UNIDAD 6:

ARQUITECTURA AVANZADA

INTRODUCCIÓN

En las primeras unidades de la materia se estudiaron los circuitos combinacionales y secuenciales en forma aislada, luego se integraron en una estructura básica que llamamos **máquina elemental**, el hardware. También se estudió el lenguaje de máquina de la máquina elemental (software). La tarea de escribir un programa en lenguaje de máquina la realizó el alumno (utilizando los mnemónicos definidos), y la tarea de corregir y convertir el programa en una secuencia de unos y ceros y además "cargarlos" adecuadamente en una posición de memoria para luego ejecutar el mismo, también la realizó el alumno a través de la consola del operador de la máquina elemental.

También se estudió que el programa en lenguaje de máquina podía ser ejecutado directamente por el hardware (unidad de control cableada). Veremos que existe una alternativa: el programa puede interpretado por un microprograma residente en una memoria ROM dentro de la unidad de control (unidad de control microprogramada).

En la segunda parte de la unidad se estudian las generalidades de la estructura avanzada de las computadoras.

MICROPROCRAMACIÓN

A fin de introducir el concepto de microprogramación vamos a suponer que tenemos nuestra propia MAQ. A, pero deseamos tener una computadora diferente (MAQ. B) Algo que podría hacerse es escribir un programa en la MAQ. A que tenga como entrada el conjunto de instrucciones de la MAQ. B, así nuestra máquina podría desempeñarse como una MAQ. B; pero, seguramente, más lenta. El programa podría llamarse **Intérprete** y podría decirse que la MAQ.A está "emulando" a la MAQ.B. Si además suponemos que la MAQ. A está especialmente adaptada para interpretar instrucciones de otras máquinas y es una máquina muy *rápida* y posee una **memoria privada** (ROM) para almacenar el programa intérprete, podemos decir que la MAQ.A está **microprogramada**. Al programa intérprete lo llamamos **firmware** o **microprograma.**

La microprogramación (Maurice Wilkes) puede entonces entenderse como un método de diseño de la lógica de un procesador central ajustado a las características mencionadas.

Vamos a transformar la CPU de la máquina elemental vista en la unidad 4, en una máquina microprogramada. Vamos a reemplazar la *Unidad de Control (cableada)* de la máquina elemental por una *Unidad de Control microprogramada*. El esquema general puede verse en la fig. 1. La ejecución de cada macroinstrucción, como también el ciclo de búsqueda, son realizados por los microprogramas residentes en la micro-ROM. La dirección de la primera microinstrucción a ejecutar, es proporcionada por el código de operación de la macroinstrucción, es decir será alguna de las 16 primeras posiciones.

Cada microinstrucción está compuesta de 45 bits, divididos en seis campos:

Acción, Test, Envíe, Reciba, Falso, Éxito.

El significado de cada uno puede verse en la TABLA 2. El campo Acción esta relacionado con las órdenes que debe dar la Unidad de Control (leer la memoria, escribir la memoria, etc.), el campo Test se relaciona con la necesidad de chequear el estado de la máquina en un momento dado (Bit 15 del acumulador, señal de overflow, etc.). Los campos Envíe y Reciba tienen que ver con enviar los contenidos de los registros al bus o levantarlos del mismo. Por último los campos falso y éxito están relacionados con el resultado del chequeo indicado por el campo Test y definen la próxima microinstrucción a ejecutar. Nótese que la máquina interior no posee contador de programa. En la TABLA 1 puede verse el contenido de la ROM, es decir los microprogramas correspondientes a cada instrucción.

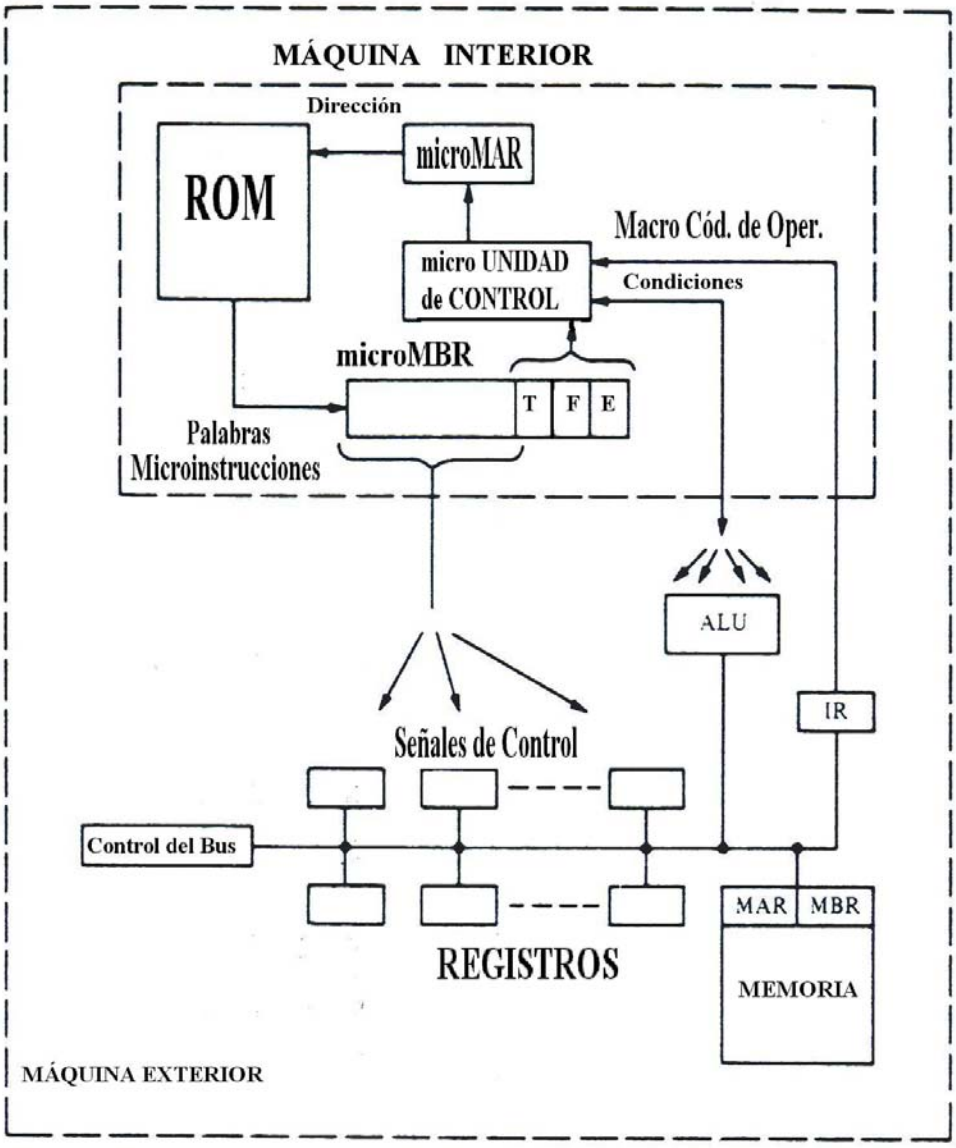


FIGURA 1. Esquema de la máquina elemental microprogramada

En la fig. 2 se muestra el circuito que controla al micro-MAR, que no es otra cosa que la micro-Unidad de Control. Obsérvese que el contenido del micro-MAR (los 8 biestables D de la figura) puede tener cuatro orígenes:

