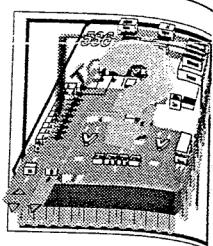


## 1.14

# 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 cumplen?*

Si bien con importantes mejoras en la velocidad de procesamiento, la mayoría de los procesadores actuales procesan en base al esquema de la figura 1.7, denominado “modelo de Von Neumann”<sup>1</sup>, 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 subprocessos 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).

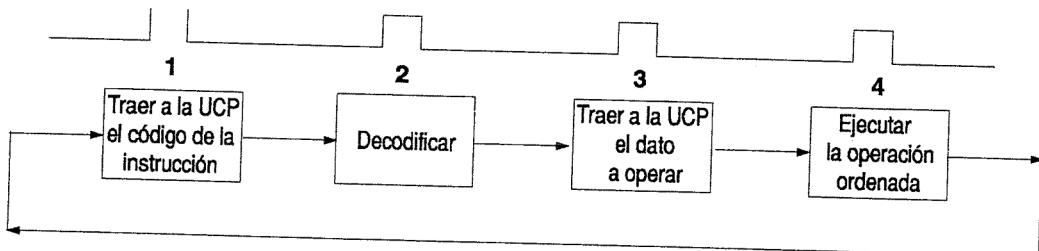


Figura 1.84

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 progresiva 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 ( $I_1$ ), y luego la siguiente ( $I_2$ ), 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”<sup>2</sup> en el procesador 8086.

<sup>1</sup> En el presente también se denomina “escalar o “secuencial”, generalizable a cualquier computador que opera en forma secuencial sobre los datos.

<sup>2</sup> 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 subprocessos que se pueden realizar en forma independiente. En una cadena de este tipo, cuando se termina un subprocesso de fabricación de una unidad (como ser el de pintura), la misma es desplazada al lugar donde se realiza siguiente subprocesso 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 subprocesso realizado sobre la unidad anterior.

De esta forma *se realizan simultáneamente todos los subprocessos* 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<sup>1</sup>) 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.<sup>2</sup>

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  $t_1, t_2, \dots$ ) 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<sup>3</sup> 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<sup>4</sup> 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”.

<sup>1</sup> Suponiendo que esta última etapa sea la que dura 5 minutos y otras mucho menos, ella determina el ritmo de lavado.

<sup>2</sup> Del mismo modo, en el lavadero citado puede requerirse un lugar entre dos subprocessos, 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 subprocesso.

<sup>3</sup> Desde la memoria caché, si está en ella (sino habrá que pedirlo a la memoria principal) o desde un registro de la UCP.

<sup>4</sup> 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”

