Tema 6

Computadores RISC

La arquitectura RISC (Reduced Instruction Set Computer, computador de conjunto reducido de instrucciones) ha sido una de las más importantes innovaciones en la arquitectura de computadores.

6.1 Características de la ejecución de instrucciones

La programación de las primeras computadoras digitales se realizó mediante instrucciones consistentes en códigos numéricos que indicaban a los circuitos de la máquina los estados correspondientes a cada operación: el lenguaje máquina.

Sin embargo, pronto los primeros usuarios de estas computadoras se dieron cuenta de la ventaja de escribir sus programas mediante claves mnemotécnicas, más fáciles de recordar que los códigos numéricos del lenguaje máquina. Estas claves constituyen el lenguaje ensamblador de la máquina.

La utilización del lenguaje ensamblador hace la programación mas sencilla, ya que el manejo de los códigos numéricos en forma de claves textuales hace mas "humana" la programación, proporcionando un interfaz más amigable.

El uso del lenguaje ensamblador para escribir un programa no penaliza de ninguna manera la eficiencia de ejecución del programa, dado que las claves mnemotécnicas tienen una correspondencia inmediata con un código numérico, por lo que puede decirse que el rendimiento es el mismo que si el programa se hubiera escrito directamente en lenguaje máquina.

Sin embargo, la utilización del lenguaje ensamblador para escribir programas tiene ciertos inconvenientes, como son:

- **Falta de fiabilidad**: Es frecuente que los programas, tanto de sistema como de aplicación, continúen mostrando nuevos errores después de años de funcionamiento.

- **Dificultad de mantenimiento**: Si bien es complicado realizar cambios para el mantenimiento de grandes programas escritos en ensamblador, pasado cierto tiempo desde su escritura la complejidad aumenta exponencialmente, máxime si quien tiene que hacer los cambios no es el programador inicial.

JUAN F. GUERRERO MARTÍNEZ
JOSÉ V. FRANCÉS VILLORA

- **Conocimiento de la arquitectura**: Cada máquina posee un lenguaje ensamblador, y se requiere que el programador conozca previamente la arquitectura del máquina.

Obviamente esto condicionaba la complejidad de los desarrollos y su mantenimiento. Por ello, la respuesta desarrolló lenguajes de programación de alto nivel. Estos lenguajes de alto nivel (*high-level language*, HLL) permiten al programador expresar los algoritmos de manera más concisa y se encargan de buena parte de los pormenores.

Sin embargo desde la aparición de los primeros compiladores, la programación en ensamblador se siguió utilizando durante décadas para, entre otros, sistemas basados en uC o DSPs. Incluso en sistemas complejos las zonas más críticas a veces se programan en lenguaje ensamblador.

¿Por que? Desgraciadamente, la aparición del compilador como solución ocasiono otro problema, conocido como *salto semántico*: la diferencia entre las operaciones que proporcionan los lenguajes de alto nivel (HLL) y las que proporciona la arquitectura del computador. Los síntomas de este salto semántico incluían:

- Ineficiencia de la ejecución.
- Tamaño excesivo del programa compilado en lenguaje máquina.

Estas desventajas se plasman, por ejemplo, en el hecho de que la eficiencia del compilador del DSP TI TMS320C2x no esté muy por encima del 60%, lo cual quiere decir que si el mismo programa se programara en lenguaje ensamblador ¡¡se ejecutaría en aproximadamente la mitad de tiempo!! ¡¡necesitando el código menor cantidad de memoria!! No resulta, pues, extraño, que el paso a la utilización del compilador no fuera inmediata.

Obviamente, la solución a este problema pasaba por 'cerrar el *gap'*, es decir, reducir y eliminar el salto semántico. Intentar que la eficiencia de ejecución y el tamaño del programa fuera la misma tanto si se programaba en ensamblador como si se programaba mediante un compilador.

La respuesta de los diseñadores fueron arquitecturas que intentaban cerrar este hueco. Las arquitecturas propuestas incluían grandes conjuntos de instrucciones, decenas de modos de direccionamiento e incluso sentencias HLL implementadas en hardware (microprogramadas). Todas ellas arquitecturas CISC (*Complexe Instruction Set Computer*).

6.2 Estudios de Características y patrones de ejecución

Mientras tanto, durante algunos años se hicieron algunos estudios para determinar las características y patrones de ejecución de las instrucciones máquina generadas por programas escritos en HLL. Fueron los resultados de estos estudios los que originaron la búsqueda de una aproximación diferente, es decir, una arquitectura que diera soporte a los HLL de una forma más natural, intentando reducir al máximo posible el hueco.

La comprensión de los resultados de estos estudios y de la forma que tienen los compiladores de gestionar las variables en pila (Apéndice A), son importantes para comprender la línea de razonamiento que llevó al desarrollo de la arquitectura RISC. Por ello, los siguientes apartados constituyen una revisión de las características de la ejecución de instrucciones. Esta revisión está dividida en:

- *Operaciones*: Se determina la función de las instrucciones que el procesador ejecuta y su interacción con memoria.
- *Operandos*: Tipos de operandos y su frecuencia de uso.
- *Procedimientos*: Llamadas y retornos de procedimiento.

6.2.1 Operaciones

Se han hecho una gran cantidad de estudios para analizar el comportamiento de los programas escritos en HLL. Estos han sido importantes para comprender por qué se produce el *gap* semántico, e intentar evitarlo. Sin embargo, como veremos a continuación, las conclusiones a las que se llega puede ser contradictorias dependiendo de la naturaleza de los estudios realizados.

6.2.1.1 Aparición Dinámica de Operaciones HLL

Se han hecho diversos estudios para analizar el comportamiento de programas escritos en lenguajes de alto nivel.

Lenguaje	PASCAL *Programas Científicos*	FORTRAN *Programas Estudiantes*	PASCAL *Servicios Sistema*	C *Servicios Sistema*
Assign	74	67	45	38
Loop	4	3	5	3
Call	1	3	15	12
If	20	11	29	43
Goto	2	9	-	3
Otras	-	7	6	1

Tabla 6.1: Frecuencia dinámica relativa de operaciones de HLL.

La tabla 6.1 recoge los resultados de la medida de la aparición de distintos tipos de operaciones durante la ejecución. En otras palabras, a partir del código fuente de un programa escrito en HLL, se cuenta el número de operaciones que se ejecutan durante el tiempo de vida de la aplicación. Se cuentan asignaciones, llamadas, saltos, condicionales, ... y se calcula la respectiva frecuencia de aparición en el código, en valor porcentual (nótese que los valores de la tabla 6.1 son valores porcentuales), de cada tipo de operación.

La tabla 6.1 refleja los resultados de varios estudios. Se han realizado diferentes estudios utilizando distintos HLL e incluso distinto tipo de programas. Es decir, algunos se centran sobre el código de servicios del sistema, otros sobre ejercicios de estudiantes, otros sobre programas que usan cálculos científicos de uso general, ...

Aunque hay, obviamente, diferencias entre los valores obtenidos en función del tipo de HLL y de la aplicación, hay bastante concordancia en varios aspectos:

- Las sentencias de asignación predominan, lo cual indica que el movimiento de datos tiene gran importancia.
- La aparición de sentencias condicionales (IF, LOOP) también es importante.
- Sin embargo, las llamadas y saltos (CALL, GOTO) suponen sólo una pequeña fracción de las sentencias ejecutadas durante el tiempo de vida del programa.

6.2.1.2 Frecuencia dinámica de instrucciones máquina relativas a operaciones HLL

Los resultados de la tabla 6.1 son muy indicativos al diseñador, debido a que las instrucciones que se ejecutan más a menudo deberían ser las instrucciones que se implementen de una forma más óptima.

Sin embargo, los anteriores resultados no revelan qué sentencias consumen más tiempo en la ejecución de un programa.

Para calcular esto, Patterson realizó un estudio consistente en compilar varios programas sobre diferentes plataformas y determinó el número medio de instrucciones máquina y referencias a memoria para cada tipo de sentencia. Lo que se persigue es determinar cuales son las sentencias de lenguaje HLL que producen la ejecución de la mayor parte de instrucciones en lenguaje máquina.