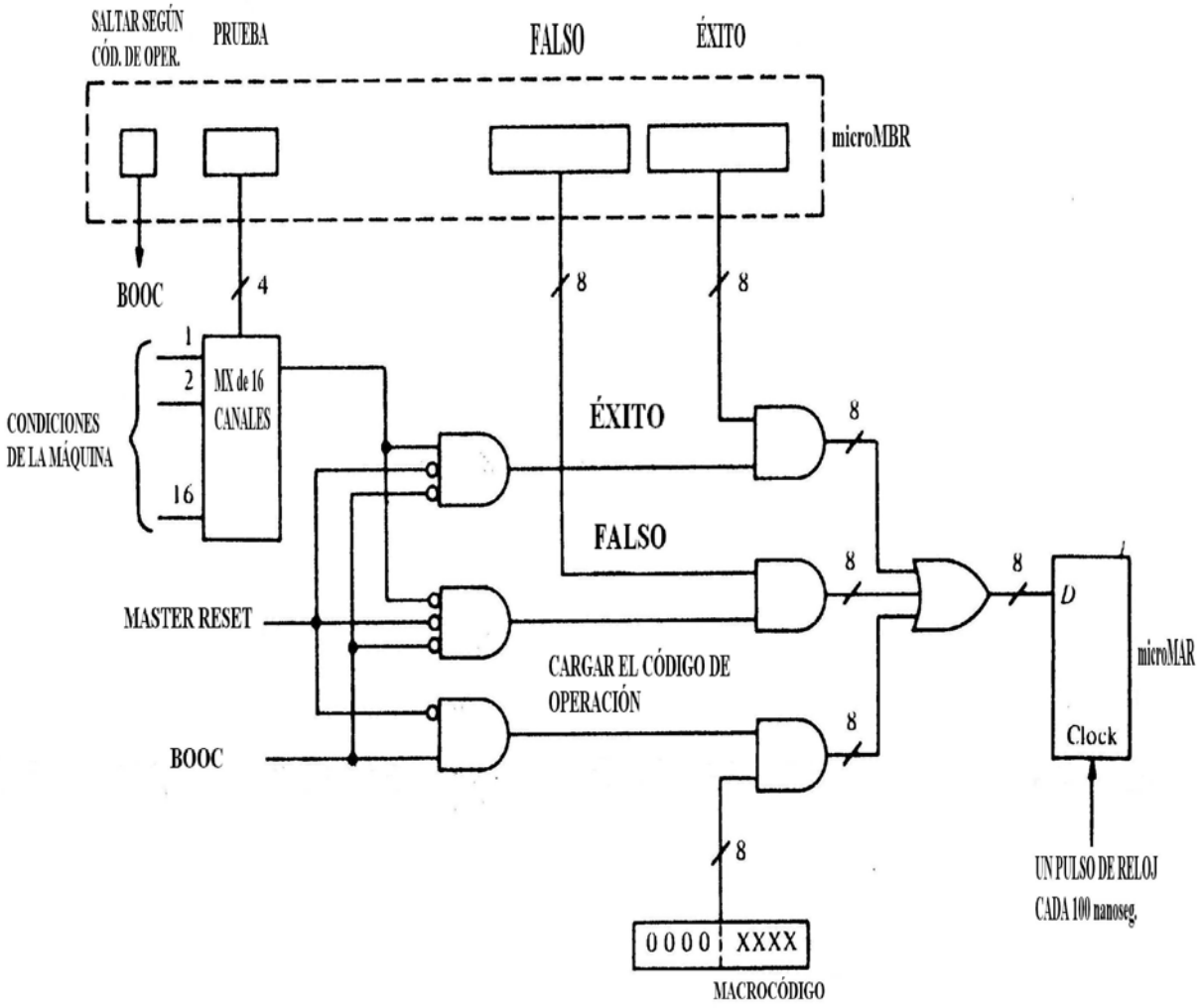
- Contenido del campo FALSO, en caso que la salida del multiplexor sea cero
- Contenido del campo ÉXITO, en caso que la salida del multiplexor sea 1.
- El código de operación de la macrointrucción residente en el registro de instrucciones, en el caso que el bit 5 del campo acción sea 1
- Cero, en el caso que RESET sea 1 (esta señal proviene del botón Master Reset en la consola del operador.).

Los dos primeros casos se dan cuando la Unidad de Control esta ejecutando un microprograma, ya sea correspondiente a una macroinstrucción o al ciclo de búsqueda.

A modo de ejemplo describiremos el ciclo de búsqueda en nuestra nueva máquina. El microprograma correspondiente comienza en la dirección rotulada RNI (ver TABLA 1). El campo Acción no tiene contenido indicando que ninguna acción es necesaria, El campo Envíe envía el PC al bus, el campo Reciba carga el MAR y el Z desde el bus, el campo teste verifica si alguien ha apretado el botón STOP de la consola, si está apretado la próxima microinstrucción es la que está en la dirección rotulada HALT, si no está apretado, la próxima microinstrucción es RNI 1. De RNI 1 a RNI 5, se realiza el incremento del PC y la carga del IR con el contenido de la posición de memoria direccionada. La RNI 6 da la orden de cargar el micro-MAR con el código de operación de la macroinstrucción ya almacenada en el IR, esto lo hace con el bit 5 en 1 (ver el circuito de control del micro-MAR en la fig. 2).

Es conveniente realizar el análisis de los todos los microprogramas a fin de comprender el funcionamiento completo. Al final de la Unidad se discuten las ventajas y desventajas de las máquinas microprogramadas que, debido a la facilidad que da el firmware, poseen gran cantidad de instrucciones; en contraposición de máquinas no microprogramadas que poseen, entre otras características, pocas instrucciones. En el punto siguiente se ejemplifica una CPU microprogramada con el microprocesador Intel 8088.

Figura 2 - Unidad de Control Microprogramada



Dirección	Acción	Envíe	Reciba	Test	Falso	Éxito
0	-----	-----	-----	-----	HALT	-----
1	-----	-----	-----	-----	ADD	-----
2	-----	-----	-----	-----	XOR	-----
3	-----	-----	-----	-----	AND	-----
4	-----	-----	-----	-----	IOR	-----
5	-----	-----	-----	-----	NOT	-----
6	-----	-----	-----	-----	LDA	-----
7	-----	-----	-----	-----	STA	-----
10	-----	-----	-----	-----	SRJ	-----
11	-----	-----	-----	-----	JMA	-----
12	-----	-----	-----	-----	JMP	-----
13	-----	-----	-----	-----	INP	-----
14	-----	-----	-----	-----	OUT	-----
15	-----	-----	-----	-----	RAL	-----
16	-----	-----	-----	-----	CSA	-----
17	-----	-----	-----	-----	NOP	-----
Microprograma de Búsqueda						
RNI	LEER	PC	MAR, Z	STOP	RNI 1	HALT
RNI 1	-----	+ 1	Y	-----	RNI 2	-----
RNI 2	-----	-----	-----	-----	RNI 3	-----
RNI 3	-----	SUM	PC	LIBRE	RNI 4	RNI 5
RNI 4	-----	-----	-----	LIBRE	RNI 4	RNI 5
RNI 5	-----	MBR	IR	-----	RNI 6	-----
RNI 6	BOOC	-----	-----	-----	-----	-----
Microprograma de la Instrucción HLT						
HALT	TRA = 0	-----	-----	START	H1	RNI
H 1	-----	-----	-----	EXAM	H 2	EXAM
H 2	-----	-----	-----	DEPOSIT	H 3	DEPOS
H 3	-----	-----	-----	Load PC	HALT	LPC
Microprograma del Botón EXAMINAR						
EXA	LEER	PC	MAR, Z	-----	E1	-----
E1	-----	+1	Y	-----	E2	-----
E2	-----	-----	-----	-----	E3	-----
E3	-----	SUM	PC	LIBRE	E4	E5
E4	-----	-----	-----	LIBRE	E4	E5
E5	-----	MBR	IR	-----	E6	-----
E6	-----	-----	-----	EXAM	H2	E6
Microprograma del Botón Depositar						
DEP	-----	PC	MAR, Z	-----	D1	-----
D1	-----	+1	Y	-----	D2	-----
D2	-----	SUM	PC	-----	D3	-----
D3	ESCRIBIR	SR	MBR	-----	D4	-----
D4	-----	-----	-----	LIBRE	D4	D5
D5	-----	-----	-----	DEPOSIT	H3	D5
Microprograma del Botón Cargue PC						
LPC	-----	SR	PC	-----	L1	-----
L 1	-----	-----	-----	Load PC	HALT	L1