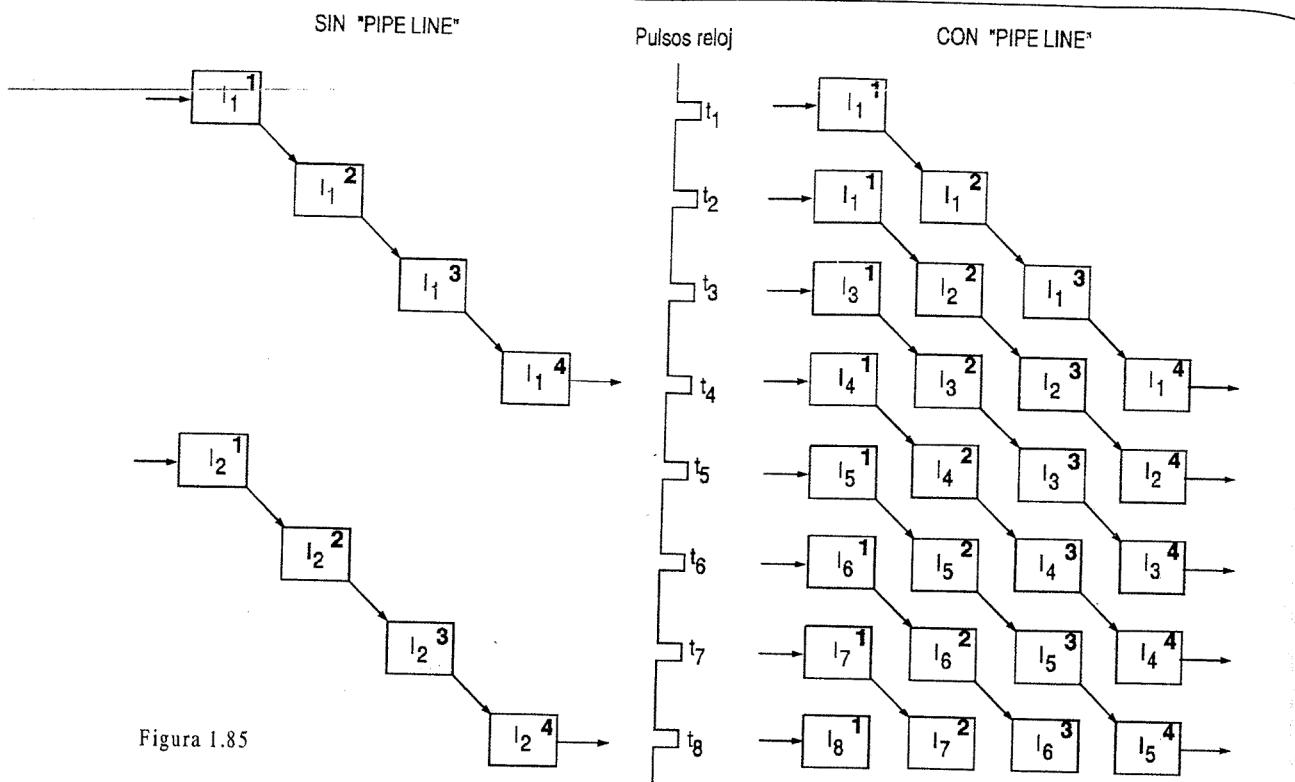


Figura 1.85

## ¿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 pueden 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<sub>1</sub>, I<sub>2</sub>, I<sub>3</sub>, I<sub>4</sub> 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 microcódigos 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.

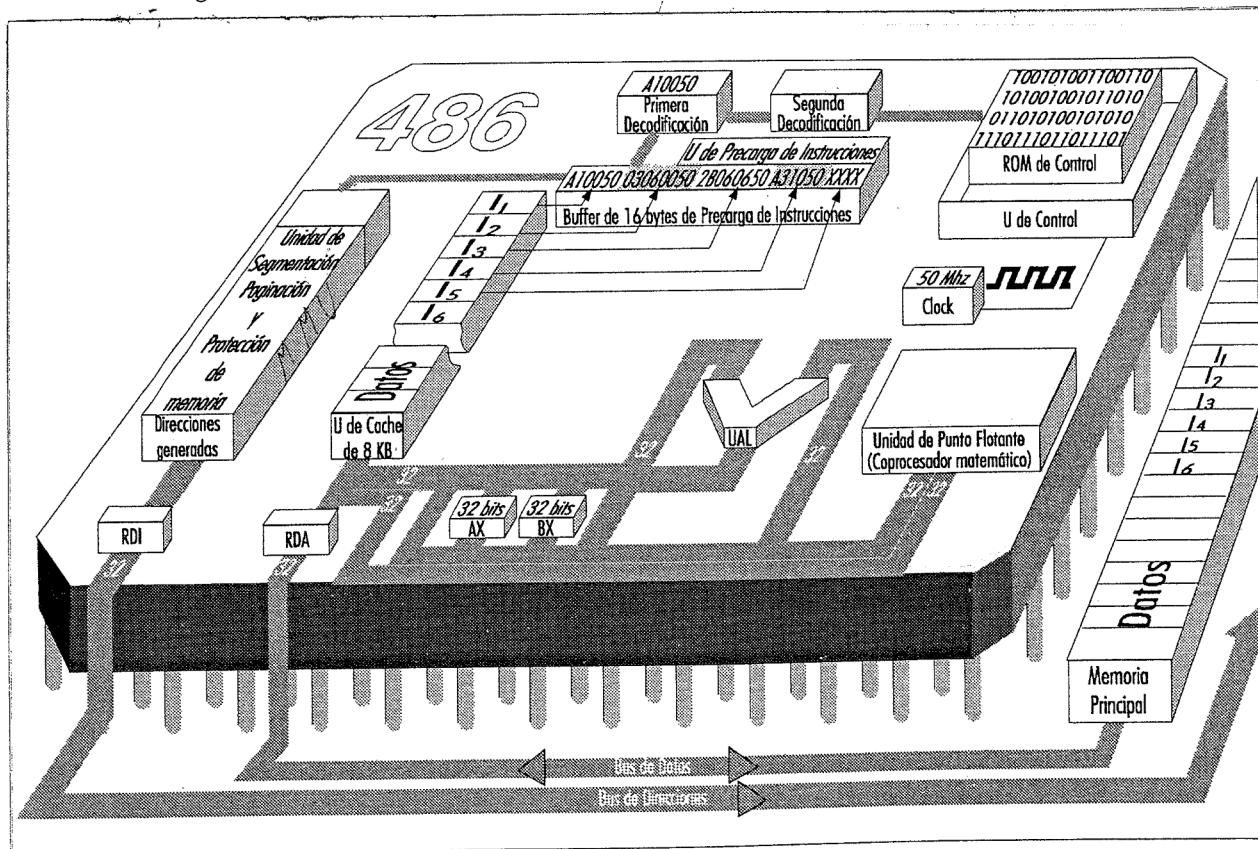


Figura 1.86

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<sup>1</sup>, 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  $I_2, I_3, I_4$ <sup>2</sup>, 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 las 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<sup>3</sup>. 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 primera 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".

<sup>1</sup> 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.

<sup>2</sup> Cuando no hay un contenido en un caché, su controlador solicita a la memoria el mismo y los que están en las direcciones siguientes.

<sup>3</sup> 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<sup>1</sup>, 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  $I_2$ ,  $I_3$ ,  $I_4$ <sup>2</sup>, 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 las 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<sup>3</sup>. 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 primera decodificación.

En la figura 1.89 se ha incluido cómo progresan 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".

<sup>1</sup> 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 él mismo, se obtendría  $I_1$  de memoria principal.

<sup>2</sup> Cuando no hay un contenido en un caché, su controlador solicita a la memoria el mismo y los que están en las direcciones siguientes

<sup>3</sup> 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.