La tabla 6.2 muestra la frecuencia relativa de aparición para varias sentencias HLL. Para ello se cuentan las instrucciones en lenguaje máquina, durante la ejecución del programa, producidas por cada sentencia. *Por eso se trata de estadísticas de frecuencia dinámica*.

En las dos últimas columnas de la tabla 6.2 también aparece el número de referencias a memoria ponderadas, para cada tipo de sentencia. Es decir, el valor porcentual del número de referencias a memoria que produce cada sentencia durante la ejecución.

En la tabla 6.2 aparece en las columnas de aparición dinámica los mismos resultados de los estudios de Patterson que aparecen en la tabla 6.1. Estas se añaden para poder establecer una comparación con los resultados representados en las columnas siguientes.

Los resultados de la tabla 6.2 indican algo muy diferente de lo que un diseñador podría interpretar de la tabla 6.1. En aquella se refleja cómo el programa en ejecución dedica un porcentaje importante de las instrucciones máquina generadas a ejecutar sentencias tipo llamada

y bucle (CALL, LOOP), y por lo tanto serían las instrucciones que claramente tendrían que optimizarse. Esta conclusión es contradictoria, pues la aparición dinámica (también en tabla 6.2) parecía indicar justamente lo contrario.

	Aparición Dinámica		Instrucciones Máquina Ponderadas		Referencias a Memoria Ponderadas	
Lenguaje	**PASCAL**	**C**	**PASCAL**	**C**	**PASCAL**	**C**
Assign	45	38	13	13	14	15
Loop	5	3	42	32	33	26
Call	15	12	31	33	44	45
If	29	43	11	21	7	13
Goto	-	3	-	-	-	-
Otras	6	1	3	1	2	1

Tabla 6.2: Frecuencia dinámica relativa ponderada de operaciones de HLL.

Puede concluirse de estos estudios que las sentencias tipo bucle y llamada, son sentencias que en ejecución tienen una aparición dinámica muy baja y que, sin embargo, son las que mayor cantidad de instrucciones máquina generan (del orden del 70% en la tabla), y por tanto, su ejecución ocupa una parte muy importante del tiempo del programa.

Pero aún hay más, las columnas de referencias a memoria ponderadas nos indican que la ejecución de estas sentencias implica la gran parte de las referencias a memoria del programa.

Obviamente, el resultado de estos estudios establece que las sentencias HLL cuya ejecución ha de ser optimizada serán claramente las llamadas y bucles.

6.2.2 Operandos

	PASCAL	C	Promedio
Constantes enteras	16	23	20
Variables escalares	58	53	55
Estructuras y Matrices	26	24	25

Tabla 6.3: Resultados de estudio sobre porcentaje dinámico de operandos.

En la tabla 6.3 se muestran los resultados de un estudio de Patterson, que considera la frecuencia dinámica de aparición de diferentes tipos de variables. Los resultados indican que la

mayoría de las referencias se hacen a variables escalares simples (alrededor del 75%), y que más del 80% de los datos escalares eran variables locales.

- Dado que el acceso a operandos es una operación que se realiza frecuentemente, es importante que la arquitectura se diseñe para prestar un rápido acceso a operandos. Estos resultados muestran que el mejor candidato para la optimización sería el acceso variables escalares, más en concreto el almacenamiento y acceso a variables escalares locales.

6.2.3 Procedimientos

Del estudio de las características de ejecución de los procedimientos y los diagramas llamada/retorno de estos, se obtienen interesantes conclusiones.

6.2.3.1 Argumentos y variables locales

Anteriormente vimos, tabla 6.2, cómo las llamadas y retornos de procedimientos son las operaciones que más tiempo consumen en los programas HLL compilados. Si consideramos realizar estas operaciones más eficientemente debemos caracterizar el comportamiento de los procedimientos en tiempo de ejecución. Para ello hay dos aspectos que debemos considerar:

- El número de parámetros y variables.
- La profundidad de anidamiento.

Porcentaje de llamadas a procedimientos con:	Compiladores, programas de composición de textos	Reducios programas no numéricos.
> 3 argumentos	0-7%	0-5%
> 5 argumentos	0-3%	0%
> 8 palabras para argumentos y variables escalares locales	1-20%	0-6%
> 12 palabras para argumentos y variables escalares locales	1-6%	0-3%

Tabla 6.4: Estudio de argumentos y variables escalares locales en procedimientos.

Un estudio de Tanenbaum concluyó que a la práctica totalidad de los procedimientos llamados dinámicamente, el 98%, se les pasaba menos de seis argumentos y que el 92% de ellos usaban menos de 6 variables escalares locales. La tabla 6.4 muestra el resultado de otro estudio parecido y que, como vemos llega a resultados similares.

El resultado viene a decir que el número de palabras necesarias en el registro de activación de un procedimiento, para contener los parámetros del procedimiento y las variables escalares

locales, no es muy elevado. Como podemos ver en la tabla, con 12 palabras sería suficiente en la mayor parte de los casos.

6.2.3.2 Comportamiento llamadas/retornos

Los estudios de Patterson acerca del comportamiento de las llamadas/retorno durante la ejecución de un programa indican que es raro tener una secuencia larga ininterrrumplida de llamadas a procedimiento seguidas por la correspondiente secuencia de retornos. De hecho, los programas suelen quedar confinados a una ventana de profundidad bastante estrecha en el nivel de anidamiento de procedimientos.

Esto se entiende mejor a partir del diagrama de la figura 6.1. Éste muestra el resultado del análisis del comportamiento de un determinado programa en tiempo de ejecución. Pero sólo se centra en las llamadas y retornos a procedimientos (o funciones). Cada vez que se realiza una llamada aumenta la profundidad de anidamiento, y decrece cuando se produce un retorno. Así, en la figura 6.1 se disponen las abcisas en unidades de llamadas/retornos, siendo las ordenadas el tiempo en unidades de llamada/retorno.

En la figura 6.1 se tiene el patrón de llamadas/retorno obtenidas de la ejecución de un determinado programa. Como puede verse, se ha definido una ventana de profundidad de nivel 5. Cada vez que una llamada o retorno salen de dicha ventana, la ventana se reajusta.

Este estudio de Patterson demostró que una ventana de profundidad 8, sólo necesitaría desplazarse en menos de un 1% de las llamadas o retornos.

Esto indica que las referencias a a instrucciones tienden a localizarse en unos pocos procedimientos a lo largo de periodos de tiempo bastante largos. Esto es una confirmación del principio de localidad. Aunque éste no es el objeto de citar aquí este estudio, como veremos más adelante.

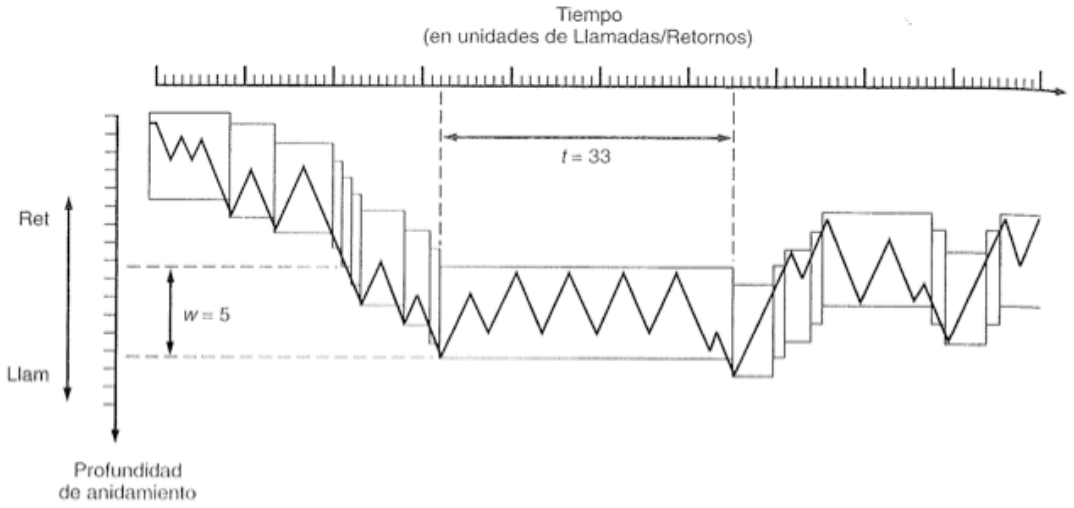


Figura 6.1: Estudio del comportamiento del patrón llamadas/retornos de un programa.
(Organización y Arquitectura de Computadores. W. Stallings)

6.2.4 Consecuencias

A partir de los estudios de la frecuencia dinámica relativa a instrucciones máquina de la tabla 6.2 se desprende que el programa en ejecución dedica un porcentaje importante de las instrucciones máquina generadas a ejecutar sentencias tipo llamada y bucle (CALL, LOOP), y por lo tanto serían las instrucciones que claramente tendrían que optimizarse. Esta conclusión es contradictoria a la tendencia del momento, que indicaba que justamente estas eran las que menos peso tendrían a la hora optimizar la arquitectura.

