de ella al registro **RI**. Mientras esto sucede la **UC** puede cambiar el valor del **IP** para que contenga la dirección de la siguiente instrucción a ejecutar, que sigue a continuación en memoria.
2. Con el código de la instrucción en **RI** la **UC** lo decodificará, con lo cual quedará determinado qué hardware intervendrá en cada uno los pasos siguientes que correspondan a la ejecución propiamente dicha.
3. Obtener el dato a operar si el mismo está en memoria, para lo cual primero se debe determinar su dirección y luego pasarlo al registro **RDA** mediante una lectura de memoria. Si se debe escribir (un resultado) en la memoria, se debe pasar en este paso su dirección al registro **RDI**.
4. Realizar la operación que ordena la instrucción.

¿ Cómo son los 4 pasos anteriores en la ejecución de I_1 ?

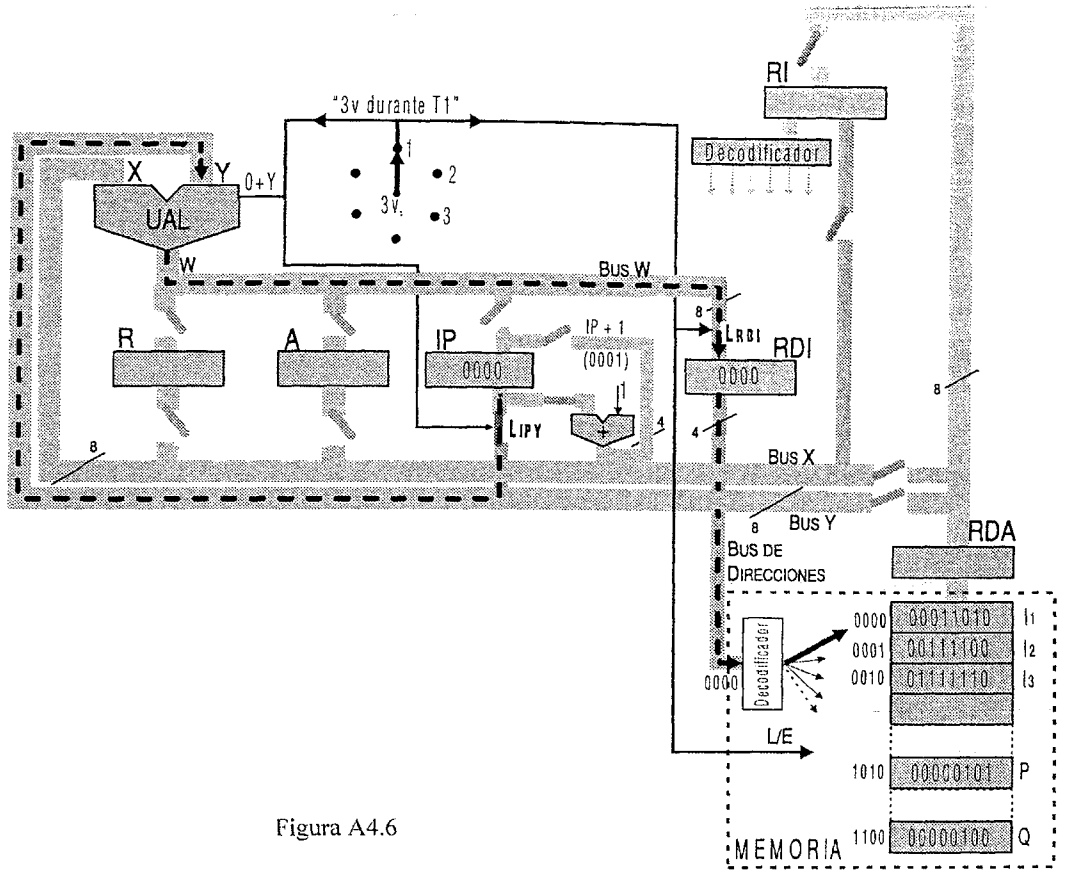


Figura A4.6

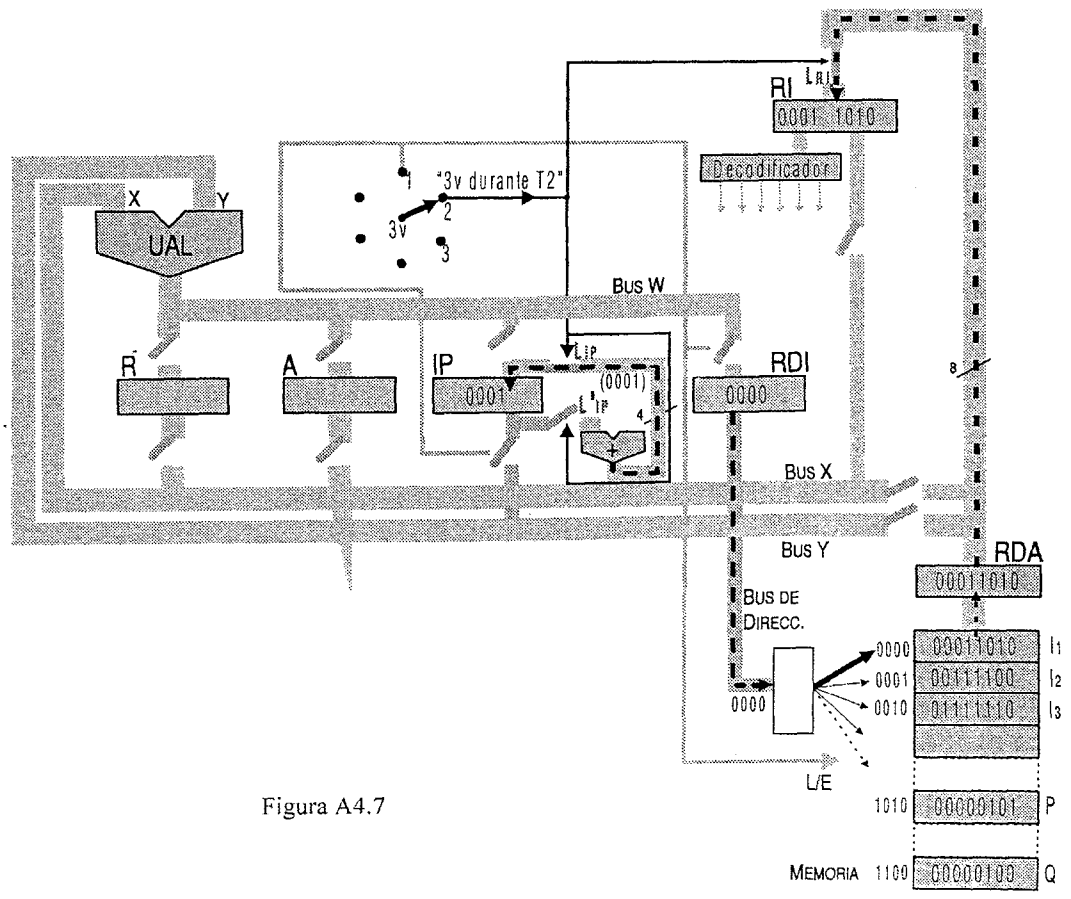


Figura A4.7

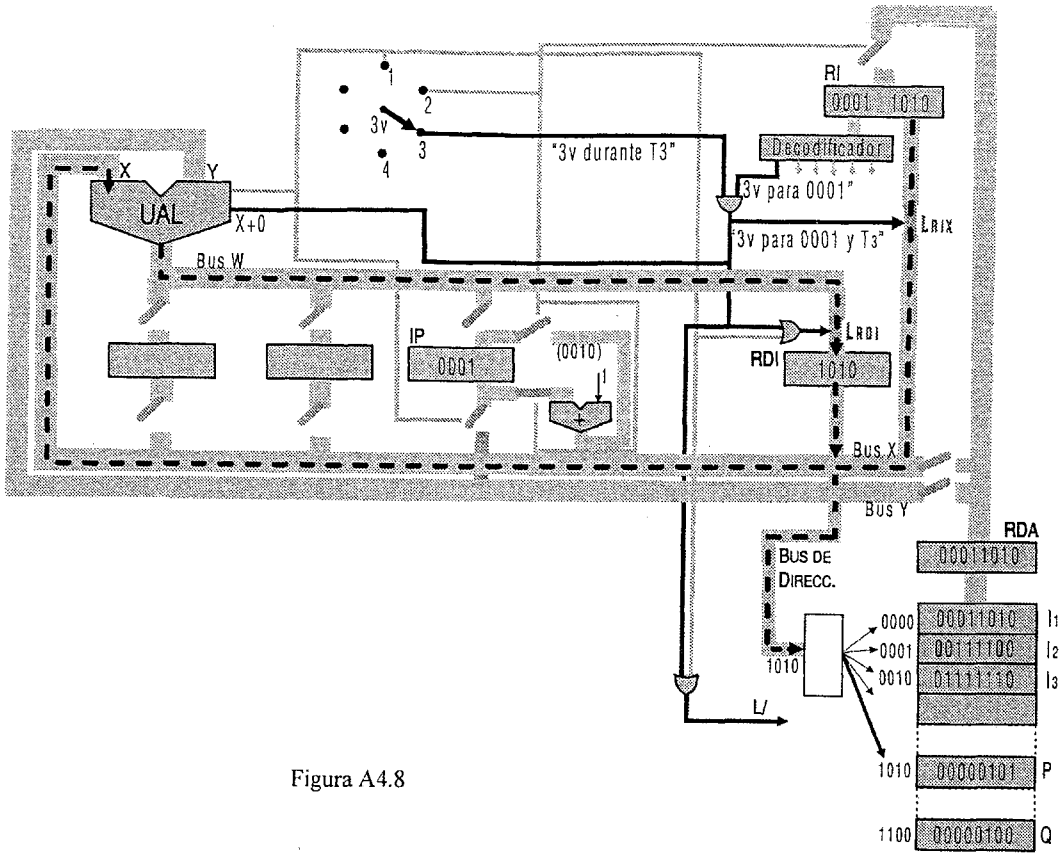


Figura A4.8

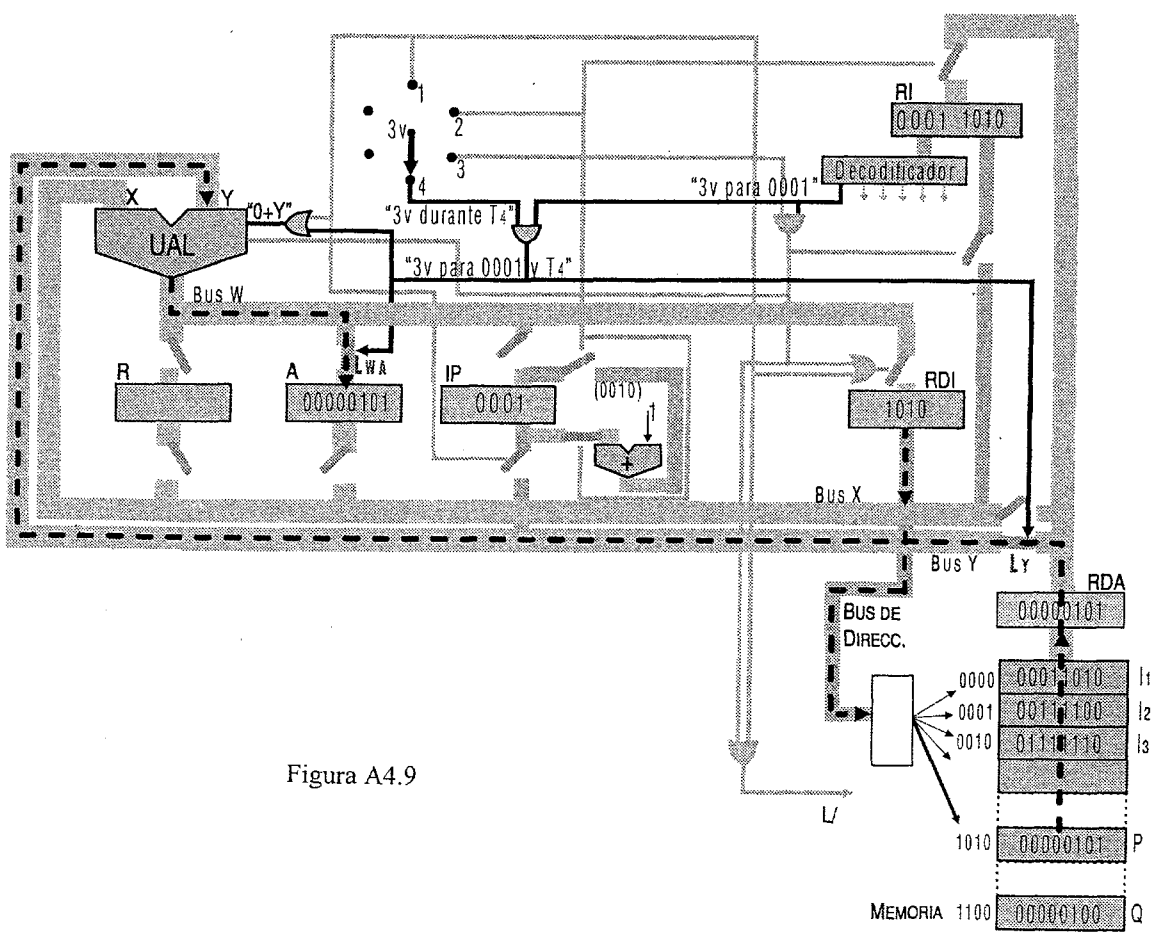


Figura A4.9

Paso 1 (Obtención del código de la instrucción I1 en **RI**)
En este paso la llave rotativa temporizadora debe estar primero en la posición 1 durante T1 = 20 nseg., tiempo durante el cual la línea "3v durante T1" está en 3v; y luego en la posición 2 durante T2 = 20 nseg. siguientes.
Puesto que I1 está en memoria, hay que localizarla en ésta mediante su dirección siempre contenida en el registro **IP** (fig. A4.6). Puesto que el único registro que está conectado al bus de direcciones que va a memoria es **RDI**, en él se debe constituir cualquier dirección (de instrucción o de dato) a la que se quiere acceder en memoria. Por lo tanto, se debe pasar una copia de **IP** al **RDI**. Para ello la línea de control "3v durante T1" [3] cerrará las llaves $L_{IP,Y}$, lo cual permite que una copia del contenido del **IP** pase al bus **Y**. Asimismo, dicha línea también actúa sobre la entrada "0 + Y" de la UAL para ordenar esa suma, de modo que sea W = Y. Así la copia del contenido del **IP** que viajaba por el bus **Y** pasa sin alteración (pues se le sumó 0) al bus **W**.
Dado que por otra parte "3v durante T1" actúa cerrando las llaves L_{RDI}, el contenido de **IP** (0000, dirección de I1) que viaja por el bus **W** pasará al **RDI**, con lo cual también viajará por el bus de direcciones hacia memoria. Entonces en la memoria comenzará el tiempo de acceso[4] para la localización de la instrucción (I1 en este caso), siendo que además se necesita que le llegue una orden de lectura por la línea L/ que debe estar en 3v, también merced a la línea "3v durante T1".

Cuando termina el tiempo T1 todas las llaves cerradas durante el mismo se abren, y la llave rotativa pasa a la posición 2 (figura A4.7), con lo cual queda en 3v durante T2 = 20 nseg. la línea de control "3v durante T2". Esta cierra las llaves $L_{M,RI}$, de modo que cuando llegue a **RDA** el código de máquina de la instrucción pedida (en este caso 00011010, copia del contenido de la dirección 0000), dicho código pase sólo a **RI**.
Paralelamente con esto "3v durante T2" cierra las llaves L_{IP}, y abre las L'_{IP} (que volverán a su posición anterior cuando termine T2). Entonces la salida del sumador que suma 1 al contenido del **IP** (que mientras L'_{IP} estaba cerrada era 0001 = 0000 + 1) comunicará su valor al **IP**. Este quedará así con 0001, preparado con la dirección de I2, para cuando se pida esta instrucción.
En definitiva, luego de las acciones determinadas por las líneas de control "3v durante T1" y "3v durante T2" se habrá obtenido en **RI** el código 00011010 de I1 (instrucción a ejecutar), y en el **IP** la dirección de I2.
En la descripción anterior debe observarse que en cada transferencia de información que ha tenido lugar *sólo se han cerrado las llaves que permiten dicho movimiento*, siendo que las restantes quedan abiertas. Esto se logra en los tiempos apropiados (T1, T2, T3, ...) merced a las líneas de control citadas que comandan dichas llaves.
Paso 2 (Decodificación de la orden que determina los pasos a seguir)
Al mismo tiempo que en T2 llega la instrucción 00011010 al **RI**, sus primeros 4 bits (0001), que constituyen el "**código de operación**" (cod-op) que ordena la operación a realizar, entran (fig A4.7) al circuito **decodificador**
Este tiene tantas salidas como instrucciones diferentes existan, pero sólo una estará en 3v según sea el cod-op presente en el **RI**. Esa línea determinará los pasos a seguir, según se verá a continuación.
Así (fig. A4.8), toda vez que el cod-op sea 0001 (0 y 0 y 0 y 1, detectable mediante una And no dibujada) **sólo** la línea "3v para 0001" queda en 3v, mientras que las restantes salidas del decodificador quedan en 0 volts. Esta situación subsistirá hasta que se termina de ejecutar la instrucción (tiempo T4 para esta instrucción I1).

Paso 3 (Obtención del dato)
Hasta acá, luego de los tiempos T1 y T2 se obtuvo en **RI** una copia del código 00011010 de I1 que estaba en memoria. Como se verá a continuación, en el tiempo T3 se direccionará el dato (P = 5 = 00000101) que está en memoria (en la dirección 1010); y en el tiempo T4 este dato pasará de **RDA** al registro **A** (paso 4).
El paso 3 empieza cuando la llave rotativa está en la posición 3 en la que permanecerá T3 = 20 nseg, con lo cual quedará en 3v durante ese tiempo la línea "3v durante T3". En el paso 2 la línea "3v para 0001" del decodificador ya está en 3v, representando la orden "enviar al registro **A** una copia del dato que está en la dirección de memoria (en este caso 1010) que acompaña a la orden 0001". Esta línea determina que deberá ocurrir en el circuito en los tiempos T3 y T4, para llevar a cabo la orden que ella representa, según se apreciará.
Durante el tiempo o pulso T3 de ejecución de I1, las señales "3v para 0001" y "3v durante T3" están simultáneamente en 3v. Entonces (fig. A4.8), la línea de salida designada "3v para 0001 y T3" de la compuerta AND de las que ellas son entradas estará en 3v durante T3 según se planteó en la fig. A4.4. Entonces (fig. A4.8), la salida "3v para 0001 y T3" deberá ordenar cerrar durante T3 las $L_{RI,X}$ y L_{RDI} [5], y ordenará que la UAL realice W = X + 0, permitiendo así que la dirección del dato (1010) que está en la mitad de **RI** pase primero al bus **X**, y de éste al bus **W**, desde donde pasa al **RDI**. De este registro llegará directamente a memoria a través del bus de direcciones (de forma semejante, en el paso 1 la dirección de I1 llegó a memoria, pero provista por el **IP**). Entonces en la memoria comenzará el tiempo de acceso para la localización del dato (00000101=5 = P) que está en la

[3] En los gráficos que siguen las líneas de control que están en 3v se dibujarán en trazo negro.
[4] Tiempo que transcurre desde que la dirección se formó en **RDI** hasta que una copia de la información correspondiente a esa dirección llegue a **RDA**
[5] Dado que L_{RDI} debe cerrarse durante T_1 cuando "3v durante T_1" está en 3v o también cuando "3v para 0001 y T_3" está en 3v (como se da ahora), estas líneas deben ser entradas de una **OR** (que no aparece por razones didácticas en la fig A4.6) cuya salida comanda a L_{RDI}

~~dirección (1010)~~ dada por **RDI**. La línea L/ debe estar en 3v siendo la salida de una compuerta OR (agregada en relación con la fig A4.6, pues se necesita ordenar leer en el paso 1 ó en el 3). De este modo L./ estará en 3v, si en la Or está en 3v su línea de entrada "3v durante T1" (como sucedió en el paso 1) o también si está en 3v su línea de entrada "3v para 0001 y T3" conforme ocurre ahora.

Paso 4 (Realización de la operación ordenada)
Cuando la llave rotativa pase a la posición 4 (fig, A4.9) la línea "3v durante T4" queda en 3v, y tiene lugar el último paso de la ejecución de I1, durante el cual una copia del dato que en T4 llega a **RDA** pase al registro **A**. Para ello la línea de salida de AND "3v para 0001 y T4" cerrará las **LY**, así la salida de **RDA** podrá comunicar su contenido al bus **Y**. De este bus debe pasar al bus **W** para llegar al registro **A**, lo cual también requiere que la salida "3v para 0001 y T4" ordene a la UAL efectuar W = 0 + Y, y que las $L_{W,A}$ se cierren.
La operación W = 0 + Y se ordena haciendo que la línea "0 + Y" que entra a la UAL esté en 3v. Como esto debe ocurrir en durante T1 de la fase de pedido descripta o también en el presente, la salida OR de entradas "3v durante T1", "3v para 0001 y T4" (no dibujada en la fig. A4.6) servirá en ambos casos para ordenar 0 + Y.
De esta forma, la señal "3v para 0001 y T4" determina que una copia de **RDA** pase al registro **A**, quedando éste con el valor del dato (00000101 = 5 = P), por lo que así finaliza la ejecución de I1

¿Cómo son los 5 pasos anteriores en la ejecución de I2 de código de operación 0011?

Paso 1 (Obtención del código de la instrucción I2 en **RI**)
Como se determinó, durante el tiempo T2 de ejecución de I1, el **IP** había cambiado

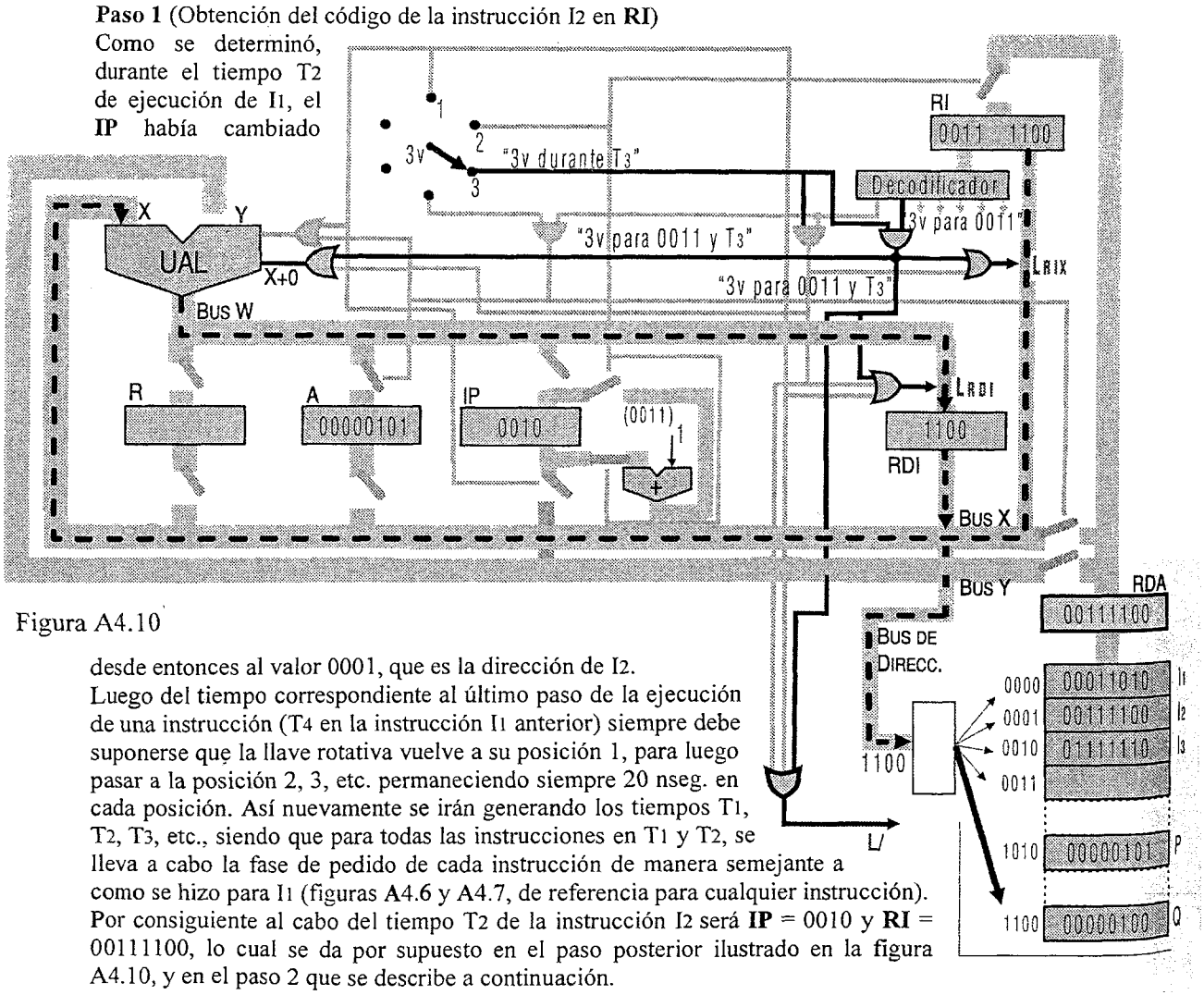


Figura A4.10

desde entonces al valor 0001, que es la dirección de I2.
Luego del tiempo correspondiente al último paso de la ejecución de una instrucción (T4 en la instrucción I1 anterior) siempre debe suponerse que la llave rotativa vuelve a su posición 1, para luego pasar a la posición 2, 3, etc. permaneciendo siempre 20 nseg. en cada posición. Así nuevamente se irán generando los tiempos T1, T2, T3, etc., siendo que para todas las instrucciones en T1 y T2, se lleva a cabo la fase de pedido de cada instrucción de manera semejante a como se hizo para I1 (figuras A4.6 y A4.7, de referencia para cualquier instrucción). Por consiguiente al cabo del tiempo T2 de la instrucción I2 será **IP** = 0010 y **RI** = 00111100, lo cual se da por supuesto en el paso posterior ilustrado en la figura A4.10, y en el paso 2 que se describe a continuación.

Paso 2 (Decodificación de la orden que determina los pasos a seguir)
A la par que durante T2 llega la instrucción 00111100 al **RI**, sus primeros 4 bits (0011), cod-op de la operación a realizar, entran al circuito decodificador. En éste, toda vez que el cod-op sea 0011 (**0 y 0 y 1 y 1**) **sólo** la línea "3v para 0011" queda en 3v mientras que las restantes salidas estarán en 0 volts. Esta situación perdura hasta que se termina de ejecutar la instrucción (tiempo T5 para esta instrucción I2).

Paso 3 (Obtención del dato en RDA)

En los tiempos T1 y T2 se obtuvo en **RI** una copia del código 00111100 de I2 que estaba en memoria. En este paso, en el tiempo T3 se obtendrá en **RDA** el dato (Q = 4 = 00000100) que está en memoria (en la dirección 1100). En el tiempos T4 este dato se sumará al contenido registro **A** (paso 4).

El paso 3 (fig. A4.10) es igual al paso 3 de I1 (fig. A4.8). Empieza con la llave rotativa en la posición 3, quedando así en 3v la línea "3v durante T3" (T3 = 20 nseg.).

Ya en el paso 2 la línea "3v para 0011" del decodificador está en 3v para determinar que deberá ocurrir en el circuito en los tiempos T3, T4 y T5, para llevar a cabo la orden que esta línea representa: "sumarle al registro **A** una copia del dato que está en la dirección de memoria (en este caso 1100) que acompaña a la orden 0011, y el resultado asignarlo al registro **A** en reemplazo del valor que tenía ".

Durante el tiempo T3 de ejecución de I2, las señales "3v para 0011" **y** "3v durante T3" están simultáneamente en 3v. Por lo tanto, como en la fig. A4.8, la línea de salida "3v para 0011 **y** T3" de la compuerta AND de las que ellas son entradas estará en 3v durante T3 . Entonces (fig. A4.10), dicha salida "3v para 0011 **y** T3" deberá ordenar cerrar durante T3 las $L_{RI,X}$ y L_{RDI}[6], y ordenará que la UAL realice W = X + 0, permitiendo así que la dirección del dato (1100) que está en la mitad de **RI** pase primero al bus **X**, de

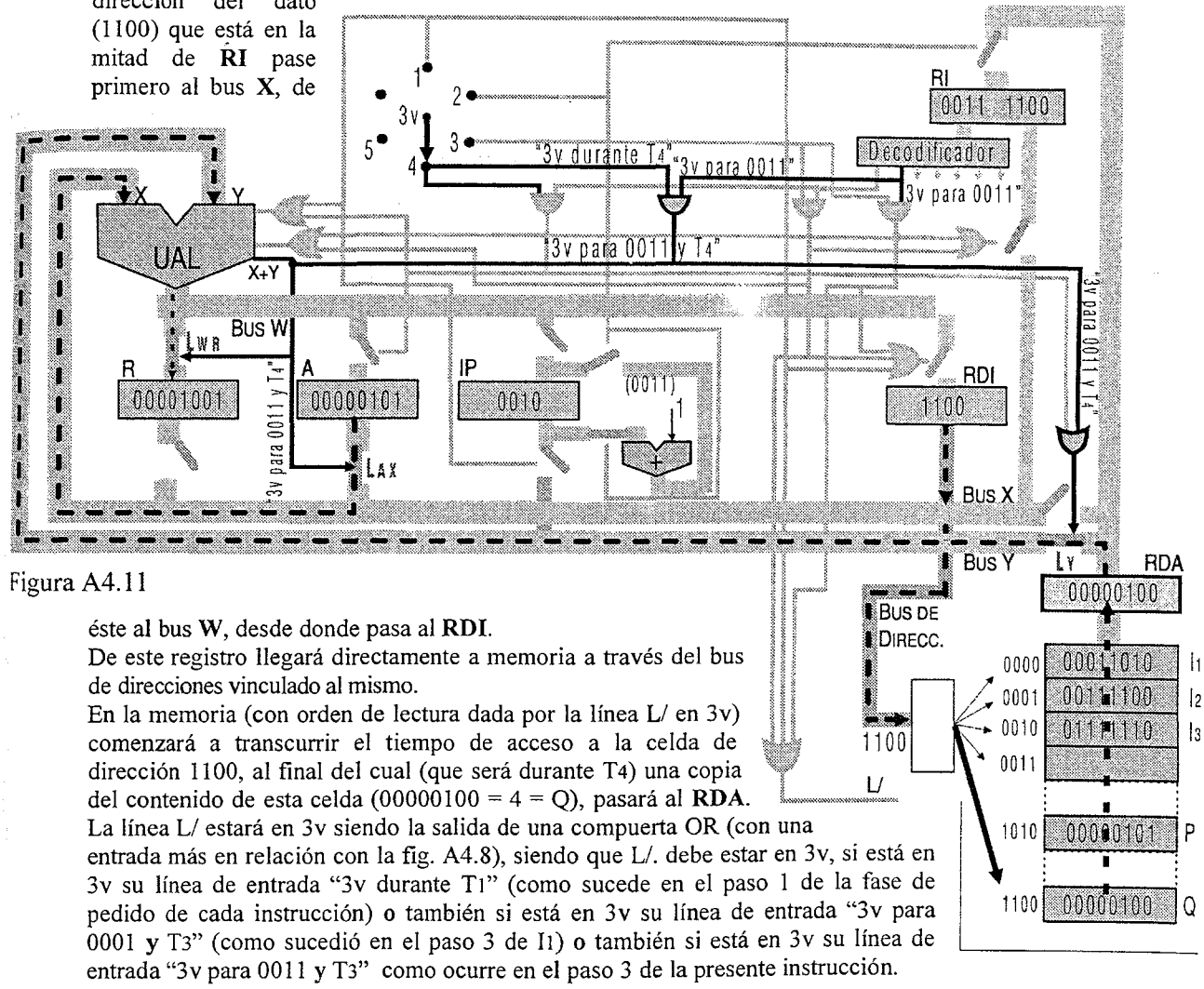


Figura A4.11

éste al bus **W**, desde donde pasa al **RDI**.