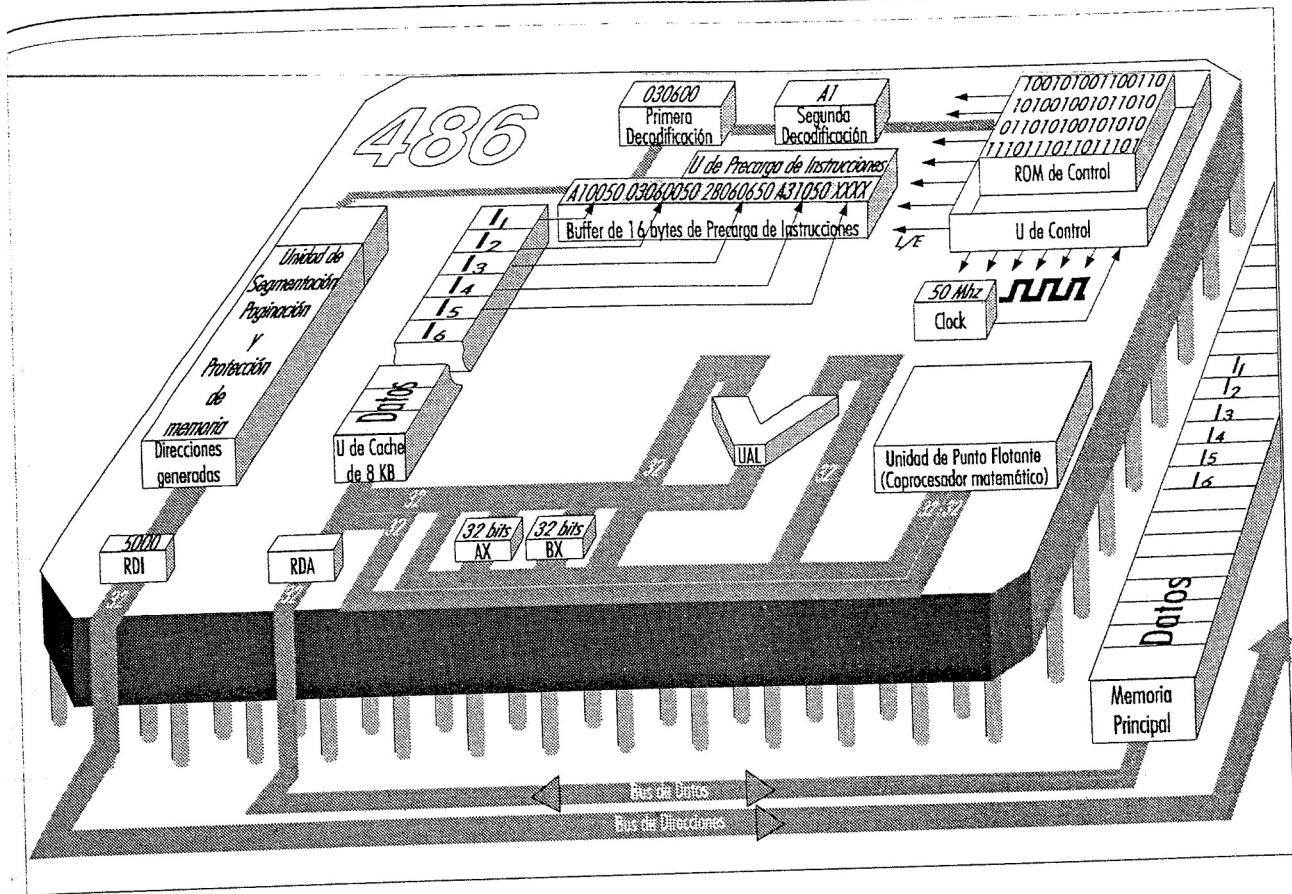


Figura 1.87

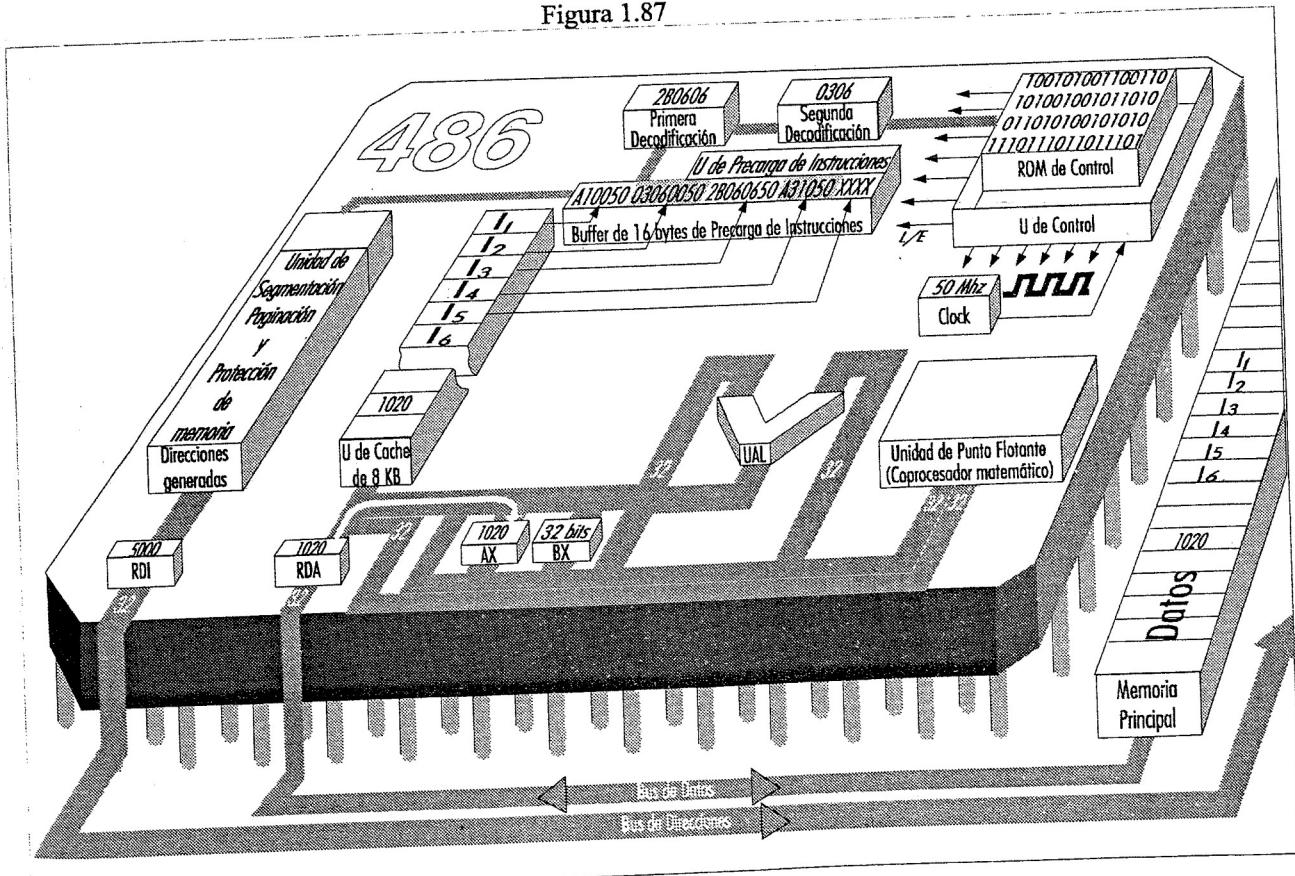


Figura 1.88

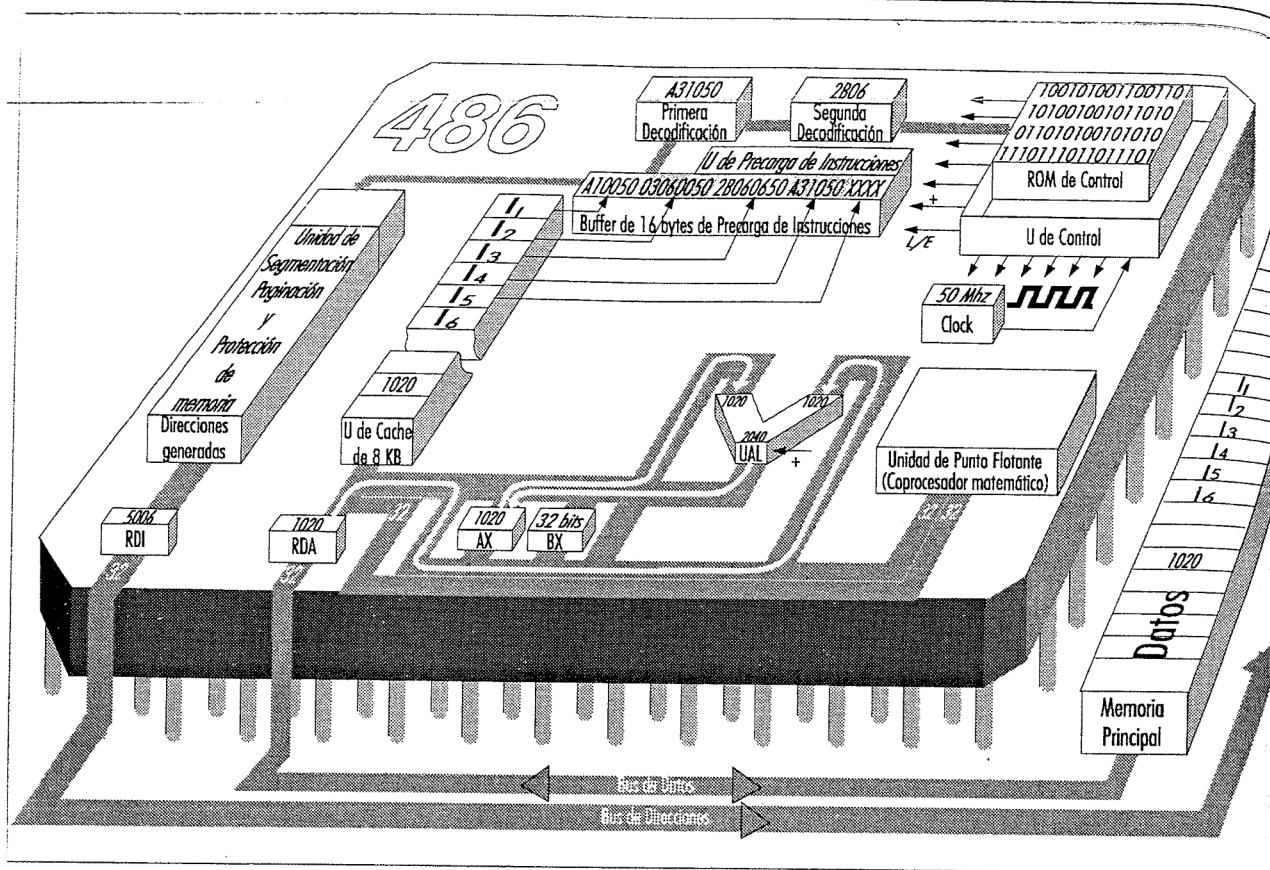


Figura 1.89

### *¿Qué tiene en común y cómo funciona 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<sub>1</sub> a I<sub>10</sub>, 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<sup>1</sup>.