De hecho, las sentencias tipo bucle y llamada son sentencias que aunque tienen una aparición dinámica muy baja, son las que mayor cantidad de instrucciones máquina generan (del orden del 70% en la tabla 6.2), y por tanto, su ejecución ocupa una parte muy importante del tiempo del programa.

La ventaja de optimizar dichas sentencias es doble, pues los estudios, tabla 6.2, también indican que la ejecución de estas sentencias implica la mayor parte de las referencias a memoria del programa.

En este sentido, cabe notar que el tiempo de acceso a memoria constituye una gran penalización debido a que introduce tiempos de espera. Este tiempo depende de la plataforma de ejecución, pero para hacernos una idea el tiempo de acierto de lectura en caché puede ser de varios ciclos (hasta 4 ó 5 ciclos dependiendo de la arquitectura, y eso suponiendo que el dato está en la caché).

Se puede medir el comportamiento promedio de un programa sobre cierta arquitectura a partir de los accesos promedio a memoria y registro por instrucción. Un ejemplo es el estudio dinámico de la ejecución del DEC-10 de la que se desprende que cada instrucción referencia, en promedio, 0.5 operandos de memoria y 1.4 registros.

Conocida la penalización que supone para la arquitectura el acceso a memoria, cabe concluir que la optimización debería producirse en la dirección de reducir los accesos a memoria y aumentar la utilización de registros, que no tiene ningún tipo de penalización de acceso. De los estudios sobre la frecuencia dinámica de aparición de los diferentes tipos de operandos, se desprende que dado que la mayoría de las referencias se hacen a variables escalares simples (alrededor del 75%) y que de ellas más del 80% de los datos escalares son variables locales, lo que se tiene que optimizar es, sobre todo, el acceso a variables escalares locales.

Una posible solución a esto sería aumentar el número de registros y maximizar y optimizar su uso, para poder almacenar allí las variables escalares locales e incrementar así la eficiencia.

Finalmente, es un dato remarcable que, en cuanto a llamadas a procedimientos, el 98% posee menos de seis argumentos y de estos el 92% posee menos de seis variables escalares locales. Además, con 12 palabras sería suficiente para almacenar parámetros y variables escalares locales, en la gran mayoría de los casos. Lo que nos permite establecer la cantidad de memoria necesaria para la activación de cada procedimiento.

Finalmente, también hay que prestar atención a los cauces de instrucciones, debiendo tender a organizaciones sin penalización y políticas de predicción de saltos.

Pues bién, estas serán las claves para introducir alguna mejora en la organización del computador de forma que optimice las prestaciones (a través de la mejora de las características que más tiempo consumen), y reducir el gap semántico.

6.3 Utilización de un gran banco de registros

Del estudio de los resultados que se acaban de exponer, se puede llegar a la conclusión de que intentar realizar una arquitectura con un gran conjunto de instrucciones cercano al de los HLL no es lo más efectivo. En lugar de eso, la solución pasaría por diseñar una arquitectura que optimizara las claves, obtenidas de los estudios anteriores, que más tiempo consumen. Este es el planteamiento de la arquitectura RISC.

Así, la arquitectura RISC apareció, teniendo en común tres elementos clave:

- Usan un gran número de registros y/o el uso del compilador para optimizar el uso de los registros. La finalidad es optimizar las referencias a operandos. De esta forma, si el operando ocupa un registro se reducen las transferencias a memoria, y por tanto el tiempo de acceso y la eficiencia del programa.
- Optimización de la segmentación de instrucciones. Debido a la alta proporción de instrucciones de bifurcaciones condicionales y de llamada a procedimientos, hay que prestar atención al diseño de los cauces de instrucciones, para intentar evitar, en la medida de lo posible, penalizaciones por vaciado del cauce.
- Conjunto reducido de instrucciones. Se caracteriza por ser limitado y sencillo.

El almacenamiento en registro es el almacenamiento más rápido que existe. Es más rápido que la memoria principal y que la memoria caché. Esto es porque emplea direcciones mucho más cortas que las de memoria y porque se encuentra integrado en el mismo chip que la ALU y la Unidad de Control. También es por ello que el almacenamiento en registro suele ser bastante más reducido que el de la memoria caché o el de memoria principal.

Para reducir el número de accesos a memoria e incrementar el número de accesos a registro. Son posibles dos aproximaciones a este problema, una basada en software, donde es el compilador el que intenta maximizar y optimizar el uso de los registros, y otra basada en hardware, consistente en usar más registros que puedan contener más variables.

En los siguientes subapartados examinaremos ambas aproximaciones.

6.3.1 Solución Hardware: Ventanas de registros

6.3.1.1 Ventanas de registros

Dado que se plantea la necesidad de reducir los accesos a memoria, es obvio que la solución pasa por incrementar el conjunto de registros.

De hecho, dado que la mayoría de las referencias a variables es a variables escalares locales, el almacenamiento en registro se utilizaría en su mayor parte para almacenar variables locales al procedimiento. Sin embargo, esto plantea un problema, dado que las variables locales sólo se utilizan cuando un procedimiento está activo. Cuando éste realice una llamada, los registros ocupados por sus variables locales pierden la localidad y no volverán a ser utilizados hasta que el procedimiento llamado retorne. Esto implica que durante ese tiempo dichos registros quedarán inutilizables a menos que su contenido se guarde en memoria para que se puedan reutilizar.

Esta posibilidad implica transferencias de registro a memoria y no es deseable, a menos que sea totalmente necesario, pues una cadena de llamadas puede agotar los registros disponibles.

A continuación se plantea la estrategia planteada por el grupo RISC de Berkeley, que se usó para realizar la primera computadora comercial puramente RISC.

La solución se basa en los resultados presentados en el apartado 6.2. Sabemos que la gran mayoría de los procedimientos utilizan pocos parámetros de llamada y variables globales. Los estudios en 6.2.3.1 llegaron a la conclusión que alrededor de un 90% de los procedimientos llamados necesitaban menos de 12 palabras para almacenar parámetros y variables locales. En este sentido la solución fue dividir el banco de registros en pequeños conjuntos de registros, denominados ventanas de registros, dedicados cada uno a la activación de un procedimiento diferente, conteniendo parámetros y variables locales a dicho procedimiento.

Por otro lado, dado que en el estudio 6.2.3.2 se demostró que utilizando una ventana de profundidad de nivel 8, ésta sólo necesitaría desplazarse en menos de un 1% de las llamadas o retornos. Es decir, se indica cómo la profundidad de activación de los procedimientos fluctúa dentro de un rango relativamente pequeño, por lo que en este caso sería suficiente tener 8 ventanas de registros (aunque esta necesidad puede cambiar dependiendo del tipo de programas que vayan a ejecutarse en dicha máquina).

Todas las ventanas de registros utilizadas son de tamaño fijo. Pero, ¿Cómo se organizan estas ventanas? Resulta obvio que, debido a la visibilidad de las variables locales al procedimiento, solo habrá una ventana de registros activa, de forma que cuando se realice una llamada se producirá una conmutación de la ventana activa, donde almacenar las variables locales y parámetros del procedimiento llamado, y desde donde no podrá accederse a ninguna variable de cualesquier otra ventana de registros.

De esta forma, cada ventana de registros se asigna a un procedimiento distinto. Una llamada a un procedimiento hace que el procesador conmute automáticamente la ventana y active otra. Solo serán visibles aquellas variables contenidas en la ventana activa, no siendo accesibles las variables en cualesquier otra ventana, a menos que debido a sucesivos retornos se conviertan en activa.