De este registro llegará directamente a memoria a través del bus de direcciones vinculado al mismo.

En la memoria (con orden de lectura dada por la línea L/ en 3v) comenzará a transcurrir el tiempo de acceso a la celda de dirección 1100, al final del cual (que será durante T4) una copia del contenido de esta celda (00000100 = 4 = Q), pasará al **RDA**.

La línea L/ estará en 3v siendo la salida de una compuerta OR (con una entrada más en relación con la fig. A4.8), siendo que L/. debe estar en 3v, si está en 3v su línea de entrada "3v durante T1" (como sucede en el paso 1 de la fase de pedido de cada instrucción) **o** también si está en 3v su línea de entrada "3v para 0001 **y** T3" (como sucedió en el paso 3 de I1) **o** también si está en 3v su línea de entrada "3v para 0011 y T3" como ocurre en el paso 3 de la presente instrucción.

Paso 4 (Realización de la operación ordenada y almacenamiento del resultado en el registro temporario **R**)

Cuando la llave rotativa pasa a la posición 4 (fig. A4.11) la línea "3v durante T4" queda en 3v. Entonces tendrá lugar otro paso de la ejecución de I2. en el cual dicha línea en conjunción con la línea "3v para 0011" pondrán en 3v la línea de salida "3v para 0011 y T4" de la AND cuyas entradas son esas dos líneas.

---

[6] Dado que L_{RDI} debe cerrarse durante T_1 cuando "3v durante T_1" está en 3v (como en el paso 1 de cualquier instrucción **o** también cuando "3v para 0001 y T_3" está en 3v (como en el paso 3 de I_1) **o** también como en este paso 3 de I_2 (con "3v para 0111 y T_3" en 3v), estas 3 líneas deben ser 3 entradas de una **OR** (que eran 2 en la fig. A4.8) cuya salida comanda a L_{RDI}

Análogamente, para $L_{RI,X}$, ella debe cerrarse durante T_3 en el paso 3 de I_1 **o** de I_2 (presente instrucción), o sea si está en 3 v la línea "3v para 0001 y T_3" o "3v para 0011 y T_3", por lo que estas 2 líneas deben ser 2 entradas de una **OR** (no dibujada en la fig A4.8) cuya salida comanda a $L_{RI,X}$

Esta salida determinará que la UAL sume el registro **RDA** (al cual durante T4 llega el dato desde 1100) con el **A**, y que el resultado se guarde en el registro temporario **R** (que no será necesario en el modelo que se expone a continuación de éste). Se requerirá un quinto paso a fin de que durante T5 una copia del registro **R** pase al **A**.

Para el paso 4 la línea de salida de AND "3v para 0011 y T4" debe ordenar por un lado cerrar las L_Y [7] así la salida de **RDA** podrá comunicar su contenido al bus **Y**, y por otro cerrar $L_{A,X}$ de forma que el registro **A** podrá pasar una copia de su contenido al bus **X**. Al mismo tiempo "3v para 0011 y T4" debe ordenar a la UAL efectuar W = X + Y (mediante "X + Y"), y que $L_{W,R}$ se cierre a fin de que el resultado de la UAL, siempre presente en el bus **W**, pueda llegar al registro **R**.

De esta manera hasta acá (T4), la señal "3v para 0011 y T4" ha determinado que una copia de **RDA** se sume en la UAL con una copia del registro **A**, y que el resultado pase al registro **R**, quedando éste con el valor de la suma (00001001 = 9 = 5 + 4 = P + Q).

Paso 5 (Almacenamiento del resultado en el registro ordenado)

Cuando la llave pasa a la posición 5 (fig. A4.12) la línea "3v durante T5"

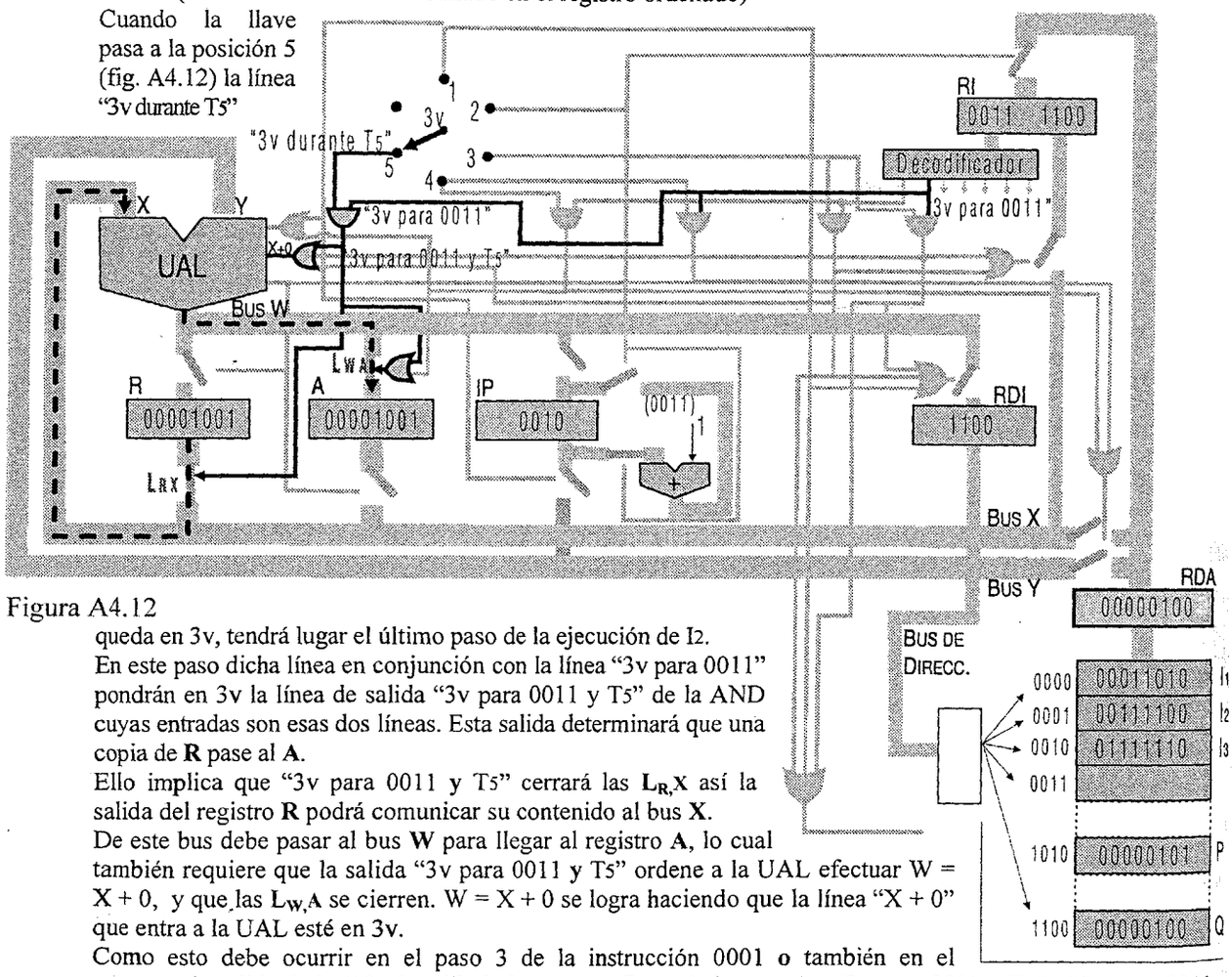


Figura A4.12

queda en 3v, tendrá lugar el último paso de la ejecución de I2.

En este paso dicha línea en conjunción con la línea "3v para 0011" pondrán en 3v la línea de salida "3v para 0011 y T5" de la AND cuyas entradas son esas dos líneas. Esta salida determinará que una copia de **R** pase al **A**.

Ello implica que "3v para 0011 y T5" cerrará las $L_{R,X}$ así la salida del registro **R** podrá comunicar su contenido al bus **X**.

De este bus debe pasar al bus **W** para llegar al registro **A**, lo cual también requiere que la salida "3v para 0011 y T5" ordene a la UAL efectuar W = X + 0, y que las $L_{W,A}$ se cierren. W = X + 0 se logra haciendo que la línea "X + 0" que entra a la UAL esté en 3v.

Como esto debe ocurrir en el paso 3 de la instrucción 0001 o también en el presente, la salida de la OR (no dibujada en otras figuras) de entradas "3v para 0001 y T3" , "3v para 0011 y T5" servirá para ambos casos.

De esta forma, la señal "3v para 0011 y T5" determina que una copia de **R** pase al registro **A**, quedando éste con el valor del resultado (00001001 = P + Q), por lo que así finaliza la ejecución de I2

¿ Cómo son los 4 pasos anteriores en la ejecución de I_3 ?

Paso 1 (Obtención del código de la instrucción I2 en RI)

Como se determinó, durante el tiempo T2 de ejecución de I2, el **IP** había cambiado desde entonces al valor 0010, que es la dirección de I3.

---

[7] Siendo que L_Y debe cerrarse en este caso, cuando la línea "3v para 0011 y T_4" está en 3v o también cuando la línea "3v para 0001 y T_4" está en 3v (como en el paso 4 de I_1) se ha agregado en la fig. A4.11 (en relación con la fig A4.9) una compuerta **OR** cuyas 2 entradas son esas 2 líneas, para contemplar que su salida comande a L_Y

Luego del tiempo correspondiente al último paso de la ejecución de una instrucción (T5 en la instrucción I2 anterior) debe suponerse que la llave rotativa vuelve a su posición 1, para luego pasar a la posición 2, 3, etc. permaneciendo siempre 20 nseg. en cada posición. Así nuevamente se irán generando los tiempos T1, T2, T3, etc., siendo que para todas las instrucciones en T1 y T2, se lleva a cabo la fase de pedido de cada instrucción de manera semejante a como se hizo para I1 (figuras A4.6 y A4.7, de referencia para cualquier instrucción).
Por consiguiente al cabo del tiempo T2 de la instrucción I3 será **IP** = 0011 y **RI** = 01111110, lo cual se da por supuesto en el paso 3 ilustrado en la fig. A4.13.

Paso 2 (Decodificación de la orden que determina los pasos a seguir)

A la par que durante T2 llega la instrucción 01111110 al **RI**, sus primeros 4 bits (0111), cod-op de la operación a realizar, entran (fig. A4.13) al

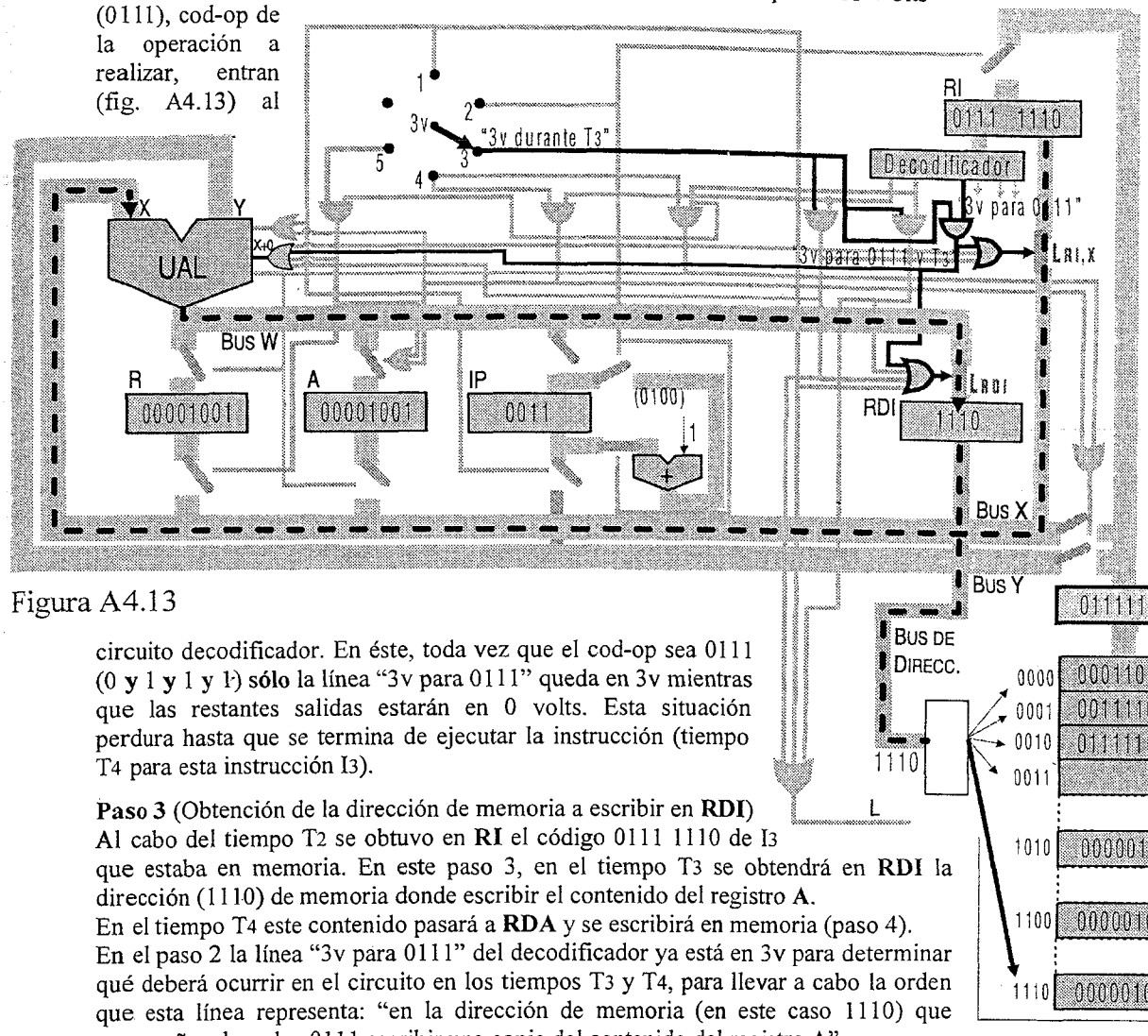


Figura A4.13

circuito decodificador. En éste, toda vez que el cod-op sea 0111 (0 y 1 y 1 y 1) **sólo** la línea "3v para 0111" queda en 3v mientras que las restantes salidas estarán en 0 volts. Esta situación perdura hasta que se termina de ejecutar la instrucción (tiempo T4 para esta instrucción I3).

Paso 3 (Obtención de la dirección de memoria a escribir en **RDI**)
Al cabo del tiempo T2 se obtuvo en **RI** el código 0111 1110 de I3 que estaba en memoria. En este paso 3, en el tiempo T3 se obtendrá en **RDI** la dirección (1110) de memoria donde escribir el contenido del registro **A**.
En el tiempo T4 este contenido pasará a **RDA** y se escribirá en memoria (paso 4).
En el paso 2 la línea "3v para 0111" del decodificador ya está en 3v para determinar qué deberá ocurrir en el circuito en los tiempos T3 y T4, para llevar a cabo la orden que esta línea representa: "en la dirección de memoria (en este caso 1110) que acompaña a la orden 0111 escribir una copia del contenido del registro **A**".
El paso 3 empieza cuando la llave rotativa está en la posición 3 en la que permanecerá T3 = 20 nseg, con lo cual quedará en 3v durante ese tiempo la línea "3v durante T3".
La dirección donde escribir el contenido de **A** se obtendrá de **RI**, siguiendo el mismo proceso que se hizo en el paso 3 de las instrucciones I1 e I2 (figuras A4.8 y A4.10).
Durante el tiempo T3 de ejecución de I3, las señales "3v para 0111" y "3v durante T3" están simultáneamente en 3v. Por lo tanto, como en la fig. A4.8, la línea de salida "3v para 0111 y T3" de la compuerta AND de las que ellas son entradas estará en 3v durante T3 según se definió en la fig. A4.4. Entonces (fig. A4.13), dicha salida "3v para 0111 y T3" deberá ordenar cerrar durante T3 los contactos $L_{RI,X}$ y L_{RDI}[8], y que la UAL realice

[8] Dado que L_{RDI} debe cerrarse durante T_1 cuando "3v durante T_1" está en 3v (como en el paso 1 de cualquier instrucción o también cuando "3v para 0001 y T_3" está en 3v (como en el paso 3 de I_1) o también como en el paso 3 de I_2, o también como en este paso 3 de I_3 (con "3v para 0111 y T_3" en 3v), estas 4 líneas deben ser 3 entradas de una OR (que eran 3 en la fig A4.10) cuya salida comanda a L_{RDI}

W = X + 0, permitiendo así que la dirección (1110) que está en la mitad de **RI** pase primero al bus **X**, de éste al bus **W**, desde donde pasa al **RDI**. De este registro llegará directamente a memoria a través del bus de direcciones vinculado al mismo. Así termina el desarrollo del paso 3.

Paso 4 (Realización de la operación ordenada)

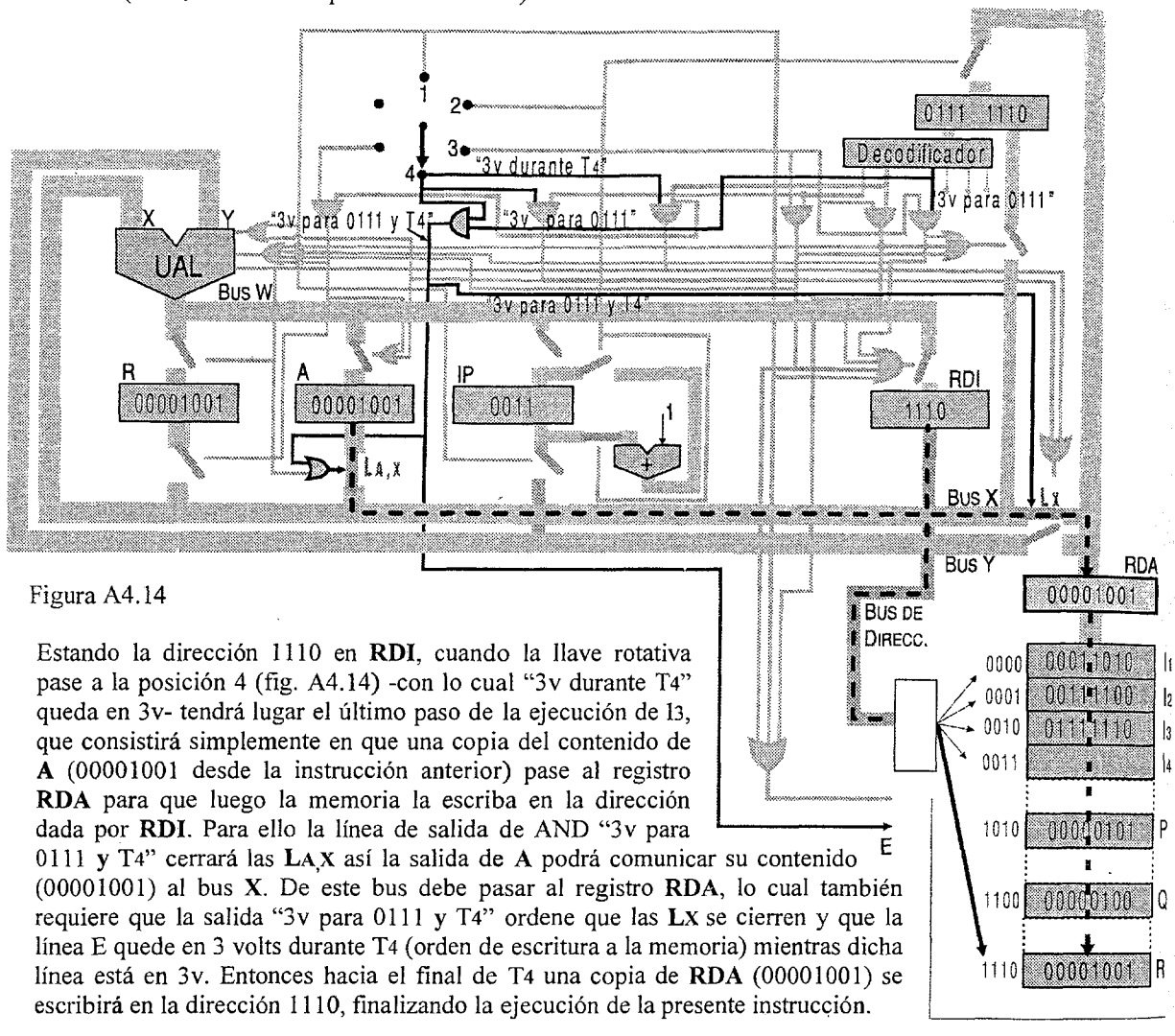


Figura A4.14

Estando la dirección 1110 en **RDI**, cuando la llave rotativa pase a la posición 4 (fig. A4.14) -con lo cual "3v durante T4" queda en 3v- tendrá lugar el último paso de la ejecución de I3, que consistirá simplemente en que una copia del contenido de **A** (00001001 desde la instrucción anterior) pase al registro **RDA** para que luego la memoria la escriba en la dirección dada por **RDI**. Para ello la línea de salida de AND "3v para 0111 y T4" cerrará las **LA,X** así la salida de **A** podrá comunicar su contenido (00001001) al bus **X**. De este bus debe pasar al registro **RDA**, lo cual también requiere que la salida "3v para 0111 y T4" ordene que las **Lx** se cierren y que la línea E quede en 3 volts durante T4 (orden de escritura a la memoria) mientras dicha línea está en 3v. Entonces hacia el final de T4 una copia de **RDA** (00001001) se escribirá en la dirección 1110, finalizando la ejecución de la presente instrucción.

De esta forma se ha calculado mediante 3 instrucciones la expresión R = P + Q, habiendo quedado el resultado (00001001) guardado en memoria en la dirección asignada a la variable R.

¿ *Cómo queda en definitiva el circuito del procesador desarrollado ?*

En la figura A4.14 están integradas las figuras A4.6 a A4.14. Se invita al lector a repetir la ejecución de las instrucciones antes tratadas en este nuevo esquema más general. El circuito de la **UC** está constituido por líneas de un solo conductor que pueden ser entradas de compuertas And, cuyas salidas van a compuertas Or, siendo que las salidas de estas últimas son las líneas de control. El conexionado AND-OR de la **UC** puede redibujarse según una matriz, como en la fig. A4.33 del Modelo de UCP rápida para instrucciones tipo RISC.

Análogamente, para $L_{RI,X}$, ella debe cerrarse durante T_3 en los pasos 3 de I_1 o de I_2 o de I_3 (presente instrucción), o sea si está en 3 v la línea "3v para 0001 y T_3" o "3v para 0011 y T_3" o "3v para 0111 y T_3", por lo que estas 3 líneas deben ser 3 entradas de una **OR** (dibujada en la fig A4.10 con 2 entradas) cuya salida comanda a $L_{RI,X}$

Complemento de la Unidad 1

CODIFICACION Y OPERACIÓN DE ENTEROS Y REALES FLAGS DE LA UAL

INTRODUCCIÓN

Hasta el presente hemos representado números, los hemos operado e interpretado resultados suponiendo que en un programa en alto nivel se pueden definir variables desitgnadas supuestamente "MAGNITUDES", que a nivel de máquina se representan en memoria como **binarios naturales**, los cuales se han tratado en el Apéndice 1 de la Unidad 1.

Para variables tipo "*Enteros*" y "*Reales*" supuestamente definibles en un programa en alto nivel, este complemento está dedicado a la representación binaria de las mismas en memoria, a las operaciiones con dichas representaciones, y a la interpretación de resultados de estas operaciones. Así se tratarán:

a) "ENTEROS': corresponden a números positivos o negativos en base diez, (±N) simbolizables en pantalla con los símbolos + y − , cuya magnitud **N** es un número natural. A nivel de máquina se representan en memoria mediante *binarios naturales* que representan enteros. Estos también se conocen como "*binarios con bit de signo*" o "*binarios signados*", por que el valor 0 ó 1 del bit extremo izquierdo de dichos binarios naturales (bit de signo) indica si representa un entero positivo o negativo, respectivamente. Son operados en la UAL como binarios naturales.

b) "REALES": corresponden por ejemplo a enteros positivos y negativos, fraccionarios, irracionales, que en memoria se representan como números binarios en "punto (coma en castellano) flotante", en una notación exponencial del tipo $N = \pm m \times 10^{\pm p}$ y son operados en el Coprocesador Matemático o Unidad de Punto Flotante (FPU).

En síntesis:

PROGRAMA EN ALTO NIVEL	*BAJO NIVEL (MAQUINA)*
Magnitudes	**Binarios naturales**
Enteros	**Binarios naturales considerados con bit de signo (signados)**
Reales	**Binarios en punto flotante**

Al final de este complemento se trata la codificación binaria BCD, que en alto nivel corresponde a variables tipo "PACKED"

Dado que, como se verá, los binarios que representan a enteros se suman y restan como binarios naturales, en relación con la resta de éstos, ya planteada su mecánica en el Apéndice 1 de la Unidad 1, a los fines de una mejor comprensión de la representación y resta de enteros, se volverá a ver más en detalle la resta de binarios naturales.

Resta de números naturales sin pedir prestado, mediante la suma del minuendo más el complemento al módulo del sustraendo

Primero plantearemos este método para números en base diez y luego para binarios.

Con el objetivo de usar un (circuito) sumador *también* para restar, restaremos números naturales:

a) mediante una suma

b) sin "pedir prestado" de una posición a otra.

En lo que sigue primero efecturaremos en base diez 543 – 84 mediante una suma, lo cual originará una segunda resta que se puede realizar sin "pedir prestado". Luego de mostrar el "artificio" se indicará la forma práctica de concretar a) y b).

Sea A – B = (543 – 084) = R (resultado a determinar).

Si sumamos a ambos lados 1000, que tiene tantos ceros como dígitos tiene A, el resultado R quedará excedido en 1000:

(543 – 084) + 1000 = R + 1000 (paso 1) reordenando: 543 + (1000 – 084) = R + 1000

La resta (1000 – 084) = 916 requiere "pedir prestado", siendo 916 el complemento (lo que le falta) a 84 para ser 1000, pudiéndose simbolizarse C_{84} = 916; siendo 1000 la cantidad total de números decimales (de 000 a 999) que pueden formarse con n=3 dígitos, denorminada **"módulo"** (1000 = M). O sea $C_{84} = 916 = C_B$ es el complemento al "módulo" o a "la base" de 84 = B

Calcularemos C_{84} = (1000 – 084) = 916 sin "pedir prestado" dado que (1000 – 084) = 1 + 999 – 084

(999 – 084) = 915 no requiere "pedir prestado", siendo 5 lo que falta a 4 para ser 9; 1 lo que falta a 8 para ser 9, etc.

En definitiva haremos 543 + (1000 – 084) = 543 + (999 – 084) + 1 = 543 + 915 + 1 = 1459 = R + 1000, como se indica a continuación en grisado. Como se justificará, será R = 459 que resulta simplemente de descartar el uno de 1459, sin tener que realizar un paso extra para restar el exceso de 1000.

A = 543 ⟶ 543 = = 543

B = 084 ⟶ + 915 } 916 = 1000 – 84 = C_{84} = C_B = + 1000 – 84

1459 = 1000 + 459 = = 1000 + (543 - 084) = 1000 + R

R

JUSTIFICACIÓN POR QUÉ SE DESCARTA EL UNO Y EL RESULTADO SON LOS DÍGITOS QUE LE SIGUEN:

Las sumas de la izquierda y derecha son iguales, dado que 543 = 543 y 915 + 1 = 916 = C_B =1000 – 84, la suma derecha permite poner en evidencia que 1459 *es la resta pedida* (543 – 84) = R *excedida en 1000* (paso 1). Resulta así que el exceso de 1000 que está sumado a R es simplemente el uno que descartamos de 1459, por lo que **R = 459** es el resultado de (543 – 84), como se indica a continuación:

1459 = 1000 + 459 = 1000 + (543 - 084)

R

Se explica ahora las ventajas de usar como exceso la unidad seguida de ceros (módulo M = 1000 en este caso): por un lado como valor sumado a R es fácil de eliminar, sin necesidad de hacer 1459 – 1000; y por otro si al módulo se le resta uno (módulo menos uno) permite hacer la resta M – B = C_B = (1000 – 084) como (999 – 084) + 1 sin "pedir prestado".

Dado que 999 = 1000 – 1 = M–1, el 915 = 999 – 84 = C'_B, es lo que le falta al 84 = B para ser 999 = M–1, o sea es el complemento al "módulo menos uno" (C'_B), llamado también complemento a "la base menos uno": C'_B = M–1 – B

Siendo C_B = (1000 – 084) = (999 – 084) + 1 = C_B + 1, resulta en general que *para calcular el complemento al módulo de un número sin "pedir prestado", primero calculamos su complemento al módulo menos uno, y a éste le sumamos uno.*

Generalización: (A – B) = R

(A – B) + M = R + M (M = 10^n) (A – B) + M = A + (M – B) = A + C_B = A + C'_B + 1 = R + M

siendo: C'_B = M–1 – B; **n** es la cantidad de dígitos de A; y 10 la base en cualquier sistema numérico (10 es dos en binario)

Resta de naturales binarios

Repitiendo el planteo recién visto para naturales en base diez, lo aplicaremos para restar en formato n = 8 los binarios naturales 10001100 y 00100110 que corresponden a las magnitudes 140 y 38 en base diez.

1) Esto es, efectuaremos A – B = (10001100 – 00100110) = R como se efectúa en un computador, según se indica a continuación en grisado:

A = 140 = 10001100 ⟶

B = 38 = 00100110 ⟶

R

10001100
+ 11011001
1
01100110

R

El método consiste, pues en ***sumarle al minuendo A el número que resulta de invertir cada uno de los bits del sustraendo B, y en la misma operación sumar uno***. Para hallar el resultado R ***se descarta el uno fuera de formato, y se toman los bits que siguen a dicho uno.***

R = 01100110 = 102 Verificación 102 = 140 – 38

A continuación *justificaremos* la metodología empleada.

Si en 1) sumamos a ambos lados 100000000, con tantos ceros como dígitos tiene A, el resultado R quedará excedido en 100000000 = 256_d = M, que también es la cantidad de números binarios que pueden formarse con 8 bits (de 00000000 a 11111111):

(10001100 – 00100110) + 100000000 = R + 100000000 (paso 1); reordenando:

10001100 + (100000000 – 00100110) = R + 100000000

La resta (100000000 – 00100110) requiere "pedir prestado". Para evitar esto, en lugar de hacer (100000000 – 00100110) = 256 – 38 = M – B = C_B efectuaremos 255 – 38 + 1 que en binario es 11111111 – 00100110 + 1, siendo 11111111 = M–1 = 255.

Justificaremos primero porqué invertimos cada bit del sustraendo B y sumamos uno:

11111111 = M – 1	La resta 11111111 – 00100110 = M–1 – B = C'_B no requiere "pedir prestado" pues siempre se tendrá
00100110 = B	1 – 0 = 1 ó 1 – 1 = 0 *ni tampoco hace falta realizarla, pues cada bit del resultado* (C'_B) *tiene valor inverso*
11011001 = C'_B	*inverso que el correspondiente bit del sustraendo B.*

Dado que en lugar de hacer (100000000 – 00100110) = 256 – 38 = M – B = C_B se hizo 11111111 – 00100110 = M–1 – B = C'_B = = 255 – 38, se debe sumar uno, como aparece en la suma en grisado.