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 intervenientes, qué operaciones realizarán.

Este tipo de instrucciones para enteros son las que pueden procesarse de a dos (una en cada "pipe line"). Instrucciones como I<sub>1</sub>, I<sub>2</sub> e I<sub>3</sub> (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 I<sub>4</sub>, 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

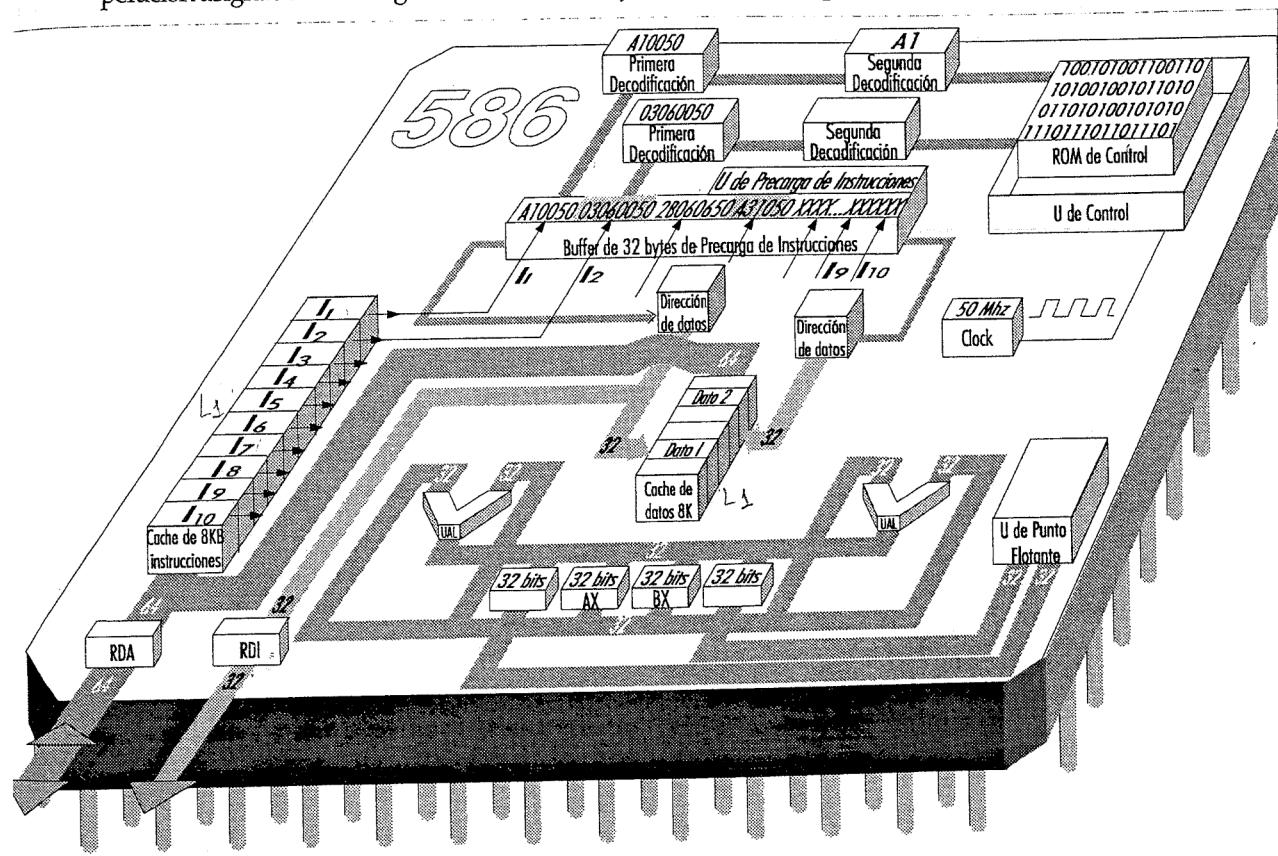


Figura 1.90

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.

<sup>1</sup> 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 exemplifica 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.