Por otra parte, dado que un procedimiento debe pasar los parámetros al registro de activación del procedimiento llamado (ver Apéndice A), la ventanas deben permitir un solapamiento parcial para realizar el paso de parámetros.

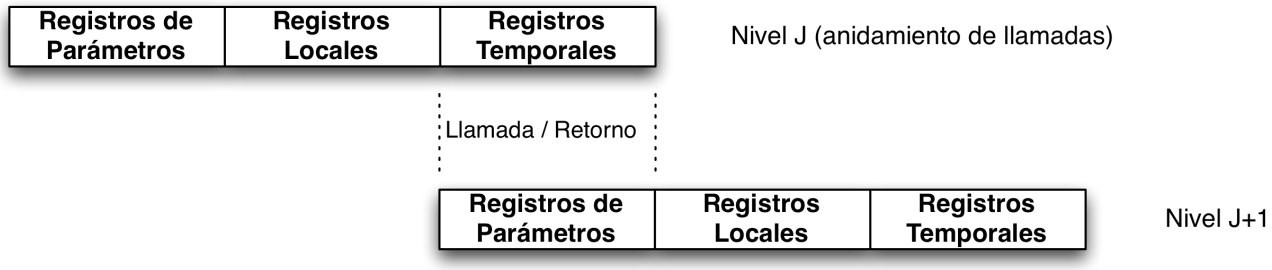


Figura 6.2: Solapamiento de ventanas de registros adyacentes.

El concepto se ilustra en la figura 6.2. Cada ventana se divide en tres áreas de tamaño fijo. Los registros de parámetros contienen los parámetros, pasados por el procedimiento que realizó la llamada, y también se utiliza para devolver a éste el resultado. El área de registros locales contiene las variables locales. El área de registros temporales sirve como almacenamiento temporal y para pasar los parámetros cuando se realiza la llamada a un procedimiento.

El área de registros temporales y el área de parámetros, de las ventanas de sucesivos niveles, se solapan. De hecho, son físicamente los mismos. Antes de realizar una llamada, el procedimiento sitúa los parámetros en el área de de registros temporales, de forma que cuando se realiza la llamada, la ventana activa conmuta y el procedimiento llamado dispone automáticamente de los parámetros. Nótese que esta ingeniosa organización permite pasar parámetros al procedimiento llamado sin que exista una transferencia real de datos (¿no es impresionante?).

Por otro lado, y dado que sólo hay un número limitado de ventanas, estas se organizan según una estructura de buffer circular como el representado por la figura 6.3. En ella vemos cómo hay un 'puntero de ventana en curso', CWP, que apunta a la ventana del procedimiento que actualmente está activo. De hecho, cuando una instrucción referencia un registro de la ventana, ésta es una referencia lógica, y el hardware utiliza este puntero como un *offset* para determinar el registro real que se está referenciando.

En la figura 6.3, el buffer circular contiene la activación de los procedimientos más recientes. Con cada llamada se conmuta a una nueva ventana activa, contigua a la anterior, desplazándose

en el sentido horario. De la misma forma, cada retorno supone un conmutación de la ventana activa a la ventana adyacente en el sentido antihorario.

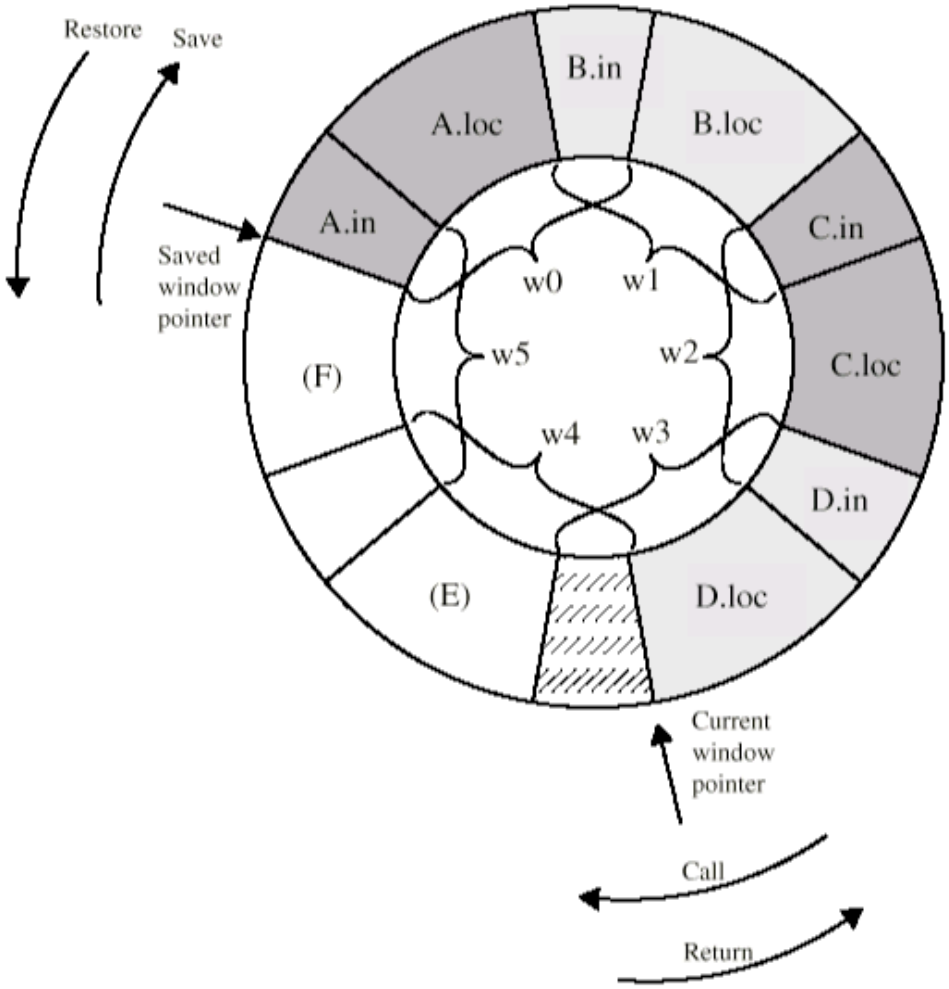


Figura 6.3: Organización de las ventanas como buffer circular.
(Organización y Arquitectura de Computadores. W. Stallings)

El 'puntero de ventana salvada', SWP, identifica la ventana guardada en memoria más recientemente. Si tras varias llamadas consecutivas de procedimientos se llena de ventanas el buffer circular, esto es, cuando tras el incremento de CWP este llega a ser igual a que SWP, se produce una interrupción cuya rutina se servicio guardará en memoria la ventana más antigua e incrementará SWP para poder proceder con la llamada en curso, y con ello con la ejecución del programa.

De los estudios de 6.2.3.1 se desprende que el número de ventanas no necesitaba ser demasiado grande. En dicho estudio una buffer circular de 8 ventanas hubiera bastado para asegurar que sólo se realizaba una salvaguarda o una restauración de la ventana a/desde memoria en un 1% de los casos.

Cuando se realiza una llamada, el puntero de ventana actual se mueve para mostrar la ventana de registro activo en curso. El puntero de ventana salvada indica donde ha de restablecerse la siguiente ventana salvada.

Pero ¿que ocurre si un procedimiento necesita más parámetros o más variables locales de las que caben en la ventana de registros? Muy sencillo, dichas variables se gestionarán en la pila, de la forma descrita en el Apéndice A.

Como ejemplo, los computadores RISC de Berkeley usaban 8 ventanas de 16 registros cada una. El computador Pyramid utilizaba 16 ventanas de 32 registros.

6.3.1.2 Variables globales

Si bien el esquema de ventanas proporciona una organización eficiente para almacenar las variables locales al procedimiento en los registros, éste no puede aplicarse a las variables globales, a las cuales debe poder accederse desde cualquier procedimiento.

La mejor solución para las variables globales es la utilización de un conjunto de registros fijos y accesibles a todos los procedimientos.

6.3.2 Solución software: Optimización de registros

El compilador también juega un gran papel en la optimización de los registros, pudiendo estudiar en tiempo de compilación cómo maximizar su uso. A este respecto, el compilador intentará mantener en registros la máxima cantidad de operandos y minimizar la necesidad de transferencias a/desde memoria.