JUSTIFICACIÓN POR QUÉ SE DESCARTA EL UNO DE LA IZQUIERDA Y EL RESULTADO SON LOS BITS QUE LE SIGUEN:

A continuación a la izquierda está repetida la suma resaltada en grisado. En ella 11011001 + 1 = 11011010 = C_B = = (100000000 – 00100110). Pensado en base diez: 217 + 1 = 218 = 256 – 38. La última resta binaria reemplaza a C_B en la suma de la derecha, que por lo tanto es igual a la de la izquierda. La suma derecha permite poner en evidencia que 01100110 *es la resta pedida.*

```
 10001100 =                          = 10001100
+11011001 }                        +
        1 } 11011010 = CB =         = 100000000 – 00100110
01100110 = 100000000 + 01100110 = 100000000 + (10001100–00100110)
                                                        R
```

Las sumas de la izquierda y derecha son iguales, dado que 10011100 = 1001100 y 11011001 + 1 = 100000000 – 00100110 = = C_B , siendo que la suma derecha permite poner en evidencia que 101100110 *es la resta pedida* (10001100 – 00100110) = R *excedida en 100000000* (paso 1). Resulta así que el exceso de 100000000 que está sumado a R es simplemente el uno que descartamos de 101100110, por lo que **R = 01100110** es el resultado de (10001100 – 00100110).

N1 Representación de enteros usando dígito de signo y el complemento de su magnitud para los negativos

N1.1 Números decimales con dígito de signo, con los negativos representados por el complemento de su magnitud

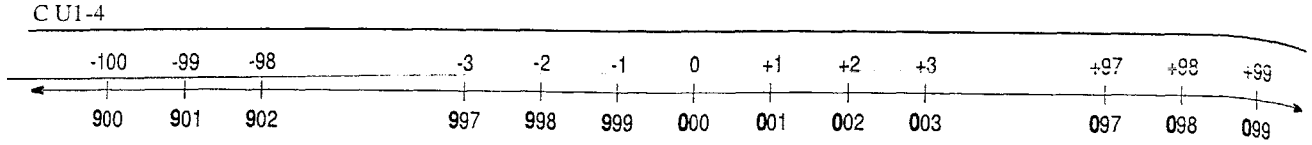
Un número decimal natural puede tener distintos significados según la convención que se trate. Así 210850 puede significar el número 210.850 ó el 21/08/50 ó 21 h 08' 50" ó 2108 artículos de código 50, etc.

Nos proponemos representar números decimales negativos y positivos sin usar los símbolos – y + empleando números decimales ***naturales***, y que además la nueva convención adoptada permita sumar los números enteros definidos en ella como si fueran naturales. Así se podrá aprovechar un mismo sumador para enteros y naturales.

Estos requerimientos sirven de base para el planteo de un problema similar en los computadores: dado que en su interior no hay un cable para indicar – ó + ¿cómo se representan por ejemplo los números negativos ?

Volviendo a los decimales, consideraremos el cuenta-vueltas de un pasacasetes con números del 000 al 999 que puede contar progresiva o regresivamente. Se trata, pues, de poder representar números negativos y positivos utilizando un subconjunto de esas 1000 combinaciones distintas que puede formar el cuenta-vueltas.

Qué ocurre si establecemos que los positivos son los que aparecen a partir del 000 cuando el cuenta-vueltas progresa del 001 hasta el 099; y que los negativos son las combinaciones que se forman cuando el cuenta-vueltas va hacia atras, de modo de asignar el 999 al –1; el 998 al –2; el 997 al –3, y así sucesivamente hasta retroceder hasta el 900 que implicaría 100 pasos hacia atrás, o sea el –100.



Con estos números de n=3 dígitos, se cumple que **(–A) + A = 0** si descartamos en el resultado el uno de 1000. Por ejemplo, la representación 997 del negativo –3 sumado a su magnitud (3), o sea: 997 + 3 = 1000 puede verse como (–3 + 1000) + 3 = **1000** = 1000 + **000**. Esto es, 997 es –3 excedido en 1000, exceso que ha pasado al 000 considerando 1000 = = 1000 + 000. Por consiguiente, si en 997 + 003 = 1000 descartamos el uno que está fuera del formato, el resultado es 000.

El cuenta-vueltas también serviría para efectuar sumas algebraicas. Así –3 + 12 implicaría a partir del 000 ir primero al **9**97 y luego avanzar 12 combinaciones, de modo de llegar al **0**09, que es el resultado de esa suma.

Se observa que el dígito izquierdo indica el signo: los positivos tienen todos un cero como dígito izquierdo, mientras que los negativos un nueve. Decimos que el signo de los números enteros así representados está en el dígito extremo izquierdo, el cual es el ***"dígito de signo"***. En esta convención los positivos mantienen directamente su magnitud en las cifras que están a la derecha de dicho dígito, pero los negativos siguen otra regla de representación, según ser verá. Como ser, el +3 se representa 003, mientras que la representación del –3 es 997.

Puesto que se dedica un dígito extra para el signo, el mayor número positivo tiene magnitud 99 (en vez de 999 usando los tres dígitos para la magnitud). El **9**00, como se vió, corresponde a un número negativo de magnitud 100.

Nos encontramos con una forma distinta de interpretar los números naturales del cuenta-vueltas. Ahora, por ejemplo el 997 como natural representa 997 unidades, pero interpretado como número con dígito de signo es el –3.

Dejaremos de lado el cuenta-vueltas y generalizaremos esta convención. Supongamos que queremos representar el número –63 sin tener que ir 63 pasos hacia atrás desde el 000 para ver qué número es entre los que empiezan con 9. Observando en la recta cada negativo y su correspondiente representación mediante el número natural que empieza con 9, resulta que la magnitud de cualquier negativo sumada al natural que lo representa **da siempre por resultado 1000.** Así, para el –1 es ***1 + 999 = 1000***; para el –3 es ***3 + 997 = 1000***; para el –99 → ***99 + 901 = 1000***; para el –100 es ***100 + 900 = 1000***, etc. (Esto equivale a lo expresado que se cumpliría que **A + (–A) = 0** si no se considera el 1 de 1000).

Por lo tanto, la representación de un número negativo es el número natural que sumado a su magnitud (también número natural) da por resultado 1000, que es la cantidad de números distintos que hay entre 000 y 999 para n=3 (designada "módulo")

Así el número natural 997 es lo que le falta a 3 para ser 1000; 901 lo que falta a 99 para ser 1000, etc.

Decimos que 997, que representa al negativo –3, es ***"el complemento a 1000"*** de 3 (lo que falta a 3 para ser 1000). Entonces, el número que representa a –63 será el complemento a 1000 (que simbolizaremos C_{63}) de su magnitud 63.

O sea debe ser: $63 + C_{63} = 1000$. Por lo que dicho número será: $C_{63} = 1000 - 63 = 93$

De manera inversa, si preguntáramos qué número negativo es el representado por el natural 937, o sea cuál es su *magnitud*, la misma sería el complemento a 1000 de 937: $937 + C_{937} = 1000$.

La magnitud será $C_{937} = 1000 - 937 = 63$. Luego 937 representa al número –63.

También podemos escribir **–63 ≡ 937** = C_{63} indicando ≡ que 937 "funciona como" o "representa" –63, siendo la igualdad formal: **937 = –63 + 1000**. O sea que la representación de un negativo por el complemento de su magnitud, *puede verse como dicho negativo* **excedido en 1000**.

Obsérvese que cada número positivo es simplemente el número natural que es su magnitud, *necesariamente* con un cero a la izquierda, pudiéndose pensar como si dicho cero fuese un signo + "pegado" a su izquierda.

A diferencia, los negativos **no son simplemente su magnitud con un nueve agregado a su izquierda**, sino el complemento al módulo (en este caso 1000) de dicha magnitud. En general será: **$-N \equiv C_N = M - N$** (M es el módulo)

N1.2 Complemento al módulo o "a la base"

Puesto que 1000 es la *cantidad de números o combinaciones distintas que pueden formarse* en el cuenta-vueltas de 3 dígitos, y técnicamente este número se denomina ***"módulo"*** (**M**), en forma genérica el "complemento a 1000" lo denominamos ***"complemento al módulo"***.

Siendo $M = 1000 = 10^3$ o sea la base a la potencia tres, también podría decirse "complemento a la base a la potencia 3", pero ha quedado ***"complemento a la base"*** como **sinónimo** de complemento al módulo.

En este caso que usamos **n** = 3 dígitos, para cualquier número **N** se cumplirá, generalizando expresiones usadas más arriba, que $N + C_N = 1000 = 10^3$

Para combinaciones de **n** dígitos, se tendrá la expresión general de un uno seguido de **n** ceros:

$$N + C_N = 10^n = 1000\ldots.0 = M$$

Las máquinas operan con números que sólo pueden tener un número fijo **n** *de dígitos* (bits en binario), o sea con *"formatos"* determinados[2] , de donde resulta que *para cada formato de* **n** *dígitos corresponde un valor de módulo igual a la unidad seguida de* **n** *ceros, número que en* ***cualquier*** *base se expresa* 10^n (10 es 2 en base dos).

Si operamos en formato de 5 dígitos (módulo 10^5 = 100000), el número entero –63 se representaría como **99937** = = 100000 – 63; mientras que +63 sería **00063**. Conforme a lo anterior, podemos decir que en la convención que usamos para representar los enteros con dígito de signo, *los negativos se representan por el complemento al módulo (también llamado "a la base") de su magnitud.* En general será: $-N \equiv C_N = M - N$

Es conveniente acostumbrarse a ver los números negativos o positivos como en un cuenta vueltas, con todos sus ***n*** dígitos, o sea por ejemplo 63 como 00063, con ceros a la izquierda completando el formato.

COMPLEMENTO AL "MÓDULO MENOS UNO"

Para obtener 937 = 1000 – 063 hay que efectuar una resta "pidiendo prestado".

Esto último puede evitarse si se hace primero 999 – 063 = 936, y al resultado se le suma uno: 936 + 1 = 937.

Siendo 1000 el módulo, es 999 el módulo menos uno; por lo que 936 es el ***complemento al "módulo menos uno"*** del número N=063, que simbolizaremos C'_{063}

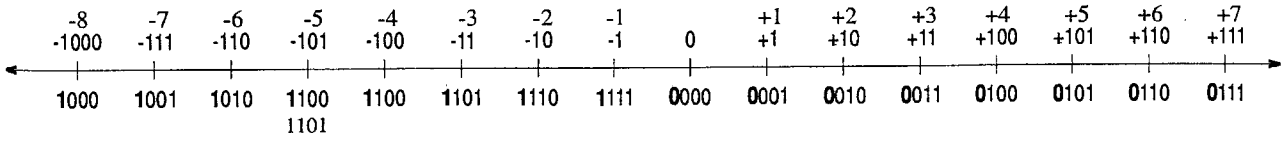
De donde resulta, que para evitar "pedir prestado" *conviene determinar el complemento al módulo de un número calculando primero su complemento al módulo menos uno y luego sumarle uno*: $C_N = C'_N + 1$

N1.3 Números binarios con bit de signo, correspondientes a "integers" (enteros) en lenguajes de alto nivel

Aplicaremos los conceptos anteriores a los números binarios naturales para representar números binarios enteros, por lo cual es muy importante tener presente en detalle todo lo visto en relación con los enteros decimales.

Podemos imaginar un "cuenta-vueltas" binario, constituído por un tambor que en su superficie tenga escritas, una debajo de otra, las 16 combinaciones binarias de 4 bits[1] que van del 0000 al 1111, el cual puede ir hacia adelante o retroceder. Si está en 0000 y retrocede una combinación se pasa a 1111, del mismo modo que un cuenta-vueltas decimal de 4 dígitos (con 10000 números decimales) pasaría de 0000 a 9999.

A partir del 0000, avanzando desde el 0001 al 0111, estos números naturales con bit extremo izquierdo **0** corresponderán a números positivos; mientras que si el tambor retrocede desde el 0000, se tendrá 1111, 1110, 1101, . . . hasta el 1000, números naturales con bit extremo izquierdo **1**, que pasarán a representar números binarios negativos que les corresponden, los cuales en el papel se simbolizan con un signo menos adelante como los negativos decimales.
Así, los negativos tienen bit de signo 1, y los positivos bit de signo 0. El número cero se conseidera positivo, por tener bit de signo 0. Podemos expresar $-110 \equiv 1010$ donde el símbolo ≡ indica que el natural 1010 "representa" al –110.



Los números binarios naturales indicados debajo de la recta representan a los binarios enteros (positivos y negativos) y sus correspondientes enteros en base diez escritos sobre la recta. Se denominan ***"binarios con bit de signo"*** o ***"binarios signados"***, *siendo que los negativos se representan por el complemento al módulo de su magnitud.*

Nótese que estamos usando los números **naturales** con su bit extremo izquierdo de valor uno **con dos significados**: así el 1101 como natural simboliza 13_D unidades, y como signado el número binario negativo $-11_B = -3_D$

---

[2] Los números de una máquina sólo pueden constar de una cantidad fija de dígitos, como es de apreciar en una calculadora manual, que en un computador típicamente pueden ser 8, 16, 32, 64, 128 bits. Cuando se opera con un número finito de dígitos puede suceder que no se cumplan ciertas propiedades de los números. Así, dado que en el cuenta vueltas ejemplificado no existen números mayores que 999, ni números menores que 000, puede que para ciertos números no se cumpla que sea: (A–B)xC = AxC – BxC o que sea (A–B) + C = A – (B + C), si en el lado derecho de esas igualdades AxC , BxC ó B+C superan 999.

[1] Si bien se ejemplifica con todas las combinaciones que pueden formarse con 4 bits –dado que así pueden tenerse a la vista todos los números que se pueden representar en este formato– el modelo es generalizable para combinaciones de *cualquier* número de bits, del mismo modo que los conceptos desarrollados para el cuenta-vueltas decimal de 3 dígitos son aplicables a conjuntos de números decimales compuestos por tantos dígitos como se requiera.

Dado que estamos usando un bit para el signo, si con 4 bits el número natural mayor que se podía representar es el ~~$1111_B = 15_B$~~ ahora el mayor positivo es el $0111_B = 7_B$, siendo el más negativo el $1000_B \equiv -1000_B = -8_D$ [2]

Puesto que el número de combinaciones binarias distintas que pueden formarse con 4 bits es $2^4{}_D = 16_D = \mathbf{10000}_B$, dicho número constituirá *el módulo* para el conjunto de combinaciones dadas, representadas antes sobre una recta.

Como en el cuenta-vueltas decimal, también se verifica que ***para cualquier número negativo la suma de su magnitud*** (número natural) ***más el número binario natural que lo representa da por resultado el módulo:*** [3]

1 1 1 1	0 1 0 1	0 1 1 1
+ 0 0 0 1	+ 1 0 1 1	+ 1 0 0 1
1 0 0 0 0	1 0 0 0 0	1 0 0 0 0 etc...

Esta relación permite determinar el número binario natural que representa a un entero negativo que se quiere representar.

Por ejemplo, si se quiere conocer para módulo 10000_B (16_D) como se representa el número -110_B, se parte de su magnitud 110 (6_D) y se halla el complemento de este número al módulo:
$C_{0110} = (10000 - 110)_B = 1010$ $(16 - 6 = 10)$, combinación que coincide con la indicada para -110 en la recta dibujada.

Obsérvese que en la presente convención para representar un negativo con bit de signo se usa el complemento al módulo de su magnitud, por lo que **no se trata simplemente de agregar un uno adelante de un número natural.**

Así, -110_B se representa con bit de signo como 1010 y **no** como 1110.

CÁLCULO DEL COMPLEMENTO AL MÓDULO A TRAVÉS DEL COMPLEMENTO AL MÓDULO MENOS UNO

La resta anterior 10000 – 110 = 1010 requiere "pedir prestado". Para evitar esto conviene –como se hizo en base diez en la sección N1.2– calcular primero el complemento al "módulo menos uno" (C'), y a éste sumarle uno ($\mathbf{C_N = C'_N + 1}$). Esto en decimal significa que en vez de hacer 16 – 6 = 10 se efectúa 15 – 6 + 1 como se hace a continuación en binario.

Puesto que operamos con números de 4 bits, el módulo es 10000 (16_D), y el módulo menos uno será 1111 (15_D) [1]
Si queremos calcular C_{0110} hacemos:

– 1111	Observando los números **0110** y **1001** = C'_{0110}, resulta que el complemento al módulo menos uno puede hacerse simplemente, **invirtiendo los unos y ceros del número a complementar por ceros y unos respectivamente, sin que sea necesario** realizar **resta alguna** como la efectuada (1111 – 0110). No debe perderse de vista que el complemento al módulo menos uno, y el uno que luego se suma en definitiva sirvern para hacer 10000 – 0110 = 1010 sin pedir prestado y sin reali zar ninguna resta. De esta forma no hay que "pedir prestado" ni tampoco efectuar resta alguna.
0110	
1001 = C'_{0110}	
+ 1	
1010 = C_{0110}	

Regla práctica para hallar en base dos el complemento al módulo

Para calcular el complemento al módulo[2] de un número binario, se *invierten* los unos y ceros del mismo por ceros y unos; y luego se *suma uno* al número así formado.

EJEMPLOS

Aplicaremos los conceptos anteriores a la representación de números enteros suponiendo una máquina que opera con números binarios de 8 bits (*"formato 8"*).

Con números de 8 bits el módulo es $2^8{}_D = 256_D = \mathbf{100000000}_B = 10^{1000}{}_B$; esto es, la unidad seguida de 8 ceros en binario, que implica que pueden formarse 256_D combinaciones binarias.

En este formato, el mayor número positivo que puede representarse es $\mathbf{01111111}_B = \mathbf{127}_D$, y el negativo de mayor magnitud es $\mathbf{10000000}_B = \mathbf{-128}_D$

Problema directo: representar con bit de signo el número $-59_D = -111011$. Para ello seguiremos los siguientes pasos:

---

2 Obsérvese al respecto que en esta convención siempre el número negativo de mayor magnitud que puede representarse, supera en uno al positivo de mayor magnitud. Esto se debe a que siendo par el número total de combinaciones, y presentar el número cero el dígito **0** como bit de signo, de hecho es un número positivo más. Esto también se verificó en el cuenta-vueltas decimal, donde los números extremos eran 99 y –100.

3 También con este subconjunto de binarios de n = 4 bits se verificaría que **A + (–A) = 0**. Así, la magnitud de un negativo sumada al binario que lo representa, como ser 101 + 1011 = 10000 (5 + 11 = 16) puede verse como 101 + (–101 + 10000) = 10000 = 10000 + 0000. [en decimal: 5 + (–5 + 16) = 16]. Esto es, 1011 es –101 excedido en 10000, exceso que ha pasado al 000- considerando 10000 = 10000 + 0000 .
Por consiguiente, si en 101 + 1011 = 10000 descartamos el uno que está fuera del formato, el resultado es 0000.

1 *En general, para números de n bits, el módulo será la unidad seguida de n ceros (10^n), y el módulo menos uno estará constituído por n unos*

2 También denominado *"complemento a la base"* o *"complemento a dos"*

1. Representar su magnitud 59 como binario natural: 111011 = 59
2. Completar los bits de la magnitud con ceros hasta completar el formato, resultando de hecho el +59 [3] 00111011 = +59
3. Cambiar los ceros por unos y los unos por ceros (complemento M–1) [4] ↓ 11000100
4. Sumar 1 al complemento al módulo menos uno + 1 = 11000101

En definitiva para formato 8 se hizo: $-59_D = -00111011 \equiv$ **11000100 + 1** = 11000101 (natural que representa al −00111011)

El número binario natural así obtenido representa –con la convención del complemento al módulo o a la base– el número entero negativo dado.

No debe perderse de vista, que *los pasos indicados, en especial 3 y 4* (ver "Cálculo del complemento a través del . . .) *sirven en esencia para obtener sin "pedir prestado" el resultado de la resta*: $C_{00111011} = 100000000 - 00111011 = 11000101$

Por lo tanto, en formato 8 resultó $-59_D = -111011_B \equiv 11000101_B$ [1] (ACLARACIÓN CONCEPTUAL)

Es importante realizar en orden los cuatro pasos indicados, so pena de no llegar al resultado correcto.

La representación de un positivo, es sencilla: **simplemente resulta de agregar a su magnitud** (número natural) **uno o más ceros a la izquierda hasta completar el formato.**
Esto es, **un número positivo debe tener un cero en su extremo izquierdo** (correspondiente a su signo).
Por ejemplo la representación del entero positivo 59, o sea +59 requiere:

1. Representamos su magnitud 59 como binario natural: 59 = 111011
2. representamos +59 con bit de signo completando con ceros el formato +59 = 00111011

Obsérvese que el natural 59 en binario es 111011, o sea empieza con 1, sin que ello signifique que es un número negativo, pues estamos suponiendo que representa un número natural. Si hubiésemos supuesto que 111011 representa un entero, sería un número negativo, pues el uno de la extrema izquierda indicaría bit de signo negativo. Esto está de acuerdo con lo que hemos dicho que un binario que empieza con uno, tiene dos significados según se lo considere natural o entero.

Problema inverso: dada una combinación binaria cuyo bit extremo izquierdo es uno, como ser 1010, *suponiendo* que representa un número entero, determinar cuál es éste expresado en base diez.
A los fines conecptuales conviene tener presente la recta con todos los binarios de 4 bits con bit de signo antes dibujada.
En principio podemos asegurar que si es un entero y tiene bit de signo uno, es un número negativo, cuya magnitud X desconocemos: $1010 \equiv -X$. Para determinar el valor de esta magnitud X sabemos que la representación de un negativo es un número natural (1001 en este caso) que es el complemento al módulo de la misma. Observando la recta citada resulta que $1010\ (10)_D$ representa al $-110\ (-6)_D$, siendo 110 el valor que buscamos. Pero no es necesario ver la recta. Dado que 1010 + 110 = 10000 = M, resulta que los números naturales 1010 y 110 son complementarios: 1010 es el complemento al módulo de 110 (magnitud del número negativo), y a su vez 110 es el complemento al módulo de 1010. Por lo tanto, conocido 1010 resulta que 110 = 10000 – 1010, resta que es más fácil hacer como 0101 + 1 = 0110.

En definitiva, $1010 \equiv -X = -(0101 + 1) = -\ \overset{8\ 4\ 2\ 1}{0\ 1\ 1\ 0} = -6_D$ Obviamente que si se hace $\overset{8\ 4\ 2\ 1}{1\ 0\ 1\ 0} \equiv -10_D$ **está mal.**

Veamos otro ejemplo: dada la combinación 11010110 suponiendo que representa un número entero, determinar cuál es éste expresado en base diez.
Puesto que el bit de signo es **1**, $11010110 \equiv -X = -(00101001 + 1) = -\ \overset{32\ 16\ 8\ 4\ 2\ 1}{1\ 0\ 1\ 0\ 1\ 0} = -42$

Si se hiciera simplemente $\overset{128\ 64\ 32\ 16\ 8\ 4\ 2\ 1}{1\ 1\ 0\ 1\ 0\ 1\ 1\ 0} = 198$ se estaría considerando al 11010110 como un número natural.

En cambio **se puede llegar al resultado −42 si al bit de signo se le da el valor −128, y a los siguientes 64, 32, 16, 8, 4, 2, 1**

---

3 *El positivo debe tener por lo menos un cero, sino implica que el número en cuestión no se puede representar.*

4 Vale decir se ha efectuado: $\overset{0\ 0\ 1\ 1\ 1\ 0\ 1\ 1}{\downarrow\ \downarrow\ \downarrow\ \downarrow\ \downarrow\ \downarrow\ \downarrow\ \downarrow}\ 1\ 1\ 0\ 0\ 0\ 1\ 0\ 0 + 1 = 11000101$
Obsérvese al respecto el cuidado que debe tenerse de partir del número positivo que en el formato dado *tenga correctamente completados los ceros de la izquierda* para dicho formato, para no incurrir en errores.

1 Usamos el símbolo ≡ para indicar que el 11000101_B por la convención adoptada representa al -111011_B siendo que **no se trata de una igualdad**, Esta sería: 11000101 = −111011 + 100000000

-128 64 32 16 8 4 2 1

Así resulta: 1 1 0 1 0 1 1 0 $= [1x(-128) + 1x64 + 0x32 + 1x16 + 0x8 + 1x4 + 1x2 + 0x1]_D = (-128 + 86)_D = -42_D$

Si se parte de un número entero como el 01100011, para determinar qué número decimal es, dado que su bit de signo es cero, se trata de un entero positivo, por lo cual la solución es sencilla, pues su magnitud es simplemente el número natural que sigue al cero:

64 32 16 8 4 2 1
0 1 1 0 0 0 1 1 = +99

Propagación de signo

Otra importante conclusión puede extraerse del siguiente ejercicio: representar –59 con 16 bits (antes representado con 8) En formato 16 será 59 = 0000000000111011, y su complemento a módulo será

1111111111000100 + 1 = **1**1111111**11000101** = –111011 = –59

Comparando con la misma representación de –59 realizada más arriba en formato 8, resulta que ahora se tiene el mismo número con ocho unos mas a la izquierda, de donde se deduce que:

> *Cuando se tiene un número con bit de signo* ***negativo****, si se agregan* ***unos*** *a la izquierda del mismo, el número no cambia; de manera análoga que si a un número* ***positivo*** *se le escriben ceros a la izquierda.*

Por lo tanto, cuando se debe pasar un número con bit de signo negativo (positivo) a un formato mayor, se le deben agregar unos (ceros) hasta completar el nuevo formato, acción que se conoce como ***"propagación del signo"***

Valores extremos representables con bit de signo en un formato dado

Determinaremos en base diez los valores límites que existen para números de 4 bits con bit de signo, y encontraremos una expresión generalizable para cualquier número de bits.

Para **n** = 4, el positivo de mayor magnitud es $+N_{max} = 0111_B = 7_D = (8-1)_D = (2^3-1)_D = (2^{4-1}-1)_D$

Generalizando para **n** bits: $+N_{max} = (2^{n-1}-1)_D$;

El negativo de mayor magnitud para n=4 es $-N_{max} = 1000_B = -8_D = (-2^{4-1})_D$; y para **n** será: $-N_{max} = (-2^{n-1})_D$

Por lo tanto, para **n** bits el rango de valores representables con bit de signo, en base diez va de -2^{n-1} a $2^{n-1}-1$

Para **n** = 8 bits resulta: $+N_{max} = 01111111 = 2^{8-1}-1 = 2^7-1 =$ **127;** y $-N_{max} = 10000000 = -2^{8-1} =$ **–128**

Por lo general cuando en muchos lenguajes de alto nivel se definen ***"integers"****, sin más aditamentos, el compilador o ;una subrutina los traduce a binarios con bit de signo en formato de* ***16*** *bits.*

Aplicando las expresiones anteriores, resulta un rango entre **–32768** y **+ 32767**, siendo que los números correspondientes son: 1000000000000000_B (15 ceros) y 0111111111111111_B (15 unos)

Resulta relevante compararlo con los máximos usando **32 bits**, correspondientes a "**Long Integers**"

$-N_{max} = -[10000...(31\ ceros)..0]_B = (-2^{31})_D \approx (-2x10^9)_D$ y $+N_{max} = [10000...(31\ ceros)..0]_B = (2^{31}-1)_D \approx 2x10^9{}_D$

Una forma rápida de hallar **x** en $2^{31} = 10^X$ es tener presente que $2^{10} = 1024 \approx 1000 = 10^3$

por lo que $2^{31} \approx 2^{10}x2^{10}x2^{10}x2^1 = 10^3x10^3x10^3x2 = 2x10^9$, valor semejante al indicado más arriba.

Valores extremos representables con binarios naturales en un formato dado

Se debe tener presente que si se usan números binarios naturales que representan **magnitudes** en alto nivel, partiendo de que para n=4 bits el rango representable va de **0000** a **1111** = $2^4-1 = 15$, si se generaliza para cualquier formato **n**, el rango irá de **cero** a 2^n-1 . *Para igual* ***n*** *se duplica la magnitud máxima respecto de los números con bit de signo*

Recapitulación de la representación de enteros como binarios con bit de signo:

- Un entero positivo en un formato dado es su magnitud (binario natural) con *bit de signo* **0** *como bit extremo izquierdo*, pudiendo existir más ceros a la izquierda de su magnitud, hasta completar dicho formato.

- Un entero negativo en un formato dado se representa por un binario natural que es *el complemento al módulo* (C_N) de su magnitud **N**, debiendo presentar por lo tanto un *bit de signo* **1** como *bit extremo izquierdo* en su representación. (Puede verse el complemento C_N como el negativo $-N$ excedido en el módulo: $\mathbf{C_N = -N + M}$)
- Un número negativo (positivo) *no cambia* si se le agregan unos (ceros) a la izquierda de su bit de signo, pasando a ser el bit extremo izquierdo el bit de signo *("propagación de signo")*
- La *magnitud* de un número *positivo* (bit de signo **0**) es *directamente* la del número natural que está a la derecha de su bit de signo. En cambio la *magnitud* de un *negativo* (bit de signo 1) se halla haciendo el *complemento al módulo* del binario natural que lo representa, para lo cual se deben invertir todos sus bits (incluído el bit de signo) y sumar uno.
- Para conocer qué número en base diez es un binario con bit de signo uno, el mismo será un número negativo, cuya magnitud X se halla invirtiendo los bits de dicho binario, incluido el bit de signo, y sumando uno. A la magnitud así hallada se le colocarán los "pesos" de cada bit para determinar la magnitud del número decimal negativo buscado.

N.2 Suma de enteros representados con dígito o bit de signo

N2.1 Suma en base diez con dígito de signo. Operatoria y ventajas

Con vista a la suma de binarios con bit de signo, realizaremos primero operaciones con decimales con dígito de signo (antes definidos), o sea con números **naturales** decimales que representan a decimales enteros sin usar los símbolos + y –. Verificaremos que mediante dichos números naturales se pueden sumar los enteros decimales que ellos representan, con menos pasos que los utilizados para sumarlos en el papel de la forma conocida.

Para tal fin realizaremos las siguientes sumas: **a)** (+8) + (+30); **b)** (–8) + (+30); **c)** (+8) + (–30); **d)** (–8) + (–30)

Como hay números de 2 dígitos, más el dígito de signo se necesitan 3 dígitos: +8 = 008; +30 = 030; –8 ≡ 992; –30 ≡ 970

a) +8 = 008
+30 = + 030
038 = +38

En este caso como la representación de cada positivo es el número natural que es la magnitud del mismo con el dígito de signo 0 a su izquierda, en esencia estamos sumando dos números naturales, obteniéndose como resultado otro número natural que también tiene un cero a su izquierda, el cual representa el +38.

b)

Operación		Justificación
–8 ≡ 992	=	= 1000 + (–8)
+30 = + 030	=	= + (+30)
1022	= 1000 + 022	= 1000 + (–8) + (+30)

De esta forma, sumando los números *naturales* 992 y 030 que representan a (– 8) y (+30) se obtiene el número natural 1022. Si descartamos el uno de 1022 queda 022 que representa a +22 = (–8) + (+30), o sea el resultado de la suma pedida.