Dirección	Acción	Envíe	Reciba	Test	Falso	Éxito
Microprograma de la Instrucción NOP						
NOP	-----	-----	-----	-----	RNI	-----
Microprograma de la Instrucción CSA						
CSA	-----	SR	A	-----	RNI	-----
Microprograma de la Instrucción RAL						
RAL	-----	A	Z	-----	RAL 1	-----
RAL 1	-----	2 * Z	A	-----	RAL 2	-----
RAL 2	-----	-----	-----	-----	RNI	-----
Microprograma de la Instrucción JMP						
JMP	-----	IR	PC	-----	RNI	-----
Microprograma de la Instrucción JMA						
JMA	-----	-----	-----	A_{15}	RNI	JMP
Microprograma de la Instrucción SRJ						
SRJ	-----	PC	A	-----	JMP	-----
Microprograma de la Instrucción NOT						
NOT	-----	A	Z	-----	NOT 1	-----
NOT 1	-----	$C_1(Z)$	A	-----	NOT 2	-----
NOT 2	-----	-----	-----	-----	RNI	-----
Microprograma de la Instrucción INP						
INP	TRA = 1	-----	-----	R = 1	INP	INP 1
INP 1	-----	LED	A	-----	INP 2	-----
INP 2	TRA = 0	-----	-----	-----	RNI	-----
Microprograma de la Instrucción OUT						
OUT	TRA = 1	-----	-----	R = 1	OUT	OUT 1
OUT 1	TRA = 0	-----	-----	-----	RNI	-----
Microprograma de la Instrucción LDA						
LDA	LEER	IR	MAR	-----	LDA 1	-----
LDA 1	-----	-----	-----	LIBRE	LDA 1	LDA 2
LDA 2	-----	MBR	A	-----	RNI	-----
Microprograma de la Instrucción STA						
STA	-----	IR	MAR	-----	STA 1	-----
STA 1	ESCRIBIR	A	MBR	-----	STA 2	-----
STA 2	-----	-----	-----	LIBRE	STA 2	RNI
Microprograma de la Instrucción ADD						
ADD	LEER	IR	MAR	-----	ADD 1	-----
ADD 1	-----	A	Z	LIBRE	ADD 2	ADD 3
ADD 2	-----	-----	-----	LIBRE	ADD 2	ADD 3
ADD 3	-----	MBR	Y	-----	ADD 4	-----
ADD 4	-----	-----	-----	-----	ADD 5	-----
ADD 5	-----	SUM	A	-----	RNI	-----
Microprograma de la Instrucción IOR						
IOR	LEER	IR	MAR	-----	IOR 1	-----
IOR 1	-----	A	Z	LIBRE	IOR 2	IOR 3
IOR 2	-----	-----	-----	LIBRE	IOR 2	IOR 3
IOR 3	-----	MBR	Y	-----	IOR 4	-----
IOR 4	-----	-----	-----	-----	IOR 5	-----
IOR 5	-----	OR	A	-----	RNI	-----

TABLA 2
MICROINSTRUCCIÓN DE MICRO-BLUE

CAMPO **ACCIÓN :** (5 BITS) Realiza una de las siguientes acciones

1er BIT : Inicie ciclo de lectura (**LEER**)
2do BIT : Inicie ciclo de escritura (**ESCRIBIR**)
3er BIT : **TRA = 1**
4to BIT : **TRA = 0**
5to BIT : Copie el cod. de op. de la macroinstrucción en el Micro-MAR. (**BOOC**)

CAMPO **ENVÍE :** (13 BITS) Envía al BUS uno de los siguientes registros

1er : Acumulador (**A**)
2do : **MBR**
3er : **PC**
4to : **SR**
5to : **+ 1**
6to : Campo de direccionamiento de **IR**
7mo : Líneas de Entrada de Datos (**LED**)
8vo : **SUM**
9no : **OR**
10mo : **XOR**
11mo : **AND**
12mo : **2 * Z**
13mo : $\mathbf{C_{1(Z)}}$

CAMPO **RECIBA :** (7 BITS) Carga uno de los siguientes registros desde el BUS

1er : Acumulador (**A**)
2do : **MBR**
3er : **PC**
4to : **Z**
5to : **IR**
6to : **Y**
7mo : **MAR**

CAMPO **TEST** : (4 BITS) Si la condición especificada es acertada, toma la próxima microinstrucción desde la dirección especificada por el campo **ÉXITO**. Si no la toma desde el campo **FALSO**. Las condiciones son las siguientes:

0000 : no realizar test.
0001 : $\mathbf{A_{15} = 1}$ (Acumulador negativo)
0010 : **R = 1** (Dispositivo de E / S listo)
0011 : Memoria **LIBRE**
0100 : Botón **STOP**
0101 : Botón **START**
0110 : Botón **Load PC**
0111 : Botón **EXAMINE**
1000 : Botón **DEPOSIT**
1001 : **OVERFLOW**
1010 al 1111 : NO DEFINIDOS

ARQUITECTURA AVANZADA

INTRODUCCIÓN

Un objetivo en el diseño de una computadora es que sea lo más rápida posible. Esto se logra haciendo más veloz el hardware, pero esto tiene sus limitaciones. Una de ellas es que la velocidad de propagación de las señales eléctricas en un conductor de cobre está en el orden de 20 cm/nseg. (en 1 nseg. recorre 20 cm), esto significa que si se pretende construir una máquina con un ciclo de instrucción de 1 nseg., la distancia total que las señales eléctricas tuvieran que viajar en la memoria, la CPU y de regreso, no deberían exceder 20 cm. Otra limitación es que a medida que aumentamos la velocidad de una máquina, también aumenta el calor generado, y al construirlas en un reducido volumen dificulta la tarea de disipar el calor. Construir máquinas rápidas es cada vez más costoso.

Sin embargo, es posible aumentar la velocidad de otra manera. Por ejemplo se puede construir una máquina con varias ALUs o varias CPUs para obtener la misma perfomance pero a un costo menor (Arquitectura en paralelo): también se puede adicionar el hardware necesario para que la CPU busque y comience a ejecutar una instrucción aún cuando no ha terminado de ejecutar la instrucción corriente (Procesamiento Pipeline); o bien adicionar una memoria rápida, no muy grande, cercana a la CPU, para almacenar datos de uso frecuente evitando tener que usar la memoria principal (Memoria Caché).

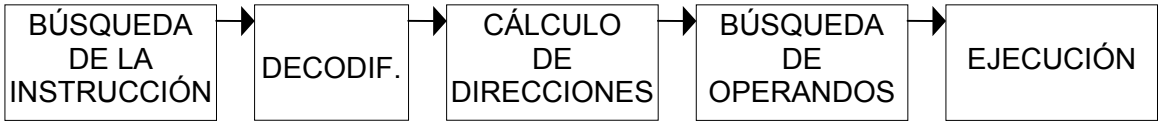
PIPELINE

Este procesamiento está basado en la idea que la Unidad de Control extrae una instrucción y la dirige a la unidades funcionales para su ejecución, mientras la Unidad de Control procede a extraer la próxima instrucción y la envía a las unidades funcionales. Así sucesivamente hasta que todas las unidades funcionales estén ocupadas. Esta estrategia general parte del hecho que el tiempo para extraer una instrucción es casi siempre mayor que el necesario para ejecutarla e implica diseñar la CPU con varias unidades de procesamiento

FIGURA 6

Ciclo	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**
Búsqueda de la instrucción	1	2	S				4	5	6	7	8	9	10
Decodificación		1	2	S				4	5	6	7	8	9
Cálculo de direcciones			1	2	S				4	5	6	7	8
Búsqueda de operandos				1	2	S				4	5	6	7
Ejecución					1	2	S				4	5	6

TABLA 5



En la fig. 6 se presenta un ejemplo de Pipeline. Consiste en cinco procesadores dentro de la CPU dedicados a funciones específicas. En la TABLA 5 se observa el proceso. En el primer ciclo se procede a buscar la instrucción 1, en el segundo ciclo se decodifica la instrucción 1 y se busca la instrucción 2, y así sucesivamente. La instrucción S, representa un salto, se observa que en el ciclo 4 no se procede a buscar la próxima instrucción ya que hasta no ejecutar la anterior, no se sabe cual es. Teniendo en cuenta que estadísticamente se verifica que las instrucciones de salto representan aproximadamente el 30 % del programa, el incremento de velocidad que podría lograrse con el pipeline (un factor de cinco en este caso) se ve reducido en el porcentaje indicado. Vale entonces la pena estudiar los tipos de saltos posibles a fin de disminuir el porcentaje.

Existen tres categorías de saltos:

- saltos incondicionales.
- saltos condicionales.
- saltos iterativos.

Una forma de disminuir el porcentaje (penalización por salto) es **ejecutar la próxima instrucción sin considerar el posible salto**. En este caso si el salto no se produce nada se ha perdido y se continua el proceso. Si el salto se produce, entonces deben eliminarse las instrucciones actualmente en línea y volver a comenzar. Se infiere que para que esta técnica sea posible, habrá que dotar al procesador de registros auxiliares que salven el contexto del sistema al momento del salto, de forma tal de restituirlos en el caso que el salto se produzca. Por otro lado esta técnica no es eficaz para el caso de los saltos incondicionales ya que estos se producen siempre.