Para ilustrar las técnicas de optimización de registros que usan los compiladores vamos a estudiar la técnica del 'Coloreado de Grafos'.

En esta técnica, el planteamiento inicial del compilador es asignar un registro simbólico a cada cantidad candidata para residir en registro. El objetivo del compilador es, entonces, asignar el número ilimitado de registros simbólicos a un número fijo de registros físicos. Para maximizar el uso de los registros físicos, el compilador buscará que varios registros simbólicos compartan un mismo registro físico, siempre y cuando su utilización no se solape en el tiempo. Si, finalmente, no se pueden asignar todos los registros simbólicos a registros físicos, aquellos registros simbólicos sin asignar deberán ser asignados a posiciones de memoria.

La técnica del 'Coloreado de Grafos' procede de la topología. En ella, el problema es construir un grafo, formado por nodos y arcos, asignando colores a los nodos de forma que nodos adyacentes tengan colores diferentes, de forma que se minimice el número de colores.

Para la construcción del grafo:

- Se dibuja un nodo por cada registro simbólico.

- Se analiza la secuencia de uso de los registros simbólicos. Para ello se acota el intervalo en el cual se usa el valor asociado a dicho registro simbólico. Se dice que en dicho intervalo temporal el registro simbólico está 'vivo'.

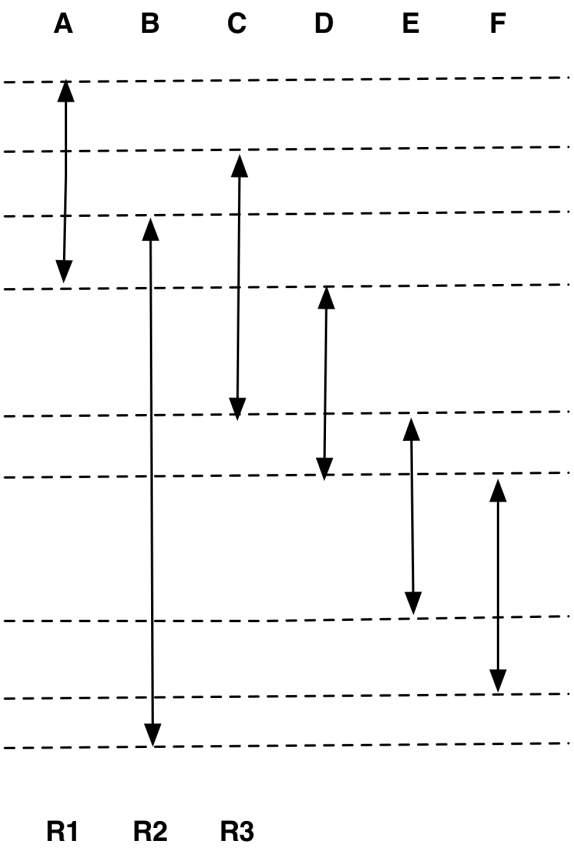
- Si dos registros simbólicos están 'vivos' durante el mismo fragmento de programa, se unen por un arco que indica su interferencia.

- Se intenta colorear el grafo con el mínimo número de colores. Se busca que como máximo se pueda colorear con n colores, con n el número de registros físicos. El gráfico se colorea de manera que nodos adyacentes tengan colores diferentes.

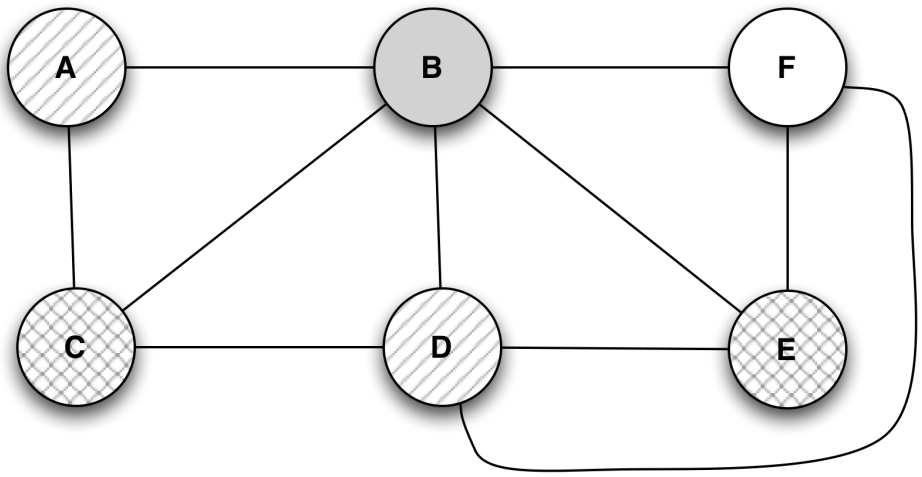
Una vez construido el grafo, necesitaremos tantos registros físicos como el numero de colores para albergar a todos los registros simbólicos. Sin embargo, si se han utilizado n colores (n igual al número de registros físicos) y han quedado registros simbólicos por colorear, éstos se asignarán a una posición de memoria. En estos casos se tienen que usar cargas y almacenamientos desde memoria para operar con ellos, cuando éstos se necesiten.

La figura 6.4 presenta un ejemplo sencillo de este proceso. En 6.4a tenemos la secuencia temporal que se obtiene de un programa con 6 registros simbólicos. La secuencia temporal indica cual es el tiempo de vida de cada uno de ellos y las interferencias entre ellos. En 6.4b se muestra el grafo de interdependencias entre registros, construido según el procedimiento descrito anteriormente. En este caso no se han usado colores, sino sombreados y tramas. Se ha intentado un posible coloreado con 3 colores porque se dispone de 3 registros físicos. Como vemos, hay un registro simbólico, el F, que queda sin colorear y, por tanto, se asignará a una posición de memoria. Por otra parte, el registro físico R1 puede albergar a los registros simbólicos A y D, coloreados igualmente en el grafo, pues no interfieren entre ellos. El registro físico R2 albergará solamente al registro simbólico B, pues no comparte coloreado con ningún otro. Y el registro físico R3 se comparte para los registros simbólicos C y E, pues también tienen tiempos de vida diferentes.

Así, queda patente cómo los compiladores juegan también un papel fundamental en maximizar el uso de registros, intentando que exista el mayor número de operandos en registro. Esto aumenta la eficiencia al reducir el número de transacciones con memoria necesarias.



(a) Intervalos temporales de uso de los registros simbólicos



(b) Grafo de interferencia entre registros.

Figura 6.4: Coloreado de grafos para optimización de registros.

6.4 Amplio banco de registros frente a caché

Un banco de registros organizado según un esquema de ventanas funciona como un pequeño y rápido buffer que contiene las variables que probablemente más se usen. Visto así, parece ser similar a una memoria caché. Entonces, ¿no sería igual utilizar una caché? La tabla 6.5 contiene las características principales de ambas soluciones, con el propósito de poder compararlas.

Mientras que la caché contiene los datos escalares usados recientemente, el banco de registros contendrá todos los datos escalares locales a los procedimientos (excepto, claro está, aquellos que hayan pasado a memoria por desbordamiento).

Cabe también indicar que el acceso a registro es más rápido que el acceso a caché y que según los estudios de Patterson rara vez ocurren los desbordamientos que producen el reajuste de la ventana de profundidad. Por otra parte, la caché hace un uso más eficiente del espacio, puesto que no se usan los registros de las ventanas que no están activas.

Por otro lado, en el banco de registros las variables se leen individualmente mediante referencias a registros, mientras que en una cache los datos se leen por bloques mediante referencias a memoria.

El banco de registros hace uso de memoria para salvaguardar una ventana cada vez que el buffer circular de ventanas se llena, reajustando así la ventana de profundidad. La memoria caché contiene sólo aquellas variables usadas recientemente, realizando la salvaguarda/restauración mediante el algoritmo de reemplazo. Sin embargo, como casi todas las caches son asociativas por conjuntos con un tamaño de conjunto pequeño, existe el peligro de que otros datos o instrucciones sobreescriban las variables usadas con frecuencia.

Banco de registros amplio	Caché
- Aloja a todos los datos escalares locales excepto desbordamiento ventana.	- Datos escalares locales recientemente usados.
- Direccionamiento de registro.	- Direccionamiento de memoria.
- Se leen variables individuales.	- Se leen bloques de memoria.
- Salvaguarda / restauración basadas en anidamiento.	- Salvaguarda / restauración basada en algoritmo de reemplazo.