Por lo tanto, *mediante una suma de naturales se pueden sumar números enteros*, con la simple corrección de descartar el uno fuera de formato, como se justifica a la derecha. Esto es, al hacer 992 + 030 = 1022, puede considerarse que se está haciendo 1000 + (–8) + (+30), siendo que 992 = –8 + 1000 contiene el –8 excedido en 1000 (módulo). Este exceso pasa al resultado 1022, siendo fácil de corregir descartando el uno de 1022. Así no hay necesidad de hacer 1022 – 1000 que implicaría perder tiempo en un paso más.

Obsérvese la ventaja de usar la convención del complemento para representar los negativos en la suma de números con signo contrario. Si tuviéramos que hacer la suma (–8) + (+30) con papel y lápiz, primero tendríamos que determinar cuál de los dos números tiene mayor magnitud (en este caso el +30), luego hacer la resta 30 - 8 = 22 cambiando en este caso el orden de los número dados, y después colocar en el resultado el signo del mayor (+22). En cambio, con esta convención *los números se operan en el orden dado, y no es necesario comparar sus magnitudes, ni efectuar una resta en lugar de una suma, ni colocar en le resultado el signo del mayor*, requiriéndose simplemente el mismo sumador que para naturales.

c)

+8 = 008
–30 ≡ + 970
978 ≡ –22

978 es un número negativo por tener dígito de signo 9 (978 = –X) cuya magnitud X se halla siendo que 978 es el complemento de X (978 + X = 1000), o sea: 978 ≡ –X = –(1000 – 978) = –22 = = (+8) + (–30). Entonces, en el 978 = –22 + 1000 está incluido el –22 excedido en 1000.
Aclaración: 1000 – 978 también puede hacerse 999 – 978 + 1 con el complemento al módulo menos uno.

Por lo tanto, sumando los números *naturales* 008 y 970 que representan a +8 y –30 se obtiene el número natural 978, que representa al –22. A continuación se verifica que al sumar dichos números naturales se suman también (+8) + (–30).

$+8 = 008 = \quad = (+8)$
$-30 \equiv {}^{+}970 = \quad = {}^{+}1000 + (-30)$
$978 = 1000 - 22 = 1000 + (-22) = 1000 + (+8) + (-30)$

Entonces, al hacer 008 + 970 = 978, puede considerarse que se está haciendo 1000 + (+8) + (−30), siendo que 970 = −30 + 1000 contiene el −30 excedido en 1000. Este exceso pasa al 978 = −22 + 1000, **que es el complemento de 22**, sin que se nece-site hacer una corrección como en el caso b); pero para hallar el resultado (−22) se debe proceder como se hizo más arriba Por lo tanto, el exceso 1000 de la representación 970 del −30 está incluido en el resultado 978 = C_{22} que representa el −22.

d)

$-8 \equiv 992$
$-30 \equiv {}^{+}970$
962

$962 \equiv -38$

962 es un número negativo por tener dígito de signo 9 (962 ≡ −X) cuya magnitud X se halla siendo que 962 es el complemento de X (962 + X = 1000), o sea: 962 ≡ −X = −(1000 − 962) = −38 = (−8) + (−30) Entonces, en el 962 = −38 + 1000 está incluido el −38 excedido en 1000.
Nuevamente sumando los números *naturales* 992 y 970 que representan a −8 y −30 se obtiene el número natural 962 = C_{38}, que representa al −38.

A continuación se verifica que al sumar dichos números naturales se suman también (−8) + (−30).

$-8 \equiv 992 = \quad = 1000 + (-8)$
$-30 \equiv {}^{+}970 = \quad = {}^{+}1000 + (-30)$
$962 = 1000 + 962 = 1000 + 1000 + (-38) = 1000 + 1000 + (-8) + (-30)$

Entonces, al hacer 992 + 970 = 1962, puede suponerse que se está haciendo 1000 + 1000 + (−8) + (−30), siendo que 992 = −8 + 1000 contiene el −8 excedido en 1000, y 970 = −30 + 1000 contiene el −30 excedido en 1000. Resulta que al sumar dos negativos (−8 y −30), mediante los números naturales que los representan (992 y 970) puede considerarse que en el resultado (1962) está el negativo (−38) que es el resultado buscado excedido en 1000 dos veces. Puede suponerse como en el caso b), que uno de los excesos es el 1000 de 1962 = 1000 + 962, el cual se resta descartando el uno de 1962. El otro exceso de 1000, como en c), **puede considerarse incluido en el** 962 = C_{38} = −38 + 1000 que representa al −38, siendo que el −38 puede obtenerse a partir del 1962.

N2.2 *Suma de enteros representados por binarios naturales con bit de signo*

Al definir antes la convención usada para representar números binarios enteros ("INTEGERS"), se vió que uno de sus objetivos era poder *sumarlos como binarios naturales*, así los circuitos desarrollados para éstos sirvan también para los primeros.
A los fines didácticos conviene comparar lo que sigue con lo desarrollado antes en N2.1 para enteros decimales.
Realizaremos en binario de 8 bits (formato 8) las siguientes sumas de "Integers" planteadas en decimal:

a) (+8) + (+43); **b)** (−8) + (+43); **c)** (+8) + (−43); **d)** (−8) + (−43) en

a) ***Suma de positivos*** (Con "papel" se quiere significar cómo sería la cuenta usando símbolos + y - en un papel)

	Papel	UAL	
$(+8_D)$	(+1000)	**00001000**	
$+(+43_D)$	+(+101011)	+ **00101011**	32 16 2 1
		00110011 →	+110011 = $+51_D$

Verificación: +51 = (+8) + (+43

En este caso como la representación de cada positivo es el número natural que es la magnitud del mismo con el bit de signo 0 a su izquierda, en esencia estamos sumando dos números naturales, obteniéndose como resultado otro número natural que también tiene un cero a su izquierda, el cual representa el +110011 = $+51_D$. En negrita aparece la cuenta que realiza la UAL.

b) ***Suma de numeros de signo opuesto con resultado positivo***

$-8_D = -1000 = -00001000 \equiv 11110111 + 1 = 11111000$ (conforme a los 4 pasos antes indicados)

$+43_D = +101011 = 00101011$ conforme a lo visto antes para n = 8

	Papel	**UAL**		**Justificación**	
−8 =	−1000 ≡	11111000	= 248	= 100000000 + (−1000)	= 256 + (-8)
+43=	+ 101011 =	+ 00101011	= 256 35	= +(+101011)	= (+43)
	291 =	100100011	= 100000000 + 00100011	= 100000000 + (−1000) + (+101011)	

De esta forma, sumando los números *naturales* 11111000 y 00100011 que representan a (−1000) y (+101011) se obtiene el número natural 100100011. Si descartamos el uno de 100100011, queda 00100011. Una subrutina para visualizar en base diez en pantalla el resultado 00100011, por tener éste bit de signo cero, lo considerará positivo, y su magnitud la determinaría así:

32 2 1
00100011 = +35 **Verificación:** +35 = (−8) + (+43

Por lo tanto, *mediante una suma de binarios naturales se pueden sumar binarioss enteros*, con la simple corrección de descartar el uno fuera de formato, como se justifica a la derecha. Esto es, al hacer 11111000 + 00100011 = 1001000011, puede considerarse que se está haciendo 100000000 + (−1000) + (+101011), siendo que 11111000 = −1000 + 100000000 contiene el −1000 excedido en 100000000 (módulo). Este exceso pasó al resultado 1001000011, siendo fácil de restar descartando el uno extremo izquierdo del mismo.

c) ***Suma de numeros de signo opuesto con resultado negativo***

−43 = −00101011 ≡ 11010100 + 1 = 11010101 (conforme a los 4 pasos antes indicados)

	Papel		UAL		Justificación		
+8 =	+1000	=	00001000	=		= (+ 1000)	= (+8)
−43 =	−101011	≡	+ 11010101	= 213		= + 100000000 + (−101011)	= 256 + (-43)
			11011101	= 100000000 + (−100011)	=	100000000 + (+1000) + (−101011)	
			221	256 35			

De este modo, sumando los números naturales 00001000 y 11010101 que representan a (+1000) y (−101011) resulta el númro natural 11011101 que tamhién, como se ha visto, por empezar con uno puede representar un entero negativo: 11011101 ≡ −X
Su magnitud X se calcula (ver problema inverso) como −X = −(00100010 + 1) = −100011 = −35
Esto es, dado que 11011101 = −X + M = −X + 100000000 resulta X = 100000000 − 11011101, y se calcula como (00100010 + 1).
Entonces, en el 11011101 = −100011 + 100000000 está incluido el −100011 excedido en 100000000. **Verificación:** (+8) + (−43) = −35

Justificación: Al hacer 00001000 + 11010101 = 11011101, puede considerarse que se está haciendo 100000000 + (+1000) + (−101011), siendo que 11010101 = −101011 + 100000000 contiene el −101011 excedido en 100000000.
Este exceso pasa al 11011101 = −100011 + 100000000, **que es el complemento de 100011**, sin que se necesite descartar el uno como en el caso b); pero para hallar el resultado (−100011) se debe proceder como se hizo más arriba.
Por lo tanto, el exceso 100000000 de la representación 11010101 del −101011 está incluido en el resultado 11011101 = C_{100011} que representa el −100011, que también puede considerarse que forma parte del 11011101.

d) ***Suma de negativos***

	Papel		UAL		Justificación	
−8 =	−1000	=	11111000	=		= 100000000 + (-1000)
−43 =	+ −101011	=	+ 11010101	=	11001101	= + 100000000 + (-101011)
			111001101	= 100000000+11001101 = 100000000+100000000+(-110011) = 100000000+100000000+(-1000)+(-101011)		

Este caso combina b) y c), puesto que se descarta el uno fuera de formato y el resultado indicado con una llave (11001101) representa un negativo, por tener bit de signo uno: 11001101 ≡ −X, cuya magnitud X se halla como en el caso c). Esto es:

3216 2 1

11001101 ≡ −X = −(00110010 + 1) = −00110011 = −51 **Verificación:** (−8) + (−43) = −51

Justificación: Al sumar 11111000 + 11010101 = 111001101 (en negrita se distingue el bit de signo), puede suponerse que se está haciendo: 100000000 + 100000000 + (−1000) + (−101011), siendo que 11111000 = −1000 + 100000000 contiene el −1000 excedido en 100000000, y 11010101 = −101011 + 100000000 contiene el −101011 excedido en 100000000. Resulta que al sumar dos negativos (−1000) y (-101011), mediante los números naturales que los representan (11111000 y 11010101) puede considerarse que en el resultado (111001101) está el negativo (−110011) que es el resultado buscado excedido en 100000000 dos veces. Puede suponerse como en el caso b), que uno de los excesos es el 100000000 de 111001101 = 100000000 + 11001101, el cual se corrige descartando el uno de 111001101. El otro exceso de 100000000, como en c), puede considerarse incluido en el 11001101 = C_{110011} = −110011 + 100000000 que representa al -110011 = −51, siendo que el −110011 puede obtenerse a partir del 111001101.

Generalización: como el número binario natural que representa un binario negativo es el complemento de su magnitud, siendo ésta igual al binario negativo excedido en el módulo, resulta que este exceso pasa al resultado de la suma que hace la UAL. Dicho exceso puede aparecer: a) como un **uno fuera de formato** que **se descarta**;
b) como formando parte del resultado si es negativo;
si se suman dos negativos resultan dos excesos, uno de los cuales es **el uno fuera del formato** que **siempre se descarta**, y el otro forma parte del resultado negativo.

Los ejemplos dados permiten confirmar que con la convención del complemento se pueden sumar dos números enteros en el orden que han sido dados, sin necesidad de determinar en caso de tener signo opuesto, cuál tiene mayor magnitud (como se hace con papel y lápiz), ni tener que restar el mayor del menor, siendo que el resultado tiene el signo del mayor.

N.3 Resta de binarios enteros con bit de signo

Los números binarios con bit de signo por ser números *naturales* también se restan como si fueran binarios naturales, por lo que es importante repasar el procedimiento de restar naturales mediante el método de sumar al minuendo el complemento al módulo del sustraendo, tratado en su implementación en el Apéndice I de la Unidad 1 de esta obra, y también más en detalle al comienzo de este complemento, en "Resta de números naturales sin pedir prestado, . . ."

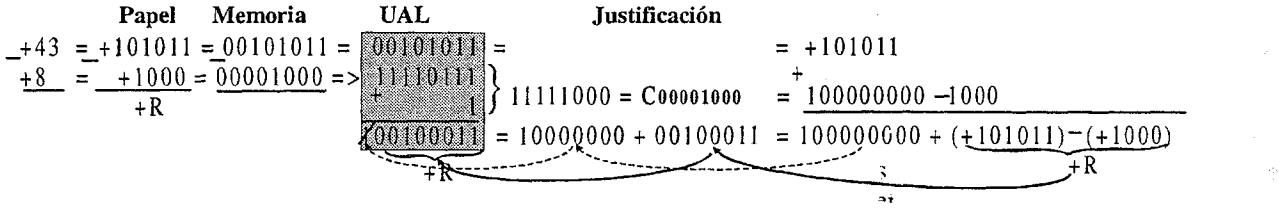
Esto es, cuando la UAL (circuito preparado para sumar o restar binarios naturales) recibe una orden de restar, al minuendo *siempre* le suma el número que resulta de invertir cada bit del sustraendo, y además en la misma operación suma un uno como tercer sumando. Si le llega como sustraendo un número que empieza con un uno, para ella será un número natural, sin la posibilidad de "darse cuenta" que es un número con bit de signo uno, proveniente de complementar anteriormente (en la traducción de decimal entero a binario signado) la magnitud de un número negativo en el formato dado. Igualmente tampoco se "da cuenta" si el minuendo representa un negativo. Eso sólo "lo sabe" el programa que ordenó la resta, y que de ser necesario interpretará el resultado como un número con bit de signo si se lo debe imprimir o visualizar como un número decimal entero.

En lo que sigue, realizaremos distintas restas con los números usados para las sumas de enteros recién vistas.

Cuando se indica "Memoria" se quiere hacer alusión a los números supuestos en memoria que se quiere restar, luego de haber sido traducidos de decimales enteros a binarios con bit de signo, siendo que si un binario que representa a un decimal negativo empieza con bit de signo uno, debe suponerse que antes se invirtieron los bits de la magnitud del binario negativo y que luego se sumó uno. La indicación "UAL" se refiere a la suma que hace la UAL para efectuar la resta de una copia de los números que están en memoria. Vale decir que "Memoria" se refiere a numeros binarios resutantes de una **traducción** de números decimales a binarios hecha (por una subrutina) en un proceso *anterior* a la cuenta que hace la UAL, originada en la ejecución de una instrucción de resta que forma parte de un programa que se está ejecutando *posteriormente* a dicha traducción. La "Justificación" sólo tiene significación teórica. "Papel" se refiere a la resta con los binarios simbolizados con los símbolos + y - con papel y lápiz. Todo lo dicho es importante para entender el proceso.

a) ***Resta de dos positivos***: (+A) – (+B)

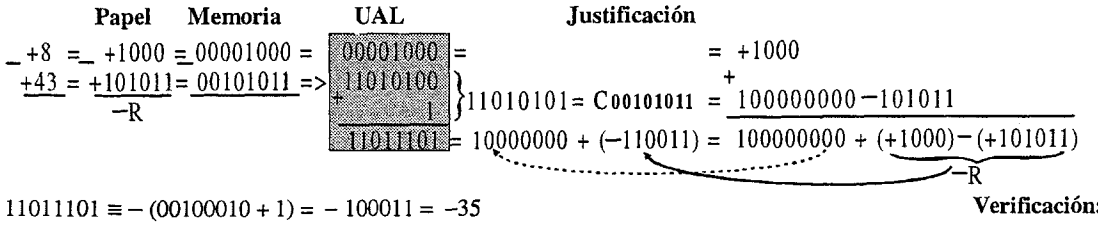
i) Caso en que A > B (Resultado R positivo)



00100011 = +100011 = +35 **Verificación:** (+43) –(+8) = +35

Por lo tanto al hacer la suma en la UAL puede considerarse que se está haciendo la resta de enteros dada excedida en 100000000. Puesto que 100100011 = 100000000 + 00100011 el exceso es el uno que está fuera de formato (flecha punteada)

ii) Caso en que A < B (Resultado R negativo)



11011101 ≡ – (00100010 + 1) = – 100011 = –35 **Verificación:** (+8) –(+43) = –35

De manera genérica: dado que los positivos son los binarios naturales que representan su magnitud con uno o más ceros a su izquierda: +A ≡ A y +B ≡ B, la resta de positivos (+A) – (+B) se deberá realizar como la resta de naturales A – B, que en la UAL se efectuará como A + C_B = A + (M – B) = M + A – B = M + (+A) – (+B) = M ± R donde se ve que la suma de naturales que hace la UAL equivale a hacer la resta de enteros pedida excedida en el módulo M, pudiéndo ser el resultado positivo o negativo.

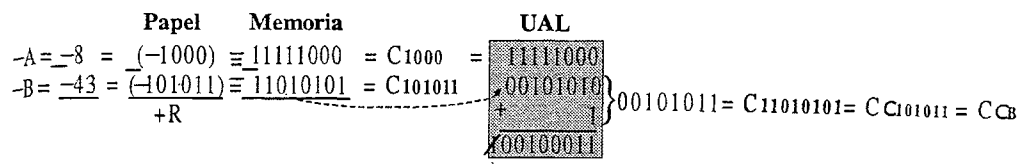
Si en magnitudes A > B (caso i), el resultado será positivo (+R) y se suma a M, por lo que aparecerá un uno fuera de formato, que se descarta para quitar dicho exceso M, dado que M es un uno seguido de tantos ceros como bits tenga el formato.

Si A < B (caso ii) el resultado es negativo (–R), por lo que el resultado M – R = –R + M que arroja la UAL al hacer A + C_B debe ser un número natural que empiece con uno (bit de signo 1), el cual es el complemento de la magnitud de –R. Entonces si se quiere determinar esta magnitud R, debe complementarse el resultado de la UAL como se hizo en el

ejemplo ii). El exceso M (100000000 en nuestro caso) pasó al número natural que empieza con uno, que es el resultado $-R+M$ generado por la UAL, el cual representa a $-R$. ($-R \equiv -R + M$).

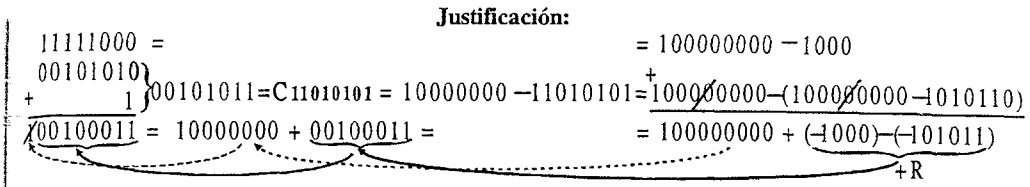
b) ***Resta de negativos***: $(-A) - (-B)$

i) Caso en que $A < B$ (Resultado R positivo)



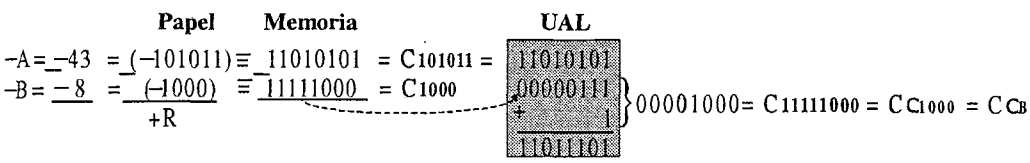
00100011 = +100011 = +35 **Verificación:** $(-8) - (-43) = +35$

Justificación:



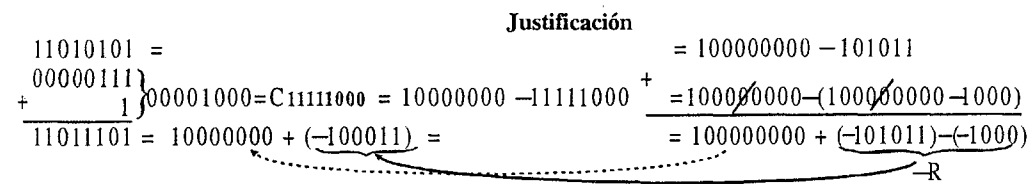
Se ha vuelto a repetir la suma de la UAL y se ha puesto de relieve que 00101010 + 1 = 00101011 = 100000000 – 11010101 es el complemento al módulo del sustraendo 11010101 que entró a la UAL, siendo a su vez 11010101 = 100000000 – 101011 el complemento al módulo de la magnitud 101011del sustraendo (– 101011) de la resta originaria. representado por dicho natural 11010101..En definitiva, 00101011 = B es el complemento del complemento de 101011 = B, habiendo resultado (– 1000) – (– 101011) como (– 1000) + 101011.

ii) Caso en que $A < B$ (Resultado R negativo)



11011101 ≡ – (00100010 + 1) = – 100011 = –35 **Verificación:** $(-43) - (-8) = -35$

Justificación



La justificación es semejante al caso i), siendo que dos excesos se anulan entre sí, y el tercer exceso de 100000000 por ser el resultado negativo de la UAL (11011101) está oculto en el mismo, al igual que el resultado (–100011) de la resta originaria (– 101011) – – (– 1000) el cual se sacó a luz al hacer 11011101 ≡ – (00100010 + 1) = – 100011. En ambos casos se verifica que la resta que hace la UAL de los naturales que representan a los negativos, es la resta de negativos originaria excedida en el módulo 100000000.

En forma genérica: $(-A) - (-B)$ se deberá realizar como la resta de naturales $C_A - C_B = (M - A) - (M - B)$ que en la UAL se efectuará como $C_A + CC_B = (M - A) + [M - (M - B)] = (M - A) + (- B) = M + (- A) + (- B) = M \pm R$ donde se ve que la suma de naturales que hace la UAL equivale a hacer la resta de enteros pedida excedida en el módulo M, pudiéndo ser el resultado positivo o negativo.

Si en magnitudes $A < B$, el resultado será positivo (+R) y se suma a M, por lo que aparecerá un uno fuera de formato, que se descarta para quitar dicho exceso M, dado que M es un uno seguido de tantos ceros como bits tenga el formato.

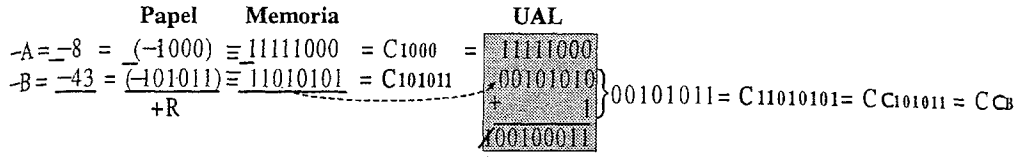
En caso que $A > B$ el resultado es negativo (–R) y el resultado M – R que genera la UAL es el complemento de la magnitud R del resultado, por lo que si se quiere determinar R se debe hallar el complemento del resultado, como se hizo en el caso ii), siendo que el exceso M pasó al número natural que empieza con uno que representa el resultado.

Nuevamente debe ponerse de relieve que la UAL para calcular – (–B) **no hace** en ningún momento "menos por menos más", aunque en definitiva suma la magnitud de B como número positivo, cumpliéndose dicha regla algebraica que usamos para operar con papel y lápiz. Para ver más claro qué ocurre, se debe tener siempre presente el **proceso** que ha tenido lugar **antes** que opere la UAL, la cual en este caso hace una resta cuando una instrucción para restar así lo

ejemplo ii). El exceso M (100000000 en nuestro caso) pasó al número natural que empieza con uno, que es el resultado $-R + M$ generado por la UAL, el cual representa a $-R$. ($-R \equiv -R + M$).

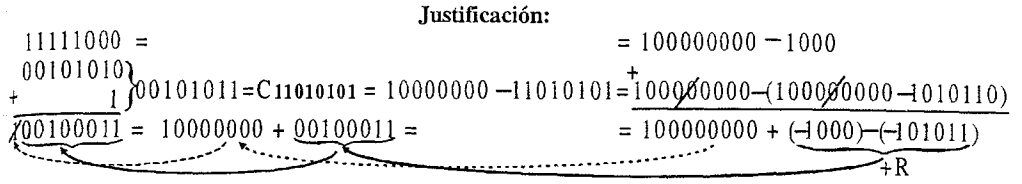
b) ***Resta de negativos***: $(-A) - (-B)$

i) Caso en que $A < B$ (Resultado R positivo)



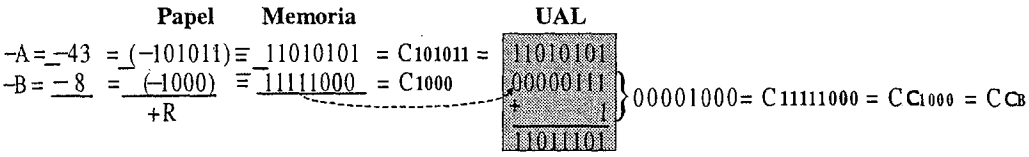
00100011 = +100011 = +35 **Verificación:** (–8) –(–43) = +35

Justificación:



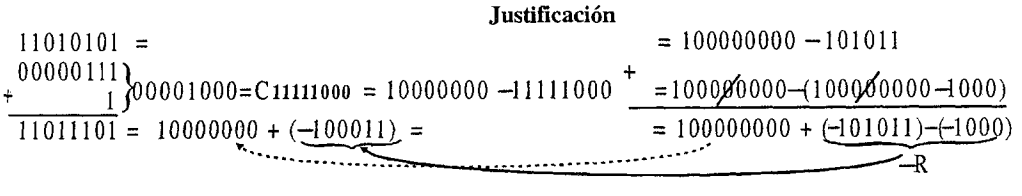
Se ha vuelto a repetir la suma de la UAL y se ha puesto de relieve que 00101010 + 1 = 00101011 = 100000000 – 11010101 es el complemento al módulo del sustraendo 11010101 que entró a la UAL, siendo a su vez 11010101 = 100000000 – 101011 el complemento al módulo de la magnitud 101011del sustraendo (– 101011) de la resta originaria. representado por dicho natural 11010101..En definitiva, 00101011 = B es el complemento del complemento de 101011 = B, habiendo resultado (– 1000) – (– 101011) como (– 1000) + 101011.

ii) Caso en que $A < B$ (Resultado R negativo)



11011101 ≡ – (00100010 + 1) = – 100011 = –35 **Verificación:** (–43) –(–8) = –35

Justificación



La justificación es semejante al caso i), siendo que dos excesos se anulan entre sí, y el tercer exceso de 100000000 por ser el resultado negativo de la UAL (11011101) está oculto en el mismo, al igual que el resultado (–100011) de la resta originaria (– 101011) – – (– 1000) el cual se sacó a luz al hacer 11011101 ≡ – (00100010 + 1) = – 100011. En ambos casos se verifica que la resta que hace la UAL de los naturales que representan a los negativos, es la resta de negativos originaria excedida en el módulo 100000000.

En forma genérica: $(-A) - (-B)$ se deberá realizar como la resta de naturales $C_A - C_B = (M - A) - (M - B)$ que en la UAL se efectuará como $C_A + CC_B = (M - A) + [M - (M - B)] = (M - A) + (- B) = M + (- A) + (- B) = M \pm R$ donde se ve que la suma de naturales que hace la UAL equivale a hacer la resta de enteros pedida excedida en el módulo M, pudiéndo ser el resultado positivo o negativo.

Si en magnitudes $A < B$, el resultado será posltivo (+R) y se suma a M, por lo que aparecerá un uno fuera de formato, que se descarta para quitar dicho exceso M, dado que M es un uno seguido de tantos ceros como bits tenga el formato.

En caso que $A > B$ el resultado es negativo (–R) y el resultado $M - R$ que genera la UAL es el complemento de la magnitud R del resultado, por lo que si se quiere determinar R se debe hallar el complemento del resultado, como se hizo en el caso ii), siendo que el exceso M pasó al número natural que empieza con uno que representa el resultado.

Nuevamente debe ponerse de relieve que la UAL para calcular – (–B) **no hace** en ningún momento "menos por menos más", aunque en definitiva suma la magnitud de B como número positivo, cumpliéndose dicha regla algebraica que usamos para operar con papel y lápiz. Para ver más claro qué ocurre, se debe tener siempre presente el **proceso** que ha tenido lugar **antes** que opere la UAL, la cual en este caso hace una resta cuando una instrucción para restar así lo

ordena. Esta instrucción forma parte de un supuesto programa que se está ejecutando. Pero antes de que éste se ejecute, el mismo debe estar en memoria y también los datos que él procesará durante su ejecución. Esto último supone, por ejemplo, que cuando los datos fueron tipeados en base diez como números enteros, el programa que los traducirá a combinaciones binarias está preparado para traducir cada entero tipeado en el número binario natural que lo representa. Si se tipea en base diez (alto nivel) un entero positivo, quedará **en memoria** traducido como un binario natural (bajo nivel) que empieza con un bit cero; y si se tipea un entero negativo quedará **en memoria** como un binario natural que empieza con bit uno, cuyo valor (incluido dicho uno) es el complemento de la magnitud del negativo tipeado (para obtenerlo, a esta magnitud pasada a binario se le invierte cada bit, y al número así formado se le suma uno, de la forma vista).

El párrafo anterior se refiere a la traducción (por medio de una subrutina para enteros) de enteros en base diez como la que se supone que ha tenido lugar en los ejemplos anteriores, de los números ±8 y ±43 a los correspondientes binarios naturales con bit de signo cero ó uno que los representan en memoria, indicados en la columna "memoria". Esta traducción, se reitera, es realizada por un programa traductor *antes* de la ejecución del programa que procesará estos datos, el cual en el momento de la traducción no se sabe cuándo se ejecutará.

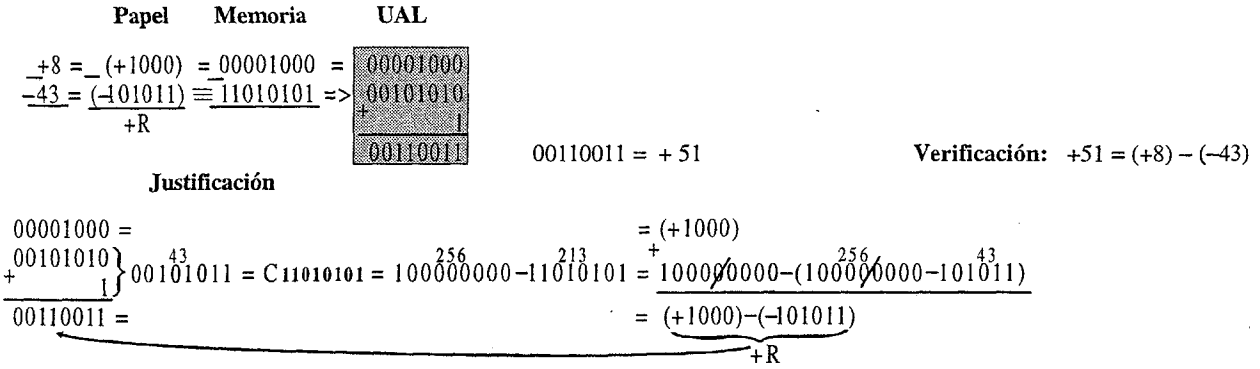
Cuando el programa procesador se ejecute, cada vez que una instrucción del mismo ordena una resta, el hardware de la UCP está proyectado para que la UC ordene a la UAL que haga una resta, para lo cual la UAL sumará al minuendo el complemento al módulo del sustraendo. Por ejemplo, si el sustraendo es 11010101, **la UAL "no sabe" si provino de la traducción del decimal natural** 213, **ó si se originó al traducir el entero** $-43 = -00101011 \equiv 11010100 + 1 = 11010101$.