Otra técnica utilizada es la **predicción de la dirección del salto**. Existen dos clases de predicciones: las **estáticas** (al momento de la compilación) y las **dinámicas** (al momento de la ejecución). En el primer caso el Compilador estima una dirección para cada una de las instrucciones de salto que genera. Por ejemplo, los saltos iterativos lo más probable es que el salto se produzca al inicio de la iteración, en los incondicionales siempre es posible estimar la dirección, en los condicionales es posible suponer una probabilidad de ocurrencia de la condición y en función de ella estimar la dirección más probable. En el segundo caso el microprograma construye una tabla de saltos y guarda en un registro el comportamiento de los mismos a fin de decidir cual es la dirección de la próxima instrucción mas probable; esto requiere agregar hardware. Con la utilización de estas técnicas se ha logrado reducir el porcentaje a un 10 %.

MEMORIA CACHÉ

Cuando se realiza una referencia a memoria, lo más probable es que las próximas referencias a memoria se realicen en las cercanías de la anterior. Mucho del tiempo de ejecución de un programa se emplea en iteraciones entre un número limitado de instrucciones. Esto se llama **principio de localidad** y representa la base de los sistemas con memoria caché. La velocidad de la memoria principal de un sistema es significativamente menor que la CPU; entonces atendiendo a lo mencionado, sería conveniente disponer de una memoria adicional (cache), que esté cerca de la CPU (controlada por ella), pequeña (para no encarecer el sistema) y definir una técnica apropiada para el llenado de la misma, teniendo en cuenta el principio de localidad.

La idea general del llenado de la caché es que cuando se referencia una posición de memoria, además de usarla , se carga en la caché, de forma tal que la siguiente vez que se

utiliza, puede accederse rápidamente. La mejora de velocidad puede formalizarse de la siguiente manera.
sea:

$t_{caché}$: tiempo de acceso de la memoria caché
t_{mem} : tiempo de acceso a la memoria principal.
k: cantidad de referencias a una posición de memoria en un tiempo corto.
h: proporción de aciertos (fracción de todas las referencias que pueden ser satisfechas fuera de la caché

tiempo medio de acceso : $t_{caché} + (1 - h)\, t_{mem} = t_{caché} + t_{mem}\, (1 / k)$

La segunda referencia a una posición de memoria puede inciarse en paralelo con una búsqueda en la caché de forma tal que de no encontrarla en esta última, el ciclo de memoria ya ha comenzado.

Existen dos técnicas para la escritura en la cache

- con *escritura en memoria*: Cada vez que se escribe en la caché, también se escribe en la memoria, esta técnica asegura que, en todo momento, coincidan.
- con *retrocopiado*: La memoria se actualiza sólo en el caso que una posición de la caché sea borrada. Esta alternativa requiere que cada posición de la caché tenga un bit especial que indique si la misma ha cambiado desde que se cargó la cache.

La primera técnica tiene el inconveniente de ocupar mucho al bus, mientras que la segunda requiere de un hardware adicional para actualizar la memoria en cualquier momento, por ejemplo cuando sea necesaria una transferencia a disco y la memoria no está actualizada.

EVOLUCIÓN DE LA ARQUITECTURA DE LAS COMPUTADORAS

INTRODUCCIÓN

Con el invento de la microprogramación de Wilkes al principio de los cincuenta la idea de hacer más y más complejo el microcódigo fue poco menos que irresistible. Las razones eran de validez. Por aquellos años las memorias eran significativamente más lentas que la CPU de forma tal que en aplicaciones que requerían uso frecuente de bibliotecas residentes en la memoria, la posibilidad de poner estas bibliotecas en el microprograma (en una rápida ROM) representaba una solución ideal. Por otro lado elevar el nivel del lenguaje de máquina incorporando instrucciones más completas, parecía no tener discusión. Además la microprogramación permite modificar y/o agregar instrucciones cambiando el microprograma (lo que significa simplemente cambiar una memoria ROM). Ya que la microprogramación no tenía discusión, a lo largo de los años se perfeccionó esta arquitectura. La micromemoria es costosa (recordemos que debe ser rápida) lo que la limita en su dimensión. Una micromemoria ancha (como la de 45 bits de la máquina elemental microprogramada) se llama *horizontal* lo que implica pocas microinstrucciones para ejecutar una instrucción, es decir una máquina más rápida). Como contrapartida se puede pensar en una microintrucción angosta, microprogramación *vertical*, más microinstrucciones por instrucción pero menor costo de la micromemoria. Note que (ver TABLA 1) existen secuencias de microinstrucciones comunes a más de una instrucción. Esto dio la idea de escribir microrutinas (usables por muchas instrucciones) que podrían residir en una microROM angosta y combinarla con una microprogramación horizontal. Existen alternativas siempre con el compromiso de costo perfomance. Esta arquitectura de máquinas se llama CISC, sigla que significa (Complex Instruction Set Computer).

Ya para los setenta la velocidad de las memorias se acerco a la de la CPU y resultaba difícil escribir, depurar y mantener los microprogramas. Además algunos especialistas comenzaron a analizar que tipo de instrucciones eran las más usadas en los programas. Los resultados fueron sorprendentes:

- El 85% de las instrucciones son de asignación, condicionales y de llamadas a procedimientos.
- El 80% de las instrucciones de asignación son de un solo término.
- El 41% de los procedimientos no tienen.
- El 80% de los procedimientos tienen 4 o menos variables locales.

Como conclusión podemos decir que si bien teóricamente es posible escribir complicados, la mayoría de los programas reales consisten en simples asignaciones, declaraciones condicionales y llamadas a procedimientos con un número reducido de parámetros. Esta conclusión es de extrema importancia en la tendencia a agregar más y más funciones al microcódigo. Mientras que el lenguaje de máquina se hace más complicado, el microprograma se hace más grande y lento. Un número elevado de modos de direccionamiento significa que su decodificación no puede realizarse en línea (lo que implicaría repetir cientos de veces el mismo microcódigo). Peor aún, se ha sacrificado velocidad a fin de incorporar instrucciones que en la práctica rara vez se usan. Se podría afirmar que una buena idea sería eliminar el microcódigo y que los programas se corran directamente por el hardware residiendo en una rápida memoria principal. Surgen así las computadoras con un numero reducido de instrucciones, llamadas máquinas RISC. Antes de Wilkes todas la maquinas eran RISC, luego la microprogramación las computadoras se hicieron más complejas y menos eficientes. Ahora, la industria está volviendo a sus raíces, construyendo máquinas sencillas y rápidas.

Una instrucción por ciclo

La denominación RISC no está del todo bien aplicada, si bien es cierto que tienen pocas instrucciones, la característica más importante es que éstas se completan en un sólo ciclo (entendiendo por ciclo la extracción de los operandos de un registro, colocarlos en el bus, ejecutarlos a través de la ALU y guardar el resultado en un registro).

	RISC	CISC
1	Instrucciones sencillas en un ciclo	Instrucciones complejas en varios ciclos
2	Sólo LOAD/STORE hacen referencia a memoria	Cualquier instrucción puede referencia memoria
3	Procesamiento serie de varias etapas	Poco procesamiento en aire
4	Instrucciones ejecutadas por hardware	Instrucciones interpretadas por el microprograma
5	Instrucciones de formato fijo	Instrucciones de formato variable
6	Pocas instrucciones y modo	Muchas instrucciones y modo
7	La complejidad está en el compilador	La complejidad está en el microprograma
8	Varios conjuntos de registros	Un sólo conjunto de registros

TABLA 6

Cualquier operación que no se lleve a cabo en un ciclo, no puede formar parte del conjunto de instrucciones.

Arquitectura de CARGA/ALMACENAMIENTO

Dado que cada instrucción debe ejecutarse en un ciclo, resulta claro que aquellas que hacen referencia a memoria representan un problema. Las instrucciones ordinarias sólo pueden tener operandos en registros (sólo está permitido el direccionamiento por registros). Las únicas instrucciones que hacen referencia a memoria son LOAD y STORE., para lograr que éstas se ejecuten en un ciclo se recurre a exigir que cada instrucción se inicie en cada

ciclo, sin importar cuando termine. Si se logra, entonces, comenzar n instrucciones en n ciclos, se habrá alcanzado un promedio de una instrucción por ciclo.