Tabla 6.5: Comparación características de un banco de registros y de una caché.

Aunque ambas opciones pueden tener sus aspectos positivos y negativos, la balanza se decanta indudablemente hacia el banco de registros por una característica muy importante, y es que el tiempo de acceso a registro es claramente más rápido.

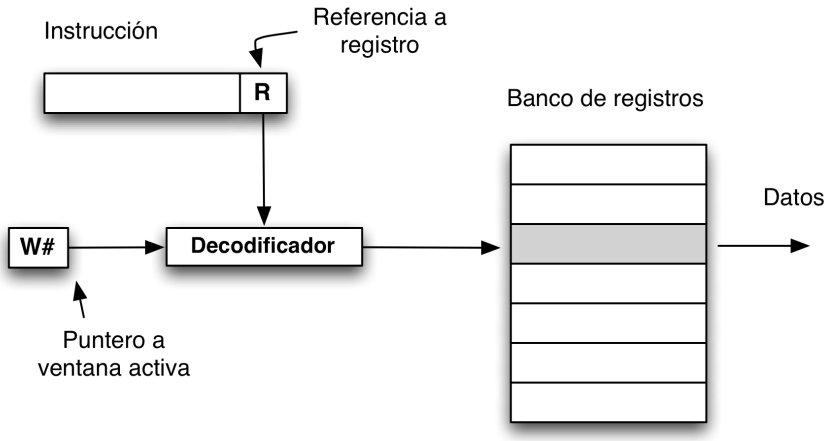


Figura 6.5: Decodificación en un banco de registros basado en ventanas.

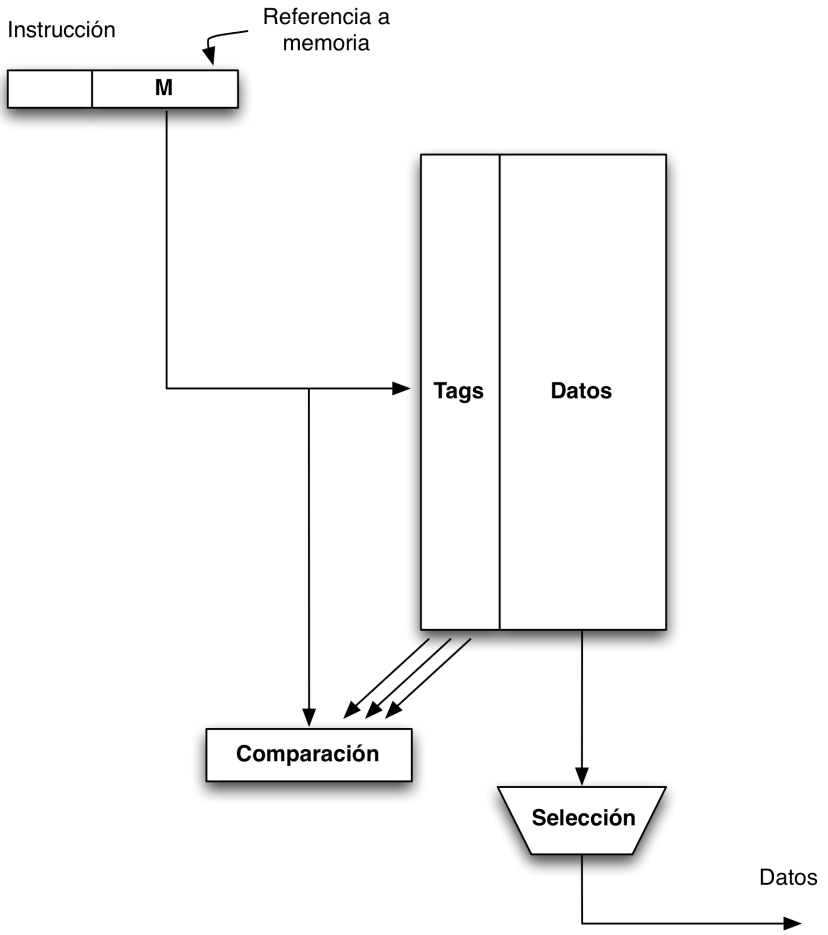


Figura 6.6: Referencia en una caché.

Otra diferencia entre la utilización de un banco de registros organizado en ventanas y una caché es la forma de decodificar la dirección. En el banco de registros basado en ventanas, la dirección del registro físico referenciado se decodifica sencillamente a partir de un número de registro lógico y el número de la ventana activa, figura 6.5. El funcionamiento de la memoria caché asociativa por conjuntos compara una parte de la dirección con una serie de etiquetas hasta seleccionar una palabra, figura 6.6.

6.5 Motivaciones contemporáneas de la arquitectura CISC

La tendencia hacia arquitecturas CISC fue motivada por el deseo de realizar máquinas que proporcionaran un mejor soporte a los HLL, simplificando los compiladores, mejorando las prestaciones ante el giro de los programadores hacia los HLL, y cerrando así el *gap* semántico.

Para ello, la tendencia inicial consistió en generar una instrucción máquina para cada sentencia de HLL. Sin embargo, esto tropezó con una realidad, las instrucciones máquina complejas son difíciles de utilizar, ya que el compilador debe encontrar los casos a los que se ajusta su utilización. El resultado final es que los compiladores raramente encuentran ocasiones en las que utilizar las instrucciones complejas, y que la mayoría de instrucciones que genera un programa compilado son, en su mayoría, las más sencillas.

Dos argumentos importantes a favor del uso de CISC, fue la esperanza de obtener programas más pequeños y rápidos:

- El hecho de que un programa sea más pequeño tiene sus ventajas. Tendrá menos instrucciones que captar, y por ello menos transferencias con memoria, reduciendo además las falta de página.

	Tamaño relativo del código (Varios programas en C)
RISC I	1,0
VAX - 11 / 780	0,8
M68000	0,9
Z8002	1,2
PDP - 11 / 70	0,9

Tabla 6.6: Tamaño del código, en relación al RISC I.

Sin embargo, aunque es cierto que un programa escrito en CISC puede tener menos instrucciones, esto no quiere decir que el número de bits que ocupa el programa sea menor. En la tabla 6.6 se muestra un estudio que compara el tamaño obtenido de ciertos programas en C, compilados en varias máquinas, incluyendo el RISC I, considerado RISC puro.

En la tabla 6.6 los tamaños de los programas están normalizados, relativos al tamaño del programa en el RISC I. Nótese como la diferencia entre el tamaño de los códigos no es sustancial. De hecho, una máquina CISC, el Z8002, generó un programas de tamaño mayor que el del propio RISC I.

Las razones de estas escasas diferencias en los tamaños resultantes son varias. Una es el hecho de que las instrucciones complejas de los CISC raramente encuentran ocasión para ser usadas. Otra es que las instrucciones RISC son más cortas que las CISC, y por ello necesitan menos bits y así generar un tamaño idéntico que las CISC utilizando más instrucciones.

- Tampoco es cierto que la ejecución de programas CISC sea más rápida. Una razón es la misma que se esgrimió anteriormente, la tendencia de los compiladores CISC a utilizar instrucciones sencillas. Por otro lado, la unidad de control CISC utiliza microprogramación para implementar instrucciones complejas, lo cual complica la unidad de control. Frente a esto, la sencillez de las unidades de control RISC permiten utilizar mayores frecuencias de reloj y, por tanto, incrementar la velocidad de ejecución.

6.6 Características de la arquitectura RISC

Hay unas pocas características comunes a todas las aproximaciones a la arquitectura de reducido conjunto de instrucciones:

- **Instrucción de un solo ciclo**.

  Las instrucciones RISC son sencillas y se ejecutan en un ciclo de reloj. De esta forma, tienen poca o ninguna necesidad de microcódigo; las instrucciones máquina suelen estar cableadas. Esto implica que no hay que acceder a la memoria de control del microprograma durante la ejecución de la instrucción, y que las instrucciones se pueden ejecutar más rápidamente, permitiendo reducir el ciclo de reloj.

- **Operaciones registro a registro**.