Ya sea en uno u otro caso le llegará como sustraendo el 11010101, al cual para restarlo invertirá cada bit y sumará uno a fin de complementarlo: 11010101 => 00101010 + 1 = 00101011. Si el programa procesador está trabajando con enteros, debe suponerse que 11010101 provino de la traducción del –43, para lo cual un programa traductor (software) tuvo que haber ordenado la inversión antes indicada para obtener el 11010101 (la UAL no lo sabe), y luego durante la ejecución de una instrucción de resta, la UAL (hardware) realiza una nueva inversión y suma uno, obteniéndose nuevamente 00101011 a partir del 11010101.

Entonces, para hacer por ejemplo (–A) – (–B) = (–8) – (–43) primero un programa traductor representó en memoria (–8) y (–43) como los binarios naturales 11111000 y 11010101; y luego, en otro momento, cuando una instrucción de resta del programa que realiza esa resta ordena realizarla, para concretarla la UAL volvió a invertir los bits de 11010101 y le sumo uno, resultando 00101011, como se hizo en el ejemplo d).

Todo sucede como si en definitiva se hubiese hecho –(–43) = 43, aunque en realidad primero por software se tradujo el (–43) = –00101011 a 11010101, y a posteriori la UAL para hacer la resta indicada lo convirtió en 00101011

c) ***Resta de un positivo menos un negativo*** (no importa la relación de sus magnitudes, el resultado será siempre positivo)

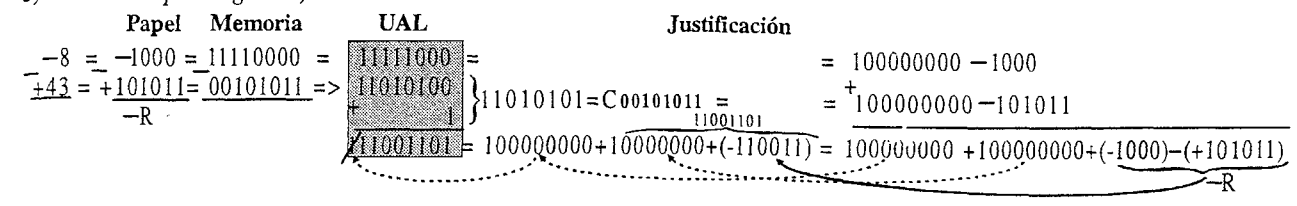


En primer lugar obsérverse que la UAL ha realizado (+1000) – (–101011) como 00001000 + 00101011 o sea en definitiva ha sumado dos positivos, como si algebraicamente se hubiera hecho "menos por menos, más" **aunque un computador jamás hace el producto de dos signos,** sino que siempre la operación (suma o resta) ordenada, y de hecho de la representación de los números resulta como si se hubiese hecho tal producto, como se hace con papel y lápiz.

Justificación de porqué los binarios enteros se restan como si fueran binarios naturales: es muy semejante a la realizada en los casos b) con sustraendo negativo, pero como el minuendo es positivo, no aparece el resultado excedido en 100000000., verificándose que al hacer la suma en la UAL corespondiente a la resta de los binarios naturales que representan a (+1000) – (–101011) también se está haciendo la resta de enteros pedida..

De manera genérica (+A) – (–B) se deberá realizar como la resta de naturales $A - C_B = A - (M - B)$, que en la UAL se efectuará como $A + CC_B = A + M - (M - B) = M - M + A + B = (+A) - (-B)$, donde resulta que la suma de naturales de la UAL equivale a hacer la resta de enteros pedida, siendo M el módulo.

*d) **Resta de un negativo menos un positivo** (result*
~~e) ado siempre negativo)~~



11001101 ≡ − (00110010 + 1) = − 110011 = −51 **Verificación:** −51 = (−8) −(+43)

Justificación de porqué los binarios enteros se restan como si fueran binarios naturales: Al hacer la suma en la UAL corespondiente a la resta de los binarios naturales que representan a (−1000) − (+101011) también se está haciendo la resta de enteros pedida excedida dos veces en 100000000. Uno de los excesos es el uno que se descarta, y el otro está formando parte del número natural que representa el resultado negativo: 11001101 = −110011 + 100000000 y que también "esconde" el −110011 como se determinó más arriba.

De manera genérica: (−A)− (+B) se deberá realizar como la resta de naturales $C_A - B = (M - A) - B$, que en la UAL se efectuará como $C_A + C_B = (M - A) + (M - B) = M + M - A - B = M + M + (-A) - (+B)$ donde se ve que la suma de naturales que hace la UAL equivale a hacer la resta de enteros pedida excedida dos veces en el módulo M.

N.4 Ejercicios

1. Explicar las ventajas de usar el complemento al módulo en la resta que hace la UAL, en lo referente a que el módulo como exceso es fácil de descartar, y a que el complemento al módulo menos uno permite realizar restas sin "pedir prestado".
2. Explicar las ventajas de la representación con la convención del bit de signo y complemento al módulo para los negativos, en comparación con las sumas y restas para enteros realizadas con papel y lápiz, y en relación del uso de un solo sumador para operar con magnitudes y enteros.
3. Dado el número con bit de signo 10111001 ≡ −N obtenido invirtiendo los bits de N y sumando 1, indicar qué operación aritmética se evita realizar operando de esta forma. Explicar por qué no se usa restar 1 y luego invertir los bits para hallar N
4. Representar el número binario con bit de signo 10110010 en formato 16. ¿Qué conclusión resulta ?
5. Suponiendo que el mismo número 10110010 corresponde a una magnitud, representarlo en formato 16.
6. ¿Cuántos bytes hacen falta para representar en binario números enteros en base diez entre −32768 y 32767 ?
7. Para un formato de 4 bytes: ¿cuál es la máxima magnitud que puede representarse, y cuál el mayor número positivo entero? Idem el número negativo de mayor magnitud.? ¿Cómo se podría hacer el cálculo rápidamente en forma aproximada sin usar calculadora.?
8. Realizar en base diez, módulo 100000 las siguientes operaciones en esa base, usando dígito de signo con la convención del complemento al módulo para los negativos.. Verificar que los resultados con dígito de signo coincidan con los que se obtendrían con los efectuados con papel y lápiz.. a) (+96) + (+180; b) (+96) + (−180); c) (−96)+(+180); d) (−96) + (−180)
9. Para el punto b) anterior justificar por qué al hacer la suma de naturales se efectúa la suma de enteros pedida excedida en 1000.
10. Dadas las siguientes sumas algebraicas en base diez, hacerlas en base dos en el menor formato posible de modo que el resultado sea representable en el mismo. Verificar si el resultado binario pasado a base diez concuerda con el esperado en esta base. a) (+18) + (+102), b) (+18) + (−102) c) (−18) + (+ 102) d) (−18) + (−102).
11. Para las sumas b), c) y d) del problema anterior, justificar en binario porqué en cada suma de naturales que hace la UAL está incluída la suma de enteros pedida excedida una o dos veces en el módulo, y dónde aparecen dichos excesos.en el resultado de la UAL.. .
12. Idem problema 10) para (+30) + (+102) y (−30) + (−102). Si no resultara la verificación, explicar a qué se debe, y determinar en qué formato mínimo es posible realizar correctamente cada operación. Verificar que es posible realizarla en formato 16.
13. Efectuar en base dos las siguientes restas algebraicas dadas en base diez, en el menor formato posible. *No es factible hacer "menos por menos = mas", etc.* Verificar si el resultado binario convertido a decimal concuerda con el esperado haciendo las operaciones en base diez. a) (+18) − (+102), b) (+18) − (−102) c) (−18) .− (+ 102) d) (−18) − (−102).
14. Justificar para las 4 sumas de binarios que hizo la UAL para hacer las restas de enteros del problema 13, que al hacer dichas sumas con binarios naturales, también se están realizando dichas restas, pudiendo existir excesos; indicando también donde aparecen los mismos en el resultado de la suma que hizo la UAL..
15. Efectuar en base dos, formato 8, la resta de magnitudes 150 − 37. Verificar si el resultado binario convertido a base diez concuerda con el esperado haciendo las operaciones en esa base. ¿Se podría haber hecho esta misma operación en formato 8 suponiendo que se opere con variables definidas como enteros, o sea: (+150) − (+37).

16. Si en base dos, formato 8, para magnitudes se ordena realizar 37 – 120 verificar qué ocurre con el resultado, y si no es el correcto, a qué se debe ello. Idem si para enteros se ordena restar (+37) – (+120).

17. Suponiendo que la UAL realizó la suma 00001000 + 11010101, indicar: a) qué magnitudes sumadas en base diez, y b) qué números enteros sumados en base diez pueden originar indistintamente dicha suma, y verificar que el resultado de la UAL pasado a base diez está bien en ambos casos.. (Esta cuenta y las siguientes ya se han realizado antes para enteros).

18. Idem para las sumas (11111000 + 00101011) y (11111000 + 11010101).

19. Suponiendo que la UAL realizó una resta mediante la suma 11111000 + 00101010 + 1, indicar: a) qué magnitudes restadas en base diez, y b) qué números enteros restados en base diez pueden originar indistintamente dicha suma en la UAL, y verificar que el resultado de la UAL pasado a base diez está bien en ambos casos

20. Idem para las siguientes sumas hechas por la UAL para restar: (00001000 + 11010100 + 1); (11111000 + 11010100 + 1). (00101011 + 11110111 + 1); (11111000 + 10000100 + 1). (Esta última suma en la UAL no fue ejemplificada anteriormente) .

Respuestas de algunos ejercicios planteados

3 100000000 – 01000111 4 1111111110110010 5 0000000010110010 6 2 bytes

7 La máxima magnitud es 4.294.967.295. El mayor entero positivo es +2.147.483.647 y el más negativo es –2.147.483.648 Por ejemplo, $2^{32} = 2^{10} \times 2^{10} \times 2^{10} \times 2^2 \approx 10^3 \times 10^3 \times 10^3 \times 4 \approx 4000 \times 10^6$ siendo $2^2 = 1024 \approx 1000 = 10^3$

8 a) 00096 + 00180 = 00276 = +276 b) 00096 + 99820 = 99916 ≡ –84 c) 999094 + 00180 = 00084 = +84
d) 99904 + 99280 = 99724 ≡ –276

9

10 a) 00010010 + 01100110 = 01111000 = 120 b) 00010010+10011010 = 10101100 ≡ –84
c) 11101110 + 01100110 ≡ 01010100 = 84 d) 11101110+10011010 ≡ 10001000 ≡ –120 9,

11

12 La verificación no resulta en formato 8 por que la maagnitud del resultado supera la máxima para ese formato.(+127 y –128 respectivamente (situación de "overflow" a tratar). En cambio el resultado concuerda con el valor esperado en formato 16.

13 a) 1010110 ≡ –84 b) 01111000 = 120 c) 10001000 ≡ –120 d)`01010100 = 84

14

15 10010110 + 11011010 + 1 = 101110001; 01110001 = 113 (siempre se descarta el 1 fuera de formato en el resultado de la UAL) (+150) – (+37).no se puede hacer en formato 8 para signados, pues para representar +150 se necesitan por lo menos 9 bits.

16 Con magnitudes no es factible hacer una resta si el minuendo es menor que el sustraendo. El resultado como número natural no tiene sentido. En cambio puede efectuarse la resta de enteros indicada, y su resultado será correcto en formato 8.

17 Cuando la UAL suma dos números como 00001000 + 11010101, ninguno de ellos sufre ningún cambio, como ocurre con el sustraendo en la resta.. a) Si dichos números binarios naturales provienen de la representación de una suma de magnitudes que por ejemplo fue tipeada en base diez, simplemente determinando que número en base diez es cada uno, resulta 00001000 = 8 y 11010101 = 213, siendo que el resultado de la suma en la UAL interpretado como magnitud: 11011101 = 221 es 8 + 213. b) Si dichos binarios naturales representan a enteros en base diez, interpretándolos de la forma vista resulta . **0**0001000 = +8 y **1**1010101 ≡ –(00101010 + 1) = –101011 = –43 , siendo el resultado **1**1011101 ≡ –(00100010 + 1) = –100011 = –35, o sea que la suma de enteros (por ej. tipeados) que produce la misma suma en la UAL (00001000 + 11010101) es (+8) + (–43) = –35

18

19 Si la UAL hizo 11111000 + 00101010 + 1 = A + CB, siendo A = 11111000, es CB = 00101010 + 1 = 00101011, por lo que el sustraendo B que entró a la UAL fue B = 11010100 + 1 = 11010101. Entonces la resta que hizo la UAL fue A – B = 11111000 – – 11010101. a) Si estos binarios naturales representan magnitudes es 11111000 = 248 y 11010101 = 213, y si del resultado de la UAL 100100011, se descarta el uno fuera de formato queda 00100011 = 35. O sea que la resta de magnitudes 248 – 213 = 35 puede originar en la UAL la suma 11111000 + 00101010 + 1. b) Si dichos binarios naturales representan a enteros, interpretándolos de la forma vista es **1**1111000 ≡ –(00000111 + 1) = –1000 = –8 y **1**1010101 ≡ –(00101010 + 1) = –101011 = –43, por lo que la resta de enteros (–8) – (–43) también origina 11111000 + 00101010 + 1 en la UAL, y el resultado de esta suma sin el uno de la izquierda, interpretado como entero es **0**0100011 = + 35 = (–8) – (–43).

20 Si la UAL hizo (00001000 + 11010100 + 1) es CB = 11010100 + 1 = 11010101 por lo que B = 00101010 + 1 = 00101011. Luego fue A – B = 00001000 – 00101011. **a)** 00001000 = 8 y 00101011 = 43 por lo que la resta de magnitudes 8 – 43 no se puede hacer (resultado 11011101 = 221 no tiene sentido en magnitudes). **b)** **0**0001000 = +8 y **0**0101011 = +43, por lo que la resta de enteros (+8) – (+ 43) tamabién origina (00001000 + 11010100 + 1) en la UAL, y el resultado de esta suma interpretada como entero es 11011101 ≡ – (00100010 + 1) = – 100011 = –35 = (+8) – (+ 43) .
Si la UAL hizo (11111000 + 10000100 + 1). es CB = 10000101, por lo que B = 01111011, o sea A – B = 11111000 – 01111011.

a) Para magnitudes 11111000 – 01111011 = 248 – 12 = 125 = 01111101 (resultado de la UAL descartando el uno izquierdo)

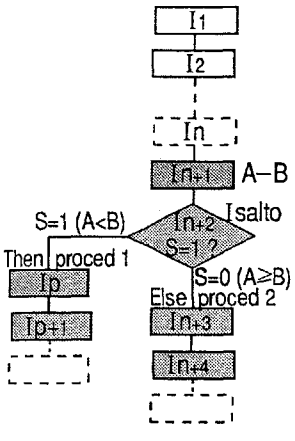
b) Para enteros **1**1111000 ≡ – (00000111 + 1) = – 1000 = –8; **0**1111011 = +125; por lo que la resta (–8) – (+ 125) = – 133 también origina (11111000 + 10000100 + 1)., pero el resultado **0**1111101 = + 125 no tiene sentido para enteros pues el verdadero resultado (– 133) supera el máximo – 128 representable en formato 8 (situación de overflow a tratar).

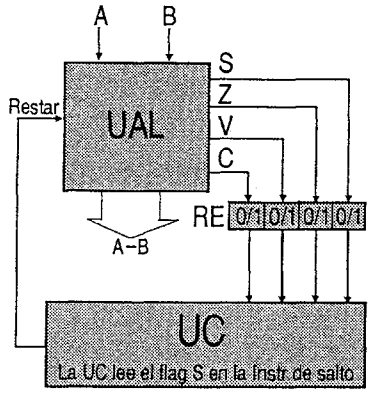
N.5 Indicadores de estado SZVC ("flags")

Su importancia: En el capítulo 1.9, figura 1.35 de la "PC por Dentro" se muestra el mecanismo de las instrucciones de salto. Ellas también se tratan en detalle y aparecen formando parte de secuencias de instrucciones en la Unidad "Assembler desde Cero". Supongamos que en un lenguaje de alto nivel formando parte de un programa para procesar números enteros aparece por ejemplo una estructura como:

"ENTEROS"
.....
IF +A < +B **THEN** hacer "procedimienato 1"
ELSE hacer "procedimiento 2"
.....

El diagrama conceptual que sigue permite apreciar, en primera instancia, cómo un procesador podría llevar a cabo con las instrucciones que él puede ejecutar lo que ordena la estructura IF. (En grisado se indica la traducción de esta estructura)





En el mismo se supone que las instrucciones de máquina **I1** a **In** que el procesador ha ejecutado o ejecutará son la traducción de las líneas de programa anteriores al IF, simbolizadas con puntos suspensivos, al igual que las que le siguen. A fin de cumplimentar el IF, para determinar si +A < +B deberá existir una instrucción supuesta **In+1** que ordene la resta A – B. Si el resultado de ésta es negativo (bit de signo uno) *es suficiente para saber que* +A < +B **sin que importe el valor concreto del resultado.**
La instrucción de salto siguiente In+2 encerrada en un rombo, pregunta si el bit de signo designado **S** del resultado anterior vale 1.

Si S=1 ordena saltar a la instrucción **Ip** de modo que el programa a ejecutar prosiga en la secuencia de instrucciones del "procedimiento 1", o sea la alternativa THEN (para lo cual hará que el registro IP tome la direccón donde se halla Ip).
En caso de ser S=0, por ser el resultado de la resta anterior un número positivo (lo cual implica que +A ≥ +B), el programa sigue por la instrucción **In+3**, primera de la secuencia del "procedimiento 2" correspondiente a la alternativa ELSE.

Por lo tanto, la instrucción **de salto** en esencia está pensada para que **la máquina tome la decisión de elegir entre** dos secuencias (caminos) alternativas que esta instrucción plantea **en función de un resultado anterior**, resumido en este ejemplo en el valor del bit de signo de dicho resultado. *Se comprende que esta instrucción es fundamental para que la máquina pueda pasar automáticamente, sin intervención humana, de una secuencia a otra según un resultado alcanzado en su interior.*

Naturalmente la UAL es la encargada de indicar si el bit de signo S del resultado de la resta que efectuó vale 0 ó 1. El indicador de signo o "**flag S**" es generado al mismo tiempo que dicho resultado, y tiene un valor que es simplemente una copia del valoar del último bit de la izquierda del formato de **n** bits o posiciones del resultado.
Junto con el flag **S** En cada operación que realiza, la UAL genera junto con el flag S otros indicadores acerca del resultado. Entre ellos ahora nos interesa: el indicador de resultado cero (**flag Z**), cuyo valor 1 ó 0 dice si el resultado fue cero o no; el indicador de desborde u "overflow" (**flag V**) cuyo valor 0 ó 1 indica si el resultado de una suma de *enteros* entró o desbordó el formato de la UAL, y el **flag C** de acarreo o "carry", cuyo valor 1 ó 0 en cada suma que realiza la UAL coincide con el valor del transporte hacia la posición n+1 fuera del formato de **n** bits o posiciones de la UAL, siendo que en cada resta de la UAL el valor de **C** es opuesto al valor de dicho transporte.

Los valores de los flags se guardan en el "Registro de Estado" (**RE**) de la UCP. Cada vez que la UC debe ejecutar una instrucción de salto, según por cuál(es) flag(s) pregunta esa instrucción, leerá su(s) valor(es) en el RE a fin de decidir si se salta a ejecutar otra instrucción (cuya dirección permite determinar la instrucción de salto), o si pasa ejecutar la instrucción que sigue a la instrucción de salto, como se representa en el diagrama visto más atrás.

En lo que sigue trataremos el significado y utilidad de cada flag, y ejemplificaremos cómo se determina su valor en *sumas y restas realizadas anteriormente.*

Como se verá, los valores de los flags **S**, **V** y **Z** intervienen en instrucciones de salto de programas que procesan números **enteros**, mientras que los valores de los flags **C** y **Z** corresponden a instrucciones de salto de programas que operan **magnitudes**. Por el valor del flag **Z** pueden preguntar instrucciones de salto de uno u otro tipo de programas.

Indicador S de signo para enteros (Flag S):

Si un programa interpreta el resultado de la UAL como un número natural *que representa a un número entero*, el valor de **S** es el valor del bit de signo del resultado:
S=0 si el resultado es positivo por presentar bit de signo 0.
S=1 si el resultado es negativo por presentar bit de signo 1.

Si la UAL suma o resta dos números de *n* bits, el valor de **S** es pues el valor del bit extremo izquierdo del resultado en el formato de *n* bits, o sea el bit del resultado que *está en la misma posición que los dos bits de signo de los sumandos.*

```
        ← n →
 −8 = + 11111000
+43 =   00101011
      100100011 → R = 00100011 = +35
      ⇓
      S=0  C=1 Z=0 V=0
```

```
 −8 = − 11111000
+43 =   00101011
```

```
        ← n →
        11111000
      + 11010100
          ------ 1
      111001101 → R = 11001101 ≡ −51
      ⇓
      S=1   C=0 Z=0 V=0
```

Junto con el flag S se dan los valores de los flags C, Z y V, a fin de que estos ejemplos sirvan también en la medida que definamos, y para poner en evidencia que **siempre la UAL genera los cuatro flags sim**ultáneamente, dado que "no sabe" si el programa que se está ejecutando es para enteros o para magnitudes.

Indicador C de acarreo ("carry") sólo para una suma de naturales (Flag C en la suma):

Cada vez que a una UAL –que opera con *n* bits o posiciones– se le ordena una suma (siendo que también suma cuando se le ordena una resta) el flag **C** es el acarreo de valor 1 ó 0 a la posición n+1, suponiendo que la suma continúa.

```
        ← n →
248 = + 11111000
 43 =   00101011
      100100011 →  100100011 = 291
      ⇓
      C=1      S=0 Z=0 V=0
```

```
        ← n →
  8 = + 00001000
213 =   11010101
      011011101 →  11011101 = 221
      ⇓
      C=0      S=1 Z=0 V=0
```

Ex profeso se ha elegido la misma cuenta en la UAL ejemplificada para el flag S, a los fines que se verifique que si un programa para magnitudes ordena sumar 248 + 43, cuando una instrucción ordene esta suma, la UAL realizará la misma suma que si otro programa para enteros ordenara (−8) + (+ 43), dado que la representación binaria del 248 es la misma que la del −8 , y la del 43 es idéntica a la del +43, entero positivo de magnitud 43.

Una aplicación del flag **C** (tratado en un ejercicio de la Unidad 3 de esta obra) se da cuando se quiere sumar en una UAL para 8 bits dos números de 16 bits. Primero se suman los 8 bits de la mitad derecha de dichos números, y luego los 8 de la mitad izquierda **más** el acarreo de la mitad derecha hacia la izquierda (valor 1 ó 0 del flag **C**).

Otra aplicación corriente se da en programas que operan con magnitudes. Cuando ordenan una suma, si **C**=1, por tratarse de un uno fuera del formato de la UAL, en general se considera **erróneo el resultado**. El valor del flag **C** se guarda en el Registro de Estado, y por lo tanto puede conocerse su valor, y asi formar el resultado correcto en 2 bytes de memoria. Pero esto no es muy práctico, por lo que si **C**=1 y el programa no lo considera, el resultado se supone erróneo, pues le falta un bit. Esto es, si en la suma anterior consideramos el uno fuera de formato, 100100011 = 291 es el resultado correcto, siendo que *cualquier suma de naturales que haaga la UAL siempre está bien si se tiene en cuenta el uno que está fuera de formato* (CUAL). Obviamente, si dicho uno el programa no lo toma en cuenta, el resultado 00100011 está mal.

Es importante subrayar que la UAL es un circuito, que "no sabe" si está sumando naturales que son magnitudes por que originariamente se tipeó 248 + 43, o naturales que representan a enteros, porque se tipeó (−8) + (+ 43) [3] eomo en el ejemplo c) anterior de la suma de enteros de signo opuesto. Puesto **que la UAL hace la misma suma en ambos ejemplos**, los valores de los flags son los mismos. En esencia la UAL siempre suma (y resta) naturales, ya sea que representen magnitudes o enteros. Por tal motivo para la misma cuenta han sido determinados los flags para enteros y naturales, como ocurre en la realidad, pues la UAL está proyectada para generar ambos tipos, a fin de que pueda ser usada para los programas que operan con enteros y los que operan con naturales.
A su vez los programas interpretan un resultado de una u otra forma. Un programa para enteros está preparado para interpretar 11001101 como un negatvo. En caso de tener que convertir este número a decimal (para mostrarlo en

[3] Lo anterior implica que las instrucciones para sumar y restar números naturales o enteros *son las mismas* Para el caso de definirse datos "reales", existen instrucciones para operar aritméticamente en punto flotante, siendo el coprocesador matemático el encargado de realizarlas, y no la UAL.

pantalla o imprimirlo) al detectar que el número empieza con el bit 1, generará el código ASCII del signo menos. Luego calculará la magnitud binaria del mismo hallando su complemento al módulo, y así determinará qué número decimal es. Conocido éste generará el código ASCII de cada uno de los dígitos que lo componen, de modo que puedan aparecer en pantalla o papel, formando el número –51.

Un programa para magnitudes interpretará 11001101como 205 asignando directamente los pesos 128, 64,, 32, etc.

También *ambos programas diferirán en las instrucciones de salto involucradas, pues son distintos para enteros que para naturales.* No debe perderse nunca de vista que el tipo de datos definidos en Pascal, C, Basic, etc (integers , magnitudes, reales, etc) determina las instrucciones que el programa compilador deja traducidas en código de máquina.

En síntesis:

La UAL en esencia opera con naturales. Pero como los binarios con bit de signo (provenientes de enteros) en la UAL se suman como naturales, y a fin de que las instrucciones de salto de los programas para números enteros puedan emplearse, de los cuatro indicadores citados **SZVC** –que genera la UAL luego de cada operación aritmética– **SVZ** se usan para enteros, y **CZ** para naturales, como se verá. **CZ** ó **SVZ** pueden usarse solos o combinados. La UAL "no sabe" si el programa en ejecución es para enteros o naturales, y tanto la suma o la resta de ambos tipos de números *las realiza de igual forma*. Luego de cada operación que efectúa, la UAL genera **SZVC**, y son las instrucciones de salto de los programas que se ejecutan, las que preguntan por el valor (0 ó 1), de **SVZ** y **CZ** según sea para enteros o naturales. Ejemplos de utilización aparecen en las secuencias de instrucciones de la Unidad 3.

Indicador C sólo para la resta de binarios naturales (Flag C en la resta):

En una UAL de formato **n** si se suman dos naturales, el valor de **C** es "lo que me llevo" si la suma continúa en la posición n+1. En la resta de naturales debe interpretarse que el valor de C es "lo que pido prestado" a la posición n+1 en la *resta originaria* de dos números naturales de **n** bits (**no** en la suma que hace UAL para restar) si la resta continuara. De esta forma, *el flag C sirve en la resta de naturales para determinar si el minuendo es mayor o menor que el sustraendo, sin importar el valor concreto del resultado, ni el orden en que se restan los números naturales.*

Ejemplificaremos primero para restas de naturales en base diez una nueva forma de usar "lo que pido prestado", para comparar naturales:

"**0**" <= _ 8520	"**1**" <= _ ^{1}2831
2831	8520
5689	4311

Si la primer resta continuara hacia la izquierda del formato de 4 dígitos, se "pediría **0** " a un supuesto quinto dígito a la izquierda del minuendo, lo cual es suficiente para afirmar que 8520 ≥ 2831 sin que interese el resultado correcto 5689. Del mismo modo en la segunda resta si ésta continuara se "pediría "**1**" para que el 2 sea 12, lo cual es suficiente para afirmar que 2831 < 8520 aunque el reultado 4311 no interesa ni tiene sentido, pues para restar naturales el minuendo debe ser mayor que el sustraendo.

El mismo criterio puede seguirse para binarios naturales, pero como la UAL no resta "pidiendo prestado" como se hace con papel y lápiz, sino que las restas se realizan mediante sumas en la UAL, vamos a ver la relación existente entre "lo que pido" si la resta contínúa, con "lo que me llevo" en la suma que se hace para restar, que para diferenciarlo del flag **C** antes definido para la suma que se hace para sumar, lo designaremos CUAL.

El valor del flag C en una resta de binarios naturales A – B será contrario a CUAL, y significará:

a) Cuando se "pide uno" prestado (**C=1**) si la resta continuara a la izquierda implica que **A < B**.
b) Cuando se "pide cero" prestado (**C=0**) si la resta continuara a la izquierda implica que **A ≥ B**.

Efectuaremos las mismas restas antes realizadas en N.3 con los naturales 11111000 – 00101011 y 00001000 – 11010101 que representaban las restas de enteros (–8) – (+43) y (+8) – (–43) respectivamente, y que ahora que dichos naturales representan magnitudes, son las restas 248 – 43 y 8 – 213, respectivamente. Como puede verificarse, las operaciones en la UAL son idénticas, por lo que volvemos a afirmar que la UAL "no sabe" si la resta que está haciendo se generó en una resta de enteros o en una de magnitudes, sólo está proyectada para que cuando reciba la orden de restar, invierta los bits del sustraendo (empiece éste con 1 ó 0) y sume uno a fin de calcular su complemento al módulo, con lo cual el resultado estará excedido en el módulo, exceso que si A ≥ B aparecerá en la UAL como un uno fuera de formato CUAL

Dado que la UAL resta a naturales que representan enteros o a naturales que representan magnitudes de la misma forma indicada, sin saber si se trata de uno u otro tipo de representación (el único que lo sabe es el programa que se está ejecutando), toda vez que CUAL =1, **este uno fuera de formato siempre debe descartarse, sea para naturales o para enteros.**

	Resta originaria	UAL
	← n →	← n →
_ 248	"0" <=_ 11111000	11111000
43	00101011	+ 11010100
		1
		111001101
		⇓
		CUAL=1 → **C=0** S=1 Z=0 V=0

	Resta originaria	UAL
	← n →	← n →
_ 8	"1" <=_00001000	00001000
213	11010101	+ 11010100
		1
		011011101
		⇓
		CUAL=0 → **C=1** S=1 Z=0 V=0

Conforme se vió para las restas de naturales, cuando hay un uno fuera de formato debe descartarse (*sean estos naturales que representan enteros o magnitudes*, por lo que el resultado de la primer resta será 11001101 = 205 = 248 − 43; y por ser **C**=0 implica que 248 ≥ 43, pues si la cuenta sigue hacia la izquierda se pidiría "0 prestado".

El resultado de la otra resta 11011101 = 221 no tiene sentido para naturales, pues al ordenar hacer 8 − 213 se ha intentado restar un número menor de otro mayor. Aunque así sea, puede establecerse que por ser **C**=1 implica que 8 < 213 (suponiendo que no se conozcan esos valores), pues se pediría "1 prestado" si la cuenta continuara hacia la izquierda.

En relación con el valor adjudicado al flag **C** se verifica que es *contrario* al valor de CUAL , por lo que de esta forma con el valor de CUAL obtenido de la suma que hace la UAL se conocerá si "pido 0 ó 1" en la resta originaria.

Demostraremos que *siempre* que la UAL hace una resta, lo que "se pide" si ella siguierra es contrario al valor de CUAL. Para ello, según se trató en la resta de naturales, para hacer A − B la UAL hace A más el complemento al módulo de B:

A + CB = A + (M − B) = M + (A − B) = 100 . . . 0 + (A − B) , dado que M para un formato **n**, es un uno seguida de **n** ceros.

Si **A ≥ B** ("**pido 0**"), es (A − B) = X ≥ 0; entonces 100 . . . 0 + (A − B) = 100 . . . 0 + X por lo que si a 100 . . . 0 se le suma X de n bits, , el uno de 100 . . . 0 se mantendrá, correspondiendo en la UAL dicho uno a **CUAL = 1**.