Todas las RISC poseen procesamiento en línea (pipeline), por ejemplo una RISC con tres procesadores tendría el aspecto de la fig. 7. En el ejemplo se puede apreciar que las

Ciclo	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Extracción de instrucciones	1	2	**L**	4	5	6	**S**	8	9	10
Ejecución de instrucciones		1	2	**L**	4	5	6	**S**	8	9
Referencia a memoria					**L**				**S**	

FIGURA 7

instrucciones ordinarias se ejecutan en dos ciclos. La instrucción LOAD (indicada con una L), se ejecuta en tres ciclos. Se ve que la instrucción 4 termina de ejecutarse antes que termine de hacerlo L que es la anterior. Lo mismo ocurre con la instrucción 8 y la S. En estos casos el Compilador verifica si la instrucción 4 es afectada por la instrucción L, si no lo es el proceso continúa sin problemas, en el caso contrario algo debe hacerse, por ejemplo el Compilador puede reemplazar a la instrucción 4 por una NOP lo que implica una degradación de la velocidad. El caso de las instrucciones de salto (JUMP) se producirá un problema similar y la solución es la misma: la instrucción que sigue a una de salto siempre comienza a ejecutarse independientemente si el salto se lleva a cabo. En todos los casos el Compilador es el responsable de colocar una instrucción útil después de una de salto, en caso de no encontrar nada adecuado, se coloca una instrucción NOP.

Las instrucciones generadas por el Compilador son ejecutadas directamente por el hardware, no son interpretadas por el microcódigo. Esta es la razón de la velocidad de las RISC. La complejidad que soporta el microcódigo en las CISC, se traslada al código de usuario en las RISC. Si una instrucción compleja implica n microinstrucciones en una CISC, en una RISC implicaría un número similar de instrucciones, pero ocuparía más memoria. Sin embargo también debe tenerse en cuenta que las instrucciones complejas representan un porcentaje menor en un programa real. Se recomienda comparar los tiempos de ciclo de instrucción de la máquina elemental con la máquina elemental microprogramada.

La razón del reducido número de instrucciones responde a la idea de simplificar el decodificador de instrucciones. En cuanto a los modos de direccionamiento conviene reducirlos al mínimo. En el formato de una instrucción RISC, puede apreciarse que es posible generar distintos modos de direccionamiento.

7	1	5	5	1	13
COD. OPER.	**C**	**DESTINO**	**ORIGEN**	**I**	**DESPLAZAMIENTO**

COD. OPER.: Código de operación de la instrucción
C: Activa o no los códigos de condición
DESTINO: Registro destino de la operación
ORIGEN: Registro fuente de la operación
I: El campo desplazamiento es tal o un registro.
DESPLAZAMIENTO: Valor de desplazamiento o un registro.

El primer operando de cualquier operación se toma del registro origen.

Para instrucciones ordinarias como ADD, los operandos dependen de cuanto vale I.
Si I = 0 , el segundo operando lo especifican los últimos cinco bits de campo desplazamiento. Representa un direccionamiento por registro.

Si I = 1 , el segundo operando es el campo desplazamiento. Representa un direccionamiento inmediato.

Para la instrucciones LOAD y STORE, el desplazamiento es sumado al registro origen para obtener la dirección de memoria. Representa un direccionamiento indexado. Si el desplazamiento es cero representa un direccionamiento indirecto por registro.

Conjunto de registros múltiples

A fin de reducir el número de cargas y almacenamientos (LOAD Y STORE), una buena parte del chip de una RISC se usa para registros (no es raro encontrar RISC con 500 registros), aprovechando que carece de firmware. La organización de estos registros es un aspecto muy importante en las RISC. El hecho que, como se mencionó anteriormente, una parte importante del tráfico a memoria es consecuencia de los llamados a procedimientos (que implica transmitir parámetros, salvar registros, etc.), dio lugar a los diseñadores de las RISC a plantear una organización de registros llamada **traslape de registros.** En general consiste en que , en un momento dado, la CPU accede a sólo un subgrupo de ellos, por lo general de 32 bits. Estos registros están divididos en 4 grupos de 8 cada uno (ver fig. 8). Los primeros

R0 . . R7	Registro origen con valor cero VARIABLES GLOBALES
R8 . . R15	PARÁMETROS DE ENTRADA
R16 . . R23	VARIABLES LOCALES
R24 . . R31	PARÁMETROS DE SALIDA

FIGURA 8

ocho registros se encuentran en todo momento accesibles a la CPU y representan los registros globales utilizables por cualquier procedimiento. En cambio desde el R8 en adelante, el grupo de 24 registros accesibles por la CPU dependerá del valor de un puntero de registros, este puntero es modificado cuando algún procedimiento es llamado desde el actual. Esta organización permite intercambiar valores entre procedimientos sin necesidad de referir a la memoria.

Comparación entre Arquitecturas RISC y CISC

Una comparación entre RISC y CISC para determinar cual es mejor, entendiendo por mejor a la más rápida, es en extremo compleja. Varios factores deben tenerse en cuenta. Podríamos hacernos algunas preguntas antes:

- ¿Qué tipo de programas corren en la máquina?, programas con pocas llamadas a procedimientos y muchos saltos se corren mejor en una CISC, por otro lado programas cortos y recursivos corren mejor en las RISC.

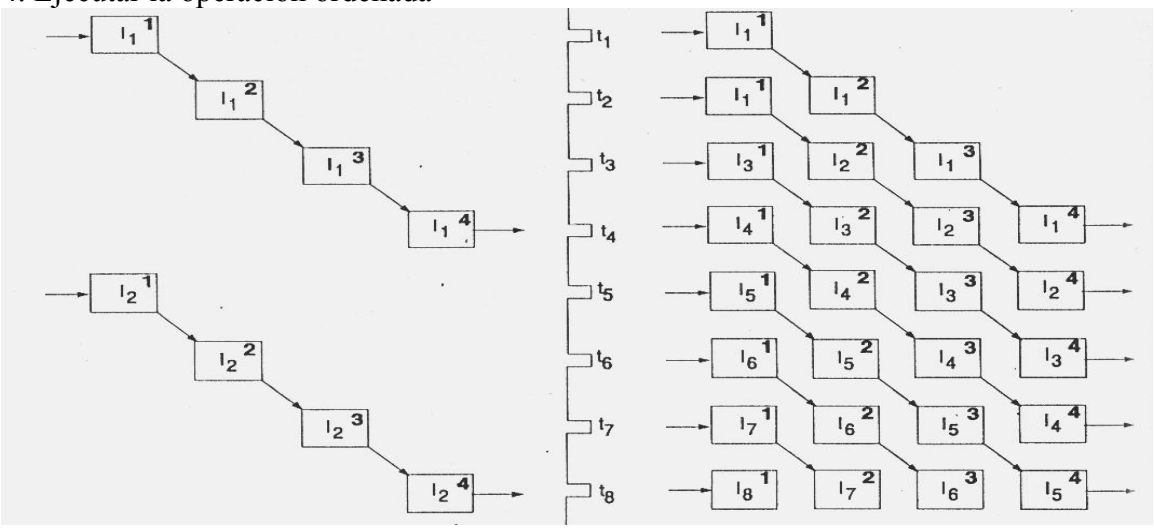
- ¿Qué tipo de compilador se usó?, es entendible que un buen compilador implica programas más rápidos.
- ¿Deberían usarse programas en punto flotante?, hemos visto que las RISC, de no contar con hardware adicional, no son buenas para este tipo de cálculo.
- ¿Deben tenerse en cuenta los recursos del sistema además de la CPU?, unidades de entrada/salida, Sistema operativo, etc..
- ¿Qué tecnología se usa para construir la CPU?. Las máquinas que podríamos comparar usan distintas tecnologías, distintos tiempos de reloj, distintos buses, etc.. ¿Como tener en cuenta estas cosas?.
- ¿Debe considerarse la cantidad de memoria utilizada?. En este sentido las RISC usan más memoria que las CISC, teniendo en cuenta que las memorias son cada vez más baratas ¿es de considerar este aspecto?
- ¿Debe medirse el tráfico de memoria?. Es decir una máquina que realice un mismo programa con menos referencias a memoria es mejor que otra, aun cuando tarde tiempos similares.