  La mayoría de las instrucciones realizan operaciones registro a registro, con la excepción de las instrucciones LOAD y STORE, que se utilizan para acceder a memoria. Esto simplifica el conjunto de instrucciones y, por tanto, simplifica aún más la unidad de control.

Los valores accedidos frecuentemente permanecen en el almacenamiento de alta velocidad.

- **Modos de direccionamiento sencillos**.

  Casi todas las instrucciones utilizan el modo de direccionamiento a registro, aunque haya otros más complejos como el de desplazamiento o el relativo al contador.

  Los modos de direccionamiento más complejos se sintetizan en base a los sencillos.

  En general las arquitecturas RISC poseen un pequeño número de modos de direccionamiento: normalmente menos de 5 modos de direccionamiento.

- **Formatos de instrucción sencillos**.

  Generalmente, las arquitecturas RISC usan uno o pocos formatos de instrucción.

La longitud de las instrucciones es siempre fija. El número de bits del campo de código de operación también es fijo. Esto tiene varias ventajas, como el tamaño de los campos es fijo, la decodificación del código y el acceso a los operandos en los registros se puede realizar simultáneamente. Además, los formatos sencillos simplifican aún más la unidad de control y el tamaño fijo optimiza la captación de instrucciones.

6.7 RISC vs. CISC

Ha habido una creciente tendencia a pensar que los diseños RISC pueden sacar provecho de la inclusión de algunas características CISC, y viceversa. El resultado es que los actuales diseños no son RISC puros. Siguiendo una línea argumental similar, los diseños CISC actuales incorporan características RISC. Esto impide una comparativa que pueda declarar la superioridad de una u otra arquitectura.

Los problemas de la comparación entre arquitecturas son varios:

- No existe una pareja de máquinas RISC y CISC directamente comparables.

- Los resultados pueden depender del conjunto de programas de prueba.

- Es difícil separar los efectos del hardware y de la habilidad del compilador.

Sin embargo, lo que sí puede estudiarse son los aspectos ventajosos de una u otra arquitectura. Las ventajas potenciales de la aproximación RISC son muchas:

- **Sencillez de la unidad de control**. La instrucciones RISC poseen una sencillez intrínseca. El hecho de poseer una longitud fija, formato de instrucción único (o pocos formatos de instrucción), pocos modos de direccionamiento, y operar básicamente con registros,

propicia la sencillez de la unidad de control (CU). Es razonable pensar que estas unidades de control carecerán de instrucciones microprogramadas y tendrán todas las instrucciones máquina cableadas. De esta forma, las instrucciones pueden ejecutarse más rápidamente que en un CISC comparable.

- **Los programas RISC son más sensibles a interrupciones**, ya que la comprobación es más frecuente que en otras arquitecturas con instrucciones complejas.

- La arquitectura **RISC puede aplicar más eficazmente las técnicas de segmentación** de instrucciones.

- **Reducción del área ocupada por la unidad de control**. En un microprocesador CISC típico la unidad de control (CU) suele ocupar una gran parte del área en silicio (hasta el 50%), mientras que un RISC necesita un área reducida. Por ejemplo, la unidad de control del RISC I de Berkeley ocupaba únicamente el 6% del área de silicio.

	CISC			RISC	
	IBM 370/168	**DEC VAX**	**Intel 486**	**Motorola 88000**	**MIPS R4000**
Número de instrucciones	208	303	235	51	94
Tamaño de instrucción (en bytes)	2-6	2-57	1-11	4	32
Modos de direccionamiento	4	22	11	3	1
Registros de uso general	16	16	8	32	32
Memoria de control (en kbits)	420	480	246	0	0

Tabla 6.7: Diferencias entre procesadores.

La tabla 6.7 muestra algunas de las diferencias entre procesadores RISC y CISC. En ésta podemos observar las diferencias entre procesadores RISC y CISC en relación a varias características.

Es destacable en RISC el reducido número de instrucciones, el bajo número de modos de direccionamiento, el tamaño de instrucción fijo, un mayor número de registros y la inexistencia de memoria de control en la unidad de control (CU).

6.8 Segmentación del cauce

La naturaleza de las instrucciones RISC es sencilla y regular. Con este tipo de instrucciones se pueden utilizar de manera eficaz los esquemas de segmentación.

Como se vió en el tema 1, la segmentación del cauce produce una mejora sustancial de las prestaciones, como máximo por un factor igual al número de vías del cauce. En la figura 6.7 se ilustra un ejemplo de cauce de tres vías. Nótese que se han introducido dos instrucciones NOOP.

Éstas muestran las típicas situaciones que penalizan y reducen la velocidad de ejecución: las dependencias de datos y el vaciado del cauce en las bifurcaciones.

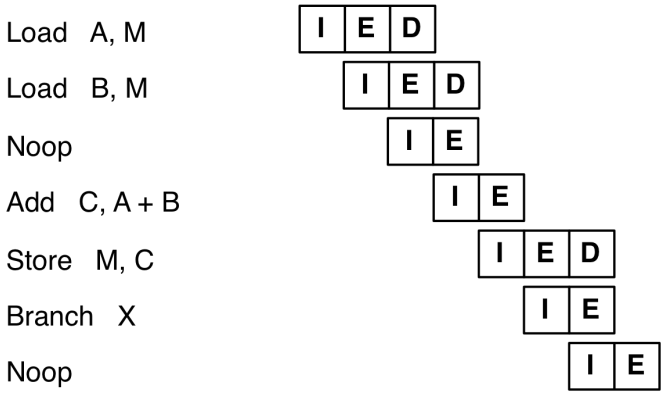


Figura 6.7: Dependencia de datos en un cauce de tres vías.

La dependencia de datos se estudiará más detalladamente en el próximo tema. En la figura 6.7, la dependencia de datos es la responsable de introducir una instrucción NOOP tras la instrucción de carga LOAD B,M. Nótese que la introducción de la instrucción NOOP evita que se ejecute la instrucción ADD C,A+B antes de que la instrucción LOAD B,M cargue el registro. Esto se hace porque la instrucción ADD debe tener un contenido válido en el registro B cuando se ejecute, y con la operación NOOP se fuerza la espera de la ejecución de la instrucción ADD para asegurar que la etapa de carga del registro desde memoria haya finalizado.

Por otro lado, la instrucción NOOP situada tras el salto indica el ciclo necesario para regularizar el cauce tras el salto. Dicho ciclo supone una penalización. Nótese que durante la etapa de ejecución del salto se adquiere la instrucción siguiente al salto, que no será ejecutada. Durante el siguiente ciclo se captará la instrucción destino del salto. Esto implica una penalización durante el llenado del cauce.

Para compensar estas dependencias, se han desarrollado las siguientes técnicas de reorganización del código:

- **Salto retardado.**

El salto retardado es una forma de evitar la penalización por vaciado del cauce inherente a los saltos, incrementando la eficacia de la segmentación. Esta técnica consiste en que el salto no tenga lugar hasta después de la ejecución de la siguiente instrucción (de ahí el nombre de retardado).

La primera columna de la tabla 6.8 ilustra el proceso normal de salto. En ella se tiene un trozo de código cuya instrucción en la dirección 102 (JUMP 105) contiene un salto a la dirección 105. Es obvio que tras el salto habrá un ciclo de penalización, porque durante la etapa de ejecución

del salto se produce la captación de la siguiente instrucción (ADD A,B). Sin embargo, esta instrucción no se ejecutará. Será durante el siguiente ciclo cuando se capte la instrucción destino del salto. En definitiva, este salto se produce con una penalización de un ciclo debido a la regularización del cauce.

Dirección	Salto normal	Salto retardado
100	Load M, A	Load M, A
101	Add 1, A	Jump 105
102	Jump 105	Add 1, A
103	Add A, B	Add A, B
104	Sub C, B	Sub C, B
105	Store A, Z	Store A, Z

Tabla 6.8: Ejemplo salto retardado.

La técnica de salto retardado se ilustra en la segunda columna de la tabla 6.8. Esta técnica asume que la instrucción después del salto también se ejecuta. Por ello las instrucciones en 101 y 102 están intercambiadas respecto al código en la primera columna. De esta forma, tras el salto se ejecuta la instrucción intercambiada, de forma que, mientras esta se ejecuta, se está captando la instrucción destino del salto. Con lo que el próximo ciclo se ejecutará la instrucción destino del salto. Nótese que utilizando esta curiosa técnica se elimina la penalización por regularización del cauce (siempre que la condición del salto no se vea afectada por la instrucción reordenada).