Si **A < B** ("**pido 1**") puede escribirse (A − B) = −(B − A); entonces 100 . . . 0 + (A − B) = 100 . . . 0 − (B − A), por lo que si a 100 . . . 0 se le resta (B − A) resultará un número sin el uno de 100 . . . 0 por lo que en la UAL será **CUAL= 0**.

La resta 248 − 43 es útil para mostrar que **el flag S no sirve para naturales**. Podría pensarse equivocadamente que si se restan dos naturales y el resultado es negativo, implica A < B. Sin embargo dicha resta arrojó S=1 siendo **248 > 43**

Para las dos restas que hizo la UAL mediante una suma, junto con el flag C se dan los valores de los flags S, Z y V, a fin de que estos ejemplos sirvan también cuando definamos Z y V; siendo además que **siempre la UAL genera los cuatro flags si**multáneamente, dado que "no sabe" si el programa que se está ejecutando es para magnitudes o para enteros.

Indicador Z de resultado cero (Flag Z para enteros y magnitudes):

Será **Z**=1 (zero "**SI**") sólo si en la suma o resta de dos número de **n** bits en la UAL los coreespondientes **n** bits del resultado son **todos** ceros. O sea un resultado R fue cero (R=0) si **Z**=1 (cero "**SI**").
Si como ocurre en la mayoría de los casos, el resultado **no** es cero, será **Z=0** (cero "**NO**").

Entonces, **Z=1** significa *"si, es cero"* o es "cierto que es cero"; y **Z=0** *"no es cero"* ó "es falso que es cero".
Dado que el número cero se representa igual para naturales o signados, el indicador **Z** puede usarse indistintamente para detectar resultado cero en ambos tipos de números.

+ (+43) = + 00101011	+ 0 = + 00000000	_43 = _ 00101011
(−43) = 11010101	0 = 00000000	43 = 00101011
1 00000000	**0 00000000**	+ 11010100
←R=0 →	←R=0 →	1
		1 00000000
		← R=0 →
Z=1 C=1 S=0 V=1	**Z=1 C=0 S=0 V=1**	**Z=1 C=0 S=0 V=1**

La suma en la UAL de (+43) + (−43) sería la misma que si se originara por los naturales 43 + 213, pues en ambos casos al sumar los dos números naturales que los representan se obatiene el módulo, pues son complementarios.
El resultado cero interpretado como representación de un entero forma parte de los positivos, acorde con lo cual **S**=0.
En la resta es C=0 pues CUAL =1. En los ejemplos dados al definir S y C resultó siempre **Z**=0, pues resultado es "no cero".

Indicador **V** *de overflow = desborde en un resultado entre enteros (Flag V sólo para enteros):*

Sea una suma o resta **de binarios naturales que reprsentan números enteros** realizada en la UAL:

a) Será **V**=1 (overflow "**SI**") si ambos números **en** la **UAL** tienen igual bit de signo, y el resultado tiene bit de signo contrario. Esto es, basta detectar en la UAL (que también suma para restar) que la suma de dos positivos (bits de signo 0) da por resultado un número negativo (bit de signo 1), o que de la suma de dos negativos (bits de signo 1) resulta un número positivo (bit de signo 0) para determinar que el resultado está mal por desborde (**V**=1) por que supera los máximos $+Nmax = 2^{n-1} - 1$ y $-Nmax = -2^{n-1}$ antes determinados (sección N1.3).

b)

b1) Será **V**=0 (overflow "**NO**") si ambos números en la UAL tienen igual bit de signo, y el resultado tiene el mismo signo que ellos. Esto es, si la suma de dos positivos da por resultado un positivo, o si la suma de dos negativos da por resultado un negativo.

b2) También será **V**=0 siempre que se suman dos naturales que representan enteros de signo opuesto, sin que importe el signo del resultado. Esto es así, dado que *si sumamos dos enteros de signo contrario en definitiva se realiza una resta de sus magnitudes*, cuyo resultado no puede superar al operando de mayor magnitud.

En las determinaciones anteriores del valor de **V** se supone que los operandos se pueden representar en el formato (un positivo debe estar representado por un natural con un bit izquierdo cero, y un negativo con un natural con bit uno, siendo que el overflow (no ó si) se refiere a si el resultado se puede representar o no en el formato de la UAL.

Puesto que el valor de **V** se determina en función de los bits de signo de los naturales que representan a enteros, el flag V *es un flag exclusivo para enteros.*

Obsérvese que la UAL no pierde tiempo en calcular si el resultado supera $|+Nmax|$ ó $|-Nmin|$. Simplemenla detecta overflow en función de los bits de signo de los operandos y del resultado.

Las operaciones siguientes muestran que *no existe ninguna relación entre el flag C de naturales y el flag V* **recién definido sólo para enteros**, dado que puede ser **V**=1 y **C**=0, así como **V**=1 y **C**=1.
Existe una tendencia, que no se condice con la definición dada para overflow, de designar con esta palabra el caso en que en una suma de magnitudes sea **C**=1. Si bien implica un uno fuera de formato, el resultado de dicha suma estaría bien si este uno es considerado por un programa para magnitudes. Pero ocurrre en la práctica, como se indicó, que tales programas no lo consideran, por lo que interpretan **C**=1 como resultado erróneo. En una suma de enteros el valor de **C** no interesa.

+80 = +**0**1010000 = 80	−80 = +**1**0101000 = 176	_ −80 = _ 10101000
+90 = **0**1011010 = 90	−90 = **1**0100110 = 166	+90 = 01011010
010101010	101001110	
V= 1 **C**=0 **S**=1 **Z**=0	**V**= 1 **C**=1 **S**=0 **Z**=0	**V**= 1 **C**=0 **S**=0 **Z**=0

```
  10101000
+ 10100110
          1
 101001110
```

Interpretando los binarios naturales que reprresentan los enteros positivos (+80) y (+90) resulta que la suma de dos positivos dió por resultado un negativo (S=1) lo cual es suficiente para afirmar que el resultado es erróneo. Se ha intentado sumar dos positivos –siendo que cada uno es representable en formato 8– pero el resultado que debía haber sido +170 supera a +Nmax = +127 en ese formato. Obsérvese que para representar el +170 se necesitan 9 bits (+170 = **0**10101010), así este positivo tiene un cero como bit de signo., y si supuestamente se hubiera operado con los operandos representado en 9 bits (en una UAL serían 16) sería **V**=0, como puede verficarse. De esto se deduce que **el overflow es un error por truncamiento**, puesto que en un formato mayor no ocurriría, siendo que los bits del resultado en el formato dado están bien.
Ahora bien, la suma anterior de enteros también pudo haberse originado de la suma de magnitudes 80 + 90 para la cual el resultado 10101010 = 170 es correcto, interpretado como número natural por un programa para magnitudes. En cambio un programa para enteros dado que 10101010 empieza con uno, lo interpretaría como −(01010101 + 1) = −01010110 = −86, resultado que no se condice con la suma de enteros ordenada (+80) + (+90) = +170. Por tal motivo, este programa luego de que una instrucción ordene una suma o resta, debe seguirle una instrucción de salto que pregunte si **V**=1, de modo que si es cierto salte a un procedimiento preparado al respecto, y que en caso de ser **V**=0 el programa siga normalmente con la instrucción siguiente a la de salto. Esta alternativa es semejante a la indicada en el diagrama anterior (rombo para **S**=1).

Igualmente en la suma (−80) + (−90), resulta que la suma de dos negativos dió por resultado un positivo (S=0) lo cual basta para afirmar que el resultado es erróneo. Este, que debía haber sido −170 supera a −Nmax = −128 en formato 8. Para representar el −170 se necesitan 9 bits (−170 ≡ **1**01010110), así este negativo tiene un uno como bit de signo.
Un programa para enteros interpretaría 01001110 como + 78 por empezar con cero, resultado obviamente errado.

En la resta (−80) − (+90) = −170 obviamente, como en la suma anterior, vuelve a ser **V**=1, y ello puede ocurrir en toda suma que haga la UAL para restar, si en ella los operandos son de igual signo y el resultado es de signo contrario.

Determinación si un entero es mayor o menor que otro aunque el resultado de la resta sea errado, usando S y V

Aunque la resta (−80) − (+90) arrojó V=1 y S=0 (**V ≠ S**), igualmente puede determinarse si A ≷ B dado que S=0 implica que en la UAL se sumaron dos negativos y el resultado fue positivo. Por lo tanto, **debió haber sido S**=1, o sea que a pesar de que se sabe que el resultado de la resta está errado (**V**=1), podemos afirmar que el mismo fue un número negativo, con lo cual se deduce que es A < B. Efectivamente, se verifica que (−80) < (+90).

Del mismo modo, si se hace (+80) − (−90) = +170 la UAL arrojaría V=1 y S=1 (**V = S**), lo cual implica que en ella se sumaron dos positivos y el resultado fue negativo. Por lo tanto, **debió haber sido** S=0, o sea que a pesar de que se sabe que el resultado de la resta está errado (V=1), podemos afirmar que el mismo fue un número positivo, con lo cual se deduce que es A ≥ B. Efectivamente, se verifica que (+80) ≥ (−90).

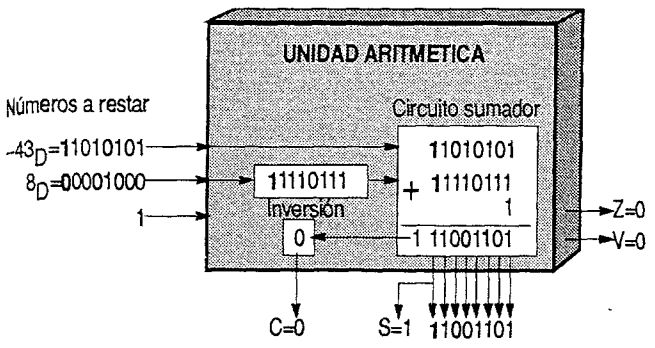
Anteriormente se vió que si el resultado de una resta de enteros arroja **V**=0 y S=1 (**V ≠ S**), como el resultado está bien y es negativo, impllica que minuendo < sustraendo, y que si en una resta de enteros resulta **V**=0 y **S**=0 (**V = S**), dado que el resultado está bien y es positivo, se infiere que minuendo ≥ sustraendo.

Por lo tanto, comparando los 4 casos descriptos, resulta, independientemente que el resultado esté bien o mal, que:
si **V = S** será siempre minuendo ≥ sustraendo.
si **V ≠ S** será siempre minuendo < sustraendo.

Cuadro final de interpretación de los flags en la suma y en la resta:

Suma de enteros: (±A) + (±B) = (±R)	**Suma de naturales (magnitudes):** A + B = R
V=0 Resultado correcto (R = OK) si **S**=0 resultado positivo (+ R). si **S**=1 resultado negativo (− R). **V**=1 Resultado erróneo (R = ε). **Z**=1 implica resultado cero (R=0) **Z**=0 implica resultado no cero (R≠0) El flag **C** no es para enteros, y su valor siempre se descarta	**C**=0 ("llevo 0" si suma continúa) Resultado correcto en la UAL y para un programa que opere con magnitudes. **C**=1 Resultado correcto en la UAL si un programa considera **C**=1 formando parte del resultado de la UAL Resultado incorrecto si dicho programa no considera **C**=1 como parte del resultado de la UAL(caso típico) También implica que "llevo 1" si la suma continuaría. **Z**=1 implica resultado cero; **Z**=1 implica resultado no cero
Resta de enteros: (±A) − (±B) = (±R)	**Resta de naturales (magnitudes):** A − B = R
V=0 es R = OK si **S**=0 resultado es + R → minuendo ≥ sustraendo si **S**=1 resultado es − R → minuendo < sustraendo **V**=1 es R = ε si **S**=0 (debió ser **S**=1) → minuendo < sustraendo si **S**=1 (debió ser **S**=0) → minuendo ≥ sustraendo En general: si **V = S** → minuendo ≥ sustraendo si **V ≠ S** → minuendo < sustraendo **Z**=1 (R=0) → minuendo = sustraendo **Z**=0 (R≠0) → minuendo ≠ sustraendo El valor de **C** no interesa para enteros y *nunca* puede formar parte del resultado de la UAL	El valor de **C** *nunca* puede formar parte del resultado de la UAL, pero sirve para conocer si A ≷ B (incluso si R = ε). **C**=0 ("pido 0 prestado" si resta originaria continúa) entonces A ≥ B, por lo que R = OK en la UAL **C**=1 ("pido 1 prestado" si resta originaria continúa) entonces A < B, por lo que R = ε para naturales. El flag **S** de enteros no sirve para determinar si A ≷ B El flag **V** de enteros no es indicativo de nada para naturales **Z**=1 (R=0) → A = B **Z**=0 (R≠0) → A ≠ B

Esquema de una resta y generación de flags en la UAL



A modo de esquema referencial, a la izquierda aparecen los tres números que entran a la UAL en una resta, (incluido el uno que se suma a los bits invertidos del sustraendo), y las operaciones que tienen lugar en su interior.
También se ejemplifica cómo se determinan los valores de los flags, y en particular la inversión de CUAL que tiene lugar en toda resta para generar el valor del flag **C**.

Cuando la UAL suma, no se produce la inversión del segundo número (sustraendo en una resta), ni de CUAL dado que en la suma **C** = CUAL; y el tercer número que entra es 0 en vez de 1, usado en la resta para sumarle uno a los bits invertidos del sustraendo.

15.1 EJERCICIOS SOBRE FLAGS

. Indicar dónde se guardan los flags y cuál es su utilidad

'. Explicar la diferencia entre los flags **V** y **C**, y si existe alguna relación entre sus valores.

. Describir dos aplicaciones del flag **C** en las sumas.

. Explicar por qué en una resta se debe invertir CUAL.

. Explicar por qué el flag **C** no se considera para enteros.

. ¿Mediante una suma se puede saber si el minuendo es mayor, igual o menor que el sustraendo ?

. Explicar por qué para cualquier suma o resta que haga la UAL, siempre genera simultáneamente los flags SZVC, siendo que SVZ sirven para enteros, y CZ para magnitudes.

. Para todas las sumas de enteros de la sección N.2.2 determinar los valores de SZVC, qué magnitudes producen la misma suma en la UAL, y qué conclusiones resultan de los valores de los flags para enteros y magnitudes.

Para todas las sumas de enteros del ejercicio 10 de N.4 determinar los valores de SZVC, qué magnitudes producen la misma suma en la UAL, y qué conclusiones resultan de los valores de los flags para enteros y magnitudes.

). Para todas las sumas de enteros del ejercicio 12 de N.4 determinar los valores de SZVC, qué magnitudes producen la misma suma en la UAL, y qué conclusiones resultan de los valores de los flags para enteros y magnitudes.

. Para las restas de la sección N.3 hallar los valores de SZVC y usarlos para determinar si el minuendo es mayor, menor o igual que el sustraendo, verificando que lo determinado se cumple para los números dados en base diez:
I) para los números enteros en base diez que originaron cada resta.
II) para las magnitudes que en base diez que originarían la misma resta, los cuales deben ser previamente determinados

. Para formate de n=4 bits encontrar ejemplos de restas de magnitudes en que si $A \geq B$ es S=1, y si $A < B$ es S=0, y qué conclusiones se extraen respecto del flag S para comparar magnitudes.

. Realizar en el menor formato posible sin que dé overflow la operación $-534 - (-256)$ indicando SZVC.

. En función del problema anterior explicar por qué se afirma que el overflow es un error por truncamiento.

. Se tiene un apellido que empieza con D y otro que empieza con R, en ese orden. Indicar como haría un computador para determinar en función de SZVC, si están en orden o no..

Respuestas de algunos ejercicios planteados

5. En una suma de enteros C siempre se descarta pues se trata de un exceso en la representación de uno de los operandos.

6. No, sólo mediante una resta.

7. La UAL no sabe si el programa que se está ejecutando es para magnitudes o enteros, por lo que está preparada para generar los flags para ambos tipos de variables. Un programa para magnitudes usará sólo CZ y uno para enteros usará SVZ.

8. a) S=0 Z=0 V=0 C=0 Enteros: V=0 S=0 → R=ok y +R Magnitudes 8 + 43 C=0 → R=ok

   b) S=0 Z=0 V=0 C=1 Enteros: V=0 S=0 → R=ok y +R Magnitudes 248 + 43 C=1 → R=ε

   c) S=1 Z=0 V=0 C=0 Enteros: V=0 S=1 → R=ok y −R Magnitudes 8 + 213 C=0 → R=ok

   d) S=1 Z=0 V=0 C=1 Enteros: V=0 S=1 → R=ok y −R Magnitudes 248 + 213 C=1 → R=ε

10, a) S=1 Z=0 V=1 C=0 Enteros: V=1 → R=ε Magnitudes 30 + 102 C=0 → R=ok=132

   b) S=0 Z=0 V=1 C=1 Enteros: V=1 → R=ε Magnitudes 226 + 154 C=1 → R=ε

11. a-i) V=0 S=0 Z=0 C=0 Enteros: V=0 S=0 → R=ok y +R y min ≥ sust (+43) ≥ (+8)
    Magnitudes 43 − 8 C=0 → min ≥ sust 43 ≥ 8 → R=ok=35

    a-ii) V=0 S=1 Z=0 C=1 Enteros: V=0 S=1 → R=ok y −R y min < sust (+8) < (+43)
    Magnitudes 8 − 43 C=1 → min < sust 8 < 43 → R=ε

    b-i) V=0 S=0 Z=0 C=0 Enteros: V=0 S=0 → R=ok y +R y min ≥ sust (−8) ≥ (−43)
    Magnitudes 248 − 213 C=0 → min ≥ sust 248 ≥ 213 → R=ok=35

    b-ii) V=0 S=1 Z=0 C=1 Enteros: V=0 S=1 → R=ok y −R y min < sust (−43) < (−8)
    Magnitudes 213 − 248 C=1 → min < sust 213 < 248 → R=ε

    c) V=0 S=0 Z=0 C=1 Enteros: V=0 S=0 → R=ok y +R y min ≥ sust (+8) ≥ (−43)
    Magnitudes 8 − 213 C=1 → min < sust 8 < 213 → R=ε

    d) V=0 S=1 Z=0 C=0 Enteros: V=0 S=1 → R=ok y −R y min < sust (−8) < (+43)
    Magnitudes 248 − 43 C=0 → min ≥ sust 248 ≥ 43 → R=ok=205

N.6 Números binarios fraccionarios

En el sistema decimal, un número fraccionario se puede expresar como un cociente o mediante una coma. Así: 1/4 = 0,25; 4/3 = 1,33..., etc. En base dos o en otra, también podemos representar, con una simbología semejante, un número que sea menor que la unidad, o que presente una parte entera y otra que es una fracción de la unidad.

Volviendo a la balanza decimal del apéndice de la Unidad 1, con 9 pesas de 1, 10, 100, ... gramos, si se necesita apreciar fracciones de un gramo, podemos suponer que existen 9 pesas de una décima de gramo (1/10 = 0,1), 9 pesas de un centésimo de gramos (1/100 = 0,01), etc... Cada 9 pesas que son fracción de gramo son **diez veces menores** (diez es la canatidad de símbolos) que las 9 existentes de la medida superior siguiente. Como se vió para parte entera cada pesa era diez veces mayor que la medida inferior siguiente. Cuando escribimos que un objeto pesa 328,205 la coma indica que el 8 que está a su izquierda son unidades, y que el 2 de su derecha son decimas de gramo.

Esta misma convención se usa en base dos, pero del mismo modo que a partir de la pesa de 1 gramo hacia arriba, cada pesa es el doble que la medida anterior menor, las pesas menores que 1 gramo van en una progresión en que cada pesa **es la *mitad*** de la medida mayor anterior, comenzando por la medida de 1/2 gramo, existiendo tanto para las mayores o menores de 1 gramo, **una** pesa de cada tipo. Expresadas en decimal las pesas binarias tendrían los gramajes:

... 32g, 16g, 8g, 4g, 2g, **1g**, 1/2g, 1/4g, 1/8g, 1/16g, ...

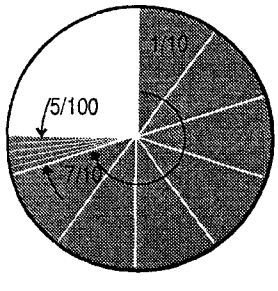
pudiendo existir infinitas medidas hacia un lado u otro. Expresadas en binario serían:

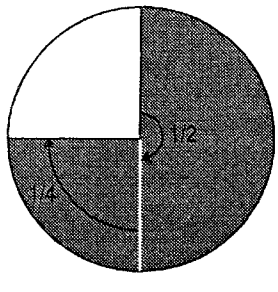
... 100000g, 10000g, 1000g, 100g, 10g, **1g**, 1/10g, 1/100g, 1/1000g, 1/10000g ...

Realizando las divisiones binarias indicadas conforme se vió en el apéndice numérico de la Unidad 1, y usando coma para separar la parte entera de la fraccionaria, también resulta en binario:

1/10 = 0,1 1/100 = 0,01 etc...

Suponiendo un objeto que pese en decimal 0,75 g. en la balanza con pesas binarias se formaría con una de 1/2g y otra de 1/4g, que expresado en binario sería: 1/10g + 1/100g = 0,1g + 0,01g = 0,11 g





Para cotejar la estructura fraccionaria en bases dos y diez. en las figuras siguientes se ha dibujado la misma fracción de un círculo. Dicha fracción en base diez se forma con 7 agrupamientos de 1/10 de círculo, más 5 de 1/100, mientras que en base dos con una fracción que es la mitad y otra igual a la cuarta parate del círculo.

Al igual que en base diez, en base dos habrá fracciones que no puede expresarse mediante un número finito de dígitos. También como veremos, puede ocurrir que una fracción representable por un número finito de dígitos en base diez, requiera infinitos en base dos.

Entonces, del mismo modo que en base diez existen la décima, centésima, milésima, ... parte de la unidad, en base dos, se define la *mitad, cuarta, octava, diciseisava, ...* parte de la unidad.

De manera inversa, dado 0,11 en binario, para concer que fracción decimal se procede como se indica a continuación:

$$0,1\,1_B = (1/2 + 1/4)_D = (0,5 + 0,25)_D = 0,75_D$$

A continuación generalizaremos el método para pasar a decimal un número binario con parte entera y decimal

$$10110,101_B = 10110,101_B = (16+4+2)+(1/2+1/8)_D = (22 + 0,625)_D = 22,625$$

Regla para convertir un número binario con parte fraccionaria a decimal:

1. El bit que está a la iquierda de la coma binaria le corresponde peso **1**
2. A partir del mismo hacia la izquierda los pesos 2, 4, 8, 16, ... se escriben sobre los bits de la parte entera; y hacia la derecha los pesos 1/2, 1/4, 1/8, 1/16, 1/32 ... sobre los bits fraccionarios
3. Se suman los pesos de las posiciones que tengan un uno, y el número se expresa en decimal.

El pasaje inverso de decimal a binario se realiza según la siguiente:

Regla para convertir un número decimal con partes entera y fraccionaria a binario:

1. Convertir a binario cada parte por separado. La parte entera se convierte por ejemplo conforme al método de las divisiones sucesivas por 2
2. La parte fraccionaria mediante sucesivas multiplicaciones por 2
3. Sumar ambas porciones antes halladas.

Ejemplo: convertir a binario el número decimal 35,62 = 35 + 0,62

1. $35_D = 100011_B$

2. Para hallar la parte fraccionaria binaria, se procede como sigue, por sucesivas multiplicaciones por 2 de cada nueva parte fraccionaria decimal

2x0,62 = 1,24 = **1** + 0,24
2x0,24 = 0,48 = **0** + 0,48
2x0,48 = 0,96 = **0** + 0,96
2x0,96 = 1,92 = **1** + 0,92
2x0,92 = 1,84 = **1** + 0,84
2x0,84 = 1,68 = **1** + 0,68
2x0,68 = 1,36 = **1** + 0,36
2x0,36 = 0,72 = **0** + 0,72
2x0,72 = 1,44 = **1** + 0,44
2x0,44 = 0,88 = **0** + 0,88
.....

Las sucesivas partes enteras halladas (en negrita) conforme van apareciendo, constituyen los bits de la parte fraccionaria binaria. O sea $0,62_D \approx 0,1001111010_B$

3. $35 + 0,62)_D \approx (100011 + 0,1001111010)_B = 100011, 1001111010_B$

En este caso se ha interrumpido arbitrariamente el proceso luego de haber obtenido una precisión de 10 bits fraccionarios, sin haber alcanzado una parte fraccionaria decimal igual a 0,00 como aparece por ejemplo si se pasa 0,75 a binario:

2x0,75 = 1,50 = **1** + 0,50
2x0,50 = 1,00 = **1** + 0,00 pudiéndose escribir la igualdad $0,75_D = 0,11_B$

Puede darse el caso que la determinación anterior no termine nunca, o sea, que un número decimal con parte fraccionaria pura, en base dos tenga su parte fraccionaria periódica.

Por ejemplo, cualquier número decimal con parte fraccionaria $0,1_D$ al convertirse en binario será periódico. Puede verificarse que $0,1_D = 0,0\mathbf{0011}_B$ (periodo en negrita)

Este método puede aplicarse para pasar de decimal a **cualquier** base **R**, con tal de realizar en decimal el método de las divisiones o multiplicaciones sucesivas por **R**, para la parte entera y fraccionaria, respectivamente

N.7 Potenciación en cualquier base

En cualquier base, siempre que se tengan **p** factores iguales de un número **n**, se podrá escribir: n x n x n x n x ... n x n = n^p según sea la base, variará la representación de **n** y **p**.

Resulta una buena ejercitación para sistemas numéricos, verificar las siguientes equivalencias:

$(1100 \times 1100 \times 1100)_B = 1100^{11}{}_B = 12^3{}_D = C^3{}_H$

$(10 \times 10 \times 10 \times 10)_B = 10000_B = 10^{100}{}_B$

$10^{100}{}_B = 2^4{}_D = (2 \times 2 \times 2 \times 2)_D = 16_D = 10_H = 2^4{}_H$

$1000_D = 10^3{}_D = 1010^{11}{}_B = (1010 \times 1010 \times 1010)_B = 1111101000_B$

$10^3{}_D = A^3{}_H = (A \times A \times A)_H = 3E8_H$

$10^3{}_H = (10 \times 10 \times 10)_H = 1000_H = 16^3{}_D = 4096_D$

$10^3{}_H = 10000^{11}{}_B = (10000 \times 10000 \times 10000)_B = 1000000000000_B$

Debe tenerse presente que **en cualquier base, la unidad seguida de p ceros puede expresar como la base a la potencia p**, ***simbolizándose 10 la base en todos los sistemas numéricos***

Así: $100000_B = (10 \times 10 \times 10 \times 10 \times 10)_B = 10^{101}$ siendo $10_B = 2_D$

$100_H = (10 \times 10)_H = 10^2{}_H$ siendo $10_H = 16_D$

N.8 Representación en punto flotante de números reales declarados como tales en alto nivel

Así como se definió una forma de representar números positivos y negativos sin usar signos más y menos, cuando se opera con números que presenten coma se requiere alguna convención representarlos mediante combinacione binarias.

Las representaciones de números reales en *"punto flotante"* [1] *("floating point"* = **FP**) –en castellano coma flotante– sirven para tal cometido, permitiendo además operaciones con magnitudes y resultados dentro de un amplio rango de valores, para aplicaciones desde las comerciales hasta los cálculos astronómicos. Además, obviamente, se puede operar con números enteros.

En esencia se trata de una representación de tipo exponencial: $\mathbf{N = \pm m \times 10^{\pm p}}$

semejante a la notación científica decimal, que permite en unos pocos bytes expresar números muy grandes o muy pequeños..

Su denominación se debe a que la posición del punto (coma), en la expresión anterior, puede desplazarse en el número **m** de la expresión anterior, si a la par se ajusta el exponente **p**, para adecuar la representación de operandos o resultados, si así se requiere.

Esto se vincula con el hecho de que los circuitos para operaciones en punto flotante (**coprocesador**) determinan en forma automática el lugar dónde va la coma en cada resultado, desentendiéndose de ello el programador. Cuando un computador no posee tales circuitos, se debe recurrir al software que opera con la **UAL** para emularlos, a costa de velocidad de procesamiento.

En notación científica estandar, los números se expresan de la forma: $N = \pm n\,E \pm p = \pm n \times 10^{\pm p}$

donde **n** es un número comprendido entre 1 y 10; **p** es un número entero.(así también representan las calculadoras de bolsillo)

Por ejemplo:

$0{,}00003 = 3.0\,\mathbf{E}-5 = 3 \times 10^{-5}$

$-246{,}36 = -2.4636\,\mathbf{E}+2 = -2{,}4636 \times 10^{2}$

$82000000000 = 8.2\,\mathbf{E}+10 = 8{,}2 \times 10^{10}$

En el papel **n** ó **p** pueden tomar cualquier valor, mientras que en el visor de una calculadora tienen valores máximos predeterminados. Supongamos que en un cierto formato **n** y **p** pueden tener hasta 5 y 2 dígitos, respectivamente; y que no escribimos la letra **E** o la base 10.

Los números anteriores podrían representarse como sigue: (+3,0000; –05) ; (–2,4636; +02) ; (+8,2000; +10)

También diremos que constan de una mantisa con 5 dígitos significativos o de precisión, y que cada factor de escala (10^{-5} 10^{2} y 10^{10} según sea) determina la verdadera posición de la coma.

Normalización:

En notación exponencial siempre es posible correr k lugares la coma a la izquierda (o derecha), si simultáneamente se incrementa (o decrementa) el exponente en un valor k, sin que cambie el valor del número representado.

Así, sumando uno a los exponentes de los números anteriores, resulta:

$0{,}3 \times 10^{-4}$; $-0{,}24636 \times 10^{3}$; $0{,}82 \times 10^{11}$

Decimos que se trata de una notación exponencial **"normalizada"**, de la forma: $\mathbf{m} \times 10^{\mathbf{p}}$

donde **m** es la *mantisa*, siendo de la forma $0{,}X_1X_2 \ldots X_n$ con $X_1 \neq 0$; o sea que debe ser distinto de cero el primer dígito que sigue a la coma. En general, será pues $0{,}1 \leq \mathbf{m} < 1$

Un número puede expresarse en forma normalizada en cualquier base.

Representación estándar para punto flotante del IEEE [1]

Existen varias convenciones para expresar números reales en punto flotante.