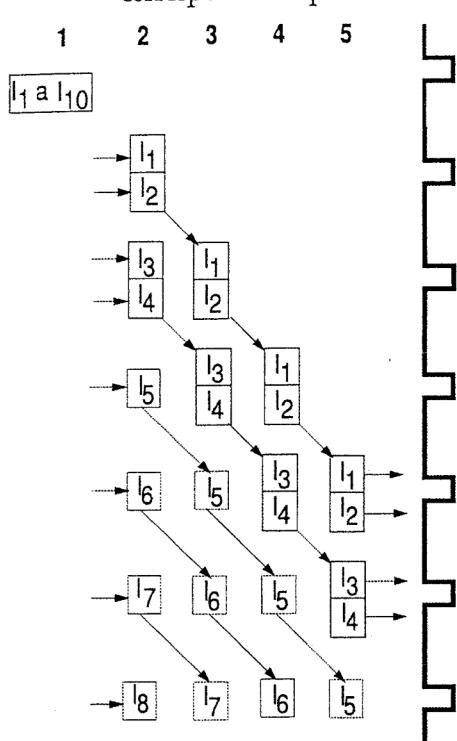


Figura 1.91

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.<sup>1</sup> Esto último se supone que no ocurre con  $I_5$  e  $I_6$ , ni con  $I_7$  e  $I_8$ , 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.

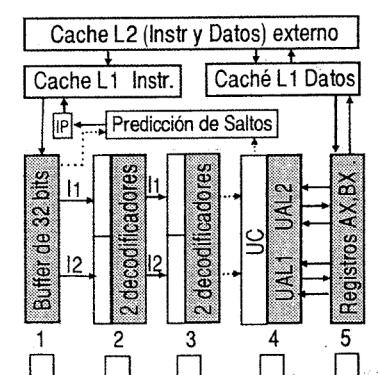


Figura 1.92

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<sup>2</sup>.

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

<sup>1</sup> 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  $I_4$ .

<sup>2</sup> 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 (códigos) 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 (códigos) 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 (*microcódigo* o *microoperación*), 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  $Z = P + P - Q$  se debe traducir a una secuencia de instrucciones de máquina como  $I_1, I_2, I_3, I_4$ . 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  $I_x$  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  $I_x$ . 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"<sup>1</sup> 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"<sup>2</sup> 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")<sup>3</sup>, 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)<sup>4</sup>. 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- constituyan 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.

<sup>1</sup> 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.

<sup>2</sup> Idem MOV CX, 2000; ADD BX, 4 CMP AX, 50

<sup>3</sup> Idem JZ 32A0 JL 0500 JMP 2800

<sup>4</sup> 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®)<sup>1</sup> ?*

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.

<sup>1</sup> 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" ( **$\mu$ ops-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  **$\mu$ ops-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  **$\mu$ ops-R**, dichos registros luego son renombrados, en correspondencia con decenas de registros (R1, R2, ... Rn) 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 temporalmente 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  $\mu$ ops-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  $\mu$ ops-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  $\mu$ ops-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  $\mu$ ops-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  $\mu$ op-R. El tercer decodificador maneja cod-ops que se traducen de 1 a 4  $\mu$ ops-R. Si se tiene una instrucción compleja que se traduce en más de 4  $\mu$ ops (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  $\mu$ ops generados en UD1 pasan en el orden correspondiente al programa a una cola de  $\mu$ ops, y las  $\mu$ ops de salto se presentan a la Unidad de Predicción Estática de Saltos (UPES) a tratar.

Estas  $\mu$ ops-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.