- **Carga retardada.**

Una técnica similar, denominada 'carga retardada' se puede utilizar para evitar las dependencias de datos que ocurren con las instrucciones LOAD, de carga desde memoria. Esta técnica supone también una reordenación del código.

La 'carga retardada' afecta a las instrucciones de carga desde memoria, LOAD. En estas instrucciones, el procesador bloquea el registro destino de la carga (que no se desbloqueará hasta que la carga finalice). Después, el procesador continúa la ejecución de las instrucciones siguientes hasta que alcance una instrucción que utilice dicho registro, en cuyo caso se detiene hasta que finalice la carga. De esta forma, el compilador puede reorganizar las instrucciones adelantando la instrucción de carga, de forma que el procesador pueda trabajar mientras se realiza la carga, evitando así esperas y aumentando la eficiencia de la ejecución.

6.9 Ejemplos

6.9.1 SPARC

SPARC (*Scalable Processor Architecture*, arquitectura de procesador escalable) es una arquitectura de Sun Microsystems inspirada en la máquina RISC I de Berkeley. De ella toma fielmente su conjunto de instrucciones y organización de registros en ventanas.

En este apartado se describe muy brevemente las organizaciones de registros y formatos de instrucción de esta arquitectura comercial, sirviendo como ejemplo de arquitectura RISC.

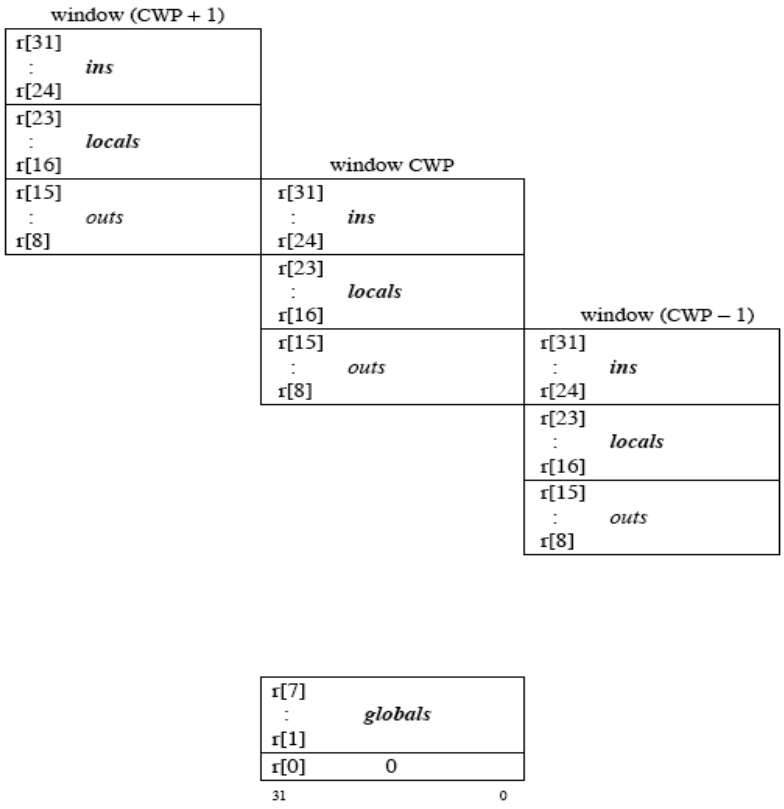


Figura 6.8: Solapamiento entre ventanas adyacentes.

El SPARC, como el RISC I de Berkeley, utiliza ventanas de registros. Cada una de las ventanas contiene 24 registros, y el número de ventanas depende de la implementación, pudiendo variar de 2 a 32 ventanas. La figura 6.8 muestra una implementación de 8 ventanas. Aunque hay 136 registros físicos, los procesos solo utilizan 32 registros lógicos. Los registros 0 a 7 son los registros globales, y los registros 8 a 31 hacen referencia a los registros de la ventana activa (nótese cómo los demás no son visibles).

La figura 6.9 muestra las ventanas de registros configuradas en forma de buffer circular. Utilizando esta arquitectura no es necesario, normalmente, guardar y restaurar registros en las llamadas y retornos de procedimiento.

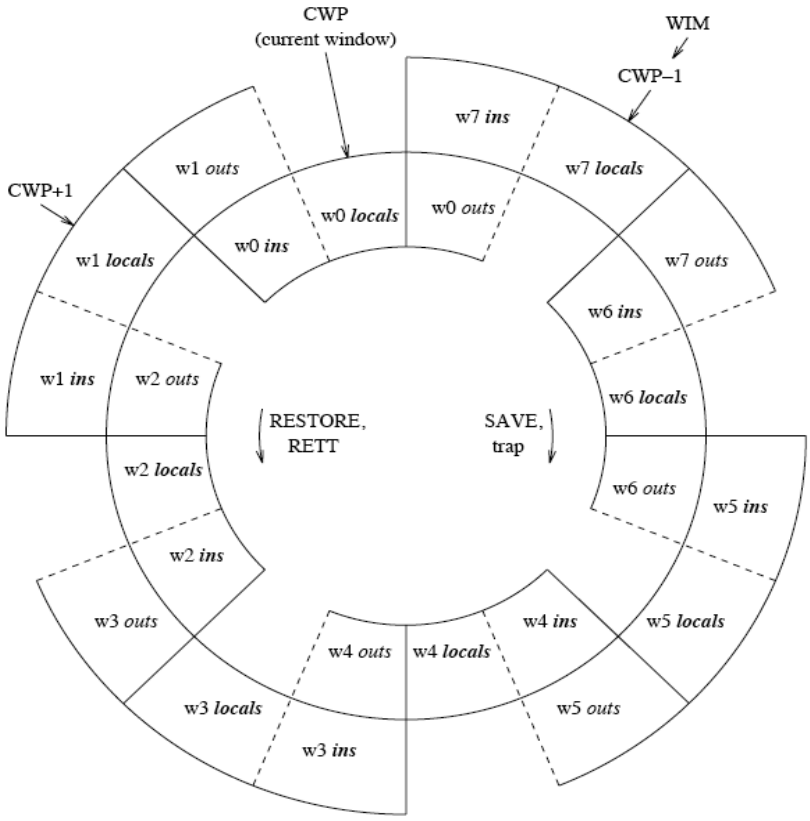
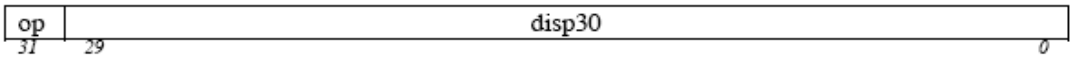


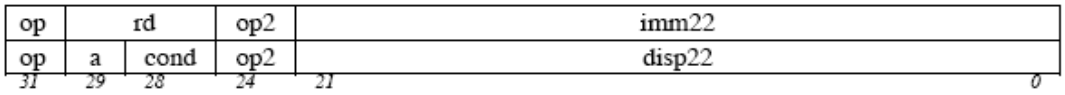
Figura 6.9: Estructura de buffer circular de ventanas.

En cuanto a los formatos de instrucción, el SPARC utiliza un conjunto de formatos de instrucción sencillo, figura 6.10. Nótese que hay tres formatos de instrucción muy específicos, una corresponde a la instrucción de llamada, otro a la instrucción de bifurcación y otro a la instrucción SETHI. Todas las demás instrucciones utilizan el formato general.

Format 1 (*op* = 1): CALL



Format 2 (*op* = 0): SETHI & Branches (Bicc, FBfcc, CBccc)



Format 3 (*op* = 2 or 3): Remaining instructions

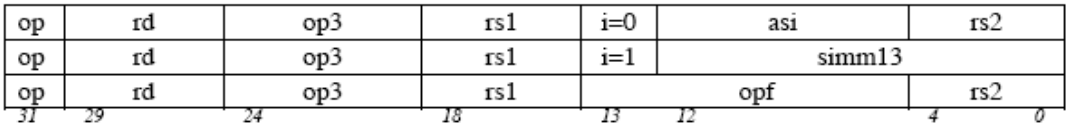


Figura 6.10: Formato de instrucción.