A los efectos de presentar la representación estándar para punto flotante del **IEEE**, ("Floating Point Standard" - **FPS**) consideraremos para los números binarios que $1_B \leq \mathbf{m} < 10_B$

o sea que **m** será de la forma: $\mathbf{m = 1{,}f}$ (**f** parte fraccionaria de **m**)

Por ejemplo, los siguientes números reales en base diez, convertidos a base dos , y en forma exponenencial normalizada serán:

$5_D = 101_B = (1{,}01 \times 100)_B = (1{,}01 \times 10^{10})_B$ [2] (se desplazó la coma $2 = 10_B$ lugares a la izquierda para normalizar)

---

[1] En contraposición los números de *"punto fijo"* o *"coma fija"* en castellano, suponen que la coma debe estar en una posición fija en un determinado formato de bits, según la conveniencia del cálculo a realizar. En particular si se supone que está luego del bit extremo derecho, se trata de un número entero. En un formato de 8 bits, estos enteros tienen un rango de 0 a 255 con resolución de una unidad entre una combinación y la siguiente. Si por ejemplo la coma se ubica a la izquierda del bit más significativo, todos los números de dicho formato serán fraccionarios, de la forma 0,..... . Entonces la resolución pasa a ser 1/256, pero el rango estará sólo entre 0 y 1. Vale decir que al aumentar el rango se pierde resolución. *A diferencia, punto flotante permite alta resolución y rango extenso.*

[1] Instituto de Ingenieros Electrónicos y Electricistas

$20_D = 10100_B = (1{,}01 \times 10000)_B := (1{,}01 \times 10^{100})_B$ (se desplazó la coma $4 = 100_B$ lugares a la izquierda para normalizar)

$(-4101{,}25)_B = (-1000000000101{,}01)_B = (-1{,}00000000010101 \times 1000000000000)_B = (-1{,}00000000010101 \times 10^{1100})_B$

(en este caso se desplazó la coma $12 = 1100_B$ lugares a la izquierda)

$0{,}015625_D = (0{,}000001)_B$[3] $= (1{,}0 \times \frac{1}{1000000})_B = (1{,}0 \times 10^{-110})_B$ (se corrió la coma $6_D = 110_B$ lugares a la derecha)

$1_D = 1_B = 1{,}0 \times 10^0{}_B$ (la coma no se corrió ningún lugar hacia derecha o izquierda)

$-0{,}5_D = 0{,}1_B = 1{,}0 \times 1/10_B = 1{,}0 \times 10^{-1}{}_B$ (se corrió la coma un lugar a la derecha)

$12{,}1_D = (12 + 0{,}1)_B = (1100 + 0{,}\underline{00011})_B$[5] $= 1100{,}\underline{00011}_B = 1{,}10000\underline{011} \times 10^{11}{}_B$

Reglas básicas de representación en FPS:

- La representación es de la forma $\mathbf{N} = \pm \mathbf{m} \times 10^{\pm p} = \pm \mathbf{1{,}f} \times 10^{\pm p}$ (siendo $10_B = 2$)
- En *simple precisión* cualquier número requiere 32 bits = 4 bytes.
- Sólo se representa la **parte fraccionaria f** de la mantisa **m**, reservándose para la misma los últimos **23 bits**, sobreentendiéndose que la parte entera es siempre *implícitamente* **1** y que existe una coma antes de **f** [6]
- El signo de la mantisa será un **bit de signo** (**S**) que vale **0** si es positiva, y **1** si es negativa, estando dicho bit el extremo izquierdo de la representación (separado de la mantisa)[7]
- Al exponente ± **p** se le debe sumar $\mathbf{127_D}$ ("exceso o desplazamiento 127"), resultando un número **c** = ± **p** + 127 para el cual se reservan **8 bits** a continuación del bit de signo citado.
- El número **cero** puede representarse con los 32 bits iguales a cero (+0), ó con el bit de signo de valor **1**, y los 31 restantes con valor cero (–0)
- Existe una convención para representar $+\infty$ y $-\infty$

Se aplicarán estas reglas a la representación en FPS de los números antes normalizados

	S	± p + 127 (8 bits)	f (23 bits)	
	+	**2+127=129**		
$5 = 1{,}01 \times 10^{10} =$	0	10000001	01000000000000000000000	$10000001_B = 129_D$ se halla sumando 127_D al exponente $p = 10_B = 2_D$
	+	**4+127=131**	**f**	
$20 = 1{,}01 \times 10^{100} =$	0	10000011	01000000000000000000000	$10000011_B = 131_D$ se halla sumando 127_D al exponente $p = 100_B = 4_D$
	–	**12+127**	**f**	$10001011_B = 139_D$ se halla sumando 127_D al exponente $p = 1100_B = 12_D$
$-4101{,}25 = -1{,}00000000010101 \times 10^{1100} =$	1	10001011	00000000010101000000000	
	+	**–6+127**	**f**	$01111001_B = 121_D$ se halla sumando 127_D al exponente $p = -0110_B = -6$
$0{,}015625 = 1{,}0 \times 10^{-110} =$	0	01111001	00000000000000000000000	
	–	**–1+127=126**	**f**	
$-0{,}5_D = 0{,}1_B = 1{,}0 \times 10^{-1} =$	1	01111110	00000000000000000000000	
	+	**0+127=127**	**f**	
$1 = 1{,}0 \times 10^0 =$	0	01111111	00000000000000000000000	
	+	**3+127=130**	**f**	
$12{,}1 = 1{,}1000\underline{\mathbf{0011}} \times 10^{11} =$	0	10000010	10000**0011**00110011**0011**001	

---

2 En cualquier base , correr la coma **n** lugares a la derecha (izquierda) es multiplicar (dividir) por un uno seguido de **n** ceros. Asimismo este último número se puede expresar como 10^n (que sería 10^{-n} si la coma se corrió **n** lugares a la izquierda)

3 Este número se halla a partir del 0,015625 por sucesivas multiplicaciones por dos (sección N.6)

5 Valor determinado anteriormente

6 Aunque f no esté normalizado, se considera que **m = 1,f** sí lo está, pues tiene dicho uno en la posición más significativa. Este bit no forma parte del formato de 32 bits. La unidad de punto flotante (coprocesador) lo incorpora cuando debe operar.

7 Se trata pues de una representación de números del tipo signo y magnitud.

Obsérvese que *no se emplea la convención de complemento a dos, ya sea para el exponente o la mantisa.* Esta se representa con bit de signo y magnitud normalizada[1] .

La coma está implícita, a la izquierda del bit más significativo de la parte fraccionaria de la mantisa. **Tampoco se representa la base $10_B = 2_D$**

Un exponente positivo más el exceso 127 puede formar un número entre:
01111111 = 127 = 0+127 y 11111110 = 254 = 127 + 127 (exponentes entre 0 y +128);
mientras que exponentes negativos excedidos en 127 forman números entre:
01111110 = 126 = −1 + 127 y 00000001 = 1 = −126 + 127 (exponentes entre −1 y −126).
Los valores máximo 11111111 y mínimo 00000000 para p+127 se reservan para representaciones especiales:

Con 11111111 y **f** = 0 se representa el infinito (+ ó − según el bit de signo)
Con 11111111 y **f** ≠ 0 se usa para indicar operaciones no válidas, como 0 por ∝
Con 00000000 y **f** = 0 se trata del número cero, como se anticipó.
Con 00000000 y **f** ≠ 0 el número está *desnormalizado:* tiene una magnitud menor que el valor mínimo que se representa en el formato normalizado.

En definitiva el intervalo de exponentes va de −126 a 127.

Siguiendo el camino inverso, puede hallarse cuál es el número real correspondiente a su representación en FP

Por ejemplo, si en la memoria se lee el número en FP como

BFC00000 = **1**01111110**1**10000000000000000000000 puede razonarse así:

a. El primer uno implica que es negativo;

b. Los 8 bits siguientes $01111110_B = 126_D = p + 127$ son el exponente excedido en 127, o sea p = 126 −127 = −1

c. Los 23 bits restantes 10000000000000000000000 constituyen la parte fraccionaria (**f**), delante de los cuales debemos agregar un uno y la coma, que se quitaron cuando se pasó de decimal a FPL.

Así se forma la mantisa **m** = **1,f** = 1, 10000000000000000000000

d. Puesto que $N = \pm \mathbf{m} \times 10^p$ ($10_B = 2_D$); en este caso $N = -1{,}11 \times 10^{-1}{}_B = -0{,}111_B = -(1/2 + 1/4 + 1/8)_D = -0{,}875_D$

Existe también el formato FPS de ***doble precisión***, con 64 bits: 11 para el exponente excedido en $\mathbf{1023}_D$, 52 bits para la precisión de la parte fraccionaria **f** de la mantisa, y uno para el signo.

Rangos de representación:
Determinaremos primero el mayor número positivo representable ($+N_{MAX}$) que en punto flotante tendrá máximo exponente (11111110 subrayado) y mantisa (23 unos): 01111111011111111111111111111111

Siguiendo los pasos a,b,c,d anteriores, resulta:

$11111110_B = 254 = p + 127$; luego p = 254 − 127 = 127

$\mathbf{m} = 1{,}11111111111111111111111_B \approx 1{,}99_D \approx 2_D$

$+N_{MAX} = m \times 2^p{}_D \approx 2 \times 2^{127}{}_D$

Si se quiere expresar $2^{127}{}_D$ como una potencia de diez, debe ser $2^{127}{}_D = 10^Y{}_D$

Aplicando logaritmos: $127 . \log_{10} = Y . \log_{10}10$ de donde $127 . 0{,}3 = Y . 1$

Resultando Y = 38,1 ≈ 38 y por lo tanto $+N_{MAX} = 2x2^{127}{}_D \approx 2x10^{38}{}_D$

10^{38} es un uno seguido de 38 ceros, por lo que pueden representarse números con hasta 38 dígitos

$-N_{MAX}$ tendrá la misma magnitud que $+N_{MAX}$ dado que su representación sólo diferirá en que empieza con **1**:

$-N_{MAX}$ = 11111111011111111111111111111111 Entonces resulta: $-N_{MAX} \approx -2x10^{38}{}_D$

Un número fraccionario positivo más pequeño ($+N_{FMIN}$) en FP se define para **p**+127 = 00000001, y con **f** = 0

[1] Esta convención prevee operaciones con mantisas no normalizadas.

$+N_{FMIN}$ = 00000000100000000000000000000000

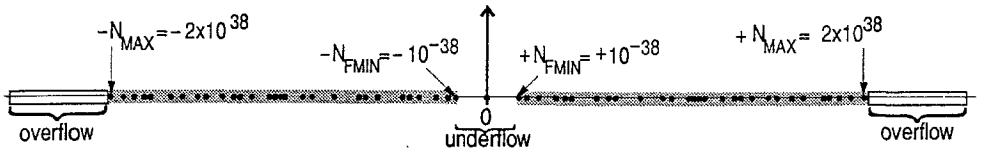
de donde 00000001 = 1 = p + 127 por lo que p = −126; m = 1,f = 1,00000000000000000000000 = 1

o sea que en decimal se tiene $+N_{FMIN} = 1x2^{-126} \approx 10^{-38}{}_D = 0,0\ldots.01_D$ (cero - coma - 38 ceros - uno)

(38 surge de aplicar logaritmos siendo ahora Y = 126 x 0,3 = 37,8 ≈ 38)

El fraccionario negativo mínimo representable correspondiente al anterior tendría igual magnitud, diferiendo en el bit de signo. Sería en decimal: $-N_{FMIN} = -1x2^{-127}{}_D \approx -10^{-38}{}_D = -0,0\ldots.01_D$ (cero - coma - 38 ceros - uno)

De acuerdo a los resultados obtenidos, se tiene el siguiente rango de representación:



Si algún dato o resultado positivo o negativo presenta parte entera con magnitud que supere N_{MAX} , se tendrá *"overflow"*, pues su exponente superaría el valor máximo 128. Y si el mismo es un fraccionario positivo o negativo de magnitud menor que 10^{-38} , se tendrá *"underflow"*.

Debe notarse que los números reales representados en PF constituyen sólo un conjunto finito de números reales racionales. Asimismo, sus valores no están uniformemente distribuídos en la recta que los representa, como denotan los puntos irregularmente dibujados, y puede deducirse de la expresión $N = mx10^{p+127}$

Manejo del underflow: desnormalización

Otra de las ventajas de la convención del IEEE, es que se pueden representar y operar números ***"desnormalizados"***, de la zona de underflow, siendo así más pequeños que el menor de los números normalizados ($1x2^{-126}$). Ellos son de la forma $(-1)^S$ x **0**,f x10 $^{-126}$ o sea que *todos* son del tipo 0,xxxxx tienen exponente −126, y se identifican por que C = 00000000. Esto último se hace para que exista una proxi-midad entre el mayor número desnormalizado 00000000111111111111111111111111 = 0,1111...11$x2^{-126}$ = 0,99999 $x2^{-126}$ y el número normalizado más pequeño ya tratado: $+N_{FMIN}$ = 00000001000000000000000000000000 = $1x2^{-126}$

El número desnormalizado más pequeño es 00000000000000000000000000000001.

Resulta, que el exponente es −126, y la parte fraccionaria 2^{-23}; por lo que su valor es $2^{-23}.2^{-126} = 2^{-149}$

Los valores determinados valen (signo por medio) para los desnormalizados negativos.

Simple precisión en punto flotante de 32 bits:

Puesto que los números extremos representables en FP son $\pm10^{38}$, ello implica que, como ser, podemos tipear en el teclado un número en base diez de 37 cifras, que será aceptado dentro del rango de representación. Pero, como se verificará, devolverá con precisión las 7 primeras de sus cifras, dado que en la representación tratada sólo se disponen de 23+1=24 bits para guardar cualquier número representable en FP (son 23+1 por el uno de la normalización que no se representa). Esta limitación de la cantidad de dígitos que se pueden representar, propia de cualquier representación de números en un computador (sean reales, integers u otros) determina que siempre se trate de "**números con precisión finita**".

Ejemplo: sea el número $4.294.967.294_D$ = $11111111111111111111111111111110_B$ que normalizado y representado en FP resulta $1,\mathbf{11111111111111111111111111111110} \times 10^{11111\ (31)}{}_B$ = 01001111011111111111111111111111 11111110

31+127=158 ←====== 23 bits ======→

Como sólo pueden almacenarse 23+1= 24 bits de la parte fraccionaria, resulta que el número queda truncado, con 8 bits fuera del formato (11111110 = 254). Suponiendo que no exista redondeo para el truncamiento dichos 8 bits se considerarían ceros, por lo que el número almacenado sería:

11111111111111111111111100000000 = 4.294.967.294 − 254 = 4.294.967.040

Se verifica, como se anticipó, que sólo se han conservado intactos 7 dígitos de precisión. Esto está de acuerdo con el hecho de que **a cada dígito decimal le corresponden 3,5 bits**, por lo que 24 bits permiten representar 24/3,5 dígitos cociente cercano a 7.

Punto flotante doble precisión (64 bits):

Un Pentium está preparado para procesar 64 bits en FP, según la convención para este formato del IEEE, el primer bit es el del signo; luego siguen 11 bits para el exponente más 1023 de exceso, quedando los 52 bits restantes para representar la parte fraccionaria del número normalizado. Esto permite una precisión de (52+1)/3,5 dígitos, cociente cercano a 15.

Sumas en el coprocesador (FPU - Floating Point Unit)

Para sumar/restar dos números, el coprocesador iguala el exponente del menor al del mayor:

$P = 1\underline{10000110}00000010100000000000000 = 1{,}00000101 \times 10^{111}$ $= 0{,}0000100000101 \times 10^{1100}$

$Q = 1\underline{10001011}00000000011111000000000 = 1{,}00000000011111 \times 10^{1100}$ $= {}^{+}\underline{1{,}00000000011111 \times 10^{1100}}$

$1{,}000010001010001 \times 10^{1100}$

V.9 Codificación y suma en BCD natural

Según se vió, para pasar del sistema decimal al binario es necesario realizar una serie de operaciones aritméticas, como en el método manual de las sucesivas divisiones por dos.

Los ***"códigos BCD"*** (*Binary-Coded-Decimal:* decimal codificado en binario) se emplean para convertir *directamente*, sin cálculo alguno, números decimales en combinaciones binarias, según determinadas convenciones, que tienen en común el hecho de que ***a cada digito decimal le corresponden 4 bits.***

La convención ***"BCD natural"*** o ***"BCD 8-4-2-1"*** atribuye a los símbolos decimales la correspondencia binaria dada por la tabla de la figura 2.2 que volvemos a repetir:

	8-4-2-1
0	0 0 0 0
1	0 0 0 1
2	0 0 1 0
3	0 0 1 1
4	0 1 0 0
5	0 1 0 1
6	0 1 1 0
7	0 1 1 1
8	1 0 0 0
9	1 0 0 1

Dado un número cualquiera en base diez, para convertirlo a binario BCD basta reemplazar cada dígito del mismo por los 4 bits correspondientes

Esta metodología de conversión es semejante a la utilizada para pasar de hexadecimal a binario para los símbolos del 0 al 9:

$$8125_D \equiv \overbrace{1000\,0001\,0010\,0101}^{8\text{-}4\text{-}2\text{-}1\ 8\text{-}4\text{-}2\text{-}1\ 8\text{-}4\text{-}2\text{-}1\ 8\text{-}4\text{-}2\text{-}1}{}_{BCD}$$
$$\underbrace{\quad}_{8}\ \underbrace{\quad}_{1}\ \underbrace{\quad}_{2}\ \underbrace{\quad}_{5}$$

mientras que en el sistema binario se representa

$$8125_D = \overbrace{1\,1\,1\,1\,1\,1\,0\,1\,1\,1\,1\,0\,1}^{4096\ \ 128\ 64\ 32\ 16\ 8\ \ 4\ \ 2\ \ 1}{}_{B}$$

Asimismo, $10_D \equiv 00010000_{BCD}$; siendo en cambio $10_D = 1010_B$

De lo anterior surge claramente, que si bien las conversiones de decimal a BCD, y de hexadecimal a binario natural, se realizan de igual forma, el significado de la combinación binaria es bien diferente. Se usó el símbolo ≡ para indicar equivalencia de representación entre BCD y decimal, y no el símbolo = , pues se trata de una convención.

El pasaje inverso de BCD a decimal consiste en separar en el número binario grupos de 4 bits, y determinar en base diez qué dígito decimal representa cada cuarteto:

Así: $0\,0\,1\,0\,1\,0\,0\,1\,0\,0\,0\,0\,0\,0\,0\,0_{BCD} \equiv 2900_D$

$\lfloor\ 2\ \rfloor\lfloor\ 9\ \rfloor\lfloor\ 0\ \rfloor\lfloor\ 0\ \rfloor$

En definitiva, podemos sistematizar estó así, para indicar que el pasaje es directo sólo en sentido horizontal:

Hexadecimal ⇐ 1 simbolo por cuarteto de bits ⇒ **Binario natural**

Decimal ⇐ 1 simbolo por cuarteto de bits ⇒ **BCD**

Para pasar de decimal a hexa, o en sentido inverso, hay que hacer las operaciones indicadas en la unidad 1.

Así: $0010100100000000_{BCD} \equiv 2900_D = B54_H = 101101010100_B$; siendo también: $0010100100000000_B = 2900_H$

La desventaja del BCD respecto a los binarios naturales, es que en BCD en un byte se pueden representar combinaciones desde 00 a 99, mientras que con los segundos se puede codificar de 0 a 255, en un byte. Asimismo, sólo conviene realizar sobre datos en BCD la suma y resta, existiendo instrucciones para corregir (ajustar) los resultados que se obtienen de sumar cuartetos BCD como si fueran números binarios. En esencia el formato BCD es un buen compromiso entre el tiempo de convertir números de ASCII a binarios –que en BCD es directo– y el tiempo que insumen sumas y restas, que en BCD es bastante corto, según podrá deducirse

Suma de números en BCD natural

*Un **método para sumar** dos números decimales codificados en BCD "natural" consiste en sumar los cuartetos que los constituyen como si fueran números naturales, y luego sumar* $6 = 0110_B$ *si la suma parcial de dos cuartetos supera* $9 = 1001_B$

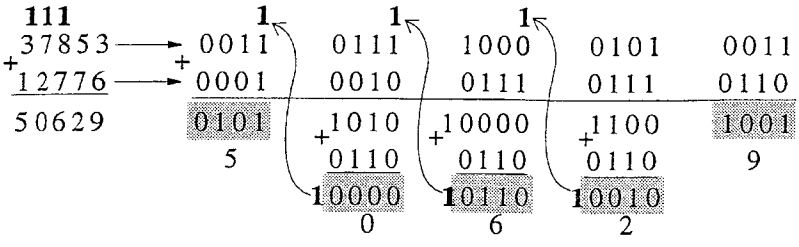
Esto último puede ocurrir de dos formas:

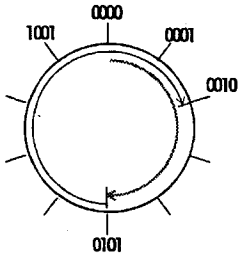
1. Si dicha suma resulta con valores 1010 a 1111 (10_D a 15_D): combinaciones que no son BCD

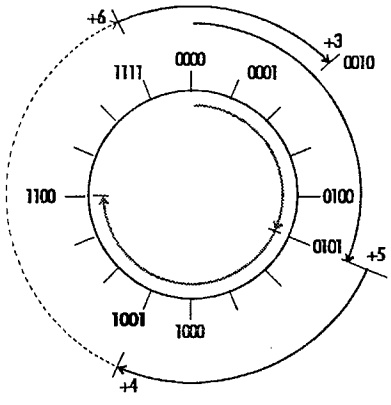
2 Si ella es 10000, 10001, 10010, 10011 (16_D a 19_D [2]): combinación BCD con **1** de acarreo

þ Se ejemplifica una suma efectuada con este método, aplicable a cualquier número de dígitos.

Las combinaciones a corregir por no ser BCD, o por que el uno fuera del cuarteto indica resultado ≥ 16, se les ha sumado 0110. En negrita aparece el uno de acarreo en decimal y en BCD.







Para justificar por qué se suma 6, apelaremos a un cuenta-vueltas binario con números del 0000 al 1111 (módulo 16), razonando como sigue. Si (dibujo de la derecha) a partir del 0000 avanzamos 5 posiciones se llega al 0101, y si luego progresamos 7 más se alcanza la 1100, habiendo realizado la suma 5 + 7 = 0101 + 0111 = 1100 con números binarios naturales.

La misma suma en el cuenta-vueltas izquierdo para decimales y BCD (con números de 0000 a 1001): si desde el 0101 se avanza 7 posiciones más se alcanza el 0010 luego de haber dado una vuelta, en correspondencia con el número 12 = $0001\ 0010_{BCD}$

Si queremos usar el cuenta-vueltas binario (con 6 pasos más que el otro) para sumar en BCD, cada vez que al sumar alcanzamos el valor 1001 debemos sumarle 6 = 0110, de modo de pasar al 1111, que está a una unidad del 0000, del mismo modo que lo está el 1001 respecto del 0000 en el cuenta-vueltas BCD.

Podemos pensar esto último como que en el cuenta vueltas binario la suma 5+7 la realizamos como 5+*4*+**6**+*3*. Esto es, descomponemos 7= *4*+*3*, para mostrar que al realizar la suma parcial 5+*4* habremos llegado al 1001 en este cuenta-vueltas, por lo que le sumamos 6=0110 para pasar al 1111, y luego sumamos las 3 unidades que faltaban para completar las 7 de la cuenta. De esta forma, como se ha dibujado, se alcanza el 0010, como debe ser en la suma BCD.

Puesto que en una suma no importa el orden de los sumandos, en vez de efectuar 5+4+6+3 se hace directamente 5+7+6 = 0101+0111+0110 como se realizó para uno de los cuartetos de la suma BCD arriba ejemplificada.

Decimal (BCD) empaquetado (packed)

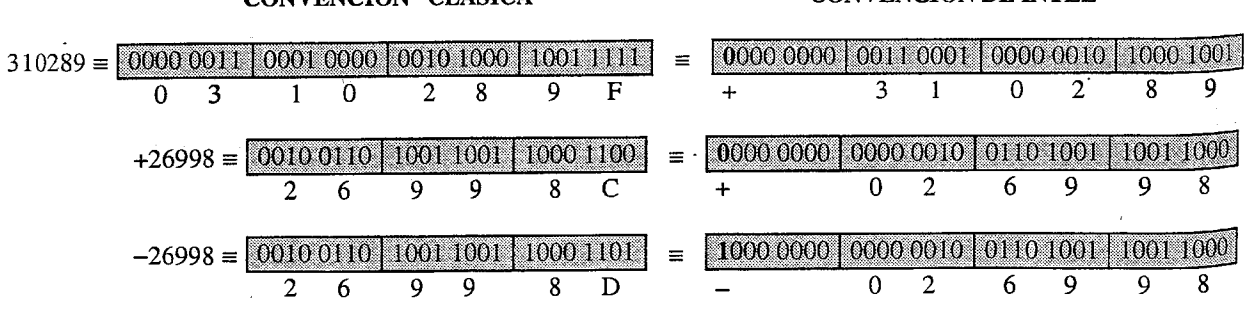
El término *"empaquetado"* se refiere a una representación numérica con uno o más bytes, siendo que en los dos cuartetos de cada byte están codificados en BCD dos dígitos decimales.

En contraposición en BCD *"desempaquetado"* cada dígito decimal ocupa un byte, como ser 3 = 00000011

Existe una representación "clásica" basada en los 4 bits por dígito del 0 al 9 del código BCD, con caracteres adicionales para indicar signo mediante otros 4 bits extras de las combinaciones restantes (1010 a 1111, de A a F) que irán en el extremo derecho. Los números positivos pueden terminar en *cualquiera* de las combinaciones: 1100 (C), 1010 (A), 1111 (F) ó 1110 (E). Para los negativos se pueden usar indistintamente 1011 (B) ó 1101 (D)

Los procesadores de Intel™ para codificar signo agregan un byte extremo izquierdo, cuyo primer bit (0 ó 1) indica el signo. Los siete bits siguientes pueden ser de valor cualquiera (elegidos ceros en los ejemplos siguientes)

þ **Ejemplos:**



[2] La suma de dos dígitos decimales a lo sumo es 19, cuando se suma 9 + 9 + 1 (1 de acarreo de la posición anterior)

EJERCICIO INTEGRADOR DE CONOCIMIENTOS PARA INTEGERS

Este ejercicio desarrollado en la figura A1.20 ha sido pensado para ver las similitudes y diferencias entre el procesamiento de variables definidas como "Magnitudes" (figura A1.15 del Apéndice 1) y las definidas como "Integres".
Por tal motivo se han elegido las mismas direcciones que en la fig. A1.15, y los mismos valores de los datos, salvo el signo.

Un programador ha desarrollado para variables que son *integers*, un programa en un cierto lenguaje de alto nivel "X", cuyas sentencias tipeó desde el teclado de un computador. Para una porción del programa ha tipeado la sentencia que más abajo se indica. Se supone que luego de ella se puede escribir el valor de las variables que se ordena operar.

INTEGERS ↵ (este símbolo ↵ aparece para indicar que se pulsó "Enter")
R = P + Q – T ↵ (sentencia a traducir)
R = 130 Q = –4103 T = –4 ↵ (valores de las variables para cuando se ejecute el programa en código de máquina.
↑ ↑ (las flechas señalan que se hizo un espacio mediante la barra espaciadora)

1. Representar cómo quedan en memoria los caracteres tipeados, a partir de la dirección 0100.
2. Si posteriormente el programador llama al programa compilador del lenguaje "X", indicar cómo el compilador deja traducida en memoria la sentencia R = P + Q – T en código de máquina, para un procesador de Intel. Para tal fin usar los códigos usados en el ejercicio integrador de la figura A1.15 del Apéndice 1 (**puesto que los enteros se suman y restan como si fueran naturales),** y escribirlos en memoria a partir de la dirección 03AC. Asignar a las variables R, P, Q y T las direcciones A23E, A240. A242 y A244 respectivamente.
3. Una vez que la sentencia R = P + Q – T fue traducida a código de máquina, se ordena su ejecución. Indicar, en el supuesto que se ejecute la secuencia traducida en el punto anterior, cómo la UAL lleva a cabos las sumas y restas que las instrucciones correspondientes ordenan, los valores de los flags SZVC generados, y cómo queda el registro AX luego que se ejecuta cada una de las instrucciones. También pasar a base diez cada resultado binario que está en AX, verificando que sea el valor esperado.
4. Representar luego de ejecutar la instrucción I_4, cómo queda en memoria el resultado asignado a la variable R, conforme ordena dicha instrucción.
5. Suponiendo que luego de la sentencia R = P + Q – T siga la sentencia PRINT "R=" R (o sea imprimir en decimal el valor de R hallado), el compilador la traducirá a varias instrucciones en código de máquina, una de las cuales llamará a una subrutina de impresión. Si ésta ordena que la impresora opere en modo texto, indicar cómo debe quedar codificada en memoria desde la dirección 7100, la información a enviar a la impresora, para que imprima en decimal el valor de R.

1. Al tipear las líneas de la porción de programa fuente en alto nivel anterior, cada carácter tipeado quedará en memoria en código ASCII, como aparece en el esquema de memoria que está a la izquierda de la fig. A1.20, La diferencia con la fig. A1.15 es que ahora se define INTEGERS, y que en los números negativos se tipea un signo menos delante de ellos.

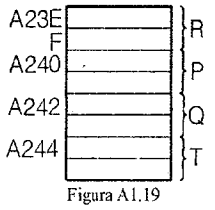


Figura A1.19

2a. Suponiendo que al ejecutarse el programa compilador, cuando éste detecta INTEGERS, una subrutina preparada para éstos, reserva dos celdas sucesivas para cada variable (fig A1.19), al igual como se hizo en la fig. A1.14 para MAGNITUDES, siendo que ahora se supone que +Nmax = 0111111111111111 = +32767

2b y **2c.** En decimal sería **R** = 130 + (–4103) – (–4) **sin poder realizar** más por menos igual a menos, ni menos por menos igual a más. Puesto que los números binarios con bit de signo se suman y restan como si fueran binarios naturales, los cod-op 0306 y 2B06 para sumar y restar binarios que ocupan dos celdas, seguirán siendo válidos para INTEGERS. Igualmente los cod-op A1 y A3, independientemente del tipo de variable ordenan transferir 2 bytes consecutivos entre memoria y AX o a la inversa. Puesto que ex-profeso para R, P, Q y T se han mantenido las mismas direcciones asignadas en la fig. A1.15 la zona de instrucciones de la fig A1.20 será la misma que la de dicha figura. A esta altura habrá terminado el proceso de traducción, realizado con las mismas consideraciones que el ejercicio integrador de la figura A1.15 del Apéndice 1.

2d. Obsérvese que en decimal la suma algebraica concreta sería R = (130) + (–4103) – (–4), pero que las instrucciones han sido traducidas respetando las operaciones que operan los números entre paréntesis, **sin que se haya realizado** "más por menos igual a menos", ni "menos por menos igual a más".
El valor decimal 130 de la variable **P** que en memoria está en ASCII como 00110001 00110011 00110000 ocupando las posiciones 0113 a 0115 (ver fig. A1.15) quedará traducido como el valor binario con bit de signo 0000000100000010_B = 130_D = 0082h que ocupará las posiciones A240 y A241 asignadas a la variable **P**. El valor decimal –4103 de **Q** que se encuentra en binario ASCII en las posiciones 0117 a 011B (4 bytes). las traducirá como el binario con bit de signo EFF9 = 1110111111111001_B = -4103_D = EFF9h que ocupará las posiciones A242 y A243. Del mismo modo, el valor –4 de T llega como 00110100 en ASCII, ocupando un byte, pasará a ser 1111111111111100_B = -4_D = FFFE en A244 y A245. El valor de la variable R recién se conocerá cuando se ejecute la secuencia **I1** a **I4** traducida.

3. Cuando se ordene ejecutar el programa que el compilador dejó en código de máquina en memoria, y la UC ejecute las instrucciones I_1 a I_4, luego de cada una de ellas, el registro AX quedará como se indica a continuación. La UAL (**que es la única que entiende binario natural**) sólo operará en las instrucciones I_2 e I_3, que ordenan sumar y restar, respectivamente, indicándose para las mismas la operación de la UAL.