En general los cálculos de desempeño favorecen a las máquinas RISC. Pero esta conclusión no debe tomarse para desestimar la arquitectura CISC, simplemente significa que las máquinas RISC se desempeñan mejor que las CISC ya construidas y que, en su diseño tuvieron que respetar la compatibilidad con procesadores anteriores (por ejemplo si el 80486 no hubiera tenido que ser compatible con el 8088, seguramente su arquitectura CISC habría sido diferente).

¿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 procesamiento, *la mayoría de los procesadores actuales procesan en base al esquema de* **"Von Neumann",** ya visto, que tiene las siguientes etapas:

1. Traer a la CPU el código de la instrucción a ejecutar
2. Decodificar dicha instrucción
3. Traer a la CPU el dato a operar
4. Ejecutar la operación ordenada



La figura arriba a la izquierda simula el orden en que avanza el proceso según el modelo original de Von Neumann. 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, y luego la siguiente, insumiendo la ejecución de ambas instrucciones 8 pulsos reloj.

Actualmente, 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 **"pipeline"** (figura de la derecha).
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.
Esto es semejante al procesamiento de cada instrucción en el modelo original de Von Neumann, 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.
Si al modelo de Von Neumann se le agrega "pipelining", la CPU mantiene su esquema básico, pero se le debe agregar circuitería adicional.
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.
La figura derecha ilustra cómo un "pipeline" 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 "pipeline' se realiza con 4 etapas. En la primera de estas instrucciones que corresponde ejecutar (I1) pasa del buffer al registro RI. En t2 el código es decodificado, y al registro RI pasa a contener el código de I2. En t3 se trae el dato a operar, se decodifica I2, y a RI llega desde el buffer el código de I3. En t4 termina de ejecutarse I1, se trae el dato a operar para I2, se decodifica I3, y llega a RI el código de I4
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 e sea, 4 veces más que con el modelo sin "pipeline' que aparece a la izquierda de la misma figura.
En general, si se tiene un "pipeline" de n etapas, teóricamente se puede procesar hasta n veces más instrucciones por segundo que sin "pipeline", 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 "pipeline ", 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 "pipeline".

¿Cómo han evolucionado los procesadores desde el 8086 hasta el 80586 (Pentium) para lograr mayor rendimiento?

La evolución de la tecnología permitido que cada vez se construyan procesadores más veloces y con mayor rendimiento.

Los principales ítems que se han adicionado y/o mejorado son:

- mayor capacidad de memoria
- el aumento del tamaño del dato a 32 bits (y hasta 64 bits actualmente)
- el uso de frecuencias del reloj que superan los 3.000 Mhz
- el «pipeline"
- la obtención anticipada de las próximas instrucciones a ejecutar
- la memoria "caché"
- el mayor número de registros de la CPU
- operación multitarea (multitasking), etc.

Al mismo tiempo, estas mejoras han permitido que los nuevos procesadores de una misma familia puedan ejecutar las instrucciones de modelos anteriores, sin tener que cambiar el software desarrollado para éstos, logrando una necesaria compatibilidad.
En relación a las mejoras citadas, por ejemplo Intel las ha incorporado en sus procesadores como sigue:

	8086	80286	80386	80486	80586 (Pentium)
Memoria en Mbytes	**1**	**16**	**4000**	**4000**	**4000**
Bus direcciones	**20**	**24**	**32**	**32**	**32**
Tamaño del dato	**16**	**16**	**32**	**32**	**32**
Frecuencia de reloj		**25**	**40**	**100**	**200**
Pipeline					**doble**
Memoria caché	**No**	**no**	**externa**	**8kb + externa**	
Registros	**14 (16 bits)**	**19 (16 bits)**	**28 (32 bits)**	**28 (32 bits)**	**28 (32 bits)**
Bus de datos	**16**	**16**	**32**	**32**	**64**
Otras mejoras		**multitasking**			**2 ALUs**

El 8086 utiliza un tipo de "pipeline", y realiza la obtención anticipada de próximas instrucciones a ejecutar.

El procesador 80286 aumenta su frecuencia de reloj hasta 25 MHz, presenta la opción de un coprocesador matemático externo opcional (80287), y puede realizar *"multitasking"* en modo protegido si usa un sistema operativo preparado para ello. Opera en memoria y en la UAL con 16 bits a la vez. Con sus 24 líneas de dirección puede acceder a 2^{24} = 16 MB de memoria principal. Opera hasta 25 Mhz.

El 80386 además de perfeccionar las innovaciones del 80286, tiene también corno opcional externo una memoria caché. Maneja 32 bits a la vez, por lo cual sus registros internos son

también de 32 bits. Las líneas de dirección son 32, pudiendo así direccionar 2^{32} = 4 GB de memoria. Opera hasta 40 MHz.

El procesador 80486DX opera con datos de 32 bits, y 32 líneas de dirección, como el 80386. Presenta un "pipeline" más elaborado, y en su interior se tiene un coprocesador matemático y un caché de 8 KB. De éste se obtienen en forma simultánea, en promedio, las 5 próximas instrucciones a ejecutar. Como en los procesadores RISC, muchas de las instrucciones (incluidas las usadas para resguardar datos en la pila) del 486 se ejecutan en un solo pulso reloj, o sea utilizan un solo microcódigo de la ROM de Control.

El **Pentium** opera internamente con 32 bits, y se comunica con el exterior a través de 64 líneas de datos y 32 de dirección. Contiene dos "pipeline" con dos ALU, por lo cual puede ejecutar simultáneamente dos instrucciones en un pulso reloj, si ambas son simples, por lo cual es un procesador "super escalar" (el modelo de Von Neumann es "escalar") con muchas concepciones RISC y "predicción de saltos condicionales". En punto flotante es cuatro veces más rápido que el 486.

El P6, así designado por Intel, opera interna y externamente con igual número de bits que el Pentium.. Presenta 3 "pipeline", con la posibilidad de ejecutar 3 instrucciones simples por pulsos reloj. Permite ejecutar las instrucciones fuera del orden establecido por el programa, las que luego son reordenadas. Su hardware convierte las instrucciones 80x86 en operaciones simples, tipo RISC. Incorpora 8 nuevos registros de 32 bits, en relación con los registros clásicos de los 8Ox86. A 150 MHz puede llegar a ser hasta un 50% más rápido que el Pentium sólo si el computador tiene un sistema operativo totalmente de 32 bits (como el OS/2 o el Windows NT, no así el Windows 95). El chip del P6, de 5,5 millones de transistores, viene adosado con otro chip que contiene un caché externo o de nivel 2 ("level 2"-L2) de 256 ó 512 KB (con 15,5 y 31 millones de transistores, respectivamente). Como en el Pentium, el caché interno (L1) está separado en uno de 8 KB para instrucciones y otro de 8 KB para datos. Otra característica del P6 es que puede conectarse directamente a otros 3 procesadores P6 para multiprocesamiento.

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, de modo de poder 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. En las arquitecturas "no Von Neumann", varias CPU pueden terminar de ejecutar juntas varias instrucciones por pulso reloj.
No debe confundirse *multiprocesamiento* con **"multiprogramación".** *'Multitasking" (*también traducible como *"multitarea"),* consiste en la ejecución alternada por una CPU 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.*