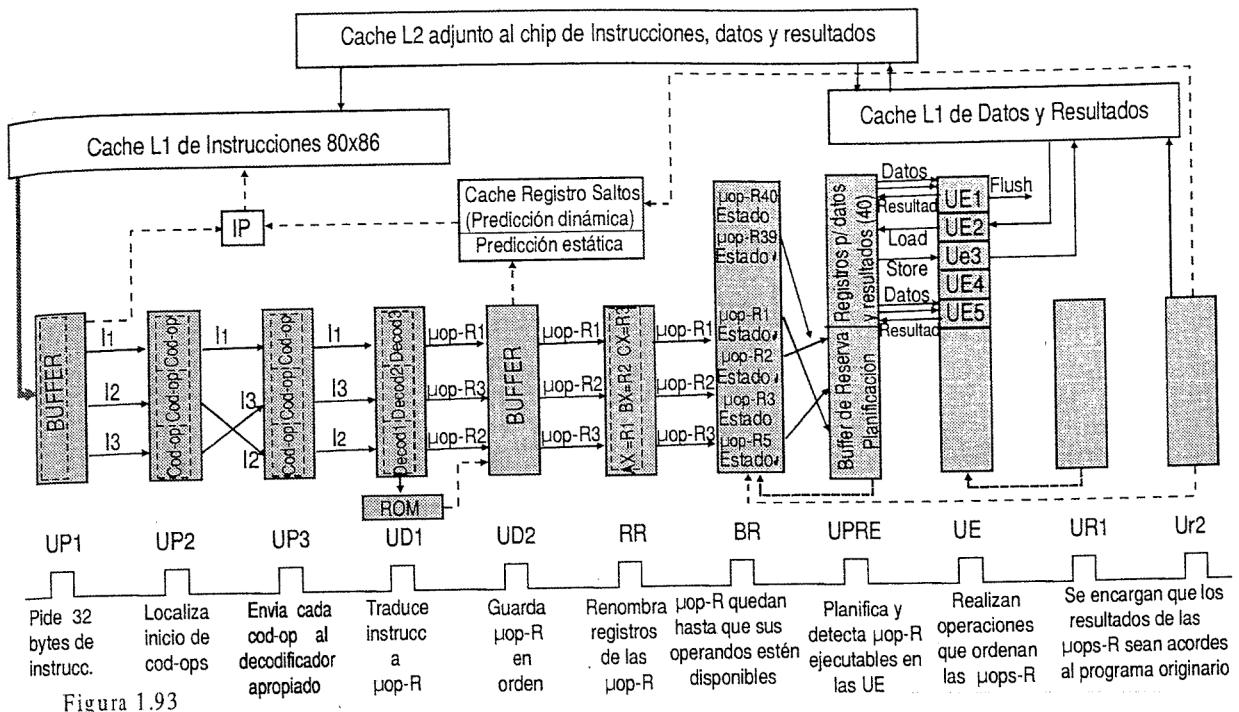


Figura 1.93

Las  $\mu$ ops-R que entraron a RR pasan ordenadas al Buffer para Suministro y Reordenamiento de  $\mu$ ops-R (**BSR**), conforme al orden que tienen en el programa las instrucciones traducidas en la decodificación de las 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  $\mu$ ops-R. Asimismo contiene los 40 registros físicos para los datos y resultados de las  $\mu$ ops-R antes citados. Junto con cada  $\mu$ op-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  $\mu$ op-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  $\mu$ op-R está 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  $\mu$ ops-R), sin importar el orden en que estaba en el **BSR**.<sup>1</sup> Si está disponible una **UE** que realiza la operación ordenada por una de estas  $\mu$ ops-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  $\mu$ ops-R cuya operación se concreta en la **UE** que corresponda, siendo que ciertos bits de cada  $\mu$ op-R ordenan que deben hacer dicha **UE**. En cada ciclo reloj sólo puede operar una **UE** por  $\mu$ op-R.

En la arquitectura P6 hay 5 posibles **UE**. Dos **UE** están dedicadas a  $\mu$ ops-R *store*, y otra **UE** para  $\mu$ ops-R *load*. La cuarta **UE** contiene -según se necesite- una UAL para enteros, circuitería (coprocesador) para  $\mu$ ops-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  $\mu$ ops-R en varias etapas del mismo cuya ejecución no debe proseguir. Cuando se ejecuta una  $\mu$ op-R de salto si su resultado no coincide con el pronosticado (ver más adelante) deben eliminarse todas las  $\mu$ ops-R que siguen a la de salto, incluidas las del buffer de reserva citado, y deben ejecutarse las  $\mu$ ops-R que correspondan. La quinta **UE** también como la anterior puede operar según se sea: a enteros (con otra UAL), a números en punto

<sup>1</sup> 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 cuando éste se soluciona. En cambio si se ejecuta fuera de orden, y el hardware 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 continuo de  $\mu$ ops-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 pueden seguir enviando instrucciones desde el decodificador. De esta forma puede darse un flujo continuo en la decodificación y en la extracción de  $\mu$ ops-R desde la **UPRE** a las **UE**. Así se puede tener capacidad de anticipación para decidir qué  $\mu$ op-R pueden ejecutarse fuera de orden.

flotante y a operandos MMX. Así es factible ejecutar hasta 5  $\mu$ ops-R juntas y 3 en promedio.

Por lo tanto, la **UPRE** evalúa cuál  $\mu$ op-R es la más conveniente para ser ejecutada en el siguiente pulso del reloj, para lo cual se determina en cada ciclo las  $\mu$ ops-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  $\mu$ ops-R ahora están ordenadas en función de las dependencias de datos y no de su orden original. Si varias  $\mu$ ops-R pueden ejecutarse, un algoritmo FIFO determina cuáles irán a las **UE**, con prioridad de las  $\mu$ ops-R de salto frente a las otras, dada su importancia en la prevención de las ramificaciones. Si se termina de ejecutar una  $\mu$ op-R, la **UPRE** actualizará el **BSR** indicando tal evento, para que dicha  $\mu$ op-R sea retirada, y la **UE** correspondiente queda disponible para otra  $\mu$ op-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 temporalmente 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  $\mu$ ops-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  $\mu$ ops-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  $\mu$ op-R ejecutada se originó en una instrucción que no debió ser ejecutada, los resultados de tal ejecución no son considerados.

Cuando todas las  $\mu$ ops-R en que se ha traducido una instrucción se han marcado como listas para ser retiradas, y se han ejecutado la(s)  $\mu$ op(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  $\mu$ op-R que corresponde ser reemplazada por otra de la cola de  $\mu$ ops-R.

Si bien las  $\mu$ op-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  $\mu$ ops-R mal ejecutadas.

A continuación, las  $\mu$ ops-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  $\mu$ ops-R (siendo que a ella entran 3 instrucciones a traducir), por lo que en cada pulso reloj se terminan de ejecutar 3  $\mu$ ops-R, lo cual se acerca a 3 instrucciones por pulso (ciclo). Es como si existieran 3 pipelines para 3  $\mu$ ops-R. La ejecución fuera de orden potencia el paralelismo interno, pues si una  $\mu$ op-R espera por un operando que tiene que generar otra  $\mu$ op-R, o por un recurso circuital para ser ejecutada, otras  $\mu$ ops-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 él 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  $\mu$ ops-R, se consigue una buena mejora en el rendimiento del procesador.

#### **Predictión 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 "dinámica" (realizada sobre la marca) 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 combinando 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, convetir, y para hacer operaciones lógicas.