Después de ejecutar I_1: AX = 0000000010000010 = 0082h que es el valor de **P** (130 en decimal)

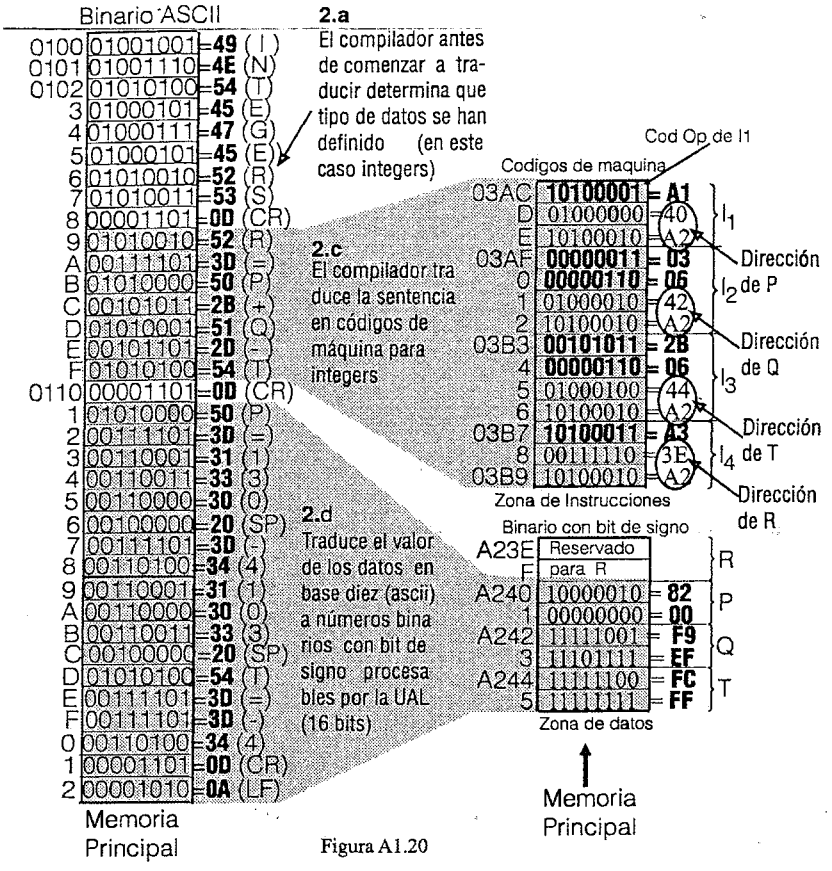


Figura A1.20

Después de ejecutar I_2:
AX = 1111000001111011 = F07Bh que es P + Q según el resultado de la UAL (–3973 en decimal).
Este valor es el resultado de sumar en la UAL: 0000000010000010 (P)
Los indicadores que genera la UAL son S=1 Z=0 V=0 C=0 +1110111111111001 (Q)
1111000001111011 = F07Bh = –3973 = (P + Q)

Después de ejecutar I_3 que ordena restar:
AX = 1111000001111111 = F07Fh que es P + Q – T = R según el resultado de la UAL (–3969 en decimal).
Este valor es el resultado de sumar en la UAL: 1111000001111011 (P + Q)
Los indicadores que genera la UAL son S=1 Z=0 V=0 C=1(resta) + 0000000000000011 (bits de T invertidos)
1
1111000001111111 = F07Fh (P + Q) – T = R

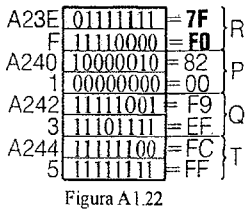


Figura A1.22

Obsérvese que al representar –4 en el paso 2d. el traductor (software) complementó el +4 binario, y en el paso 3 la UAL (hardware), para restar siempre complementa el sustraendo, por lo que el –4 antes representado en memoria, se vuelve a complementar otra vez, quedando nuevamente +4 (0000000000000011+1 = 0000000000000100 = +4.
En definitiva aritméticamente resultó que –(–4) = +4, aunque, como ordenó la sentencia, la variable T fue restada, sin importar que su valor sea negativo o positivo, y eso fue lo que se hizo en el proceso tratado.

Después de ejecutar I_4 el valor de AX no cambiará, pues I_4 ordena que una copia de AX pase a memoria:
AX = 1111000001111111 = F07Fh

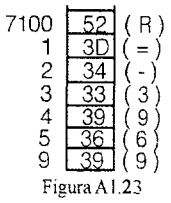


Figura A1.23

4. La ejecución de I_4 ordena que en la dirección A23E y en la siguiente (asignadas a R) copiar el valor de AX, por lo que luego de ejecutarse I_4 en memoria se tendrá lo indicado en la figura A1.22

5. En modo texto, la subrutina de impresión dejará en posiciones sucesivas de memoria los caracteres a imprimir, cada uno codificado en ASCII. Previamente dicha subrutina interpretará la orden de impresión (en este caso R = valor de R), siendo que el valor de R que está en A23E y A23F deberá pasarlo a dígitos decimales codificados en ASCII. Esto es, determinará que el valor de R, que es 1111000001111111 = –3969, el cual en ASCII resultará 00111101 00110011 00111001 00110110 00111001, como aparece en memoria.
Antes de estos códigos aparecen los códigos de R y de =, como se indica en la figura A.23

6. Cuando se ejecute mediante el Debug –como se hizo a continuación de la fig. A1.15– la secuencia de instrucciones **I1** a **I4**, se verificará que los contenidos de AX después de ejecutar cada instrucción sean los calculados más arriba y el valor de los flags que genera la UAL, siendo que NC es C=0, CY es C=1, NZ es Z=0, ZR es Z=1, NV es V=0, OV es V=1, PL es S=0, NG es S=1.

EJERCICIO INTEGRADOR DE CONOCIMIENTOS PARA REALES

Un programador ha desarrollado para variables que son ***reales***, un programa en un cierto lenguaje de alto nivel "X", cuyas sentencias ha tipeado desde el teclado de un computador. Para una porción del programa ha tipeado la sentencia que más abajo se indica. Se supone que luego de ella, se puede escribir el valor de las variables que dicha sentencia ordena operar.

REALES ↵
R = P + Q − T ↵

P = −130,5 Q = 4103,75 T = 4 ↵
(las flechas indican que se hizo un espacio mediante la barra espaciadora)

1. Indicar cómo quedan en memoria los caracteres tipeados, a partir de una dirección dada
2. Si posteriormente el programador llama al programa compilador del lenguaje "X", indicar cómo el compilador deja traducida la sentencia R=P+Q−T en código de máquina, para un procesador de Intel, trabajando en punto flotante, y escribirlos en memoria a partir de la dirección 20D1. Asignar a las variables R, P, Q y T las direcciones DC15, 2003, A200, y BC08, respectivamente.
3. Si una vez que todo el programa en el lenguaje "X" fue traducido a código de máquina (y luego "link editado" para agregarle subrutinas), se ordena ejecutarlo ("run"), indicar cuando se ejecute la secuencia traducida en el punto anterior, cómo el coprocesador lleva a cabo las sumas y restas que las instrucciones correspondientes ordenan, y cómo queda la cima de la pila que el coprocesador opera, luego de que se ejecuta cada una de las instrucciones de dicha secuencia. También pasar a base diez cada resultado binario que está en la cima de la pila, verificando que sea el valor esperado
4. Indicar luego de ejecutar la instrucción $\mathbf{I_4}$ cómo queda en memoria el resultado asignado a la variable **R**, conforme ordena dicha instrucción.
5. Suponiendo que luego de la sentencia R = P + Q − T siga la sentencia **PRINT "R=" R** (o sea imprimir R = valor hallado de R en decimal), el compilador la traducirá a varias instrucciones en código de máquina, una de las cuales llamará a una subrutina de impresión. Si ésta ordena que la impresora opere en modo texto, indicar cómo debe quedar codificada en memoria la información a enviar a la impresora, para que imprima en decimal el valor de R.
6. Ejecutar mediante el Debug la secuencia de instrucciones determinadas en el punto 2, verificando que los resultados esperados coincidan con los calculados en el punto 3.

1. No hay grandes diferencias con la concreción de este punto en el ejercicio anterior para enteros.
2. El compilador para el lenguaje "X" recorre la zona de memoria donde quedó codificado en ASCII el programa en lenguaje "X". Así va identificando cada sentencia y los datos que ella ordena operar, a fin de traducirlos en instrucciones (códigos) de máquina que el microprocesador pueda ejecutar, y códigos de datos que el coprocesador pueda operar.

2a. con esto fines debe determinar el tipo de variables (en este caso reales) que las sentencias ordenan procesar, para llevar a cabo las traducciones a instrucciones para procesar punto flotante, y códigos de datos correspondientes a éstas. Entonces, el compilador antes de traducir leerá en memoria que en ASCII está escrito "REALES".

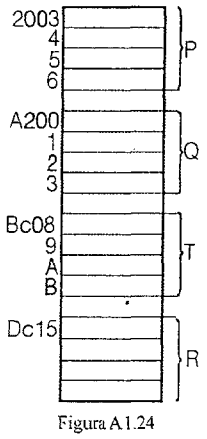


Figura A1.24

2b. el compilador reservará 4 bytes de memoria para las variables R, P, Q y T, (si fueran en doble precisión" reservaría 8 bytes) en las direcciones de memoria indicadas (figura A1.24).

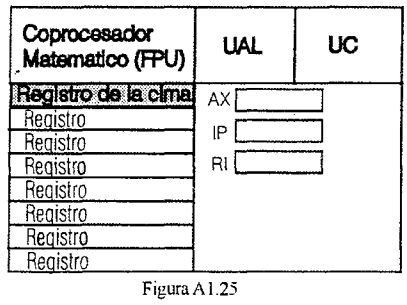


Figura A1.25

2c. Una vez asignada a cada variable una dirección de localización y las tres siguientes, el compilador traducirá R = P + Q − T en instruíciones cuyos códigos de máquina son indicados en la figura **A1.26**. Tales códigos suponen, que el coprocesador matemático (FPU) -hoy integrado al microprocesador- está ligado a una pila de registros sin denominación individual (fig. A1.25). Para la traducción operaremos con el registro que está en la cima de la pila de la misma forma que lo hicimos con AX en los ejercicios para magnitudes (fig. A1.15 del Apéndice 1 y la fig A1.20 anterior). La instrucción $\mathbf{I_1}$ (código **D906**) ordenará llevar el dato **P** que está en 2003 a la cima de la pila. $\mathbf{I_2}$ (código **D806**) ordena sumar contra la cima el dato **Q** que está en A200, y el resultado guardarlo en la cima. $\mathbf{I_3}$ (código **D826**) ordena restar contra la cima el dato **T** que está en BC08. $\mathbf{I_4}$ (código **D916**) ordena enviar una copia de la cima hacia D015, donde está **R** en memoria. La instrucción $\mathbf{I_0}$ (código **DBE3**) habilita el funcionamiento del coprocesador, **único que entiende binario que codifica punto flotante.**

2d. Una subrutina traductora preparada para "REALES" traducirá cada valor de variable decimal escrita en ASCII en el número en punto flotante de 32 bits equivalente. Así, el valor decimal −1305 de la variable **P** que en memoria está en ASCII quedará traducida en punto flotante 11000011000000101000000000000000 = **C3028000** = $-130{,}5_D$ que ocupará las 4 posiciones asignadas a la variable **P**. El valor decimal −4103,75 de **Q** que el compilador encuentra en binario ASCII en las traducirá como el número en binario punto flotante como 11000101000000000111111000000000 = **C5803E00** = $-4103{,}75_D$, que ocupará las cuatro posiciones asignadas a la variable **Q**. Del mismo modo, el valor 4 de T que al compilador llega como 00110100 en ASCII, ocupando un byte, pasará a ser $01000000100000000000000000000000_B$ = **40800000** = 4_D que el compilador escribirá en las cuatro direcciones asignadas a **T**. Para la variable **R**, cuyo valor recién se conocerá cuando se ejecute el programa que se está traduciendo, el compilador le reserva las 4 posiciones indicadas.

3. Cuando se ordene ejecutar el programa que el compilador dejó en código de máquina en memoria, y la UC ejecute las instrucciones I_1 a I_4, luego de cada una de ellas, la cima de la pila quedará como se indica a continuación. El coprocesador operará las instrucciones I_2 e I_3, que ordenan sumar y restar, respectivamente, de la forma que se indica más abajo.

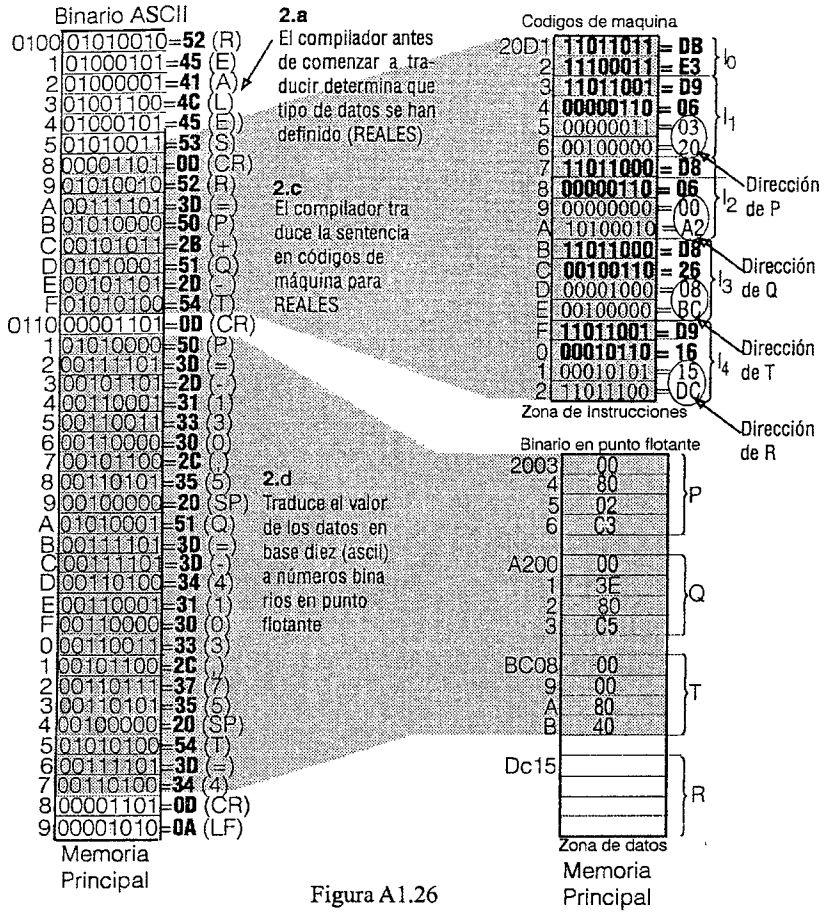


Figura A1.26

Después de ejecutar I_1:
Cima = 11000011000000101000000000000000 = C3028000 = $-130{,}5_D$

Después de ejecutar I_2:
Cima = 11000101100001000101001000000000 = C5845200, que es P + Q = $-4234{,}25_D$

Este valor es el resultado de sumar en el coprocesador como sigue, conforme se vio en la sección N8, para sumar números en FP (punto c) se debe igualar (desplazando la coma) el exponente menor al mayor:

P = 11000011000000101000000000000000 = $1{,}00000101 \times 10^{111}$ = $0{,}0000100000101 \times 10^{1100}$
Q = 11000101100000000011111000000000 = $+\ 1{,}00000000011111 \times 10^{1100}$
$1{,}00001000101001 \times 10^{1100}$

P + Q = $1{,}00001000101001 \times 10^{1100}$ = 11000101100001000101001000000000 = C5845200

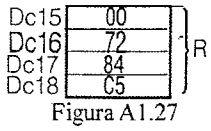


Figura A1.27

Después de ejecutar I_3
Cima = 11000101100001000111001000000000 = C5847200, que es P + Q –T = $-4238{,}25_D$
El coprocesador procede para la resta, de manera semejante a la suma, primero igualando los exponentes y luego restando mediante el complemento del sustraendo. Después de ejecutar I_4 el valor de la cima no cambiará, pues I_4 ordena que una copia de la cima pase a memoria.
Cima = 11000101100001000111001000000000 = C5847200

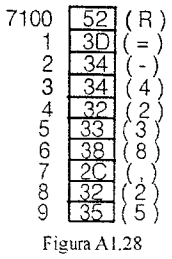


Figura A1.28

4. I_4 ordena que en la dirección D015 y en las tres siguiente (asignadas a R) copiar el valor de la cima, por lo que luego de ejecutarse I_4, en memoria se tendrá como indica la figura 1.27

5. En modo texto, la subrutina de impresión dejará en posiciones sucesivas de memoria los caracteres a imprimir, codificados en ASCII. Previamente dicha subrutina interpretará la orden de impresión (en este caso R = valor de R), siendo que el valor de R que está en D015 y siguientes deberá pasarlo a dígitos decimales codificados en ASCII. Esto es, determinará que el valor de R, que es 11000101100001000111001000000000 es el –4238,25, el cual en ASCII resultará como se indica en la figura A1.28. Antes de estos códigos aparecen los códigos de R y de = , como se indica en dicha figura.

6. Como en las unidades 1 y 3 indicaremos en letra itálica los símbolos que se tipean, y con ↵ la acción de pulsar la tecla Enter. En negrita aquello que interesa ver en la UCP a los fines del ejercicio, y en menor tamaño de número o letra, el valor que puede ser distinto de una PC a otra.

C:\> *DEBUG* ↵
– *E 2003* ↵
309D:2003 1F.*00* 26.*80* 7B.*02* 1F.*C3* ↵ Así se ha escrito P = C3028000 en las direcciones 2003 a 2006
En 2003 se supuso que existía 1F, 26 en 2004, etc.El valor 309D implica una dirección de referencia, que en general será XXXX
– *E A200* ↵
309D:A200 3F.*00* E6.*3E* AB.*80* FF.*C5* ↵ Así se ha escrito Q = C5803E00 en las direcciones A200 a A203

– *E BC08* ↵
309D:BC08 33.*00* EF.*00* AA.*80* 22.*40* ↵ Así se ha escrito T = C5803E00 en las direcciones BC08 a BC0B

Escritura en memoria de las 5 instrucciones a ejecutar
– *E 20D1* ↵
309D:20D1 35.*DB* 00.*E3* 00.*D9* AF.*06* 35.*03* 00.*20* 00.*D8*
309D:20D8 FF.*06* DE.*00* 68.*A2* AA.*D8* 95.*26* EE.*08* 38.*BC* 40.*D9*
309D:20E0 *67.16 35.15* E7.*D0* ↵
–*R IP*↵
IP 0100 En general en IP puede haber XXXX
: *20D1* ↵ Así se ha escrito 20D1 en el registro IP para que cambie el valor 0100 que tenía por 20D1, que es donde se ha escrito la instrucción DBE3 en el paso anterior, como se indica en la unidad 1.
– *R* ↵
AX=0000 BX=0000 CX=0005 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=309D ES=309D SS=309D CS=309D **IP=20D1** NV UP EI PL NZ NA PO NC
309D:**20D1** **DBE3** **FINIT**

Este paso tiene como fin verificar que todo está en orden antes de ejecutar la instrucción. Como ser, que la próxima instrucción a ejecutar es DBE3
– *T* ↵ (Orden de ejecución de la instrucción DBE3)
AX=0000 BX=0000 CX=0005 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=309D ES=309D SS=309D CS=309D **IP=20D3** NV UP EI PL NZ NA PO NC
309D:**20D3** **D9060320** **FLD DWORD PTR [2003]**

– *T* ↵ (Orden de ejecución de la instrucción D9060320)
AX=0000 BX=0000 CX=0005 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=309D ES=309D SS=309D CS=309D **IP=20D7** NV UP EI PL NZ NA PO NC
309D:**20D7** **D80600A2** **FADD DWORD PTR [A200]**

– *T* ↵ (Orden de ejecución de la instrucción D800600A2)
AX=0000 BX=0000 CX=0005 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=309D ES=309D SS=309D CS=309D **IP=20DB** NV UP EI PL NZ NA PO NC
309D:**20DB** **D82608BC** **FSUB DWORD PTR [BC08]**

– *T* ↵ (Orden de ejecución de la instrucción D82608BC)
AX=0000 BX=0000 CX=0005 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=309D ES=309D SS=309D CS=309D **IP=20DF** NV UP EI PL NZ NA PO NC
309D:**20DF** **D91615DC** **FST DWORD PTR [DC15]**

– *T* ↵ (Orden de ejecución de la instrucción D91615D0)
AX=0000 BX=0000 CX=0005 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=309D ES=309D SS=309D CS=309D IP=20E3 NV UP EI PL NZ NA PO NC
Este tercer renglón no interesa

Siendo que el Debug sólo muestra los registros vinculados a la UAL (que para estas instrucciones no se usan) y a la UC, y que los registros ligados al coprocesador son "invisibles", para todas las instrucciones ejecutadas el único registro visible que ha cambiado es el IP, permaneciendo sin variar los restantes, incluyendo los flags.

– *E DC15* ↵ (Examinar las posiciones DC15 a DC18 correspondientes a la variable R)
309D:DC15 *.72 .84 .40*
309D:DC18 *.C5*

C5847200 = 11000101100001000111001000000000 = –1,00001000111001000000000x10^{1100} = –1000010001110,01 = –4238,25 Se verifica que el valor de R que está en FP es el esperado.

Dado que el registro de la cima del coprocesador es "invisible" para el Debug, si se quiere conocer su contenido, se debe copiar el mismo a la memoria, mediante la instrucción $\mathbf{I_4}$ de código **D91615DC** y luego examinar la memoria mediante *E D015* como se hizo más arriba. Por lo tanto, si luego del código de $\mathbf{I_1}$ e $\mathbf{I_2}$ se intercala el código de $\mathbf{I_4}$ (D91615DC), o sea si la sucesión de instrucciones en memoria es DBE3 ($\mathbf{I_0}$) D9060320 ($\mathbf{I_1}$) D91615DC ($\mathbf{I_4}$) D80600A2 ($\mathbf{I_2}$) D91615DC ($\mathbf{I_4}$) D82608BC ($\mathbf{I_3}$) D91615DC ($\mathbf{I_4}$) se podrá conocer -examinando la memoria los sucesivos contenidos de la cima mediante *E DC15*- si en la cima está P; si luego está P + Q; y por último, como ya está planeado en la secuencia de la fig A1.26, si en R está P + Q – T.

- Constitución de la Sociedad. Objeto. Inscripción. Accionistas fundadores. Cómo ingresaron Williams y Mercado. Red Cell / WPP/ Williams y Mercado: relación y función que cada uno cumplía y cumple.

- El 31 de mayo de 2004, Dylan William y Martin Mercado ("Williams" y "Mercado", respectivamente) remitieron una carta a WPP Group USA ("WPP") en la que comunicaron su disposición a suscribir un Joint Venture Agreement y se establece el pago de una multa por el monto de AR$900.000 para el supuesto que ocurran determinados supuestos especialmente previstos. Dicha carta fue consentida por WPP (Anexo I). Entre quién y quién era el contrato? Finalidad del mismo? Se firmó? Qué relación tienen Red Cell SA, WPP y el Joint Venture Agreement?

- Williams y Mercado comenzaron a trabajar en relación de dependencia con Red Cell S.A. (la "Sociedad"). Williams tenía a su cargo el puesto de Gerente General y Mercado el de Gerente General Creativo, conforme fuera ratificado en la ~~el acta de~~ asamblea No. 3 de la Sociedad celebrada el ~~del~~ 30 de octubre de 2007 (la "Asamblea") (Anexo II). Cómo estaba formado el órgano de administración de Red Cell?.

- Al considerarse ~~En~~ el punto 7 del Orden del Día de la Asamblea, se resolvió destinar parte de los aportes realizados por el accionista Berkeley Square Holding B.V. ("Berkeley") a aumentar el capital de la Sociedad según el siguiente detalle: (i) Berkeley: 80%; (ii) Mercado: 10%; y (iii) Williams: 10%. Los aportes eran todos de Berkeley? O también aportaron Mercado y Williams? Como fue que suscribieron e integraron el aumento de capital? Así fue como pasaron a ser accionistas? Por qué se les ofreció ser accionistas? Fue como "premio" por su desempeño o recompensa por algo?

- El aumento de capital resuelto en la Asamblea fue inscripto ante IGJ el [*], con lo que queda evidenciado que es falsa la afirmación contenida en la denuncia respecto a ..... (*"Si bien dicha inscripción tiene fecha de diciembre de 2007 es claro y se nota a simple vista que fue hecha el mismo día que la inscripción anterior dado que se trata de la misma letra" ?? no se entiende. Cual es la inscripción anterior?*)

- La decisión asamblearia es obligatoria para los accionistas y directores desde que se la adopta salvo impugnación art. 251 o derecho de receso. La ~~mencionada~~ inscripción ante IGJ del aumento de capital no solo da fecha cierta a la suscripción, sino que da a esta publicidad y oponibilidad a terceros. Williams conociendo dicha decisión no hizo nada. Nótese que no solo revestía el carácter de accionista sino que además era Presidente y como tal se encontraba no sólo facultado sino que hasta obligado a impugnar.

- Por otra parte, la ~~referida~~ suscripción e integración del aumento en los términos indicados fue consentida por Williams quien no impugnó la decisión y suscribió actas posteriores [*] (Anexo III) ~~lo que implica que tuvo conocimiento de la Asamblea en cuestión~~. Williams no puede invocar su propia torpeza para continuar sosteniendo sus argumentos. Debe recordarse al respecto la teoría de los actos propios .....(PONER LETRA DE TELECOM)

- "Haber incluido al Sr. Williams como accionista en un 10 % de Red Cell SA. recién para el tiempo de su disolución (lo cual lleva afrontar un eventual quiebre de la sociedad en su porción accionaria)": No fue así. 30/10/07 ingresa como accionista y disolución y liquidación es resuelta el xxxxxx. Por otra parte ni la Sociedad ni sus accionistas están obligados a no disolverla o liquidarla, es una posibilidad y derecho que otorga la ley y nada dice en contrario el estatuto de Red Cell.

- El 25/6/08) se realizó una auditoría externa por parte del estudio Jorge Andrés Thomas y Asociados que no menciona al Sr. Williams como accionista, sino únicamente a Berkeley Square Holding BV (80 %) y al Sr. Mercado (20 %) ???

- Para concretar tal situación, expuso que se habría concretado una emisión de acciones en contra de la reglamentación de la sociedad, razón por la cual también los sucesos podrían encuadrarse dentro del art. 301 del Código Penal, (actos contrarios a la ley y el estatuto consentidos por el director, gerente, administrador, liquidador de una sociedad): Diría por qué no se emitieron acciones en contra de la reglamentación de la sociedad (fundaría con ley de sociedades y estatuto de Red Cell). Por otra parte y aún cuando supusiéramos que las decisiones eran contrarias a la ley, estatuto o reglamento, el remedio está establecido en la LSC: acción de impugnación, que no sólo la podia iniciar como accionista sino también como Presidente. Hay una via idónea que es la societaria. Se trata de un conflicto societario que escapa a la justicia penal.

- Asimismo relató que los administradores de la sociedad -titulares del paquete mayoritario que representaba el estudio Brons & Salas- derivaron las principales cuentas de la firma a otra sociedad de ellos (Young & Rubicam), donde comenzaron a desempeñarse los creativos de Red Cell (administración fraudulenta -art. 173 inciso 7° del CP-): SE OLVIDA QUE EL ADMINISTRABA?!!! Se refiere a administradores de la Sociedad y el Presidente era él.

- En el intercambio de emails que se acompañan como Anexo IV, surge que Williams no solo tenía conocimiento de que las actividades de la Sociedad serían asumidas por otra empresa del Grupo WPP, sino que estaba a cargo de su negociación. NO SE SI LO PONDRÍA TAN ASI. No estaríamos "reconociendo" lo que el invoca.

- Es falso que Williams no tuviese conocimiento respecto de la decisión de Berkeley de disolver la Sociedad conforme surge de la CD que se adjunta como Anexo V, en la que se le solicita convocar a asamblea para considerar, entre otros, dicho punto. Precisamente a dichos fines y en su carácter de Presidente fue que se le solicitó proceda a convocar a asamblea para considerar la cuestión.

- Ante la ausencia del Presidente de la Sociedad (ausencia o negativa?), Berkeleys procedió a solicitar la convocatoria ante la IGJ, organismo que hizo lugar al pedido de convocatoria a asamblea efectuado por Berkeley (Anexo VI) en tanto los requisitos de admisibilidad de dicha medida se encontraban por demás reunidos. ~~y~~ La asamblea fue celebrada el 31 de marzo de 2009 (Anexo VII). Del

INDICE ALFA-TEMATICO DE LA UNIDAD 1

texto de esta última surge que es incorrecta la objeción de Williams respecto a que es una situación extraña que se haya consignado que la disolución fue aprobada por la totalidad de los accionistas presentes. Esto fue así ya que Williams no notificó asistencia a la asamblea y su presencia fue en el acto se debió a su carácter de Presidente de la Sociedad (no como accionista). NO se entiende bien. Qué sostiene Williams? Por qué dice que es extraño que se haya consignado que fue aprobada por la totalidad de accionistas presentes?

- En cuanto al supuesto perjuicio patrimonial derivado de la calidad de accionista de Williams, es evidente que este último no tiene claro que su responsabilidad patrimonial se limita al monto integrado para la suscripción de las acciones (que ni siquiera fue desembolsado por él conforme surge de la asamblea del 30 de octubre de 2007. No se si lo diría tan así porque no es precisamente por ello que el niega su carácter de accionista?. Otra prueba de que no deberá hacerse cargo de ninguna deuda es el nuevo aumento de capital dispuesto mediante asamblea del 19 de mayo de 2009 en la cual Berkeley asume la obligación de integrar el monto necesario para hacer frente a las obligaciones de la Sociedad (Anexo VIII).

- En relación a este hecho habría que delimitar la concreta conducta que habría llevado a cabo el Sr. Mercado en dicha inscripción, la cual no fue puntualizada en la denuncia: MERCADO no incluyó o dejó de incluir a Williams como accionista. Ello fue producto de lo resuelto en la Asamblea cuyas decisiones todos los accionistas consintieron.

- Finalmente también indicó que la "nueva sociedad" Young & Rubicam, atendía clientes en competencia con otros permanentes de la firma Red Cell, generando entonces un supuesto quebranto y desviando la clientela de está última empresa a favor de Young & Rubicam (concurrencia desleal -art. 159 del CP.-): HABRIA QUE DECIR ALGO. VER QUÉ VA A DECIR BRONS al respecto.

BIBLIOGRAFIA PRINCIPAL

Angulo J., Funke E, *386 y 486 Microprocesadores avanzados de 32 bits* . Ed. Paraninfo. 1992
Brey B. *The Intel Microprocessors* Fifth Edition Prentice Hall, Inc 2000
BYTE, The Magazine of Technology Integration. Colecciones 1990-1996
Ginzburg M *Introducción a las Técnicas Digitales con Circuitos Integrados* Biblioteca Técnica Superior 9ª ed. 2005
Hennessy J., Patterson D. *Arquitectura de Computadores. Un enfoque cuantitativo* Mc Graw Hill 1993
Hennessy J., Patterson D *Computer Organization & Design. The Hardware/Software Interface* Morgan Kaufmann Publishers, Inc
Intel *Pentium Pro Family, Developers Manual* 1997
Intel *IA 32 Intel Architecture* 2003 - (P6 Family to Pentium 4)
Intel: www.intel.eom/technology/hyperthread//index.htm (Papers sobre hyperthreading)
Messmer Hans-Peter *The Indispensable PC Hardware Book* Second Edition-Addison-Wesley 1995
Norton P. *Inside the PC* Brady Publishing 1993
Stallings W. *Computer Organization and Architectures.* Fifth Edition Me Millan Publishing Co. 2000
Tanenbaum A. *Structured Computer Organization* Fourth Edition Prentice Hall International, Inc. 1999
Tischer M. *PC Intern, System Programming* Data Becker-Abacus, 1995
Triebel W, *The 80386DX Microprocessor, Hardware, Software & Interfacing* E d. Prentice Hall 1992