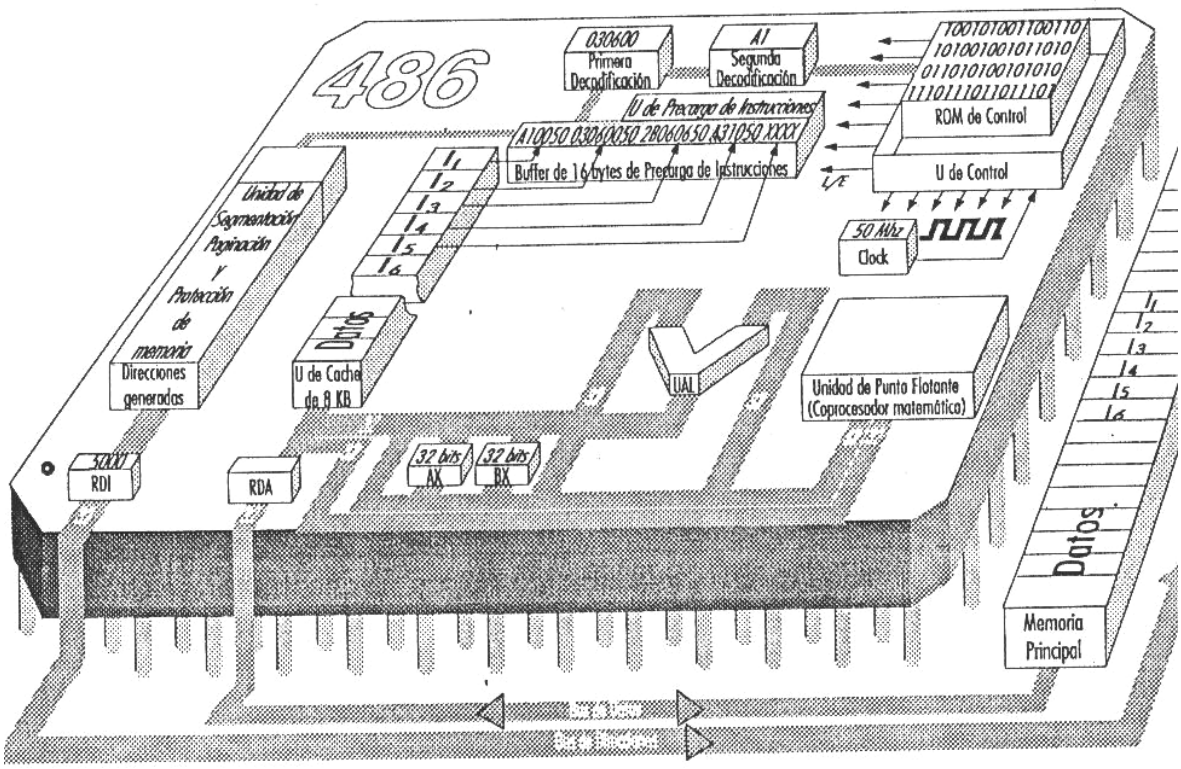
¿ Cómo funciona básicamente un microprocesador 486 ?

A continuación describiremos los principales bloques que están en el interior de un procesador 486, y las funciones que cumplen.
En la figura 1.86 aparecen los siguientes sub-bloques y bloques:

- Los registros de direcciones (RD l) y de datos (RDA) pertenecen 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. Las instrucciones y datos leídos en memoria pasan al caché interno de 8 KB del procesador

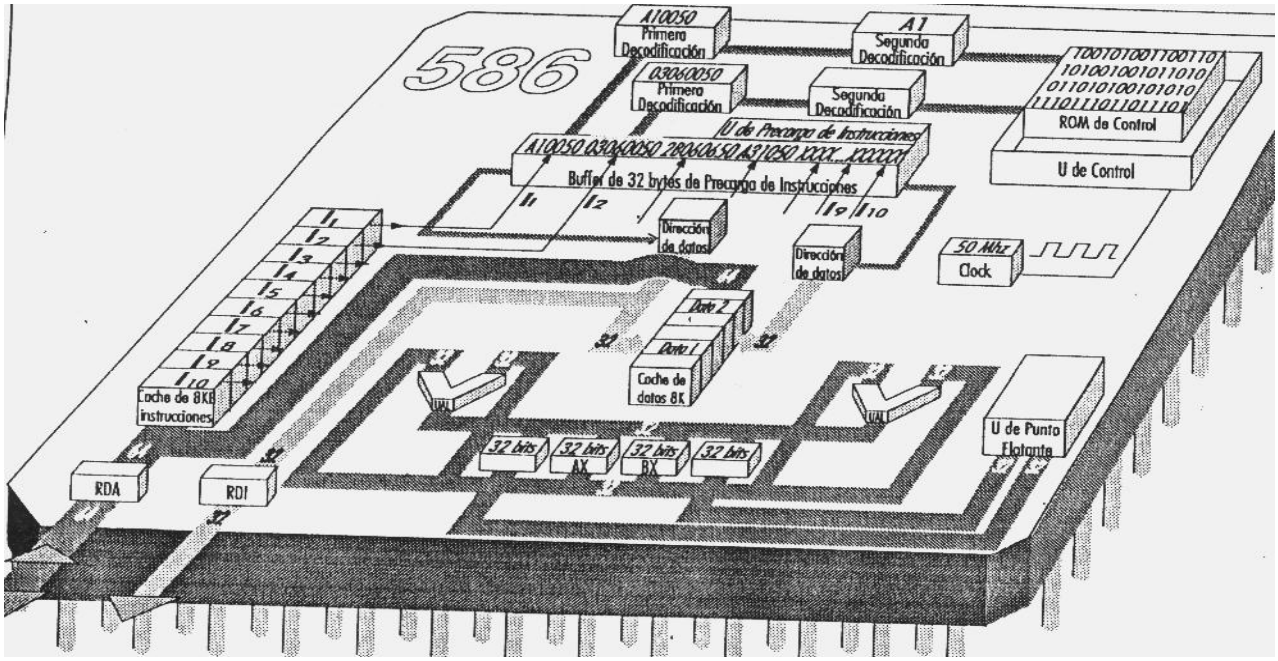
- La *Unidad de caché* de 8 KB guarda las instrucciones y datos que seguramente serán requeridos próximamente. Por una parte, a través de un bus de 128 líneas se pueden leer del caché 16 bytes que pasan a un buffer de la Unidad de pre-carga de instrucciones, correspondientes en promedio a unas 5 instrucciones a ejecutar, que así llegan juntas para entrar al "pipeline. Por otra, el caché puede ser leído para que se envíen 32 bits de datos a la ALU, o a un registro de la CPU ó 64 bits de datos a la Unidad de Punto Flotante (FPU en inglés). En una escritura van hacia el caché 32 6 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.
- La *Unidad de Control (UC)* mediante líneas que salen de ella, activa. las operaciones que con cada pulso reloj deben realizar los distintos bloques de la CPU, 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; esta unidad se encarga de ello, así como de la protección contra escrituras no permitidas en zonas reservadas de memoria.



Todas estas unidades participan en el "pipeline" 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 "pipeline" a dos buffers (de 16 bytes cada uno) de la Unidad de Pre-carga, , para formar una “cola"
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).
3. Segunda Decodificación: el código de operación identificado en el paso anterior es ahora decodificado. Esto permite determinar la secuencia de **microcódigo contenida** en la ROM de Control, merced a la cual la UC generará las señales de control, que enviará por las líneas que salen de ella, para que cada unidad que controla, ejecute una parte de la instrucción con cada pulso reloj. Si la instrucción es simple se ejecuta en un solo pulso. Al mismo tiempo que pasa esto por esta etapa de1 "pipeline", otros tres bytes del código entran a la etapa de primera codificación.
4. Ejecución: se ejecuta una operación en la ALU por ejemplo, leyendo el dato a operar en el caché. Paralelamente con la acción recién descripta se van ejecutando nuevas tareas en la primera y segunda decodificación.
5. Almacenamiento de resultados: esta es la etapa final de1 "pipeline" completándose la ejecución de la instrucción. En ésta, el resultado de la ALU se almacena, así como los "flags" que ella también genera, resultantes de la operación, en el registro de estado.

¿Qué tiene en común y cómo funciona el Pentium en relación con la operación de un 486 ?
La siguiente figura da cuenta de un esquema básico de un Pentium basado en el del 486:



Como en el 386 y el 486, en el Pentium las instrucciones para enteros, siguen un "pipeline" de 5 etapas.
Para la etapa de pre-carga 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 hacia un buffer de la Unidad de pre-carga, que puede almacenar 32 bytes (existen dos de estos buffers). 0 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 (pan 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 "pipeline" 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 "pipeline") o si sólo seguirá una por el "pipeline" 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 codificación.
Una instrucción para números en punto flotante opera con datos de 64 bits, que ocupan los dos "pipelines" para números enteros (de 32 bits cada uno), por lo que ella no puede procesarse junto con otra instrucción.
Estas instrucciones pasan por las cinco etapas correspondientes a instrucciones para enteros, y además requiere 3 etapas de un "pipeline" exclusivo para punto flotante. Puede decirse que el Pentium presenta un "pipeline" 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.
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 segunda decodificación.
Si en la primer decodificación se determina que el par de instrucciones identificadas son simples, y que la segunda en **orden no depende** M 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.*

¿Qué características tienen los procesadores CISC ?