El **Pentium III** (500 Mhz) presenta las extensiones "Streaming SIMD" con 70 nuevas instrucciones para mejoramiento de imágenes, sonido, video 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 descrita que le sirve de base.

Los actuales sistemas operativos y otros programas de usuario dividen la carga de trabajo en *processos* 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 límites para 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 fin sólo 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. Esto de por sí posibilita duplicar los Mhz de funcionamiento, siendo que se debe operar a esta mayor velocidad para que 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 arquitectura 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 ordenadas 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 dirección 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 las µops-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 de µops-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  $\mu$ ops-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  $\mu$ ops-R a ejecutar, éstas deben aparecer en las salidas del TC. Dichas  $\mu$ ops-R (de igual longitud) irán a la etapa de Renombramiento de Registros (RR) de donde pasan según el orden original a la "Cola de  $\mu$ ops-R", junto con la dirección de la instrucción que los originó. Lo mismo ocurre con las  $\mu$ ops-R generadas por la ROM. Esta cola oficia de buffer intermedio 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  $\mu$ ops-R de cada "thread".

Si para "HT" se debe acceder simultáneamente al TC para obtener líneas con  $\mu$ ops-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 esté 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  $\mu$ ops-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  $\mu$ ops-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 dirigir 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  $\mu$ ops-R a la UT y también a la "Cola de  $\mu$ ops-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, temporalmente 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  $\mu$ ops-R que se van a ejecutar próximamente, y cada vez que en el TC hay un fallo, la UPD traducirá en  $\mu$ ops-R las instrucciones del BL2 (*a razón de una por vez*) que irán al TC y también a la Cola de  $\mu$ ops-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  $\mu$ ops-R se inicia con una asignación ("allocation") en la etapa RR que renombra los registros 80x86 indicados en las  $\mu$ ops-R, que están en la "Cola de  $\mu$ ops-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  $\mu$ ops-R" hay  $\mu$ ops-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  $\mu$ op-R que se va a ejecutar de cada "thread" en qué registros "alias" estarán sus operandos (que pueden ser resultados de  $\mu$ ops-R ejecutadas antes), y en qué registro "alias" irá el resultado. Las secuencias de  $\mu$ ops-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  $\mu$ ops-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  $\mu$ ops-R se pueden ejecutar. Cada LP tiene su propia cola de 8 a 12  $\mu$ ops-R que debe seleccionar para enviar a las UE. *Las LP no distinguen entre las  $\mu$ ops-R de dos "threads"*. Las  $\mu$ ops-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  $\mu$ op-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  $\mu$ ops-R que se ejecutarán luego. Asimismo las  $\mu$ ops-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  $\mu$ ops-R, como el máximo de  $\mu$ ops-R que genera la UT es 3, queda limitado a este número la cantidad de  $\mu$ ops-R ejecutables por ciclo.

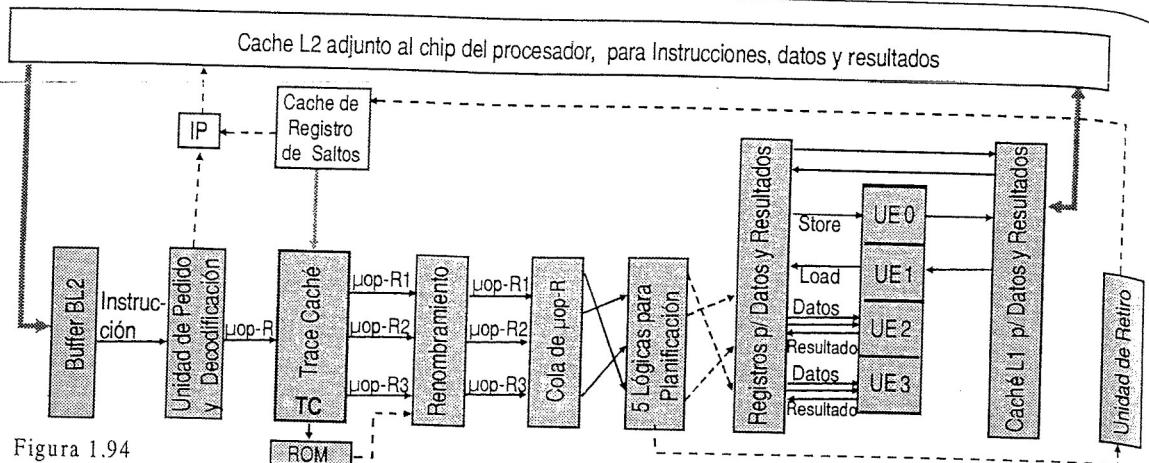


Figura 1.94

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  $\mu\text{ops-R}$  "store", y **UE1** para  $\mu\text{ops-R}$  "load", a razón de una  $\mu\text{op-R}$  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  $\mu\text{op-R}$  de salto; y en la segunda mitad de un ciclo puede operar nuevamente enteros en la UAL de dicha **UE**.

Para  $\mu\text{ops-R}$  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  $\mu\text{op-R}$  en la **UE** correspondiente, las  $\mu\text{ops-R}$  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  $\mu\text{ops-R}$  en orden en forma alternada para cada "thread".

Luego que las  $\mu\text{ops-R}$  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.