A partir de los computadores 360 y 370 de IBM, surgidos en la década del 70, la mayoría de los procesadores (CPU) de las computadoras, incluidos los de minicomputadoras y PC personales (Pentium y el P6 de Intel, y los 680x0 de Motorola han sido CISC (Complex Instruction Set Computer). Esta denominación se debe a que pueden ejecutar desde instrucciones muy simples, (como las que ordenan sumar, restar dos números que están en registros de la UCP y el resultado asignarlo a uno de esos registros), hasta instrucciones muy complejas (como los tan usados movimientos de cadenas de caracteres de gran longitud y variable, en procesamiento de textos). Las instrucciones simples, luego de su decodificación, pueden ejecutarse en un pulso reloj, mientras que las complejas requieren un número de pulsos que depende de la secuencia de pasos necesarios para su ejecución.
Como se describió, cada paso se lleva a cabo mediante una combinación binaria *(microcódigo),* que aparece en las líneas de control de la UC con cada pulso reloj, la cual activa los circuitos que intervienen en ese paso.
Las sucesivas combinaciones (microcódigos) que requiere la ejecución de una instrucción compleja, deben ser provistas por una ROM de Control que las almacena, y que forma parte de la UC.
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.

En general, cada operación que ordena una instrucción de un procesador CISC presenta variantes para ser aplicadas a diversas estructuras de datos, desde simples constantes y variables, hasta matrices y otras. Así, una instrucción que ordena sumar tiene muchas variantes (códigos) en función de la estructura de datos sobre la cual opera. Es como si existieran tantas instrucciones que ordenan una misma operación como estructuras de datos típicas se han definido para operar, lo cual se denomina *"modos de direccionamiento"* de una instrucción.
La *existencia de muchos "modos de direccionamiento " para* realizar la operación que ordena una instrucción, es otra de las características de complejidad de los procesadores CISC. Esto se manifiesta en que el repertorio ("set") de instrucciones de una máquina CISC presente un *número elevado de códigos de instrucción.* Así, una IBM 370 tiene 210 instrucciones, 300 la VAX, y 230 el 80486. Asimismo, lo anterior exige instrucciones que ocupan *distinta* cantidad de bytes en memoria.
En relación con el "pipeline", se ha supuesto que las instrucciones que se han ejecutado, todas se llevan a cabo en igual cantidad de pasos (4 pulsos reloj). Esto permite que con cada pulso reloj se termine de ejecutar una instrucción. Si la ejecución de instrucciones requiere distinta cantidad de pulsos reloj, no es factible este rendimiento ideal de una instrucción por pulso.
Resulta así, que por tener instrucciones ejecutables en diferentes cantidad de pulsos reloj, *un procesador CISC no puede aprovechar eficazmente su "pipeline en la producción de instrucciones.*

¿En qué se diferencian los procesadores RISC de los CISC ?

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% de¡ repertorio de instrucciones de un procesador CISC- constituían el 80% de programas típicos ejecutados. El 91% de las sentencias más usadas en lenguajes de alto nivel (Fortran, Pascal, Basic, C, etc.) son las del tipo *Asignar* (un valor a una variable), "IF' (condicional), *"CaII"* llamar a procedimiento), "Loop" (repetir una secuencia), que en promedio constituyen el 47%, 23%, 15%, y 6%, respectivamente.

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 semántica". Recordar al respecto, que una sentencia como Z = P + P - Q se debe traducir a una secuencia de instrucciones de máquina como I1, I2, I3, I4.
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 Ix, 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 Ix.
De existir muchas equivalencias entre sentencias en alto nivel e instrucciones de máquina, el programa traductor entre ambos niveles (compilador) sería más sencillo de fabricar, y los tiempos de compilación disminuirían, objetivos de las arquitectura CISC.
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.
La información anterior sirvió para planificar procesadores con un *repertorio de instrucciones simples* (operar dos números que están en registros de la UCP, y el resultado asignarlo a uno de esos registros) que presentan muy pocos modos de direccionamiento. Por ser simples, estas instrucciones se ejecutan en un solo pulso reloj, luego de haber sido decodificada.
A fin de poder traducir un lenguaje de alto nivel a este tipo de instrucciones, empleando un mínimo de ellas, se requiere para RISC un programa *compilador inteligente,* muy elaborado. 0 sea que es necesario un compilador (software) complejo, como contrapartida de un hardware más simple.
Para mover datos de memoria a registro, y en sentido contrario, fue necesario la existencia de instrucciones que ordenen esos movimientos, que en el lenguaje assembler de un RISC se denominan LOAD y STORE, respectivamente. Estas instrucciones se trata de usarlas lo menos posible (usando programas compiladores "inteligentes"), puesto que requieren dos pulsos para ser ejecutadas, luego de que fueron decodificadas
A diferencia, en un CISC cualquier instrucción tiene la opción de requerir un dato de memoria.
El número total de instrucciones del repertorio de un procesador RISC es reducido entro 70 y 150 instrucciones según el modelo)
Puesto que la mayoría de las instrucciones RISC se ejecutan en un pulso reloj, resulta un «pipeline" eficaz, *terminándose de ejecutar en promedio, una instrucción por pulso reloj*
En promedio pues las instrucciones tipo LOAD y STORE requieren dos pulsos luego de ser decodificadas.
Todas las instrucciones RISC son de formato fijo, por ejemplo de 4 bytes, siendo que un CISC ocupan distinta cantidad de bytes. Esto redunda en una mayor sencillez y velocidad de procesamiento.
Por constar de instrucciones cuya fase de ejecución requiere mayormente un pulso, o a lo sumo dos, *no se requiere una ROM de Control para generar el microcódigo que debe aparecer en las salidas de la UC con cada pulso reloj para comandar el procesador.*
Esto es, los bits del código de operación de cada instrucción que llega al registro de instrucción RI para ser ejecutado por un procesador RISC, sirven de base para que un circuito de la UC los convierta directamente en la combinación de bits que deben aparecer en las salidas de la UC (microcódigo) para que en próximo pulso reloj, se ejecute la instrucción en cuestión.
Por lo tanto, la UC de un RISC *no contiene ROM de control (de microcódigos).* Esto, por un lado permite ganar en velocidad, pues se evita el acceso a una ROM. Por otro lado, beneficia mucho el diseño del chip que contiene un procesador RISC, dado que la superficie que ocupa

una ROM de Control de un CISC, en un RISC es aprovechada para aumentar -como ser a 32- el número de registros de uso general de la CPU.
Un mayor número de registros permite utilizar menos instrucciones LOAD-STORE, lo cual redunda en menos accesos a memoria principal. El hecho de que la mayoría de las instrucciones sean de igual complejidad trae aparejado un mejor rendimiento del "pipeline" Otro factor que disminuye este rendimiento son la **dependencia del resultado** de una **instrucción para poder** ejecutar la siguiente, y un mismo, recurso (por ejemplo un registro de la CPU) que es requerido por varias etapas de un "pipeline". Estos factores influyen menos en un RISC que en un CISC, resultando así una mayor velocidad de procesamiento.
Por ejemplo, si el resultado de una instrucción es el dato de la siguiente, puesto que la mayoría requiere dos pulsos (fases) para ejecutarse, cuando una instrucción en el "pipe-line pasa a la fase de ejecución, las anteriores que entraron al "pipeline" ya completaron dicha fase. Sólo existe una espera, cuando el dato que opera una instrucción se cargó en un registro desde memoria en la instrucción anterior. Se trata de un "pipeline" al cual llegan instrucciones de máquina planificadas por un compilador inteligente.
Los procesadores CISC pierden mucho tiempo en las instrucciones de llamado a subrutina y en las interrupciones, dado los consiguientes accesos la pila de memoria principal que requieren. Las estadísticas indican que en el llamado a procedimientos, el 98% de las veces se utilizan menos de 6 argumentos, y el 92% de las veces menos de 6 variables locales.
Asimismo, que un procedimiento llame a otro, esté a un tercero, el tercero a un cuarto, etc., (anidamiento, o llamadas sucesivas) sólo se llega a 8 llamadas sucesivas en el 1 % de las veces.
Dado que los RISC poseen un número elevado de registros, éstos pueden usarse para el manejo de llamados, en lugar de perder tiempo para escribir y leer la pila ubicada en memoria.

BIBLIOGRAFÍA

- ORGANIZACIÓN DE COMPUTADORAS. Un Enfoque Estructurado A.S.TANENBAUM
- COMPUTER ARCHITECTURE - C. FOSTER
- MICROPROCESADORES DE 16 BITS - J.M. ANGULO
- MICROPROCESSOR AND PERIPHERAL HANDBOOK - Vol. I - INTEL